Point-free Program Calculation

Manuel Alcino Pereira da Cunha

Departamento de Informática
Escola de Engenharia
Universidade do Minho
2005

Dissertação submetida à Universidade do Minho para obtenção do grau de Doutor em Informática, ramo de Fundamentos da Computação

Acknowledgments

First of all, I would like to thank José Bernardo Barros for accepting being my supervisor. Eventually, the supervision extended to many other issues besides research, and evolved into a friendship that I really treasure. Many tanks also to Jorge Sousa Pinto, who joined this “boat” half-way through, but whose great supervision was key to the conclusion of this work.

Many other people deserve scientific acknowledgment: José Nuno for making me a point-free maniac; Bacelar for clarifying so many tricky questions; José Espírito-Santo for the valuable tip concerning the categorical abstract machine; Thorsten Altenkirch for pointing out some relevant work concerning decidability of equality; and José and Miguel for helping me tuning the pointwise to point-free translation.

For making the department (and not only) such a fun place to be, thank you again Zé, Jorge, and Becas, but also Paulo, Manuel, Carlos, Victor, Orlando, João, Nestor, and Mané. To my friends, Xano, Luís, and Paula, special thanks are due for all the support and encouragement during this work. And for being so important (and so patient) in this last year, Rosa definitively deserves a long and tender kiss. And by the way, *tu es ravissante*!

Finally, I would like to thank my sisters, Angelina and Natália, for being my biggest fans, and my parents, Glória and Manuel, for raising me so wisely and for the unconditional support in all aspects of my life. I love you all! To my grandparents, who I miss so much, I dedicate this thesis.

This thesis was partially supported by FCT research project Pure under contract POSI/ICHS/44304/2002.

"Since the beginning of the century, computational procedures have become so complicated that any progress by those means has become impossible, without the elegance which modern mathematicians have brought to bear on their research, and by means of which the spirit comprehends quickly and in one step a great many computations.

It is clear that elegance, so vaunted and so aptly named, can have no other purpose ...

Go to the roots of these calculations! Group the operations. Classify them according to their complexities rather than their appearances! This, I believe, is the mission of future mathematicians. This is the road on which I am embarking in this work."

Evariste Galois. From the preface to his final manuscript. 1832.

Abstract

Due to referential transparency, functional programming is particularly appropriate for equational reasoning. In this thesis we reason about functional programs *by calculation*, using a program calculus built upon two basic ingredients. The first is a set of *recursion patterns* that allow us to define recursive functions implicitly. These are encoded as higher-order operators that encapsulate typical forms of recursion, such as the well-known *foldr* operator on lists. 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 basic combinators are derived from standard categorical constructions, and are characterized by a rich set of equational laws. In order to be able to apply this calculational methodology to real lazy functional programming languages, a concrete category of partial functions and elements is used.

While recursion patterns are already well accepted and a lot of research has been carried out on this topic, the same cannot be said about point-free programming. This thesis addresses precisely this component of the calculus. One of the contributions is a mechanism to translate classic pointwise code into the point-free style. This mechanism can be applied to a λ-calculus rich enough to represent the core functionality of a real functional programming language. A library that enables programming in a pure point-free style within Haskell is also presented. This library is useful for understanding the expressions resulting from the above translation, since it allows their direct execution and, where applicable, the graphical visualization of recursion trees. Another contribution of the thesis is a framework for performing point-free calculations with higher-order functions. This framework is based on the internalization of some basic combinators, and considerably shortens calculations in this setting. In order to assess the viability of mechanizing calculations, several issues are discussed concerning the decidability of equality and the implementation of fusion laws.

Resumo

Devido à transparência referencial, o paradigma funcional é particularmente adequado ao raciocínio equacional. Nesta tese a manipulação de programas funcionais será feita *por cálculo*, sendo o cálculo de programas constituido por dois ingredientes fundamentais. O primeiro é um conjunto de *padrões de recursividade* que nos permite definir funções recursivas implicitamente. Estes padrões são codificados como operadores de ordem superior que ecapsulam formas de recursão típicas, tal como o bem conhecido operador *foldr* para listas. O segundo ingrediente é um estilo de programação "*point-free*", no qual os programas são definidos por combinação de funções mais simples sem nunca mencionar explicitamente os seus argumentos. Os combinadores fundamentais são derivados de construções categoriais padrão, e são caracterizados por um conjunto expressivo de leis equacionais. Para ser possível aplicar este método de cálculo a linguagens de programação funcional "lazy", foi usada uma categoria concreta onde as funções e os elementos podem ser parciais.

Ao contrário dos padrões de recursividade, que já são bem aceites e sobre os quais já se fez muita investigação, o mesmo não se pode dizer sobre a programação "point-free". Esta tese aborda precisamente este componente do cálculo. Uma das contribuições é um mecanismo que permite traduzir código "pointwise" clássico para o estilo "point-free". Este mecanismo pode ser aplicado a um λ-calculus suficientemente expressivo para representar a funcionalidade básica de uma liguagem de programação funcional real. Também se apresenta uma biblioteca que permite programar num estilo "point-free" puro dentro da linguagem Hankell. Esta biblioteca é útil para compreender as expressões que resultam da tradução acima referida, pois permite a sua execução directa e, quando aplicável, a visualização gráfica de árvores de recursividade. Outra contribuição da tese consiste numa metodologia para realizar cálculos "point-free" sobre funções de ordem superior. Esta metodologia é baseada na internalização de alguns combinadores fundamentais, e permite encurtar significativamente os cálculos. Para estabelecer a viabilidade de mecanização, também se discutem várias questões relacionadas com a decidibilidade da igualdade e a implementação de leis de fusão.

Contents

Chapter 1

Introduction

Functional programming is particularly appropriate for equational reasoning. Due to *referential transparency*, expressions in a functional programming language behave as ordinary mathematical ones, meaning that expressions denoting the same value can be interchanged without concern for the meaning of the surrounding context. This fact has been known for a long time, and put into practice for program reasoning at least since Burstall and Darlington [BD77] introduced the fold/unfold technique, and Backus [Bac78] 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 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.

In this thesis we use a different framework for reasoning about functional programs: *by calculation.* Essentially a program calculus consists of a collection of equational laws allowing to prove semantic equivalence between programs, or else to derive programs from other programs, or from their *specifications.* Quoting Backus [Bac78]:

> *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.*

Some classic strategies for program transformation have been introduced using this framework, such as Bird's *accumulation* strategy [Bir84]. 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 [THT98].

The program calculus used in this thesis is built upon two basic ingredients. The first is a set of *recursion patterns* that allow us to define recursive functions implicitly. These are encoded as higher-order operators that encapsulate typical forms of recursion such as the well-known *foldr* 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 favor 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. Since the choice of recursion patterns and point-free combinators was mainly driven by the power of the associated laws, this programming paradigm is usually known as *Algebraic Programming.*

While recursion patterns are already widely used by functional programmers, the same cannot be said about point-free programming. Although there are obvious advantages in using this style – the absence of variables and λ-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 Jeremy Gibbons on the advantages of calculating in the point-free style [Gib99]:

> *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.*

Even if programmers were forced to program in the point-free style there would still be a large amount of legacy code to which we would like to apply our reasoning methodology. As such, in this thesis the point-free style will be used just for calculations, and we assume that the programs one wants to reason about are defined in pointwise. A useful comparison here is that of mathematical transforms such as the Fourier transform or the Laplace transform, which allow to express functions in different domains in which certain manipulations are easier to perform.

This methodology raises a number of pertinent questions, most of them concerning the point-free aspect of the paradigm. As said above, recursion patterns are already well accepted, and indeed a lot of research, both theoretical and practical, has been carried out on this topic. One possible question could be: to what extent will calculations be committed to the point-free style? Readers acquainted with the subject could point out that when functions become a little more hairy, even the authors that clearly advocate this style introduce some variables in order to simplify calculations. That is the case with features nowadays so beloved such as curried or higher-order functions. To this question we have a radical answer. Although we agree that some calculations can be very long and tedious, we don't think this is an intrinsic disadvantage of point-free, but merely a consequence of the lack of adequate combinators and a solid proof methodology. As such, one of the objectives of this thesis is precisely to improve the machinery that is used to perform point-free calculations, so that they can be done entirely within this style.

Another question that immediately pops up is: how will pointwise code be translated into point-free style? Interestingly, the need for such an algorithm was recently mentioned in a functional programming mailing list, in a thread concerning the advantages/disadvantages of the point-free style [Kly05]:

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

This question leads to another goal of this thesis: to define mechanisms to translate between pointwise and point-free, and vice-versa, so that the programmer is free to choose whatever style is more adequate in any given context.

Once more, the above quotation reinforces what seems to be commonly accepted in the functional programming community, but in fact we must also ask: is it really easy to reason about programs written in point-free style? This is obviously true if reasoning just means the mechanical process of applying the equational laws as rewrite rules. But there are other (perhaps more) important issues concerning an equational theory, namely decidability. Among the many authors who advocate point-free reasoning, we failed to find a single one who addresses this issue, and found only a few who have attempted to mechanize proofs or transformations concerning point-free programs. So a third goal of the thesis is to help clarify this issue.

Another question that is relevant in any program reasoning and transformation context is: can this methodology be applied to real programming languages? The last main goal of the thesis is thus to answer this question positively. To be more specific, our target is the lazy functional programming language Haskell [Jon03]. As mentioned above, the set of basic point-free combinators is derived from standard categorical constructions. Most of the work presented here uses a concrete category of partial functions and elements, that is typically used to give semantics to lazy programming languages. In practice, this means that the calculus is not so neat, and some laws are polluted by strictness side conditions. This is the price to pay for using the appropriate semantics, which is not the case in most published work on this subject.

The thesis includes a large set of examples defined using recursion patterns and in the point-free style. To emphasize the connection with Haskell, most of the examples begin with the equivalent definition given in this language. In program transformation examples, that will also be the case with the transformed program. Finally, whenever possible, we tried to deploy our research results as practical tools and libraries that operate on Haskell programs instead of some abstract syntax. Haskell was also the language of choice to develop them.

Overview of the Thesis.

Chapter 2 presents the foundations of the algebraic programming paradigm. It starts with a brief presentation of category theory. After enumerating the basic set of point-free combinators and the associated laws, it describes how recursive types can be modeled in a categorical setting. Our presentation of recursion patterns differs slightly from the usual one. Given the concrete category underlying our presentation, we can assume the existence of just one basic recursion pattern – the *hylomorphism*. This recursion pattern, whose presentation concludes Chapter 2, has an expressive power equivalent to that of the fixpoint operator. This expressiveness is put into use in Chapter 3, where hylomorphisms are used to define some of the best known recursion patterns, namely *folds* and *unfolds*. We also show how to derive their laws from the basic set that characterizes hylomorphisms.

Some less known recursion patterns, such as *paramorphisms* or *accumulations*, are also introduced in Chapter 3. Although these recursion patterns were originally defined using folds, we show how they can be defined using hylomorphisms. This exercise reveals one of the advantages of using this recursion pattern. For some complex functions, a definition using hylomorphisms is much easier to understand than one using just folds or unfolds. In fact, a hylomorphism can be seen as the composition of a function that builds some intermediate data structure, and another one that consumes it. This producer/consumer model usually simplifies the comprehension of the defined function. As such, hylomorphisms make an interesting contribution to program understanding.

The main contributions of the thesis start in Chapter 4. This chapter describes how Bird's accumulation strategy [Bir84] can be applied in a pure point-free setting. The main problem with this technique is that it implements accumulations as higher-order folds that return a value of functional type (in Chapter 3 accumulations are defined in the uncurried form using tuple arguments). To simplify reasoning in this setting some of the basic combinators introduced in Chapter 2 are internalized as point-free definitions. This approach allows us to derive some fairly advanced examples, namely functions with two accumulating parameters. This chapter is a revised version of a paper to appear [CP05].

Many program transformation techniques, including Bird's accumulation strategy, can be implemented using fusion. This law states the conditions in which the composition of a function with a recursion pattern can be transformed into a single recursion pattern. Given its relevance, Chapter 5 presents a brief review of practical transformation systems based on this law. As will be seen, there is already some solid work on the implementation of fusion in the pointwise setting, but unfortunately very little on the point-free counterpart.

Some of the examples presented in the first chapters reinforce the idea that, indeed, it can be very difficult to program and understand definitions in the point-free style. Part of this problem could be alleviated if the programmer was able to execute and type-check his specifications. With this idea in mind, we have developed *Pointless Haskell*, a library that enables programming with recursion patterns in a pure point-free style. The implementation of this library is described in Chapter 6. The polytypic implementation of the recursion patterns is inherited from PolyP [NJ03], a generic programming library for Haskell. With the help of some type system extensions, we have implemented an implicit coercion mechanism that provides a limited form of structural equivalence between types. Although the library is embedded in Haskell, this mechanism allows us to use a syntax almost identical to the theoretical one defined in Chapter 2. This feature allows for a direct encoding of all the examples presented in this thesis. To further support program understanding, the library also includes a generic visualization mechanism for the intermediate data structures of hylomorphisms. This mechanism improves a previous version developed for visualizing recursion trees of Haskell functions [Cun03], and is implemented using the graphical debugger GHood [Rei01].

Chapter 7 defines a mechanism to derive point-free hylomorphisms from explicitly recursive pointwise definitions. The chapter begins with the definition of a simply typed λ-calculus with pairs and terminal object, which is subsequently enriched with sums, explicit recursion, recursive data types, and pattern-matching. We believe that these features are rich enough to cover the core functionality of most functional programming languages, namely Haskell. The translation to point-free style is based on the well-known equivalence between simply typed λ-calculi and cartesian closed categories suggested by Lambek [Lam80]. We show how to extend this equivalence in order to cover sums and the fixpoint operator. To handle sums we again resort to the internalization of a primitive combinator. The fixpoint operator can be encoded as a hylomorphism, but since this encoding yields expressions that are hard to manipulate, we also show how to adapt to our λ-calculus a well-known hylomorphism derivation algorithm that yields more understandable definitions. None of these ingredients is entirely new, but, to our knowledge, it is the first time they are put together in order to build an effective pointwise to point-free translation mechanism.

Chapter 8 presents a review of previous work on the decidability of equality in *almost bicartesian closed categories*, i.e, categories with products, non-empty coproducts, exponentials and terminal object. Likewise to fusion mechanization, this problem is already well studied at the pointwise level, but very few results exist for point-free. Namely, the problem has been shown to be decidable for pointwise terms using both normalization by rewriting and by evaluation. We show that a direct adaptation of the normalization procedure based on an expansionary rewriting system to point-free terms is not possible.

In spite of the negative results presented, in practice it is easy to implement a system that, given some hints from the programmer, automates many point-free calculations. In Chapter 9 we review Bird's functional calculator [Bir98], an example of one such system. We show how it can be reimplemented using a generic programming library, and how it can be extended in order to accommodate type-directed expansion rules. The chapter concludes with the presentation of a rewriting system, that can be used to simplify the point-free expressions that result from the translation defined in Chapter 7.

Chapter 10 begins with a review of the main contributions of the thesis. It also discusses the proposed reasoning methodology taking into account the initially stated objectives. The thesis ends with an enumeration of possible ideas for future work.

Chapter 2

Algebraic Programming

The origins of the algebraic programming paradigm can be traced back to 1977, when John Backus proposed, in his ACM Turing Award lecture, 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 [Bac78]. The main idea was already 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 combinators (presented later in this chapter) were already introduced in that seminal paper.

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 [Bir84, Bir87, Mee86]. 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 emphasis on the use of recursion operators, instead of the point-free combinators. As Backhouse pointed out [Bac89], 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 later showed that the concepts introduced by Bird and Meertens (notably the notion of fusion) arise naturally for any data type when viewed in a categorical setting [Mal90]. The categorical approach to data types, and functional programming in general, had been previously clarified by Hagino [Hag87]; 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 shown in this chapter, the definition of most of the combinators used in the point-free style of programming is 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 [MFP91] 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.

Our presentation of algebraic programming will be split into two chapters. This one presents the basic point-free combinators, the categorical approach to modeling recursive data-types, and the notion of hylomorphism. This fundamental recursion pattern will be used in the next chapter to implement typical recursion patterns, such as folds and unfolds.

2.1 Some Categorical Notions

The category theory concepts that will be used in this thesis are very simple, and are presented in any introductory book to the subject, such as [Pie91]. Essentially, a *category* is a collection of objects and a collection of arrows (or morphisms), such that

- an arrow f with domain A and codomain B is denoted by $f : A \to B$;
- there exists a composition operator $\cdot \circ \cdot$ that assigns to each pair of arrows $g : A \to B$ and $f : B \to C$ their composite arrow $f \circ g : A \to C$, and that obeys the following associative law
$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \textsf{Comp-Assoc}$$
- for each object A there exists an identity arrow $\mathsf{id}_A : A \to A$ satisfying the following condition
$$\mathsf{id} \circ f = f \circ \mathsf{id} = f \qquad \textsf{Id-Nat}$$

As seen in Id-Nat, we will often drop the typing information from most of the basic "polymorphic" morphisms, whenever this information can be easily derived from the context or is not relevant. In this case the complete formulation of the law would be: for any arrow $f : A \to B$, we have $\mathsf{id}_B \circ f = f \circ \mathsf{id}_A = f$.

The following is another basic fact about any category.

$$f \circ h = g \circ h \quad \Leftarrow \quad f = g \qquad \textsf{Leibniz}$$

The categorical approach to functional programming is based on a categorical account of the denotational semantics, with functions modeled by arrows in the underlying category, and types modeled by objects in that category. A lot of the research done in the area of algebraic programming is being carried out in the context of total functions and total elements. Technically, this means that the model of computation is **Set**, the category where objects are sets and morphisms are total functions. Unfortunately, this category is not a good semantic model for modern lazy functional languages, since it hardens the treatment of arbitrary recursive and partial definitions. Another problem is that, in **Set**, finite and infinite data types are different entities that cannot be combined, thus excluding, for example, functions defined by induction that work for both, and also the hylomorphism recursion pattern that will be defined in Section 2.4.

These problems can be overcome by moving to the category **CPO**, which imposes some additional structure: objects are pointed complete partial orders, and morphisms are continuous functions (more details later). The study of algebraic programming in this setting was pioneered by Meijer, Fokkinga and Paterson [MFP91], and the presentation of the material in this chapter is strongly influenced by their work.

A *partial order* (poset) is a set A that is equipped with a reflexive, transitive, and antisymmetric relation $\sqsubseteq_A$ (we will drop the subscript if it is clear from the context). The *least element* of a poset A is the $a \in A$ such that $a \sqsubseteq b$ for all $b \in A$ (notice that it may not exist, but if it exists, by antisymmetry it must be unique). A *chain* is a sequence $\{a_i \mid i \geq 0\} \subseteq A$ such that $\forall i \geq 0,\ a_i \sqsubseteq a_{i+1}$.

An *upper bound* of a chain $\{a_i \mid i \geq 0\}$ is an element a such that $\forall i \geq 0,\ a_i \sqsubseteq a$. The *least upper bound* (lub) is an upper bound that is less than or equal to every other upper bound. If a chain $\{a_i \mid i \geq 0\} \subseteq A$ has a lub it is denoted by $\bigsqcup_{i\geq 0}^{A} a_i$. A poset where every chain has a lub is a *complete partial order* (cpo). If it also has a least element (denoted by $\bot_A$) it is called a *pointed complete partial order* (pcpo).

A function $f : A \to B$ between pcpos A and B is *monotonic* if $\forall a_1, a_2 \in A$, if $a_1 \sqsubseteq_A a_2$ then $f\ a_1 \sqsubseteq_B f\ a_2$. A monotonic function $f : A \to B$ is *continuous* if for each chain $\{a_i \mid i \geq 0\} \subseteq A$ it is the case that $f\ (\bigsqcup_{i\geq 0}^{A} a_i) = \bigsqcup_{i\geq 0}^{B} (f\ a_i)$.

Since types will be modeled by pcpos, this means that their elements can be compared according to a partial order $\sqsubseteq$. The fact $x \sqsubseteq y$ will mean that x is an approximation of y, in the sense that x is less well defined than y. Completeness of a partial order implies that infinite data structures can be determined by the limit of their finite approximations. Continuity means that functions (modeled by arrows) respect these limits.

Terminal Object. The *terminal object* in **CPO** is $1 = \{\bot_1\}$. This means that for any other object A there exists a unique arrow from it to 1, namely the function that always returns $\bot_1$. We will denote that arrow by $!_A$. Since the least element of the functional type is precisely the function that always returns $\bot$, $!_A$ denotes the value

$\bot_{A\to 1}$. Given any $f : A \to B$, ! is characterized by the following properties.

$$!_1 = \mathsf{id}_1 \qquad \textsf{Bang-Reflex}$$

$$!_B \circ f = !_A \qquad \textsf{Bang-Fusion}$$

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

Strictness. From the point of view of program calculation, the major difference between using **Set** and **CPO** as the 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 **CPO**. A function $f : A \to B$ is *strict* if it preserves bottoms. In the categorical setting, a strict morphism is defined as follows.

$$f \text{ strict} \quad \Leftrightarrow \quad f \circ \underline{\bot} = \underline{\bot} \qquad \textsf{Strict-Def}$$

The composition of two strict functions is necessarily strict.

$$f \circ g \text{ strict} \quad \Leftarrow \quad f \text{ strict} \wedge g \text{ strict} \qquad \textsf{Strict-Comp}$$

The reverse implication is not true in general, but the following holds due to monotony.

$$f \text{ strict} \quad \Leftarrow \quad f \circ g \text{ strict} \qquad \textsf{Comp-Strict}$$

Sometimes we will also refer to the sub-category of **CPO** where all the functions are strict. We will denote this category by $\mathbf{CPO}_\bot$.

Functors. Another useful categorical concept is that of a functor, which will be used to model type constructors. A *functor* F is a mapping between categories (it maps objects to objects and arrows to arrows) such that

$$F\ f : F\ A \to F\ B \quad \Leftarrow \quad f : A \to B$$

$$F\ (f \circ g) = F\ f \circ F\ g \qquad \textsf{Functor-Comp}$$

$$F\ \mathsf{id}_A = \mathsf{id}_{FA} \qquad \textsf{Functor-Id}$$

For our purposes (and in general in the context of programming language semantics), *endofunctors* in **CPO** will be used, mapping types to types, and functions to functions.

The basic set of functors includes the identity functor Id, whose action on types is defined as $\mathsf{Id}\ A = A$, and on functions as $\mathsf{Id}\ f = f$. It also includes the constant functor: given a type A, the functor $\underline{A}$ is defined on types as $\underline{A}\ B = A$, and on functions as $\underline{A}\ f = \mathsf{id}_A$.

A functor is *locally continuous* if, for all objects A and B, its action on functions of type $A \to B$ is continuous. A functor F is *strictness-preserving* if, given a strict f, $F\ f$ is also strict (essentially, it should be a functor in $\mathbf{CPO}_\bot$).

A *bifunctor* $\star$ is a mapping from a pair of categories to a category; in the present context a bifunctor maps pairs of types to types, and pairs of functions to functions, verifying the conditions (infix notation is used):

$$f \star g : A \star B \to C \star D \quad \Leftarrow \quad f : A \to C \wedge g : B \to D$$

$$(f \circ g) \star (h \circ k) = (f \star h) \circ (g \star k) \qquad \textsf{Bifunctor-Comp}$$

$$\mathsf{id}_A \star \mathsf{id}_B = \mathsf{id}_{A \star B} \qquad \textsf{Bifunctor-Id}$$

Given two monofunctors F and G and a bifunctor $\star$, a new monofunctor $F \hat{\star} G$ can be defined by *lifting* $\star$ as follows:

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

Given a type A, a monofunctor $A\star$ can also be defined by *sectioning* $\star$:

$$(A\star) = \underline{A} \hat{\star} \mathsf{Id}$$

This left-sectioning corresponds to treating as a constant the first parameter of the functor. Analogously, the right-sectioning of a bifunctor can also be defined.

Natural Transformations. A *natural transformation* η between functors F and G, denoted by $\eta : F \dot{\to} G$, is a function that assigns to each type A an arrow $\eta_A : F\ A \to G\ A$ such that, for any function $f : A \to B$ the following naturality condition holds.

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

Alternatively, we could say that for each $f : A \to B$ the following diagram commutes.

$$\begin{tikzcd}
F\ A \arrow[r, "\eta_A"] \arrow[d, "F\ f"'] & G\ A \arrow[d, "G\ f"] \\
F\ B \arrow[r, "\eta_B"'] & G\ B
\end{tikzcd}$$

If each η_A is also an isomorphism then η is called a *natural isomorphism*. The concept of natural transformation can be generalized to bifunctors.

2.2 Basic Combinators

In this section we will introduce a number of type constructors (such as products and coproducts); each comes equipped with its own function combinators and laws for these combinators. In the categorical setting, type constructors are simply universal constructions (that can be generalized as functors), and the laws can all be derived from their universal properties. Most of the laws presented in this chapter have trivial proofs. Some are presented in order to exemplify the calculational style used throughout the thesis, but the majority of them are omitted.

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

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

The ordering on the elements is defined as

$$(a,b) \sqsubseteq (c,d) \quad \Leftrightarrow \quad a \sqsubseteq c \wedge b \sqsubseteq d$$

which means that the least element is $\perp_{A\times B} = (\perp_A, \perp_B)$.

On a product $A \times B$ the *projections* and the *split* function combinator (denoted by $\cdot \vartriangle \cdot$) can also be defined.

$$\begin{aligned} \mathsf{fst}_{A\times B}\ (x,y) &= x \\ \mathsf{snd}_{A\times B}\ (x,y) &= y \end{aligned} \qquad (g \vartriangle h)\ x = (g\ x, h\ x)$$

The fact that the cartesian product is a categorical product in **CPO** is justified by the following uniqueness law.

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

This law is also known as the universal law of products. Universal laws are used in category theory to implicitly characterize entities. Instead of giving the explicit pointwise definitions, we could state that the product of two objects A and B is an object $A \times B$, together with a pair of projections $\mathsf{fst} : A \times B \to A$ and $\mathsf{snd} : A \times B \to B$, such that for every object C, and arrows $g : C \to A$ and $h : C \to B$, there exists exactly one arrow from C to $A \times B$, denoted by $g \vartriangle h$, that makes the following diagram commute (uniqueness is signaled by a dashed line).

$$\begin{tikzcd} & C \arrow[dl, "g"'] \arrow[d, dashed, "g \triangle h"] \arrow[dr, "h"] & \\ A & A \times B \arrow[l, "\mathsf{fst}"] \arrow[r, "snd"'] & B \end{tikzcd}$$

The implicit definition of products via their universal law turns out to be much more effective for program calculation than the explicit pointwise definition given before. Suppose one wants to prove the following very simple property about split.

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

Given its pointwise definition, the proof goes as follows.

$$\begin{array}{ll} & \mathsf{fst} \triangle \mathsf{snd} = \mathsf{id} \\ \Leftrightarrow & \quad \{\, \eta\text{-expansion} \,\} \\ & (\mathsf{fst} \triangle \mathsf{snd})\ (x,y) = \mathsf{id}\ (x,y) \\ \Leftrightarrow & \quad \{\, \text{definition of } \cdot \triangle \cdot \text{ and } \mathsf{id} \,\} \\ & (\mathsf{fst}\ (x,y), \mathsf{snd}\ (x,y)) = (x,y) \\ \Leftrightarrow & \quad \{\, \text{definition of } \mathsf{fst} \text{ and } \mathsf{snd} \,\} \\ & (x,y) = (x,y) \end{array}$$

Using the universal law, we can avoid unfolding definitions, and the proof becomes even more straightforward.

$$\begin{array}{ll} & \mathsf{fst} \triangle \mathsf{snd} = \mathsf{id} \\ \Leftrightarrow & \quad \{\, \mathsf{Prod\text{-}Uniq}, \text{ with } g = \mathsf{fst},\ h = \mathsf{snd}, \text{ and } f = \mathsf{id} \,\} \\ & \mathsf{fst} \circ \mathsf{id} = \mathsf{fst} \wedge \mathsf{snd} \circ \mathsf{id} = \mathsf{snd} \\ \Leftrightarrow & \quad \{\, \mathsf{Id\text{-}Nat} \,\} \\ & \mathsf{fst} = \mathsf{fst} \wedge \mathsf{snd} = \mathsf{snd} \end{array}$$

Throughout the thesis, the following conventions will be adopted. Although naturality of identity is explicitly mentioned here, it will usually be omitted. We will implicitly use associativity laws, just by omitting parenthesis around operators that enjoy this property, namely composition. We will also assume that this operator binds stronger than any other combinator in order to avoid the proliferation of parentheses.

Using Prod-Uniq the following laws can easily be proved.

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

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

$$f \triangle g = h \triangle i \quad \Leftrightarrow \quad f = h \ \wedge\ g = i \qquad \mathsf{Prod\text{-}Equal}$$

It is also useful to define a product function combinator as follows.

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

Observe that this definition of product over functions allows to see product as a bifunctor. The corresponding laws can be checked as shown.

$$\mathsf{id} \times \mathsf{id} = \mathsf{id} \qquad \textsf{Prod-Functor-Id}$$

$$\begin{array}{ll} & \mathsf{id} \times \mathsf{id} \\ = & \quad \{\,\textsf{Prod-Def}\,\} \\ & \mathsf{id} \circ \mathsf{fst} \vartriangle \mathsf{id} \circ \mathsf{snd} \\ = & \quad \{\,\textsf{Prod-Reflex}\,\} \\ & \mathsf{id} \end{array}$$

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \textsf{Prod-Functor-Comp}$$

$$\begin{array}{ll} & (f \times g) \circ (h \times i) \\ = & \quad \{\,\textsf{Prod-Def}\,\} \\ & (f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd}) \circ (h \circ \mathsf{fst} \vartriangle i \circ \mathsf{snd}) \\ = & \quad \{\,\textsf{Prod-Fusion}\,\} \\ & f \circ \mathsf{fst} \circ (h \circ \mathsf{fst} \vartriangle i \circ \mathsf{snd}) \vartriangle g \circ \mathsf{snd} \circ (h \circ \mathsf{fst} \vartriangle i \circ \mathsf{snd}) \\ = & \quad \{\,\textsf{Prod-Cancel}\,\} \\ & f \circ h \circ \mathsf{fst} \vartriangle g \circ i \circ \mathsf{snd} \\ = & \quad \{\,\textsf{Prod-Def}\,\} \\ & f \circ h \times g \circ i \end{array}$$

The following absorption law fuses a product with a split.

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

Since we are working in **CPO** it is important to characterize the strictness of the split combinator and the projections.

$$\begin{array}{lcll} (f \vartriangle g) \text{ strict} & \Leftrightarrow & f \text{ strict} \wedge g \text{ strict} & \textsf{Prod-Strict} \\ \mathsf{fst} \text{ strict} & & & \textsf{Fst-Strict} \\ \mathsf{snd} \text{ strict} & & & \textsf{Snd-Strict} \end{array}$$

An example of a very useful function defined using split is the swap function.

$$\begin{array}{lcl} \mathsf{swap} & : & A \times B \rightarrow B \times A \\ \mathsf{swap} & = & \mathsf{snd} \vartriangle \mathsf{fst} \end{array} \qquad \textsf{Swap-Def}$$

This function is a natural isomorphism between the bifunctor $\cdot \times \cdot$ and $A \otimes B = B \times A$, as shown in the following calculations.

$$\begin{array}{ll} (f \times g) \circ \mathsf{swap} = \mathsf{swap} \circ (g \times f) & \textsf{Swap-Nat} \\ \mathsf{swap} \circ \mathsf{swap} = \mathsf{id} & \textsf{Swap-Iso} \end{array}$$

$$
\begin{array}{cl}
 & \mathsf{swap} \circ (g \times f) \\
= & \quad \{\, \mathsf{Swap\text{-}Def},\ \mathsf{Prod\text{-}Def} \,\} \\
 & (\mathsf{snd} \vartriangle \mathsf{fst}) \circ (g \circ \mathsf{fst} \vartriangle f \circ \mathsf{snd}) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Cancel} \,\} \\
 & f \circ \mathsf{snd} \vartriangle g \circ \mathsf{fst} \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
 & f \circ \mathsf{fst} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \vartriangle g \circ \mathsf{snd} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion} \,\} \\
 & (f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd}) \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\
= & \quad \{\, \mathsf{Swap\text{-}Def},\ \mathsf{Prod\text{-}Def} \,\} \\
 & (f \times g) \circ \mathsf{swap}
\end{array}
\qquad
\begin{array}{cl}
 & \mathsf{swap} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Swap\text{-}Def} \,\} \\
 & (\mathsf{snd} \vartriangle \mathsf{fst}) \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion} \,\} \\
 & \mathsf{snd} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \vartriangle \mathsf{fst} \circ (\mathsf{snd} \vartriangle \mathsf{fst}) \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
 & \mathsf{fst} \vartriangle \mathsf{snd} \\
= & \quad \{\, \mathsf{Prod\text{-}Reflex} \,\} \\
 & \mathsf{id}
\end{array}
$$

Another useful natural isomorphism is the one that gives evidence of the associativity of the cartesian product.

$$
\begin{array}{rcl}
\mathsf{assocr} & : & (A \times B) \times C \to A \times (B \times C) \\
\mathsf{assocr} & = & (\mathsf{fst} \circ \mathsf{fst}) \vartriangle (\mathsf{snd} \times \mathsf{id})
\end{array}
\qquad \mathsf{Assocr\text{-}Def}
$$

$$
(f \times (g \times h)) \circ \mathsf{assocr} = \mathsf{assocr} \circ ((f \times g) \times h) \qquad \mathsf{Assocr\text{-}Nat}
$$

Its inverse is defined as follows.

$$
\begin{array}{rcl}
\mathsf{assocl} & : & A \times (B \times C) \to (A \times B) \times C \\
\mathsf{assocl} & = & (\mathsf{id} \times \mathsf{fst}) \vartriangle (\mathsf{snd} \circ \mathsf{snd})
\end{array}
\qquad \mathsf{Assocl\text{-}Def}
$$

$$
\mathsf{assocr} \circ \mathsf{assocl} = \mathsf{assocl} \circ \mathsf{assocr} = \mathsf{id} \qquad \mathsf{Assocr\text{-}Iso}
$$

Sums. It is know that **CPO** does not have true coproducts. In most lazy functional languages this notion is implemented by the *separated sum*, where a new bottom element is added to the tagged reunion of the elements of both sets.

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

The ordering relation of this pcpo is defined as follows.

$$
x \sqsubseteq y \quad \Leftrightarrow \quad x = \bot \vee (\mathsf{fst}\ x = \mathsf{fst}\ y \wedge \mathsf{snd}\ x \sqsubseteq \mathsf{snd}\ y)
$$

Dually to products, *injections* and the *either* combinator (denoted by $\cdot \triangledown \cdot$) can be defined on a sum $A + B$.

$$
\begin{array}{l}
\mathsf{inl}_{A+B}\ x = (0, x) \\
\mathsf{inr}_{A+B}\ x = (1, x)
\end{array}
\qquad
\begin{array}{l}
(g \triangledown h)\ \bot = \bot \\
(g \triangledown h)\ (0, x) = g\ x \\
(g \triangledown h)\ (1, x) = h\ x
\end{array}
$$

The fact that the separated sum is not a categorical coproduct in **CPO** is reflected in the uniqueness law, that only holds for strict functions:

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

If the strictness side condition was omitted, the equation would not be valid because the right-hand side would not determine the outcome of f when applied to $\perp$. Resorting to diagrams, we could say that there exists exactly one strict arrow from $A + B$ to C that makes the following diagram commute.

$$\begin{tikzcd} A \arrow[r, "\mathsf{inl}"] \arrow[dr, "g"'] & A + B \arrow[d, dashed, "g \triangledown h"] & B \arrow[l, "\mathsf{inr}"'] \arrow[dl, "h"] \\ & C & \end{tikzcd}$$

Using uniqueness the following laws can be proved. Notice how fusion becomes "polluted" by strictness side conditions.

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

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

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

$$f \triangledown g = h \triangledown i \quad \Leftrightarrow \quad f = h \wedge g = i \qquad \textsf{Sum-Equal}$$

Likewise products, the separated sum can be turned into a bifunctor by defining its operation on arrows, which leads to the introduction of the sum function combinator:

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

Here are the corresponding functor and absorption laws.

$$\mathsf{id} + \mathsf{id} = \mathsf{id} \qquad \textsf{Sum-Functor-Id}$$

$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad \textsf{Sum-Functor-Comp}$$

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

By definition, any either is strict. However, unlike fst and snd, which are strict functions, inl and inr are not.

$$(f \triangledown g) \text{ strict} \qquad \textsf{Sum-Strict}$$

There is an interesting law that relates split with either, typically called the abides or exchange law.

$$(f \vartriangle g) \triangledown (h \vartriangle i) = (f \triangledown h) \vartriangle (g \triangledown i) \qquad \textsf{Abides}$$

$$
\begin{array}{ll}
 & (f \vartriangle g) \triangledown (h \vartriangle i) \\
= & \quad \{\, \text{Prod-Reflex} \,\} \\
 & (\mathsf{fst} \vartriangle \mathsf{snd}) \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \\
= & \quad \{\, \text{Prod-Fusion} \,\} \\
 & \mathsf{fst} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \vartriangle \mathsf{snd} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \\
= & \quad \{\, \text{Sum-Fusion, Fst-Strict, Snd-Strict} \,\} \\
 & (\mathsf{fst} \circ (f \vartriangle g) \triangledown \mathsf{fst} \circ (h \vartriangle i)) \vartriangle (\mathsf{snd} \circ (f \vartriangle g) \triangledown \mathsf{snd} \circ (h \vartriangle i)) \\
= & \quad \{\, \text{Prod-Cancel} \,\} \\
 & (f \triangledown h) \vartriangle (g \triangledown i)
\end{array}
$$

Analogously to the swap function for products, the coswap can be defined for sums, with similar laws (and proofs).

$$
\begin{array}{lcl}
\mathsf{coswap} & : & A + B \to B + A \\
\mathsf{coswap} & = & \mathsf{inr} \triangledown \mathsf{inl}
\end{array}
\qquad \text{Coswap-Def}
$$

$$(f + g) \circ \mathsf{coswap} = \mathsf{coswap} \circ (g + f) \qquad \text{Coswap-Nat}$$

$$\mathsf{coswap} \circ \mathsf{coswap} = \mathsf{id} \qquad \text{Coswap-Iso}$$

Sum is also associative, and the isomorphism is justified by the following definitions.

$$
\begin{array}{lcl}
\mathsf{coassocr} & : & (A + B) + C \to A + (B + C) \\
\mathsf{coassocr} & = & (\mathsf{id} + \mathsf{inl}) \triangledown (\mathsf{inr} \circ \mathsf{inr})
\end{array}
\qquad \text{Coassocr-Def}
$$

$$
\begin{array}{lcl}
\mathsf{coassocl} & : & A + (B + C) \to (A + B) + C \\
\mathsf{coassocl} & = & (\mathsf{inl} \circ \mathsf{inl}) \triangledown (\mathsf{inr} + \mathsf{id})
\end{array}
\qquad \text{Coassocl-Def}
$$

Unfortunately, **CPO** is not a *distributive category*, which means that $A \times (B + C)$ is not isomorphic to $(A \times B) + (A \times C)$. In fact, there are more elements in the first type (for example, given a value $(x, \bot)$ of the first type, it must be converted into $\bot$ of the second, but there is no way to invert this value in order to obtain the original x). Even so, it is possible to write a function that distributes the product over the sum and vice-versa. The point-free definition of the former needs exponentials, but the latter can be readily defined as follows.

$$
\begin{array}{lcl}
\mathsf{undistr} & : & (A \times B) + (A \times C) \to A \times (B + C) \\
\mathsf{undistr} & = & (\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr})
\end{array}
\qquad \text{Undistr-Def}
$$

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

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

Given pcpos A and B, the ordering of the continuous functions from A to B is determined by

$$f \sqsubseteq g \quad \Leftrightarrow \quad (f\ a \sqsubseteq g\ a, \forall a \in A)$$

As already mentioned in the previous section, this ordering implies that the least element of B^A is the function that always returns $\bot_B$.

Associated to the exponential B^A, are the *apply* function and the *curry* combinator (denoted by $\overline{\cdot}$).

$$\begin{aligned} \mathsf{ap}_{B^A}\ (f, x) &= f\ x \\ \overline{g}\ x\ y &= g\ (x, y) \end{aligned}$$

The following uniqueness law states that this notion of exponentiation is truly categorical in **CPO**.

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

Alternatively, we could say that for every object C, and $g : C \times A \to B$, $\overline{g}$ is the unique arrow from C to B^A that makes the following diagram commute.

$$\begin{tikzcd} B^A \times A \arrow[r, "\mathsf{ap}"] & B \\ C \times A \arrow[u, dashed, "\overline{g} \times \mathsf{id}"] \arrow[ur, "g"'] & \end{tikzcd}$$

The definition of the exponentiation combinator allows to turn this operation into a functor:

$$f^A = \overline{f \circ \mathsf{ap}} \qquad \textsf{Exp-Def}$$

Notice that when the type in superscript is not relevant the symbol $\bullet$ will be used in replacement. The following laws characterize exponentiation.

$$\overline{\mathsf{ap}} = \mathsf{id} \qquad \textsf{Exp-Reflex}$$

$$\mathsf{ap} \circ (\overline{f} \times \mathsf{id}) = f \qquad \textsf{Exp-Cancel}$$

$$\overline{f} \circ g = \overline{f \circ (g \times \mathsf{id})} \qquad \textsf{Exp-Fusion}$$

$$\overline{f} = \overline{g} \quad \Leftrightarrow \quad f = g \qquad \textsf{Exp-Equal}$$

$$\mathsf{id}^\bullet = \mathsf{id} \qquad \textsf{Exp-Functor-Id}$$

$$(f \circ g)^\bullet = f^\bullet \circ g^\bullet \qquad \textsf{Exp-Functor-Comp}$$

$$f^\bullet \circ \overline{g} = \overline{f \circ g} \qquad \textsf{Exp-Absor}$$

Left-strictness. To characterize the strictness of the curry operator, the stronger notion of left-strictness is needed. An arrow of type $A \times B \to C$ is left-strict if it returns

$\bot_C$ when the first element of the input pair is $\bot_A$. Categorically, this concept can be characterized as follows.

$$f \text{ left-strict} \quad \Leftrightarrow \quad f \circ (\bot \times \mathsf{id}) = \bot \circ \mathsf{fst} \qquad \textsf{Lstrict-Def}$$

$$f \text{ strict} \quad \Leftarrow \quad f \text{ left-strict} \qquad \textsf{Lstrict-Strict}$$

Alternatively, f is left-strict iff the following diagram commutes.

$$\begin{tikzcd} 1 \times B \arrow[r, "\mathsf{fst}"] \arrow[d, "\bot \times \mathsf{id}"'] & 1 \arrow[d, "\bot"] \\ A \times B \arrow[r, "f"'] & C \end{tikzcd}$$

An example of a strict function that is not left-strict is the projection snd.

Left-strictness and composition interact according to the following laws. Again, the last law can be derived from monotony.

$$f \circ (g \times \mathsf{id}) \text{ left-strict} \quad \Leftarrow \quad f \text{ left-strict} \wedge g \text{ strict} \qquad \textsf{Lstrict-Comp-Left}$$

$$f \circ g \text{ left-strict} \quad \Leftarrow \quad f \text{ strict} \wedge g \text{ left-strict} \qquad \textsf{Lstrict-Comp-Right}$$

$$f \text{ left-strict} \quad \Leftarrow \quad f \circ (g \times \mathsf{id}) \text{ left-strict} \qquad \textsf{Comp-Lstrict}$$

Remember that the least element of type A^B is a function that ignores its argument and always returns the least element of type A. As such, a curried function is strict if and only if its uncurried version is left-strict. The same reason justifies that ap is a left-strict function (and thus strict).

$$\overline{f} \text{ strict} \quad \Leftrightarrow \quad f \text{ left-strict} \qquad \textsf{Exp-Strict}$$

$$\mathsf{ap} \text{ left-strict} \qquad \textsf{Ap-Lstrict}$$

Distributivity. Using exponentials, functions to distribute products over sums can be defined in the point-free style as follows.

$$\begin{array}{lcl} \mathsf{distl} & : & (A + B) \times C \to (A \times C) + (B \times C) \\ \mathsf{distl} & = & \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \end{array} \qquad \textsf{Distl-Def}$$

$$\begin{array}{lcl} \mathsf{distr} & : & A \times (B + C) \to (A \times B) + (A \times C) \\ \mathsf{distr} & = & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{distl} \circ \mathsf{swap} \end{array} \qquad \textsf{Distr-Def}$$

Although **CPO** is not distributive, it is still true that

$$\mathsf{distr} \circ \mathsf{undistr} = \mathsf{id} \qquad \textsf{Distr-Iso-Left}$$

as the following calculation shows.

$$
\begin{array}{ll}
 & \mathsf{distr} \circ \mathsf{undistr} \\
= & \quad \{\, \mathsf{Distr\text{-}Def},\ \mathsf{Distl\text{-}Def},\ \mathsf{Undistr\text{-}Def} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ \mathsf{swap} \circ ((\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr})) \\
= & \quad \{\, \mathsf{Sum\text{-}Fusion},\ \mathsf{swap}\ \text{strict},\ \mathsf{Swap\text{-}Nat} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ ((\mathsf{inl} \times \mathsf{id}) \circ \mathsf{swap} \triangledown (\mathsf{inr} \times \mathsf{id}) \circ \mathsf{swap}) \\
= & \quad \{\, \mathsf{Sum\text{-}Absor},\ \mathsf{Prod\text{-}Def} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ ((\mathsf{inl} \circ \mathsf{fst} \vartriangle \mathsf{snd}) \triangledown (\mathsf{inr} \circ \mathsf{fst} \vartriangle \mathsf{snd})) \circ (\mathsf{swap} + \mathsf{swap}) \\
= & \quad \{\, \mathsf{Abides},\ \mathsf{Prod\text{-}Absor} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \circ (\mathsf{inl} \circ \mathsf{fst} \triangledown \mathsf{inr} \circ \mathsf{fst}) \vartriangle (\mathsf{snd} \triangledown \mathsf{snd})) \circ (\mathsf{swap} + \mathsf{swap}) \\
= & \quad \{\, \mathsf{Sum\text{-}Fusion},\ \mathsf{Sum\text{-}Strict},\ \mathsf{Sum\text{-}Cancel} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \circ \mathsf{fst} \triangledown \overline{\mathsf{inr}} \circ \mathsf{fst}) \vartriangle (\mathsf{snd} \triangledown \mathsf{snd})) \circ (\mathsf{swap} + \mathsf{swap}) \\
= & \quad \{\, \mathsf{Abides},\ \mathsf{Prod\text{-}Def} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \times \mathsf{id}) \triangledown (\overline{\mathsf{inr}} \times \mathsf{id})) \circ (\mathsf{swap} + \mathsf{swap}) \\
= & \quad \{\, \mathsf{Sum\text{-}Fusion},\ \mathsf{ap}\ \text{strict},\ \mathsf{Exp\text{-}Cancel} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ (\mathsf{inl} \triangledown \mathsf{inr}) \circ (\mathsf{swap} + \mathsf{swap}) \\
= & \quad \{\, \mathsf{Sum\text{-}Reflex},\ \mathsf{Sum\text{-}Functor\text{-}Comp} \,\} \\
 & \mathsf{swap} \circ \mathsf{swap} + \mathsf{swap} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Swap\text{-}Iso},\ \mathsf{Sum\text{-}Functor\text{-}Id} \,\} \\
 & \mathsf{id}
\end{array}
$$

Obviously, the inverse does not hold (although $\mathsf{undistr} \circ \mathsf{distr} \sqsubseteq \mathsf{id}$). Some additional properties about distributivity, namely naturality, are stated and proved in Appendix A.

Guards. Booleans can be defined by $\mathsf{Bool} = 1 + 1$, with

$$
\begin{array}{lcl}
\mathsf{true} & : & 1 \to \mathsf{Bool} \\
\mathsf{true} & = & \mathsf{inl}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{false} & : & 1 \to \mathsf{Bool} \\
\mathsf{false} & = & \mathsf{inr}
\end{array}
$$

Given this definition, negation is implemented by the coswap function.

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

$$
\begin{array}{lcl}
p? & : & A \to A + A \\
p? & = & (\mathsf{fst} + \mathsf{fst}) \circ \mathsf{distr} \circ (\mathsf{id} \vartriangle p)
\end{array}
\qquad \mathsf{Guard\text{-}Def}
$$

Notice that if p returns $\perp$ for some input, then $p?$ will also return $\perp$.

Cartesian Closed Categories. A category that has products, exponentials, and terminal object is called *cartesian closed*. In these categories, the set of arrows from A to B can be represented by the object B^A. Formally, this means that an $f : A \to B$ can be internalized as a point $\underline{f} : 1 \to B^A$, and vice versa. Moreover, it is possible to define the conversions between a function and its point using the basic combinators previously defined. Given any arrow $f : A \to B$ we have [McL95]

$$\underline{f} = \overline{f \circ \mathsf{snd}} \qquad \textsf{Pnt-Def}$$

$$f = \mathsf{ap} \circ (\underline{f} \circ ! \vartriangle \mathsf{id}) \qquad \textsf{Pnt-Cancel}$$

Notice that in the first law, the type of snd is $1 \times A \to A$. The second law can be proved as follows.

$$\begin{aligned}
& \mathsf{ap} \circ (\underline{f} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{ \textsf{Pnt-Def} \} \\
& \mathsf{ap} \circ (\overline{f \circ \mathsf{snd}} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{ \textsf{Prod-Absor} \} \\
& \mathsf{ap} \circ (\overline{f \circ \mathsf{snd}} \times \mathsf{id}) \circ (! \vartriangle \mathsf{id}) \\
= & \quad \{ \textsf{Exp-Cancel} \} \\
& f \circ \mathsf{snd} \circ (! \vartriangle \mathsf{id}) \\
= & \quad \{ \textsf{Prod-Cancel} \} \\
& f
\end{aligned}$$

By looking just at the last step of the proof, it may seem that ! could be replaced by any other function with similar results. However, due to the specific type of snd in this particular case, one must guarantee that the first element of the pair is of type 1; but there is only one function that given a A returns 1, namely $!_A$.

Surprisingly (or maybe not), points allow us to internalize in the point-free calculus some pointwise like definitions. The basic idea is that application can be modeled by composition as follows.

$$f \circ \underline{x} = \underline{f\ x} \qquad \textsf{Pnt-Comp}$$

Consider, for example, the pointwise definition of the exponentiation combinator $f^\bullet\ g = f \circ g$. Given Pnt-Comp, this fact can be expressed and proved in the point-free calculus as follows.

$$f^\bullet \circ \underline{g} = \underline{f \circ g} \qquad \textsf{Exp-Pnt}$$

$$\begin{aligned}
& f^\bullet \circ \underline{g} \\
= & \quad \{ \textsf{Pnt-Def} \} \\
& f^\bullet \circ \overline{g \circ \mathsf{snd}} \\
= & \quad \{ \textsf{Exp-Absor} \} \\
& \overline{f \circ g \circ \mathsf{snd}} \\
= & \quad \{ \textsf{Pnt-Def} \} \\
& \underline{f \circ g}
\end{aligned}$$

2.3 Recursive Data Types

In a typed functional programming language a new data type is defined by declaring its constructors and the respective types. For example, in Haskell the following types can be defined.

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

The first one is a monomorphic type that represents natural numbers, and the later are polymorphic ones representing lists and binary trees containing elements of an arbitrary type.

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

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

For binary trees we have to decide where to put parentheses.

$$\begin{array}{rcl} \mathsf{empty} & : & 1 \to \mathsf{Tree}\ A \\ \mathsf{node} & : & A \times (\mathsf{Tree}\ A \times \mathsf{Tree}\ A) \to \mathsf{Tree}\ A \end{array}$$

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

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

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

$$\begin{array}{rcl} \mathsf{F}_{\mathsf{Nat}} & = & \underline{1} \hat{+} \mathsf{Id} \\ \mathsf{in}_{\mathsf{Nat}} & = & \mathsf{zero} \triangledown \mathsf{succ} \end{array} \qquad \begin{array}{rcl} \mathsf{F}_{\mathsf{List}} & = & \underline{1} \hat{+} \underline{A} \hat{\times} \mathsf{Id} \\ \mathsf{in}_{\mathsf{List}} & = & \mathsf{nil} \triangledown \mathsf{cons} \end{array} \qquad \begin{array}{rcl} \mathsf{F}_{\mathsf{Tree}} & = & \underline{1} \hat{+} \underline{A} \hat{\times} (\mathsf{Id} \hat{\times} \mathsf{Id}) \\ \mathsf{in}_{\mathsf{Tree}} & = & \mathsf{empty} \triangledown \mathsf{node} \end{array}$$

A recursive data type T is then defined by taking the fixed point of its base functor F_T. Reynolds proved that in **CPO**, given a locally continuous and strictness-preserving base functor F, there exists a unique data type $T = \mu F$ and two unique strict functions

$\mathsf{in}_T : F\ T \to T$ and $\mathsf{out}_T : T\ \to F\ T$ that are each other's inverse [Rey77].

$$\begin{aligned}
&\mathsf{in}_T \circ \mathsf{out}_T = \mathsf{id}_T \ \wedge\ \mathsf{out}_T \circ \mathsf{in}_T = \mathsf{id}_{FT} && \textsf{In-Out-Iso}\\
&\mathsf{in}_T \text{ strict} && \textsf{In-Strict}\\
&\mathsf{out}_T \text{ strict} && \textsf{Out-Strict}
\end{aligned}$$

Fokkinga and Meijer [FM91] showed that all polynomial, and even all regular functors, are locally continuous and strictness-preserving. A *polynomial functor* is either the identity functor, a constant functor, a lifting of the sum and product bifunctors, or the composition of polynomial functors. This guarantees that, for example, all the following data types are well defined.

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

A *regular functor* can also be built from type functors, a concept that will be presented later in Section 3.1.1. An example of a data type built from a regular functor is that of *rose trees*, where each node may have an arbitrary number of children. Since all functors used in this thesis are locally continuous and strictness-preserving, this fact will usually be omitted from proofs.

To see what elements belong to a data type defined by fixed point, let us take the natural numbers as an example. A pcpo that satisfies the equation $\mathsf{Nat} \cong 1 + \mathsf{Nat}$ can informally be defined as $\mathsf{Nat} = 1 + (1 + (1 + \ldots))$. We can picture the ordering relation of this pcpo as follows.

$$\begin{tikzcd}
\mathsf{zero}\ \bot_1 & (\mathsf{succ} \circ \mathsf{zero})\ \bot_1 & (\mathsf{succ}^n \circ \mathsf{zero})\ \bot_1 & \\
\bot_{\mathsf{Nat}} \arrow[u] \arrow[r] & \mathsf{succ}\ \bot_{\mathsf{Nat}} \arrow[u] \arrow[r, dotted] & (\mathsf{succ}^n)\ \bot_{\mathsf{Nat}} \arrow[u] \arrow[r, dotted] & \infty
\end{tikzcd}$$

The elements in the upper row can be interpreted as the natural numbers, that is, $0 = \mathsf{zero}\ \bot_1$, $1 = (\mathsf{succ} \circ \mathsf{zero})\ \bot_1$, and in general $n = (\mathsf{succ}^n \circ \mathsf{zero})\ \bot_1$. Notice that the order in the poset is a definedness order, and so, these elements are unrelated. Besides the natural numbers, Nat also contains a chain of "partial numbers". An upper bound of this chain $\infty = \bigsqcup_n ((\mathsf{succ}^n)\ \bot_{\mathsf{Nat}})$ must be added in order to make the poset complete. This element satisfies $\infty = \mathsf{succ}\ \infty$, and denotes the infinite number. Similarly, the data type $\mathsf{List}\ A$ contains, not only all totally defined finite lists, but also chains of partial lists with shape $\mathsf{cons}\ (a_0, \mathsf{cons}\ (a_1, \ldots \bot))$, whose upper bounds are totally defined infinite lists.

We have already seen what the in functions for this data types are. But what about their destructors, the out functions? Given a predicate $\mathsf{iszero} : \mathsf{Nat} \to \mathsf{Bool}$ that tests if a natural is equal to zero, and a function $\mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat}$ that determines the predecessor, we have for naturals

$$\begin{array}{rcl} \mathsf{out}_{\mathsf{Nat}} & : & \mathsf{Nat} \to 1 + \mathsf{Nat} \\ \mathsf{out}_{\mathsf{Nat}} & = & (! + \mathsf{pred}) \circ \mathsf{iszero?} \end{array}$$

For lists, given a similar predicate $\mathsf{isnil} : \mathsf{List}\ A \to \mathsf{Bool}$, and the typical destructors $\mathsf{head} : \mathsf{List}\ A \to A$ and $\mathsf{tail} : \mathsf{List}\ A \to \mathsf{List}\ A$ we have

$$\begin{array}{rcl} \mathsf{out}_{\mathsf{List}} & : & \mathsf{List}\ A \to 1 + A \times \mathsf{List}\ A \\ \mathsf{out}_{\mathsf{List}} & = & (! + (\mathsf{head} \vartriangle \mathsf{tail})) \circ \mathsf{isnil?} \end{array}$$

Remark. This presentation excludes non-regular data types and mutually recursive ones. Several authors have shown how to extend this theory of algebraic programming to mutually recursive data types [Mei92, SF93, IHT98]. There is also some work on extending it to nested data types [BP99, MG01]. These are parameterized data types in which the parameter changes in the recursive call, and are interesting because they can capture some structural invariants. For example, the following Haskell data type can be used to store perfectly balanced binary trees with labels in the leafs.

```
data Balanced a = Leaf a | Fork (Balanced (a,a))
```

2.4 Hylomorphisms

The *hylomorphism* recursion pattern was first defined in [FM91]. Given a functor F, a function $g : F\ B \to B$, and a function $h : A \to F\ A$, a hylomorphism is defined as the following recursive function, using the fixpoint operator μ.

$$\begin{array}{rcl} [\![g, h]\!]_{\mu F} & : & A \to B \\ [\![g, h]\!]_{\mu F} & = & \mu(\lambda f. g \circ F f \circ h) \end{array} \qquad \mathsf{Hylo\text{-}Def}$$

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

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

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

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

Example 2.1 (Factorial). Consider the following typical definition of the factorial function in Haskell, where `Nat` is the data type defined in the previous section, and `mult :: (Nat,Nat) -> Nat` implements multiplication.

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

Assuming that constants are functions from $\mathsf{1}$, this recursive definition verifies the following equations.

$$\mathsf{fact}\ (\mathsf{zero}\ \bot) = \mathsf{succ}\ (\mathsf{zero}\ \bot) \ \wedge\ \mathsf{fact}\ (\mathsf{succ}\ n) = \mathsf{mult}\ (\mathsf{succ}\ n, \mathsf{fact}\ n)$$

By applying the definitions of composition and the split combinator we can push the variables out of the expressions. one will be used as a shortcut to $\mathsf{succ} \circ \mathsf{zero}$.

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

η-reduction then enables us to transform these equations into the point-free style.

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

With some simple calculations we can rearrange this expression as follows.

$$\begin{array}{ll}
 & \mathsf{fact} \circ \mathsf{zero} = \mathsf{one}\ \wedge\ \mathsf{fact} \circ \mathsf{succ} = \mathsf{mult} \circ (\mathsf{succ} \vartriangle \mathsf{fact}) \\
\Leftrightarrow & \quad \{\,\mathsf{Sum\text{-}Equal}\,\} \\
 & \mathsf{fact} \circ \mathsf{zero} \triangledown \mathsf{fact} \circ \mathsf{succ} = \mathsf{one} \triangledown \mathsf{mult} \circ (\mathsf{succ} \vartriangle \mathsf{fact}) \\
\Leftrightarrow & \quad \{\,\mathsf{Sum\text{-}Fusion},\ \mathsf{fact}\ \text{strict},\ \mathsf{Prod\text{-}Absor}\,\} \\
 & \mathsf{fact} \circ (\mathsf{zero} \triangledown \mathsf{succ}) = \mathsf{one} \triangledown \mathsf{mult} \circ (\mathsf{id} \times \mathsf{fact}) \circ (\mathsf{succ} \vartriangle \mathsf{id}) \\
\Leftrightarrow & \quad \{\,\mathsf{in}_{\mathsf{Nat}} = \mathsf{zero} \triangledown \mathsf{succ},\ \mathsf{Sum\text{-}Absor}\,\} \\
 & \mathsf{fact} \circ \mathsf{in}_{\mathsf{Nat}} = (\mathsf{one} \triangledown \mathsf{mult}) \circ (\mathsf{id} + \mathsf{id} \times \mathsf{fact}) \circ (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \\
\Leftrightarrow & \quad \{\,\mathsf{In\text{-}Out\text{-}Iso}\,\} \\
 & \mathsf{fact} = (\mathsf{one} \triangledown \mathsf{mult}) \circ (\mathsf{id} + \mathsf{id} \times \mathsf{fact}) \circ (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}
\end{array}$$

This means that factorial can be determined by the fixpoint

$$\mathsf{fact} = \mu(\lambda f.(\mathsf{one} \triangledown \mathsf{mult}) \circ (\mathsf{F}_{\mathsf{List}}\ f) \circ (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}})$$

and hence defined as the hylomorphism

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

Example 2.2 (Length). To give another example, this informal method to derive a hylomorphism will be applied to the following function that determines the length of a list.

```
length :: List a -> Nat
length Nil        = Zero
length (Cons h t) = Succ (length t)
```

This definition satisfies the following pointwise equations.

$$\mathsf{length}\ (\mathsf{nil}\ \bot) = \mathsf{zero}\ \bot\ \wedge\ \mathsf{length}\ (\mathsf{cons}\ (h,t)) = \mathsf{succ}\ (\mathsf{length}\ t)$$

These can be converted into point-free style by using snd to "forget" the head of the list.

$$\mathsf{length} \circ \mathsf{nil} = \mathsf{zero}\ \wedge\ \mathsf{length} \circ \mathsf{cons} = \mathsf{succ} \circ \mathsf{length} \circ \mathsf{snd}$$

By similar calculations to the factorial example, these equations can be converted into

$$\mathsf{length} = \mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{length}) \circ (\mathsf{id} + \mathsf{snd}) \circ \mathsf{out}_{\mathsf{List}}$$

which means that length can be defined as the following hylomorphism.

$$\begin{array}{lcl} \mathsf{length} & : & \mathsf{List}\ A \to \mathsf{Nat} \\ \mathsf{length} & = & [\![\mathsf{in}_{\mathsf{Nat}}, (\mathsf{id} + \mathsf{snd}) \circ \mathsf{out}_{\mathsf{List}}]\!]_{\mathsf{Nat}} \end{array}$$

Laws. Most of the fundamental laws about hylomorphisms follow directly from similar laws about fixpoints, or can be proved by fixpoint induction. That is the case of the following, first presented in [FM91].

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

$$[\![g, h]\!]_{\mu F} \circ [\![i, j]\!]_{\mu F} = [\![g, j]\!]_{\mu F} \quad \Leftarrow \quad h \circ i = \mathsf{id} \qquad \mathsf{Hylo\text{-}Compose}$$

$$[\![g, h]\!]_{\mu F} \text{ strict} \quad \Leftarrow \quad g \text{ strict} \wedge h \text{ strict} \qquad \mathsf{Hylo\text{-}Strict}$$

The following fact is related to the existence of recursive data types, and its original proof using the fixpoint operator instead of hylomorphisms is also due to Reynolds [Rey77].

$$[\![\mathsf{in}_{\mu F}, \mathsf{out}_{\mu F}]\!]_{\mu F} = \mathsf{id}_{\mu F} \qquad \mathsf{Hylo\text{-}Reflex}$$

Finally, we also have the interesting shifting law [MFP91], that can be used to change the shape of recursion.

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

These laws allow us to reason about recursive definitions using the same calculational style that the laws about the basic combinators enabled for the non-recursive case.

Example 2.3 (From). In order to exemplify their usage let us prove that

$$\mathsf{length} \circ \mathsf{from} = \mathsf{id}$$

where from is a function that given a natural n generates a list with all numbers from $n-1$ down to 0, and that can be defined as

$$\begin{array}{lcl} \mathsf{from} & : & \mathsf{Nat} \to \mathsf{List\ Nat} \\ \mathsf{from} & = & [\![\mathsf{in}_{\mathsf{List}}, (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{List\ Nat}} \end{array}$$

The calculation goes as follows. Notice that we have specialized length to lists of naturals.

$$\begin{array}{ll}
 & [\![\mathsf{in}_{\mathsf{Nat}}, (\mathsf{id} + \mathsf{snd}) \circ \mathsf{out}_{\mathsf{List}}]\!]_{\mathsf{Nat}} \circ [\![\mathsf{in}_{\mathsf{List}}, (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{List\ Nat}} \\
= & \quad \{ \mathsf{Hylo\text{-}Shift},\ \mathsf{id} + \mathsf{snd} : \underline{1} \hat{+} \underline{\mathsf{Nat}} \hat{\times} \mathsf{Id} \dot{\to} \underline{1} \hat{+} \mathsf{Id} \} \\
 & [\![\mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{snd}), \mathsf{out}_{\mathsf{List}}]\!]_{\mathsf{List\ Nat}} \circ [\![\mathsf{in}_{\mathsf{List}}, (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{List\ Nat}} \\
= & \quad \{ \mathsf{Hylo\text{-}Compose},\ \mathsf{In\text{-}Out\text{-}Iso} \} \\
 & [\![\mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{snd}), (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{List\ Nat}} \\
= & \quad \{ \mathsf{Hylo\text{-}Shift},\ \mathsf{id} + \mathsf{snd} : \underline{1} \hat{+} \underline{\mathsf{Nat}} \hat{\times} \mathsf{Id} \dot{\to} \underline{1} \hat{+} \mathsf{Id} \} \\
 & [\![\mathsf{in}_{\mathsf{Nat}}, (\mathsf{id} + \mathsf{snd}) \circ (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{Nat}} \\
= & \quad \{ \mathsf{Sum\text{-}Functor\text{-}Comp},\ \mathsf{Prod\text{-}Cancel} \} \\
 & [\![\mathsf{in}_{\mathsf{Nat}}, (\mathsf{id} + \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{Nat}} \\
= & \quad \{ \mathsf{Sum\text{-}Functor\text{-}Id},\ \mathsf{Hylo\text{-}Reflex} \} \\
 & \mathsf{id}_{\mathsf{Nat}}
\end{array}$$

As seen in this example, natural transformations are sometimes the main ingredient of a hylomorphism, and its identification is fundamental during calculation. Due to this fact, hylomorphisms are sometimes presented in the so-called *triplet form*, where the natural transformation is explicitly factored out [TM95].

Acid Rain. Takano and Meijer [TM95] defined a different kind of fusion rule, more suitable to be applied in contexts of program deforestation (i.e., optimization through the elimination of intermediate data structures). This rule generalizes the foldr/build rule, first defined in [GLJ93], to work on any regular data type, and is usually known as the acid rain theorem.

$$
\begin{gathered}
[\![g, \mathsf{out}_{\mu F}]\!]_{\mu F} \circ [\![\tau(\mathsf{in}_{\mu F}), h]\!]_{\mu G} = [\![\tau(g), h]\!]_{\mu G} \\
\Leftarrow \\
\tau : \forall A.(F\ A \to A) \to G\ A \to A
\end{gathered}
$$

The first difference to Hylo-Fusion is that the function to be fused is not arbitrary, and should itself be expressed as a hylomorphism. The other difference is that the hylomorphism parameters are also not arbitrary. In particular, one of them must result from the application of a polymorphic function transformer to the constructors of the intermediate data structure.

The dual of this rule, also presented in [TM95], was (much) later rebaptized as the destroy/unfoldr rule [Sve02].

$$
\begin{gathered}
[\![g, \sigma(\mathsf{out}_{\mu F})]\!]_{\mu G} \circ [\![\mathsf{in}_{\mu F}, h]\!]_{\mu F} = [\![g, \sigma(h)]\!]_{\mu G} \\
\Leftarrow \\
\sigma : \forall A.(A \to F\ A) \to A \to G\ A
\end{gathered}
$$

Expressiveness. An interesting result by Meijer and Hutton [MH95] shows how hylomorphisms can be used to compute arbitrary fixpoints, and thus provide the full power of recursion. In fact, the fixpoint operator can itself be implemented as a hylomorphism. The insight to this result is to notice that $\mu\ f$ is determined by the infinite application $f\ (f\ (f\ \ldots))$, whose recursion tree is an infinite list of functions f, subsequently consumed by application.

Infinite lists, or streams, can be defined as

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

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

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

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

$$
\begin{array}{lcl}
\mathsf{fix} & : & A^A \to A \\
\mathsf{fix} & = & [\![\mathsf{ap}, \mathsf{id} \vartriangle \mathsf{id}]\!]_{\mathsf{Stream}\ A^A}
\end{array}
\qquad \textsf{Fix-Def}
$$

According to [AL91], a morphism $\mu_A : A^A \to A$ is a fixpoint operator for A in a cartesian closed category if it verifies $\mu = \mathsf{ap} \circ (\mathsf{id} \vartriangle \mu)$. After expanding the definitions of composition, split, and ap we see that this corresponds to the expected pointwise equation $\mu\ f = f\ (\mu\ f)$. By applying Hylo-Cancel and Prod-Absor to fix it can be proved that it is indeed a fixpoint operator.

$$\mathsf{fix} = \mathsf{ap} \circ (\mathsf{id} \vartriangle \mathsf{fix}) \qquad \textsf{Fix-Cancel}$$

2.5 Summary

This chapter presented the fundamentals of algebraic programming, using the **CPO** category as a model for computation. Most of the material is well known, and is presented in many introductory texts to the subject, like [MFP91, BdM97, Gib02]. The notion of left strictness does not appear in literature, but will be very useful to reason about higher-order functions. A slightly different approach was followed in presenting recursion operators, by restricting the basic set to a single primitive pattern: the hylomorphism. In the next chapter it will be used to define folds, unfolds, and all the remaining typical recursion operators. Some very simple calculations and examples were presented, to smooth the transition to the more elaborate ones to appear in the remaining of the thesis.

Chapter 3

Recursion Patterns as Hylomorphisms

The advantages of using generic operators to capture typical patterns of recursion are widely recognized. Some of these are [SF93]:

- *Abstraction.* They allow the specification of algorithms to be independent from the types of the values they operate on. They make possible the statement, proof and use of generic theorems.

- *Structure.* By using a structured programming paradigm, tasks like program understanding or program transformation can be made easier.

Although many recursion patterns are described in the literature, only the hylomorphism was presented in the previous chapter. Since the fixpoint operator can itself be defined as a hylomorphism, this recursion pattern provides the full power of recursion. This means that, in principle, there is no need to define other recursion operators. However, having a collection of more structured recursion patterns can help in program calculation and transformation, because they can be characterized by more specific laws than the hylomorphism.

In particular, due to their generality, hylomorphisms lack one of the fundamental laws to reason about programs - uniqueness. As seen for the basic combinators in Section 2.2, this law gives a precise algebraic characterization of when a function can be expressed by a given combinator, and can be used as the "swiss army knife" for proving properties about it.

In this chapter we show how to define most of the typical recursion patterns using hylomorphisms. Usually, when working in **CPO**, recursion patterns are defined directly by fixpoint [MFP91]. By using hylomorphisms, the laws that characterize them can be derived from the basic set of laws presented in the previous chapter, thus avoiding the use of fixpoint induction. Since this also applies to uniqueness, this chapter can also be seen as a quest to find "well behaved" hylomorphisms.

Likewise to the basic combinators, namely products and sums, the categorical notion of dual also applies to recursion patterns. For example, folds and paramorphisms (corresponding to the operator that captures primitive recursion) will be presented, together with their duals, unfolds and the less known apomorphisms. By skipping the proofs, which are essentially the same, the presentation of duals is more concise.

3.1 Catamorphisms

One of the fundamental patterns of recursion is iteration, where recursive data types are "consumed" by replacing their constructors by arbitrary functions. This recursion pattern is usually called fold or *catamorphism*. In Haskell it is predefined for lists as the function `foldr`. Adapting that definition to the lists defined in Section 2.3 we get

```
fold_List :: (a -> b -> b) -> b -> List a -> b
fold_List f z Nil        = z
fold_List f z (Cons h t) = f h (fold_List f z t)
```

From this definition, it is clear that folding over a list

$$\texttt{Cons}\ x_1\ \texttt{(Cons}\ x_2\ \texttt{(Cons}\ x_3\ \texttt{(... Nil)))}$$

yields

$$\texttt{f}\ x_1\ \texttt{(f}\ x_2\ \texttt{(f}\ x_3\ \texttt{(... z)))}$$

The informal approach presented in Section 2.4 can be used to convert this definition into a hylomorphism. By uncurrying `f` and treating `z` as a point, both parameters can be combined into a single function $g : 1 + A \times B \to B$. Given this change, the fold for lists can alternatively be defined by the following equations.

$$\mathsf{fold}_{\mathsf{List}}\ g \circ \mathsf{nil} = g \circ \mathsf{inl}$$
$$\mathsf{fold}_{\mathsf{List}}\ g \circ \mathsf{cons} = g \circ \mathsf{inr} \circ (\mathsf{id} \times \mathsf{fold}_{\mathsf{List}}\ g)$$

By using $\mathsf{in}_{\mathsf{List}}$ to pack the constructors, these definitions can be simplified as follows. Notice that both g and $\mathsf{fold}_{\mathsf{List}}\ g$ are strict functions, because g is defined using the either combinator and the fold diverges when presented with a totally undefined list.

$$\begin{array}{ll}
 & \mathsf{fold}_{\mathsf{List}}\ g \circ \mathsf{nil} = g \circ \mathsf{inl} \wedge \mathsf{fold}_{\mathsf{List}}\ g \circ \mathsf{cons} = g \circ \mathsf{inr} \circ (\mathsf{id} \times \mathsf{fold}_{\mathsf{List}}\ g) \\
\Leftrightarrow & \quad \{\,\mathsf{Sum\text{-}Equal}\,\} \\
 & \mathsf{fold}_{\mathsf{List}}\ g \circ \mathsf{nil} \triangledown \mathsf{fold}_{\mathsf{List}}\ g \circ \mathsf{cons} = g \circ \mathsf{inl} \triangledown g \circ \mathsf{inr} \circ (\mathsf{id} \times \mathsf{fold}_{\mathsf{List}}\ g) \\
\Leftrightarrow & \quad \{\,\mathsf{Sum\text{-}Fusion},\ \mathsf{fold}_{\mathsf{List}}\ g \text{ strict},\ g \text{ strict}\,\} \\
 & \mathsf{fold}_{\mathsf{List}}\ g \circ (\mathsf{nil} \triangledown \mathsf{cons}) = g \circ (\mathsf{inl} \triangledown \mathsf{inr} \circ (\mathsf{id} \times \mathsf{fold}_{\mathsf{List}}\ g)) \\
\Leftrightarrow & \quad \{\,\mathsf{in}_{\mathsf{List}} = \mathsf{nil} \triangledown \mathsf{cons},\ \mathsf{Sum\text{-}Def}\,\} \\
 & \mathsf{fold}_{\mathsf{List}}\ g \circ \mathsf{in}_{\mathsf{List}} = g \circ (\mathsf{id} + \mathsf{id} \times \mathsf{fold}_{\mathsf{List}}\ g) \\
\Leftrightarrow & \quad \{\,\mathsf{Leibniz},\ \mathsf{In\text{-}Out\text{-}Iso},\ \mathsf{F}_{\mathsf{List}} = \underline{1} \hat{+} \underline{A} \hat{\times} \mathsf{Id}\,\} \\
 & \mathsf{fold}_{\mathsf{List}}\ g = g \circ (\mathsf{F}_{\mathsf{List}}\ (\mathsf{fold}_{\mathsf{List}}\ g)) \circ \mathsf{out}_{\mathsf{List}}
\end{array}$$

This calculation shows that

$$\mathsf{fold}_{\mathsf{List}}\ g = [\![g, \mathsf{out}_{\mathsf{List}}]\!]_{\mathsf{List}\ A}$$

The same reasoning can be applied to folding over naturals.

```
fold_Nat :: (b -> b) -> b -> Nat -> b
fold_Nat f z Zero     = z
fold_Nat f z (Succ n) = f (fold_Nat f z n)
```

By combining both parameters in a single function $g : 1 + \mathsf{Nat} \to \mathsf{Nat}$, this function can also be implemented using a hylomorphism.

$$\mathsf{fold}_{\mathsf{Nat}}\ g = [\![g, \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{Nat}}$$

Definition. Given a function of type $g : F\ A \to A$, the catamorphism operator which implements iteration over the data type μF can be generically defined as follows (following the Dutch tradition, it is denoted using the banana-brackets notation).

$$\begin{array}{lcl}
(\!| g |\!)_{\mu F} & : & \mu F \to A \\
(\!| g |\!)_{\mu F} & = & [\![g, \mathsf{out}_F]\!]_{\mu F}
\end{array} \qquad \mathsf{Cata\text{-}Def}$$

Example 3.1 (Sum). One of the simplest examples of a catamorphism over lists is the sum function. To sum all the elements of a list nil can be replaced by zero, and cons by plus : Nat × Nat → Nat.

$$\begin{array}{lcl}
\mathsf{sum} & : & \mathsf{List\ Nat} \to \mathsf{Nat} \\
\mathsf{sum} & = & (\!| \mathsf{zero} \triangledown \mathsf{plus} |\!)_{\mathsf{List\ Nat}}
\end{array}$$

Example 3.2 (Length). As seen in Example 2.3, the length function can also be defined as the hylomorphism $[\![\mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{snd}), \mathsf{out}_{\mathsf{List}}]\!]$. By definition this means that

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

Example 3.3 (Insertion Sort). Given a function $\mathsf{insert} : A \times \mathsf{List}\ A \rightarrow \mathsf{List}\ A$ that inserts an element in an ordered list, the insertion sort algorithm can be easily implemented by a catamorphism.

$$\begin{array}{lcl} \mathsf{isort} & : & \mathsf{List}\ A \rightarrow \mathsf{List}\ A \\ \mathsf{isort} & = & (\!|\mathsf{nil} \triangledown \mathsf{insert}|\!)_{\mathsf{List}\ A} \end{array}$$

Example 3.4 (Flatten). Another example of a catamorphism over lists is the flatten function, that converts a list of lists into a single list. $\mathsf{cat} : \mathsf{List}\ A \times \mathsf{List}\ A \rightarrow \mathsf{List}\ A$ is the function that concatenates two lists.

$$\begin{array}{lcl} \mathsf{flatten} & : & \mathsf{List}\ (\mathsf{List}\ A) \rightarrow \mathsf{List}\ A \\ \mathsf{flatten} & = & (\!|\mathsf{nil} \triangledown \mathsf{cat}|\!)_{\mathsf{List}\ (\mathsf{List}\ A)} \end{array}$$

Example 3.5 (Inorder). An example of a catamorphism over binary trees is the inorder traversal.

$$\begin{array}{lcl} \mathsf{inorder} & : & \mathsf{Tree}\ A \rightarrow \mathsf{List}\ A \\ \mathsf{inorder} & = & (\!|\mathsf{nil} \triangledown \mathsf{cat} \circ (\mathsf{id} \times \mathsf{cons}) \circ \mathsf{assocr} \circ (\mathsf{swap} \times \mathsf{id}) \circ \mathsf{assocl}|\!)_{\mathsf{Tree}\ A} \end{array}$$

Laws. From Cata-Def it is trivial to derive the following laws about catamorphisms from the corresponding laws about hylomorphisms.

$$(\!|\mathsf{in}_{\mu F}|\!)_{\mu F} = \mathsf{id}_{\mu F} \qquad \textsf{Cata-Reflex}$$

$$(\!|g|\!)_{\mu F} \circ \mathsf{in}_{\mu F} = g \circ F\ (\!|g|\!)_{\mu F} \qquad \textsf{Cata-Cancel}$$

$$f \circ (\!|g|\!)_{\mu F} = (\!|h|\!)_{\mu F} \quad \Leftarrow \quad f \circ g = h \circ F\ f \wedge f \text{ strict} \qquad \textsf{Cata-Fusion}$$

Assuming a strictness-preserving functor we have

$$(\!|g|\!)_{\mu F} \text{ strict} \quad \Leftrightarrow \quad g \text{ strict} \qquad \textsf{Cata-Strict}$$

The $\Leftarrow$ implication follows directly from Hylo-Strict and Out-Strict. For the other implication we could argue as follows.

$$\begin{array}{ll} & (\!|g|\!) \text{ strict} \\ \Rightarrow & \quad \{\ \textsf{In-Strict, Strict-Comp}\ \} \\ & (\!|g|\!) \circ \mathsf{in} \text{ strict} \\ \Rightarrow & \quad \{\ \textsf{Cata-Cancel}\ \} \\ & g \circ F\ (\!|g|\!) \text{ strict} \\ \Rightarrow & \quad \{\ \textsf{Comp-Strict}\ \} \\ & g \text{ strict} \end{array}$$

Similarly to sums, catamorphisms also obey a uniqueness law constrained by some strictness side conditions.

$$f = \llparenthesis g \rrparenthesis_{\mu F} \;\wedge\; g \text{ strict} \quad\Leftrightarrow\quad f \circ \mathsf{in}_{\mu F} = g \circ F f \;\wedge\; f \text{ strict} \qquad \textsf{Cata-Uniq}$$

The $\Rightarrow$ implication follows trivially from Cata-Cancel and Cata-Strict. The other implication can also be easily proved as follows.

$$\begin{array}{ll}
 & f \circ \mathsf{in}_{\mu F} = g \circ F f \;\wedge\; f \text{ strict} \\
\Rightarrow & \quad \{\, \textsf{Cata-Fusion} \,\} \\
 & f \circ \llparenthesis \mathsf{in}_{\mu F} \rrparenthesis_{\mu F} = \llparenthesis g \rrparenthesis_{\mu F} \\
\Leftrightarrow & \quad \{\, \textsf{Cata-Reflex} \,\} \\
 & f = \llparenthesis g \rrparenthesis_{\mu F}
\end{array}$$

Strictness of g then follows from Cata-Strict.

Categorically, this uniqueness law means that $\mathsf{in}_{\mu F}$ is an initial F-*algebra* in the category $\mathbf{CPO}_\perp$. An F-algebra is a function of type $F\ A \to A$, where A is called the carrier of the algebra. Given a strict $g : FA \to A$, $\llparenthesis g \rrparenthesis_{\mu F}$ is the unique strict function that makes the following diagram commute.

$$\begin{tikzcd}
\mu F \arrow[d, dotted, "\llparenthesis g \rrparenthesis_{\mu F}"'] & F\ (\mu F) \arrow[l, "\mathsf{in}_{\mu F}"'] \arrow[d, "F\ \llparenthesis g \rrparenthesis_{\mu F}"] \\
A & F\ A \arrow[l, "g"]
\end{tikzcd}$$

Unfortunately, Cata-Uniq does not say anything about non-strict functions, but as argued by Fokkinga and Meijer [FM91] it is not possible to relax any of the strictness conditions without breaking the law. As an example of this lack of expressive power, let us try to derive Cata-Fusion from Cata-Uniq. In **Set** this is typically achieved by the following calculation.

$$\begin{array}{ll}
 & f \circ \llparenthesis g \rrparenthesis_{\mu F} = \llparenthesis h \rrparenthesis_{\mu F} \\
\Leftrightarrow & \quad \{\, \textsf{Cata-Uniq} \,\} \\
 & f \circ \llparenthesis g \rrparenthesis_{\mu F} \circ \mathsf{in}_{\mu F} = g \circ F\ (f \circ \llparenthesis g \rrparenthesis_{\mu F}) \\
\Leftrightarrow & \quad \{\, \textsf{Cata-Cancel, Functor-Comp} \,\} \\
 & f \circ g \circ F\ \llparenthesis g \rrparenthesis_{\mu F} = h \circ F\ f \circ F\ \llparenthesis g \rrparenthesis_{\mu F} \\
\Leftarrow & \quad \{\, \textsf{Leibniz} \,\} \\
 & f \circ g = h \circ F\ f
\end{array}$$

In order to perform the same calculation in **CPO** $f \circ \llparenthesis g \rrparenthesis$ must be a strict function, but Cata-Fusion only requires f to be strict. Due to this problem, in [FM91] both uniqueness and fusion are independently proved using fixpoint induction.

Expressiveness. Concerning the expressiveness of catamorphisms, it can be proved that at least any strict injective function (a function with left inverse) can be defined as a catamorphism.

$$f = \llparenthesis f \circ \mathsf{in}_{\mu F} \circ F\ g \rrparenthesis_{\mu F} \quad \Leftarrow \quad g \circ f = \mathsf{id} \wedge f \text{ strict}$$

This result follows from Cata-Uniq. Notice that given the strictness of f, g is necessarily strict. This means that $f \circ \mathsf{in}_{\mu F} \circ F\ g$ is also strict and uniqueness can be applied. A complete characterization of catamorphisms in the **Set** category can be found in [GHA01]. The authors give a necessary and sufficient condition to decide if a set-theoretic function can be written as a fold, but currently it is not known how to extend this result to **CPO**.

Given this result, out can be defined as a catamorphism using its inverse in as follows.

$$\begin{aligned} \mathsf{out}_{\mu F} &: \mu F \to F\ (\mu F) \\ \mathsf{out}_{\mu F} &= \llparenthesis F\ \mathsf{in}_{\mu F} \rrparenthesis_{\mu F} \end{aligned} \qquad \text{Out-Def}$$

The proof that this function is indeed the inverse of in is given by the following calculations.

$$\begin{aligned} & \mathsf{in} \circ \mathsf{out} \\ = & \quad \{\ \text{Out-Def}\ \} \\ & \mathsf{in} \circ \llparenthesis F\ \mathsf{in} \rrparenthesis \\ = & \quad \{\ \text{Cata-Fusion, In-Strict}\ \} \\ & \llparenthesis \mathsf{in} \rrparenthesis \\ = & \quad \{\ \text{Cata-Reflex}\ \} \\ & \mathsf{id} \end{aligned}$$

$$\begin{aligned} & \mathsf{out} \circ \mathsf{in} \\ = & \quad \{\ \text{Out-Def}\ \} \\ & \llparenthesis F\ \mathsf{in} \rrparenthesis \circ \mathsf{in} \\ = & \quad \{\ \text{Cata-Cancel}\ \} \\ & F\ \mathsf{in} \circ F\ \llparenthesis F\ \mathsf{in} \rrparenthesis \\ = & \quad \{\ \text{Functor-Comp, Out-Def}\ \} \\ & F\ (\mathsf{in} \circ \mathsf{out}) \\ = & \quad \{\ \mathsf{in} \circ \mathsf{out} = \mathsf{id},\ \text{Functor-Id}\ \} \\ & \mathsf{id} \end{aligned}$$

Banana Split. Another important law about catamorphisms relates them with the split combinator, and was first presented by Fokkinga [Fok89]. Due to the notation used for representing catamorphisms it is also called the *banana split* law. It has practical usage in program optimization, since it can be used to transform a pair of traversals over a data structure into a single one.

$$\llparenthesis g \rrparenthesis_{\mu F} \vartriangle \llparenthesis h \rrparenthesis_{\mu F} = \llparenthesis g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd} \rrparenthesis_{\mu F} \qquad \text{Cata-Split}$$

$$
\left[\begin{array}{ll}
 & (\!|g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd}|\!)_{\mu F} \\
= & \quad \{\ \mathsf{Prod\text{-}Reflex}\ \} \\
 & (\mathsf{fst} \vartriangle \mathsf{snd}) \circ (\!|g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd}|\!)_{\mu F} \\
= & \quad \{\ \mathsf{Prod\text{-}Fusion}\ \} \\
 & \mathsf{fst} \circ (\!|g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd}|\!)_{\mu F} \vartriangle \mathsf{snd} \circ (\!|g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd}|\!)_{\mu F} \\
= & \quad \{\ \mathsf{Cata\text{-}Fusion},\ \mathsf{Fst\text{-}Strict}\ \} \\
 & \left[\begin{array}{ll}
 & \mathsf{fst} \circ (g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd}) = g \circ F\ \mathsf{fst} \\
 = & \quad \{\ \mathsf{Prod\text{-}Cancel}\ \} \\
 & g \circ F\ \mathsf{fst} = g \circ F\ \mathsf{fst}
 \end{array}\right. \\
 & (\!|g|\!)_{\mu F} \vartriangle \mathsf{snd} \circ (\!|g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd}|\!)_{\mu F} \\
= & \quad \{\ \mathsf{Cata\text{-}Fusion},\ \mathsf{Snd\text{-}Strict}\ \} \\
 & \left[\begin{array}{ll}
 & \mathsf{snd} \circ (g \circ F\ \mathsf{fst} \vartriangle h \circ F\ \mathsf{snd}) = h \circ F\ \mathsf{snd} \\
 = & \quad \{\ \mathsf{Prod\text{-}Cancel}\ \} \\
 & h \circ F\ \mathsf{snd} = h \circ F\ \mathsf{snd}
 \end{array}\right. \\
 & (\!|g|\!)_{\mu F} \vartriangle (\!|h|\!)_{\mu F}
\end{array}\right.
$$

Example 3.6 (Average). The classic example of using Cata-Split consists in transforming the following clear, but inefficient, function to determine the average of a list into a single pass version. $\mathsf{div} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ computes integer division.

$$
\begin{array}{lcl}
\mathsf{average} & : & \mathsf{List\ Nat} \to \mathsf{Nat} \\
\mathsf{average} & = & \mathsf{div} \circ (\mathsf{sum} \vartriangle \mathsf{length})
\end{array}
$$

After some simplifications, the resulting function is

$$
\mathsf{average} = \mathsf{div} \circ (\!|(\mathsf{zero} \triangledown \mathsf{plus} \circ (\mathsf{id} \times \mathsf{fst})) \vartriangle (\mathsf{zero} \triangledown \mathsf{succ} \circ \mathsf{snd} \circ \mathsf{snd})|\!)_{\mathsf{List\ Nat}}
$$

3.1.1 Type Functors.

The base functor of a polymorphic data type can be seen as the sectioning of a bifunctor. The base bifunctor that corresponds to a polymorphic data type $T A$ will be denoted by $\star_T$. For example, for lists such bifunctor can be defined as

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

allowing us to state

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

For each polymorphic data type, a new *type functor* can be defined, whose action on functions corresponds to the standard map function of that type. The action of a type functor on objects is the data type definition itself $T A = \mu(A \star_T)$. Given a function

$f : A \to B$, the map function can be defined generically using a catamorphism.

$$\begin{array}{lcl} T\ f & : & T\ A \to T\ B \\ T\ f & = & (\!|\mathsf{in}_T \circ (f \star_T \mathsf{id})|\!)_{TA} \end{array} \qquad \textsf{Map-Def}$$

After simplification with Sum-Absor, this definition gives the following map function for lists.

$$\mathsf{List}\ f = (\!|\mathsf{nil} \triangledown \mathsf{cons} \circ (f \times \mathsf{id})|\!)_{\mathsf{List}\ A}$$

After converting it into pointwise Haskell the expected definition is obtained.

```
map :: (a -> b) -> List a -> List b
map f Nil        = Nil
map f (Cons h t) = Cons (f h) (map f t)
```

The following law allows catamorphisms to be fused with maps (g is a function of type $A \to B$).

$$(\!|f|\!)_{TB} \circ T\ g = (\!|f \circ (g \star_T \mathsf{id})|\!)_{TA} \qquad \textsf{Cata-Map-Fusion}$$

This law can be proved using Hylo-Shift.

$$\begin{array}{cl}
 & (\!|f|\!)_{TB} \circ T\ g \\
= & \quad \{\ \textsf{Map-Def}\ \} \\
 & (\!|f|\!)_{TB} \circ (\!|\mathsf{in}_T \circ (g \star_T \mathsf{id})|\!)_{TA} \\
= & \quad \{\ \textsf{Cata-Def}\ \} \\
 & [\![f, \mathsf{out}_T]\!]_{TB} \circ [\![\mathsf{in}_T \circ (g \star_T \mathsf{id}), \mathsf{out}_T]\!]_{TA} \\
= & \quad \{\ \textsf{Hylo-Shift},\ g \star_T \mathsf{id} : (A\star_T) \dot{\to} (B\star_T)\ \} \\
 & [\![f, \mathsf{out}_T]\!]_{TB} \circ [\![\mathsf{in}_T, (g \star_T \mathsf{id}) \circ \mathsf{out}_T]\!]_{TB} \\
= & \quad \{\ \textsf{Hylo-Compose}\ \} \\
 & [\![f, (g \star_T \mathsf{id}) \circ \mathsf{out}_T]\!]_{TB} \\
= & \quad \{\ \textsf{Hylo-Shift},\ g \star_T \mathsf{id} : (A\star_T) \dot{\to} (B\star_T)\ \} \\
 & [\![f \circ (g \star_T \mathsf{id}), \mathsf{out}_T]\!]_{TA} \\
= & \quad \{\ \textsf{Cata-Def}\ \} \\
 & (\!|f \circ (g \star_T \mathsf{id})|\!)_{TA}
\end{array}$$

Example 3.7 (Squares). Consider the following two-pass function for computing the sum of the squares of a list, where $\mathsf{sq} = \mathsf{mult} \circ (\mathsf{id} \vartriangle \mathsf{id})$.

$$\begin{array}{lcl} \mathsf{sumsq} & : & \mathsf{List\ Nat} \to \mathsf{Nat} \\ \mathsf{sumsq} & = & \mathsf{sum} \circ \mathsf{List\ sq} \end{array}$$

By applying Cata-Map-Fusion and Sum-Absor it can be optimized into the following single pass implementation.

$$\begin{aligned}\mathsf{sumsq} &: \mathsf{List\ Nat} \to \mathsf{Nat}\\ \mathsf{sumsq} &= (\!|\mathsf{zero} \triangledown \mathsf{plus} \circ (\mathsf{sq} \times \mathsf{id})|\!)\end{aligned}$$

This is a well-known example in the program transformation community, where it is usually implemented using fold/unfold rules [BD77]. In Chapter 4 an example of using such technique will be presented.

Type functors can be used to define more complex types, such as rose trees, where lists are used to capture the arbitrary branching factor. The base functor of this data type is regular, but not polynomial.

$$\mathsf{Rose}\ A = \mu(\underline{A} \mathbin{\hat{\times}} \mathsf{List})$$

Example 3.8 (Preorder). The preorder traversal on rose trees can be defined as follows.

$$\begin{aligned}\mathsf{preorder} &: \mathsf{Rose}\ A \to \mathsf{List}\ A\\ \mathsf{preorder} &= (\!|\mathsf{cons} \circ (\mathsf{id} \times \mathsf{flatten})|\!)_{\mathsf{Rose}\ A}\end{aligned}$$

3.2 Anamorphisms

The recursion pattern corresponding to the dual of iteration is a standard way of producing values of a recursive data type. It is usually designated as unfold or *anamorphism*. In Haskell it is predefined for native lists as the function `unfoldr`. This definition can be adapted to the `List` data type.

```
unfold_List :: (b -> Maybe (a,b)) -> b -> List a
unfold_List h b = case h b of Nothing -> Nil
                              Just (a,c) -> Cons a (unfold_List h c)
```

The data type `Maybe a` is isomorphic to $1 + A$ and is predefined as

```
data Maybe a = Nothing | Just a
```

The function `h` has a dual role. It dictates whether to stop or to continue to generate a list (respectively, by returning `Nothing` or `Just` something). In the latter case, it also specifies the value to be placed at the head of the list, and the seed that will be used to generate the tail. For example, the function from of Example 2.3 can be defined as the following unfold.

```
from :: Nat -> List Nat
from = unfold_List h
    where h Zero     = Nothing
          h (Succ n) = Just (n,n)
```

In order to transform the definition of unfold into a hylomorphism it should first be rearranged as follows.

```
unfold_List :: (b -> Maybe (a,b)) -> b -> List a
unfold_List h b = aux (h b)
    where aux Nothing      = Nil
          aux (Just (a,c)) = Cons a (unfold_List h c)
```

From this definition, it is clear that $\mathsf{unfold}_{\mathsf{List}}\ h = \mathsf{aux} \circ h$. Concerning aux we can reason as follows.

$$\begin{array}{ll}
 & \mathsf{aux} \circ \mathsf{inl} = \mathsf{nil} \wedge \mathsf{aux} \circ \mathsf{inr} = \mathsf{cons} \circ (\mathsf{id} \times \mathsf{unfold}_{\mathsf{List}}\ h) \\
\Leftrightarrow & \quad \{\ \text{Sum-Equal}\ \} \\
 & \mathsf{aux} \circ \mathsf{inl} \triangledown \mathsf{aux} \circ \mathsf{inr} = \mathsf{nil} \triangledown \mathsf{cons} \circ (\mathsf{id} \times \mathsf{unfold}_{\mathsf{List}}\ h) \\
\Leftrightarrow & \quad \{\ \text{Sum-Fusion, aux strict, Sum-Absor}\ \} \\
 & \mathsf{aux} \circ (\mathsf{inl} \triangledown \mathsf{inr}) = (\mathsf{nil} \triangledown \mathsf{cons}) \circ (\mathsf{id} + \mathsf{id} \times \mathsf{unfold}_{\mathsf{List}}\ h) \\
\Leftrightarrow & \quad \{\ \text{Sum-Reflex}, \mathsf{in}_{\mathsf{List}} = \mathsf{nil} \triangledown \mathsf{cons}, \mathsf{F}_{\mathsf{List}} = \underline{1} \hat{+} \underline{A} \hat{\times} \mathsf{Id}\ \} \\
 & \mathsf{aux} = \mathsf{in}_{\mathsf{List}} \circ (\mathsf{F}_{\mathsf{List}}\ (\mathsf{unfold}_{\mathsf{List}}\ h))
\end{array}$$

This calculation shows that this unfold satisfies the equation

$$\mathsf{unfold}_{\mathsf{List}}\ h = \mathsf{in}_{\mathsf{List}} \circ (\mathsf{F}_{\mathsf{List}}\ (\mathsf{unfold}_{\mathsf{List}}\ h)) \circ h$$

suggesting that it can be implemented by the following hylomorphism.

$$\mathsf{unfold}_{\mathsf{List}}\ h = [\![\mathsf{in}_{\mathsf{List}}, h]\!]_{\mathsf{List}\ A}$$

A similar reasoning can be applied to the unfold of the natural numbers data type

```
unfold_Nat :: (a -> Maybe a) -> a -> Nat
unfold_Nat h a = case h a of Nothing -> Zero
                             Just b  -> Succ (unfold_Nat h b)
```

in order to get the following definition.

$$\mathsf{unfold}_{\mathsf{Nat}}\ h = [\![\mathsf{in}_{\mathsf{Nat}}, h]\!]_{\mathsf{Nat}}$$

Definition. As seen in these examples, given a function $h : A \to F\ A$, the anamorphism parameterized by g can be generically defined as follows.

$$\begin{array}{lcl}
[\!(h)\!]_{\mu F} & : & A \to \mu F \\
[\!(h)\!]_{\mu F} & = & [\![\mathsf{in}_{\mu F}, h]\!]_{\mu F}
\end{array} \qquad \text{Ana-Def}$$

Example 3.9 (From). Given the definition as a hylomorphism in Example 2.3 and Ana-Def, it is clear that from can be defined as an anamorphism.

$$\begin{array}{lcl}
\mathsf{from} & : & \mathsf{Nat} \to \mathsf{List\ Nat} \\
\mathsf{from} & = & [\!((\mathsf{id} + \mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}})\!]_{\mathsf{List\ Nat}}
\end{array}$$

Example 3.10 (Length). The same can be said about the length function presented in Example 2.2, which can be defined as

$$\begin{aligned} \mathsf{length} &: \mathsf{List}\ A \to \mathsf{Nat} \\ \mathsf{length} &= [\!(\, (\mathsf{id} + \mathsf{snd}) \circ \mathsf{out}_{\mathsf{List}} \,)\!]_{\mathsf{Nat}} \end{aligned}$$

This is an example of a function that can be defined both as a catamorphism from lists (see Example 3.2) or as an anamorphism to naturals.

Example 3.11 (Map). Another example of a function that can be defined either as a catamorphism or as an anamorphism is the map function, defined for polymorphic data types in Section 3.1.1. Given a data type $T\ A = \mu(A{\star_T})$, and a function $f : A \to B$, it can also be defined as

$$\begin{aligned} T\ f &: T\ A \to T\ B \\ T\ f &= [\!(\, (f \star_T \mathsf{id}) \circ \mathsf{out}_T \,)\!]_{T\ B} \end{aligned}$$

Example 3.12 (Repeat). As seen in Section 2.3, the data type List A contains also totally defined infinite lists with elements of type A. As such, it is possible to define the following anamorphism that generates infinitely many copies of a given value. The idea is to take two copies of the input, one to put in the head of the list, and another to seed the generation of the tail.

$$\begin{aligned} \mathsf{repeat} &: A \to \mathsf{List}\ A \\ \mathsf{repeat} &= [\!(\, \mathsf{inr} \circ (\mathsf{id} \vartriangle \mathsf{id}) \,)\!]_{\mathsf{List}\ A} \end{aligned}$$

Example 3.13 (Plus). A more substantial example is the definition of plus as an anamorphism. In this case, it helps to first define the Haskell version using `unfold_Nat`.

```
plus :: (Nat, Nat) -> Nat
plus = unfold_Nat h
    where h (Zero  , Zero  ) = Nothing
          h (Succ n, Zero  ) = Just (n, Zero)
          h (Zero  , Succ n) = Just (Zero, n)
          h (Succ n, Succ m) = Just (Succ n, m)
```

The generation of the result stops when both arguments are zero. Otherwise it proceeds by decreasing one of the components of the seed. In the point-free style, pattern matching on a type is achieved by judicious use of the out function. In this particular case, since the input is a pair of naturals, $\mathsf{out} \times \mathsf{out}$ will be used in order to inspect both arguments. To get the expected four alternatives, the result of this inspection can be rearranged using the following auxiliary definition.

$$\begin{aligned} \mathsf{aux} &: (1 + \mathsf{Nat}) \times (1 + \mathsf{Nat}) \to 1 \times 1 + (\mathsf{Nat} \times 1 + (1 \times \mathsf{Nat} + \mathsf{Nat} \times \mathsf{Nat})) \\ \mathsf{aux} &= \mathsf{coassocr} \circ (\mathsf{distl} + \mathsf{distl}) \circ \mathsf{distr} \end{aligned}$$

The result of this function must then be converted into a value of type $1 + \mathsf{Nat} \times \mathsf{Nat}$. Notice that the rearranging function was already defined in order to make this conversion easy, namely by grouping all the alternatives that yield a new seed into the right summand. The final definition is

$$\begin{aligned}
\mathsf{plus} &: \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{plus} &= [\![(! + (\mathsf{id} \times \mathsf{zero} \triangledown (\mathsf{zero} \times \mathsf{id} \triangledown \mathsf{succ} \times \mathsf{id}))) \circ \mathsf{aux} \circ (\mathsf{out} \times \mathsf{out})]\!]_{\mathsf{Nat}}
\end{aligned}$$

Example 3.14 (Zip). A similar example is the zip function, that given a pair of lists generates a list of pairs. A slightly different auxiliary function will be used, because in this case the generation of the result only proceeds if both lists are non-empty, which leads to a different rearranging of the pattern matching alternatives. Given

$$\mathsf{aux} = \mathsf{coassocl} \circ (\mathsf{distl} + \mathsf{distl}) \circ \mathsf{distr}$$

zip can be defined as follows.

$$\begin{aligned}
\mathsf{zip} &: \mathsf{List}\ A \times \mathsf{List}\ B \to \mathsf{List}\ (A \times B) \\
\mathsf{zip} &= [\![(! + (\mathsf{fst} \times \mathsf{fst}) \vartriangle (\mathsf{snd} \times \mathsf{snd})) \circ \mathsf{aux} \circ (\mathsf{out} \times \mathsf{out})]\!]_{\mathsf{List}\ (A \times B)}
\end{aligned}$$

Laws. From the definition of the anamorphism as a hylomorphism it is possible to derive the following laws to reason about this recursion pattern.

$$[\![\mathsf{out}_{\mu F}]\!]_{\mu F} = \mathsf{id}_{\mu F} \qquad \mathsf{Ana\text{-}Reflex}$$

$$\mathsf{out}_{\mu F} \circ [\![h]\!]_{\mu F} = F\ [\![h]\!]_{\mu F} \circ h \qquad \mathsf{Ana\text{-}Cancel}$$

$$[\![g]\!]_{\mu F} \circ f = [\![h]\!]_{\mu F} \quad \Leftarrow \quad g \circ f = F\ f \circ h \qquad \mathsf{Ana\text{-}Fusion}$$

$$[\![h]\!]_{\mu F} \text{ strict} \quad \Leftarrow \quad h \text{ strict} \qquad \mathsf{Ana\text{-}Strict}$$

The uniqueness law for anamorphisms follows trivially from Ana-Cancel and Ana-Fusion.

$$f = [\![g]\!]_F \quad \Leftrightarrow \quad \mathsf{out}_F \circ f = F f \circ g \qquad \mathsf{Ana\text{-}Uniq}$$

Since there are no strictness side conditions, this uniqueness law means that $\mathsf{out}_{\mu F}$ is a final F-*coalgebra* (a function of type $A \to F\ A$ for a carrier A) not only in the **CPO**$_\perp$ category, but also in **CPO**. Given any $h : A \to F\ A$, $[\![h]\!]_{\mu F}$ is the unique function that makes the following diagram commute.

$$\begin{tikzcd}
A \arrow[r, "h"] \arrow[d, dotted, "{[\![h]\!]_{\mu F}}"'] & F\ A \arrow[d, "{F\ [\![h]\!]_{\mu F}}"] \\
\mu F \arrow[r, "\mathsf{out}_{\mu F}"'] & F\ (\mu F)
\end{tikzcd}$$

Expressiveness. Using Ana-Uniq it is easy to prove that any surjective function (a function with right-inverse) can be defined as an anamorphism.

$$f = [\![F\ g \circ \mathsf{out}_{\mu F} \circ f]\!]_{\mu F} \quad \Leftarrow \quad f \circ g = \mathsf{id}$$

This result leads to the following definition of in as an anamorphism.

$$\begin{array}{lcl} \mathsf{in}_{\mu F} & : & F\ \mu F \rightarrow \mu F \\ \mathsf{in}_{\mu F} & = & [\![F\ \mathsf{out}_{\mu F}]\!]_{\mu F} \end{array} \qquad \text{In-Def}$$

The banana split can be dualized to anamorphisms, but the resulting law is not useful for program optimization because in either case the resulting data structure is only traversed once.

$$[\![g]\!]_{\mu F} \triangledown [\![h]\!]_{\mu F} = [\![F\ \mathsf{inl} \circ g \triangledown F\ \mathsf{inr} \circ h]\!]_{\mu F} \qquad \text{Ana-Either}$$

3.2.1 Splitting Hylomorphisms

Equipped with these recursion patterns, it is now possible to present one of the most essential laws about hylomorphisms – the one that states their factorization into the composition of a catamorphism with an anamorphism.

$$[\![g, h]\!]_F = (\!|g|\!)_F \circ [\![h]\!]_F \qquad \text{Hylo-Split}$$

This law follows directly from the definitions and Hylo-Compose. Notice that, traditionally, hylomorphisms are defined using this equation, instead of directly using fixpoints.

Hylo-Split can be used to expose the call tree of a recursive definition as an intermediate data structure, and to present the hylomorphism as the composition of two functions: one that builds this data structure from the input (the anamorphism), and another that traverses it in order to produce the result (the catamorphism). This fact has some nice implications in program understanding. For example, looking at the factorial definition as a hylomorphism, it is clear that this function is just the product of all integers from the input n down to 1: the catamorphism is a function that multiplies all numbers in a list, and the anamorphism is the function that generates the list with all values between n and 1.

3.3 Paramorphisms

While catamorphisms encode the recursion pattern of iteration, *paramorphisms* encode the notion of primitive recursion [Mee92]. The difference is that the latter can use both the recursive call on a substructure of the input and the substructure itself to compute the result. For lists it can be implemented as follows.

```
para_List :: (a -> b -> List a -> b) -> b -> List a -> b
para_List f z Nil        = z
para_List f z (Cons h t) = f h (para_List f z t) t
```

Definition. The definition of a paramorphism as a hylomorphism is known at least since [MFP91]. The idea is that the anamorphism should make a copy of the parameter of a recursive invocation to be passed intact into the catamorphism. This means that, unlike the definition of catamorphism, the intermediate data type will no longer be equal to the data type being consumed. Given an input of type μF, the functor that generates the intermediate data structure is $F \circ (\mathsf{Id} \mathbin{\hat{\times}} \underline{\mu F})$: every recursive occurrence of the original type is replaced by a new recursive occurrence and a copy of the older one that will be left intact when recursing. For example, a paramorphism over naturals will have as intermediate data structure an element of type $\mu(\underline{1} \mathbin{\hat{+}} \mathsf{Id} \mathbin{\hat{\times}} \underline{\mathsf{Nat}})$, which is isomorphic to a list of naturals.

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

$$\begin{array}{lcl} \langle\!| g |\!\rangle_{\mu F} & : & \mu F \to A \\ \langle\!| g |\!\rangle_{\mu F} & = & \llbracket g, F\ (\mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mu F} \rrbracket_{\mu(F \circ (\mathsf{Id} \hat{\times} \underline{\mu F}))} \end{array} \qquad \text{Para-Def}$$

A diagram provides a better understanding of this definition.

$$\begin{tikzcd} \mu F \arrow[r, "\mathsf{out}_{\mu F}"] \arrow[d, "\langle\!| g |\!\rangle_{\mu F}"'] & F\ (\mu F) \arrow[r, "F\ (\mathsf{id} \vartriangle \mathsf{id})"] & F\ (\mu F \times \mu F) \arrow[d, "F\ (\langle\!| g |\!\rangle_{\mu F} \times \mathsf{id})"] \\ A & & F\ (A \times \mu F) \arrow[ll, "g"] \end{tikzcd}$$

In Meertens' original definition [Mee92], a paramorphism is encoded using a catamorphism that simultaneously produces a pair with the result and a (recursively computed) copy of the input. In the end, the second component of the result is discarded.

$$\langle\!| g |\!\rangle_{\mu F} = \mathsf{fst} \circ (\!| g \vartriangle \mathsf{in} \circ F\ \mathsf{snd} |\!)_{\mu F} \qquad \text{Para-Cata}$$

This definition is less comprehensible than the one using hylomorphisms, but will be useful to prove the uniqueness law of paramorphisms. Uniqueness will then provide a trivial proof that both definitions are indeed equivalent.

Example 3.15 (Factorial). The most well-known example of a paramorphism is the factorial function: for a nonzero input n, it is defined as the result of multiplying n by the recursive result $\mathsf{fact}\ (n-1)$.

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

Example 3.16 (Sorted). Testing if a list is sorted is also a paramorphism, because the head of the list must be compared with the head of the tail. $\mathsf{and} : \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}$ implements logical conjunction, $\mathsf{le} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Bool}$ tests if the first number is less or equal then the second, and $\mathsf{aux} = \mathsf{assocl} \circ (\mathsf{id} \times \mathsf{swap})$.

$$\begin{array}{lcl}\mathsf{sorted} & : & \mathsf{List}\ A \to \mathsf{Bool} \\ \mathsf{sorted} & = & \langle\!|\mathsf{true} \triangledown \mathsf{and} \circ ((\mathsf{true} \circ ! \triangledown \mathsf{le} \circ (\mathsf{id} \times \mathsf{head})) \circ (\mathsf{isnil} \circ \mathsf{snd})? \times \mathsf{id}) \circ \mathsf{aux}|\!\rangle_{\mathsf{List}\ A}\end{array}$$

Laws. Given Para-Def, most of the laws about paramorphisms can be derived from the laws of hylomorphisms.

$$\langle\!|\mathsf{in}_{\mu F} \circ F\ \mathsf{fst}|\!\rangle_{\mu F} = \mathsf{id}_{\mu F} \qquad \textsf{Para-Reflex}$$

$$\langle\!|g|\!\rangle_{\mu F} \circ \mathsf{in}_{\mu F} = g \circ F\ (\langle\!|g|\!\rangle_{\mu F} \vartriangle \mathsf{id}) \qquad \textsf{Para-Cancel}$$

$$f \circ \langle\!|g|\!\rangle_{\mu F} = \langle\!|h|\!\rangle_{\mu F} \quad \Leftarrow \quad f \circ g = h \circ F\ (f \times \mathsf{id}) \wedge f \text{ strict} \qquad \textsf{Para-Fusion}$$

$$\langle\!|g|\!\rangle_{\mu F} \text{ strict} \quad \Leftrightarrow \quad g \text{ strict} \qquad \textsf{Para-Strict}$$

For example, Para-Reflex can be proved as follows.

$$\begin{array}{cl} & \langle\!|\mathsf{in}_{\mu F} \circ F\ \mathsf{fst}|\!\rangle_{\mu F} \\ = & \quad \{\ \textsf{Para-Def}\ \} \\ & [\![\mathsf{in}_{\mu F} \circ F\ \mathsf{fst}, F\ (\mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mu F}]\!]_{\mu(F \circ (\mathsf{Id}\, \hat{\times}\, \underline{\mu F}))} \\ = & \quad \{\ \textsf{Hylo-Shift},\ F\ \mathsf{fst} : F \circ (\mathsf{Id}\ \hat{\times}\ \underline{\mu F}) \xrightarrow{.} F\ \} \\ & [\![\mathsf{in}_{\mu F}, F\ \mathsf{fst} \circ F\ (\mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mu F}]\!]_{\mu F} \\ = & \quad \{\ \textsf{Functor-Comp, Prod-Cancel, Functor-Id}\ \} \\ & [\![\mathsf{in}_{\mu F}, \mathsf{out}_{\mu F}]\!]_{\mu F} \\ = & \quad \{\ \textsf{Hylo-Reflex}\ \} \\ & \mathsf{id}_{\mu F} \end{array}$$

These laws can also be proved using the definition Para-Cata, but some proofs become more lengthy. For example, the proof of Para-Cancel is immediate from Para-Def, but using Para-Cata we must argue as follows. Notice the need to prove that the second component of the result of the catamorphism is indeed the (recursive) identity function.

$$
\left[\begin{array}{ll}
 & \langle\!|g|\!\rangle \circ \mathsf{in} \\
= & \quad \{\, \mathsf{Para\text{-}Cata} \,\} \\
 & \mathsf{fst} \circ \llparenthesis g \vartriangle \mathsf{in} \circ F\ \mathsf{snd} \rrparenthesis \circ \mathsf{in} \\
= & \quad \{\, \mathsf{Cata\text{-}Cancel} \,\} \\
 & \mathsf{fst} \circ (g \vartriangle \mathsf{in} \circ F\ \mathsf{snd}) \circ F\ \llparenthesis g \vartriangle \mathsf{in} \circ F\ \mathsf{snd} \rrparenthesis \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel},\ \mathsf{Prod\text{-}Reflex} \,\} \\
 & g \circ F\ ((\mathsf{fst} \vartriangle \mathsf{snd}) \circ \llparenthesis g \vartriangle \mathsf{in} \circ F\ \mathsf{snd} \rrparenthesis) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion},\ \mathsf{Para\text{-}Cata} \,\} \\
 & g \circ F\ (\langle\!|g|\!\rangle \vartriangle \mathsf{snd} \circ \llparenthesis g \vartriangle \mathsf{in} \circ F\ \mathsf{snd} \rrparenthesis) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion} \,\} \\
 & \quad \left[\begin{array}{ll}
 & \mathsf{snd} \circ (g \vartriangle \mathsf{in} \circ F\ \mathsf{snd}) \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
 & \mathsf{in} \circ F\ \mathsf{snd}
\end{array}\right. \\
 & g \circ F\ (\langle\!|g|\!\rangle \vartriangle \llparenthesis \mathsf{in} \rrparenthesis) \\
= & \quad \{\, \mathsf{Cata\text{-}Reflex} \,\} \\
 & g \circ F\ (\langle\!|g|\!\rangle \vartriangle \mathsf{id})
\end{array}\right.
$$

The uniqueness law for paramorphisms is specified as follows.

$$
f = \langle\!|g|\!\rangle_{\mu F} \wedge g \text{ strict} \quad \Leftrightarrow \quad f \circ \mathsf{in}_{\mu F} = g \circ F\ (f \vartriangle \mathsf{id}) \wedge f \text{ strict} \qquad \mathsf{Para\text{-}Uniq}
$$

Unfortunately this law cannot be derived from the previous ones, as was the case with catamorphisms and anamorphisms. Instead we must resort to the original proof by Meertens [Mee92], based on the definition Para-Cata and the uniqueness of catamorphisms. The proof of the $\Rightarrow$ implication is the same of Para-Cancel stated above. The other implication can be proved as follows.

$$
\left[\begin{array}{ll}
 & f \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
 & \mathsf{fst} \circ (f \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Cata\text{-}Uniq},\ \mathsf{Prod\text{-}Strict},\ f \text{ strict},\ \mathsf{id} \text{ strict} \,\} \\
 & \quad \left[\begin{array}{ll}
 & (f \vartriangle \mathsf{id}) \circ \mathsf{in} \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion} \,\} \\
 & f \circ \mathsf{in} \vartriangle \mathsf{in} \\
= & \quad \{\, f \circ \mathsf{in} = g \circ F\ (f \vartriangle \mathsf{id}) \,\} \\
 & g \circ F\ (f \vartriangle \mathsf{id}) \vartriangle \mathsf{in} \\
= & \quad \{\, \mathsf{Functor\text{-}Id} \,\} \\
 & g \circ F\ (f \vartriangle \mathsf{id}) \vartriangle \mathsf{in} \circ F\ \mathsf{id} \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
 & g \circ F\ (f \vartriangle \mathsf{id}) \vartriangle \mathsf{in} \circ F\ \mathsf{snd} \circ F\ (f \vartriangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion} \,\} \\
 & (g \vartriangle \mathsf{in} \circ F\ \mathsf{snd}) \circ F\ (f \vartriangle \mathsf{id})
\end{array}\right. \\
 & \mathsf{fst} \circ \llparenthesis g \vartriangle \mathsf{in} \circ F\ \mathsf{snd} \rrparenthesis \\
= & \quad \{\, \mathsf{Para\text{-}Cata} \,\} \\
 & \langle\!|g|\!\rangle
\end{array}\right.
$$

Meertens' original proof did not include the strictness side conditions. Essentially, these are equivalent to Para-Strict, which can be proved as follows.

$$\begin{array}{ll}
 & g \text{ strict} \\
\Rightarrow & \quad \{\,\textsf{In-Strict, Snd-Strict, Strict-Comp, Prod-Strict}\,\} \\
 & g \vartriangle \textsf{in} \circ F\ \textsf{snd} \text{ strict} \\
\Rightarrow & \quad \{\,\textsf{Cata-Strict}\,\} \\
 & (\!| g \vartriangle \textsf{in} \circ F\ \textsf{snd} |\!) \text{ strict} \\
\Rightarrow & \quad \{\,\textsf{Fst-Strict, Strict-Comp, Para-Cata}\,\} \\
 & \langle\!| g |\!\rangle \text{ strict}
\end{array}
\qquad
\begin{array}{ll}
 & \langle\!| g |\!\rangle \text{ strict} \\
\Rightarrow & \quad \{\,\textsf{In-Strict, Strict-Comp}\,\} \\
 & \langle\!| g |\!\rangle \circ \textsf{in} \text{ strict} \\
\Rightarrow & \quad \{\,\textsf{Para-Cancel}\,\} \\
 & g \circ F\ (\langle\!| g |\!\rangle \vartriangle \textsf{id}) \text{ strict} \\
\Rightarrow & \quad \{\,\textsf{Comp-Strict}\,\} \\
 & g \text{ strict}
\end{array}$$

Since the definition using a hylomorphism satisfies Para-Cancel, by uniqueness both definitions are equal, at least for strict parameters.

Expressiveness. Since the expressive power of paramorphisms is greater than that of catamorphisms, it is possible to express the latter using the former.

$$(\!| g |\!)_{\mu F} = \langle\!| g \circ F\ \textsf{fst} |\!\rangle_{\mu F} \qquad \textsf{Cata-Para}$$

The proof is similar to that of Para-Reflex.

$$\begin{array}{ll}
 & \langle\!| g \circ F\ \textsf{fst} |\!\rangle_{\mu F} \\
= & \quad \{\,\textsf{Para-Def}\,\} \\
 & [\![g \circ F\ \textsf{fst}, F\ (\textsf{id} \vartriangle \textsf{id}) \circ \textsf{out}_{\mu F}]\!]_{\mu(F \circ (\textsf{Id}\, \hat{\times}\, \underline{\mu F}))} \\
= & \quad \{\,\textsf{Hylo-Shift},\ F\ \textsf{fst} : F \circ (\textsf{Id}\ \hat{\times}\ \underline{\mu F}) \dot{\rightarrow} F\,\} \\
 & [\![g, F\ \textsf{fst} \circ F\ (\textsf{id} \vartriangle \textsf{id}) \circ \textsf{out}_{\mu F}]\!]_{\mu F} \\
= & \quad \{\,\textsf{Functor-Comp, Prod-Cancel, Functor-Id}\,\} \\
 & [\![g, \textsf{out}_{\mu F}]\!]_{\mu F} \\
= & \quad \{\,\textsf{Cata-Def}\,\} \\
 & (\!| g |\!)_{\mu F}
\end{array}$$

As a final remark, it can be proved that any function of the appropriate type can be defined as a paramorphism.

$$g = \langle\!| g \circ \textsf{in}_{\mu F} \circ F\ \textsf{snd} |\!\rangle_{\mu F}$$

As the proof shows, this result is only of theoretical interest. Notice that the resulting function is not even recursive.

$$\begin{array}{ll}
 & \langle\!| g \circ \textsf{in}_{\mu F} \circ F\ \textsf{snd} |\!\rangle_{\mu F} \\
= & \quad \{\,\textsf{Para-Cancel, In-Out-Iso}\,\} \\
 & g \circ \textsf{in}_{\mu F} \circ F\ \textsf{snd} \circ F\ (\langle\!| g \circ \textsf{in}_{\mu F} \circ F\ \textsf{snd} |\!\rangle_{\mu F} \vartriangle \textsf{id}) \circ \textsf{out}_{\mu F} \\
= & \quad \{\,\textsf{Functor-Comp, Prod-Cancel}\,\} \\
 & g \circ \textsf{in}_{\mu F} \circ F\ \textsf{id} \circ \textsf{out}_{\mu F} \\
= & \quad \{\,\textsf{Functor-Id, In-Out-Iso}\,\} \\
 & g
\end{array}$$

3.4 Apomorphisms

The *apomorphism* is the dual recursion pattern of the paramorphism, and therefore it can express functions defined by primitive corecursion. Apomorphisms were introduced independently by Vene and Uustalu [VU98] and Vos [Vos95]. While in an anamorphism the coalgebra determines the arguments of the recursive calls, the parameter of an apomorphism can also be used to determine directly the result without recursion. For example, for naturals this recursion pattern can be encoded in Haskell as follows.

```
apo_Nat :: (a -> Maybe (Either a Nat)) -> a -> Nat
apo_Nat h a = case h a of Nothing        -> Zero
                          Just (Left b)  -> Succ (apo_Nat h b)
                          Just (Right n) -> Succ n
```

Definition. Given a function $h : A \to F\ (A + \mu F)$, an apomorphism $\llparenthesis h \rrparenthesis$ can be defined generically using a hylomorphism.

$$\begin{array}{lcl} \llparenthesis h \rrparenthesis_{\mu F} & : & A \to \mu F \\ \llparenthesis h \rrparenthesis_{\mu F} & = & \llbracket \mathsf{in}_{\mu F} \circ F\ (\mathsf{id} \triangledown \mathsf{id}), h \rrbracket_{\mu(F \circ (\mathsf{Id} \hat{+} \underline{\mu F}))} \end{array} \qquad \text{Apo-Def}$$

The typing information is summarized in the following diagram.

$$\begin{tikzcd} A \arrow[rr, "h"] \arrow[d, "\llparenthesis h \rrparenthesis_{\mu F}"'] & & F\ (A + \mu F) \arrow[d, "F\ (\llparenthesis h \rrparenthesis_{\mu F} + \mathsf{id})"] \\ \mu F & F\ (\mu F) \arrow[l, "\mathsf{in}_{\mu F}"] & F\ (\mu F + \mu F) \arrow[l, "F\ (\mathsf{id} \triangledown \mathsf{id})"] \end{tikzcd}$$

Dually to paramorphisms, apomorphisms were initially defined using anamorphisms and sums [VU98].

$$\llparenthesis h \rrparenthesis_{\mu F} = [\!(h \triangledown F\ \mathsf{inr} \circ \mathsf{out}_{\mu F})\!]_{\mu F} \circ \mathsf{inl} \qquad \text{Apo-Ana}$$

Example 3.17 (Plus). Using apomorphisms a more natural definition of plus can be given. Unlike the definition using anamorphisms (presented in Example 3.13), where both arguments were fully traversed recursively, with apomorphisms one of the arguments can be immediately returned when the other reaches zero. The implementation using apo_Nat is

```
plus :: (Nat, Nat) -> Nat
plus = apo_Nat h
    where h (Zero,   Zero) = Nothing
          h (Succ n, Zero) = Just (Right n)
          h (n,    Succ m) = Just (Left (n,m))
```

In the point-free style it can be defined as follows. Some extra work is needed in order to simulate the non disjunct pattern matching.

$$\begin{array}{lcl} \mathsf{plus} & : & \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{plus} & = & [\![(\mathsf{id} + \mathsf{coswap}) \circ \mathsf{coassocr} \circ (\mathsf{fst} + \mathsf{in} \times \mathsf{id}) \circ \mathsf{distr} \circ (\mathsf{out} \times \mathsf{out})]\!]_{\mathsf{Nat}} \end{array}$$

Example 3.18 (Snoc). Appending an element to the end of a list can also be implemented as an apomorphism. When the end of the list is reached, the singleton list with the element is returned.

$$\begin{array}{lcl} \mathsf{snoc} & : & A \times \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{snoc} & = & [\![\mathsf{inr} \circ \mathsf{undistr} \circ \mathsf{coswap} \circ (\mathsf{id} \times \mathsf{nil} + \mathsf{aux}) \circ \mathsf{distr} \circ (\mathsf{id} \times \mathsf{out})]\!]_{\mathsf{List}\ A} \end{array}$$

This definition uses the following auxiliary rearranging function.

$$\begin{array}{lcl} \mathsf{aux} & : & A \times (B \times C) \to B \times (A \times C) \\ \mathsf{aux} & = & \mathsf{assocr} \circ (\mathsf{swap} \times \mathsf{id}) \circ \mathsf{assocl} \end{array}$$

Example 3.19 (Insert). Another good example of an apomorphism is the function that inserts an element in an ordered list, since when the correct place is found the function can return immediately.

$$\begin{array}{lcl} \mathsf{insert} & : & A \times \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{insert} & = & [\![\mathsf{inr} \circ \mathsf{undistr} \circ (\mathsf{inr} \triangledown (\mathsf{inr} \triangledown \mathsf{inl})) \circ \mathsf{aux} \circ \mathsf{distr} \circ (\mathsf{id} \times \mathsf{out})]\!]_{\mathsf{List}\ A} \end{array}$$

The definition of the auxiliary function is quite complex, mainly due to the need to rearrange values.

$$\mathsf{aux} = \mathsf{id} \times \mathsf{nil} + ((\mathsf{id} \times \mathsf{cons}) \circ \mathsf{assocr} + \mathsf{assocr} \circ (\mathsf{swap} \times \mathsf{id})) \circ \mathsf{distl} \circ (\mathsf{le?} \times \mathsf{id}) \circ \mathsf{assocl}$$

Laws. From the definition Apo-Def the following laws can be easily derived. They can also be derived from Apo-Ana with some additional work.

$$[\![F\ \mathsf{inl} \circ \mathsf{out}_{\mu F}]\!] = \mathsf{id}_{\mu F} \qquad \text{Apo-Reflex}$$

$$\mathsf{out}_{\mu F} \circ [\![h]\!]_{\mu F} = F\ ([\![h]\!]_{\mu F} \triangledown \mathsf{id}) \circ h \qquad \text{Apo-Cancel}$$

$$[\![f]\!]_{\mu F} \circ g = [\![h]\!]_{\mu F} \quad \Leftarrow \quad f \circ g = F\ (g + \mathsf{id}) \circ h \qquad \text{Apo-Fusion}$$

$$[\![h]\!]_{\mu F} \text{ strict} \quad \Leftarrow \quad h \text{ strict} \qquad \text{Apo-Strict}$$

The uniqueness law for apomorphisms is specified as follows.

$$f = [\![h]\!]_{\mu F} \quad \Leftrightarrow \quad \mathsf{out}_{\mu F} \circ f = F\ (f \triangledown \mathsf{id}) \circ h \qquad \text{Apo-Uniq}$$

Likewise to paramorphisms, a direct proof of uniqueness cannot be given using definition Apo-Def and the basic hylomorphism laws. Instead, definition Apo-Ana and the uniqueness law of anamorphisms must be used. The proof is similar to that of paramorphisms and will be omitted (see [VU98] for details). Since Apo-Cancel follows from Apo-Def, by uniqueness this definition is equivalent to Apo-Ana.

Expressiveness. The fact that apomorphisms are at least as expressive as anamorphisms is clear in the following law.

$$[\![h]\!]_{\mu F} = [\![F\ \mathsf{inl} \circ h]\!]_{\mu F} \qquad \text{Ana-Apo}$$

Finally, similarly to paramorphisms, it can be proved that any function of the appropriate type can be defined as an apomorphism.

$$h = [\![F\ \mathsf{inr} \circ \mathsf{out}_{\mu F} \circ h]\!]_{\mu F}$$

3.5 Accumulations

Accumulations are binary functions that use the second parameter to store intermediate results. The so called “accumulation technique” is typically used in functional programming to derive efficient implementations of some recursive functions. In the next chapter we will present several examples of using this technique, but with accumulations defined in the curried style as higher order catamorphisms. Alberto Pardo defined (in the **Set** category) a generic version of an uncurried accumulation recursion pattern, that can be used for any inductive data type [Par03].

This recursion pattern can be implemented in Haskell for the `Nat` data type as follows. Unlike the previous recursion patterns, this operator is parameterized by two functions. The first is used to combine the recursive result and the accumulating parameter in order to produce the result. The second determines how the accumulator is modified between recursive calls.

```
accum_Nat :: ((Maybe a, b) -> a) -> (b -> b) -> (Nat, b) -> a
accum_Nat g t (Zero,   x) = g (Nothing, x)
accum_Nat g t (Succ n, x) = g (Just (accum_Nat g t (n, t x)), x)
```

For example, `plus` can easily be implemented with this recursion operator, by treating the second argument as an accumulating parameter. There are two possible implementations. The first does not take advantage of the accumulator, and uses it as a constant parameter.

```
plus :: (Nat, Nat) -> Nat
plus = accum_Nat g id
```

```
    where g (Nothing, x) = x
          g (Just n,  x) = Succ n
```

The second increments the accumulator while traversing the first argument, and essentially corresponds to a tail-recursive implementation of addition.

```
plus :: (Nat, Nat) -> Nat
plus = accum_Nat g Succ
    where g (Nothing, x) = x
          g (Just n,  x) = n
```

Definition. This recursion pattern can be expressed as a hylomorphism using as intermediate structure a *labeled variant* of the input data type – all nodes of the input are tagged with a value of the same type as the accumulating parameter. Assuming that this type is X, the labeled variant of μF can be obtained by $\mu(F \mathbin{\hat{\times}} \underline{X})$. For example, an accumulation over naturals uses the intermediate data type $\mu((\underline{1} \mathbin{\hat{+}} \mathsf{Id}) \mathbin{\hat{\times}} \underline{X})$. If **CPO** was distributive, this data type would be isomorphic to non-empty lists with elements of type X.

Given $g : F A \times X \to A$, and $\tau : F\ (\mu F) \times X \to F\ (\mu F \times X)$, an accumulation $\{\!|g, \tau|\!\}$ can be defined generically as the following hylomorphism.

$$\begin{aligned} \{\!|g,\tau|\!\}_{\mu F} &: \mu F \times X \to A \\ \{\!|g,\tau|\!\}_{\mu F} &= \llbracket g, (\tau \mathbin{\triangle} \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \rrbracket_{\mu(F \hat{\times} \underline{X})} \end{aligned} \qquad \textsf{Accum-Def}$$

The typing information is depicted in the following diagram.

$$\begin{tikzcd} \mu F \times X \arrow[r, "\mathsf{out}_{\mu F} \times \mathsf{id}"] \arrow[d, "\{\!|g,\tau|\!\}_{\mu F}"'] & F\ (\mu F) \times X \arrow[r, "\tau \triangle \mathsf{snd}"] & F\ (\mu F \times X) \times X \arrow[d, "F\ \{\!|g,\tau|\!\}_{\mu F} \times \mathsf{id}"] \\ A & & F\ A \times X \arrow[ll, "g"] \end{tikzcd}$$

As can be seen in this definition, τ is the function responsible for propagating the accumulator to the recursive calls. This function should be *proper for accumulation*, that is, it should be a natural transformation of type

$$\tau : F \mathbin{\hat{\times}} \underline{X} \mathrel{\dot{\to}} F \circ (\mathsf{Id} \mathbin{\hat{\times}} \underline{X}) \qquad \textsf{Tau-Nat}$$

which guarantees that the propagated accumulation does not depend on the recursive values of the data type. τ also cannot modify the shape of the data type, or the data contained in it, as captured by the following restriction.

$$F\ \mathsf{fst} \circ \tau = \mathsf{fst} \qquad \textsf{Tau-Cancel}$$

As shown in [Par00], this recursion pattern can also be defined in curried form as a higher-order catamorphism, that is, one that returns a value of functional type. To recover the expected type this catamorphism must be uncurried. According to Exp-Uniq, that can be done by wrapping it with $\mathsf{ap} \circ (\cdot \times \mathsf{id})$. The resulting definition is

$$\{\!|g, \tau|\!\}_{\mu F} = \mathsf{ap} \circ ((\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!)_{\mu F} \times \mathsf{id}) \qquad \text{Accum-Cata}$$

Although rather incomprehensible, this alternative definition is useful to prove the uniqueness law, and can be proved equivalent to Accum-Def.

Example 3.20 (Plus). As seen above, plus is one of the simplest examples of definitions through accumulation. For naturals, in order for τ to be proper for accumulation, it should have shape $\mathsf{fst} + \mathsf{id} \times \phi$, where $\phi : X \to X$. This is the reason why in the above Haskell implementation a function of this type was required as parameter. The tail recursive plus can be defined as follows.

$$\begin{array}{lcl} \mathsf{plus} & : & \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{plus} & = & \{\!|(\mathsf{snd} \triangledown \mathsf{fst}) \circ \mathsf{distl}, (\mathsf{fst} + \mathsf{id} \times \mathsf{succ}) \circ \mathsf{distl}|\!\}_{\mathsf{Nat}} \end{array}$$

Example 3.21 (Reverse). It is possible to define an efficient tail recursive reverse function using accumulations. While traversing the input list, the current element is added at the head of the accumulator. For lists, in order for τ to be proper for accumulation, it should have shape $(\mathsf{fst} + \mathsf{assocr} \circ (\mathsf{fst} \vartriangle \phi \circ (\mathsf{fst} \times \mathsf{id}))) \circ \mathsf{distl}$, where $\phi : A \times X \to X$. This last function reflects the fact that the new value of the accumulator can only depend on the head of the list and on its own current value. In the reverse function ϕ will be instantiated with cons. The accumulation is encoded in the following auxiliary function.

$$\mathsf{aux} = \{\!|(\mathsf{snd} \triangledown \mathsf{snd} \circ \mathsf{fst}) \circ \mathsf{distl}, (\mathsf{fst} + \mathsf{assocr} \circ (\mathsf{fst} \vartriangle \mathsf{cons} \circ (\mathsf{fst} \times \mathsf{id}))) \circ \mathsf{distl}|\!\}_{\mathsf{List}\ A}$$

The reverse function just needs to initialize the accumulator with the empty list.

$$\begin{array}{lcl} \mathsf{reverse} & : & \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{reverse} & = & \mathsf{aux} \circ (\mathsf{id} \vartriangle \mathsf{nil} \circ !) \end{array}$$

Laws. Again, most of the laws about generic accumulations can be derived from the definition as a hylomorphism. For example, the reflexivity law is defined and proved as follows.

$$\{\!|\mathsf{in}_{\mu F} \circ \mathsf{fst}, \tau|\!\}_{\mu F} = \mathsf{fst} \qquad \text{Accum-Reflex}$$

$$
\begin{array}{ll}
 & \{\!|\mathsf{in}_{\mu F} \circ \mathsf{fst}, \tau|\!\}_{\mu F} \\
= & \quad \{\ \mathsf{Accum\text{-}Def}\ \} \\
 & [\![\mathsf{in}_{\mu F} \circ \mathsf{fst}, (\tau \vartriangle \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id})]\!]_{\mu(F \hat{\times} \underline{X})} \\
= & \quad \{\ \mathsf{Hylo\text{-}Shift},\ \mathsf{fst} : F \hat{\times} \underline{X} \dot{\to} F\ \} \\
 & [\![\mathsf{in}_{\mu F}, \mathsf{fst} \circ (\tau \vartriangle \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id})]\!]_{\mu F} \\
= & \quad \{\ \mathsf{Prod\text{-}Cancel}\ \} \\
 & [\![\mathsf{in}_{\mu F}, \tau \circ (\mathsf{out}_{\mu F} \times \mathsf{id})]\!]_{\mu F} \\
= & \quad \{\ \mathsf{Hylo\text{-}Fusion}\ \} \\
 & \quad \left[\begin{array}{ll}
 & F\ \mathsf{fst} \circ \tau \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \\
 = & \quad \{\ \mathsf{Tau\text{-}Cancel}\ \} \\
 & \mathsf{fst} \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \\
 = & \quad \{\ \mathsf{Prod\text{-}Def},\ \mathsf{Prod\text{-}Cancel}\ \} \\
 & \mathsf{out}_{\mu F} \circ \mathsf{fst}
 \end{array}\right. \\
 & [\![\mathsf{in}_{\mu F}, \mathsf{out}_{\mu F}]\!]_{\mu F} \circ \mathsf{fst} \\
= & \quad \{\ \mathsf{Hylo\text{-}Reflex}\ \} \\
 & \mathsf{fst}
\end{array}
$$

The following can also be easily proved.

$$\{\!|g, \tau|\!\}_{\mu F} \circ (\mathsf{in}_{\mu F} \times \mathsf{id}) = g \circ (F\ \{\!|g, \tau|\!\}_{\mu F} \circ \tau \vartriangle \mathsf{snd}) \qquad \mathsf{Accum\text{-}Cancel}$$

$$f \circ \{\!|g, \tau|\!\}_{\mu F} = \{\!|h, \tau|\!\}_{\mu F} \quad \Leftarrow \quad f \circ g = h \circ (F\ f \times \mathsf{id}) \land f \text{ strict} \qquad \mathsf{Accum\text{-}Fusion}$$

As expected, these laws can also be proved using the definition Accum-Cata. For example, Accum-Cancel can be proved as follows.

$$
\begin{array}{ll}
 & \{\!|g, \tau|\!\} \circ (\mathsf{in} \times \mathsf{id}) \\
= & \quad \{\ \mathsf{Accum\text{-}Cata},\ \mathsf{Prod\text{-}Functor\text{-}Comp}\ \} \\
 & \mathsf{ap} \circ ((\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!) \circ \mathsf{in} \times \mathsf{id}) \\
= & \quad \{\ \mathsf{Cata\text{-}Cancel},\ \mathsf{Prod\text{-}Functor\text{-}Comp}\ \} \\
 & \mathsf{ap} \circ (\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})} \times \mathsf{id}) \circ (F\ (\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!) \times \mathsf{id}) \\
= & \quad \{\ \mathsf{Exp\text{-}Cancel}\ \} \\
 & g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd}) \circ (F\ (\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!) \times \mathsf{id}) \\
= & \quad \{\ \mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Cancel}\ \} \\
 & g \circ (F\ \mathsf{ap} \circ \tau \circ (F\ (\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!) \times \mathsf{id}) \vartriangle \mathsf{snd}) \\
= & \quad \{\ \mathsf{Tau\text{-}Nat}\ \} \\
 & g \circ (F\ \mathsf{ap} \circ F\ ((\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!) \times \mathsf{id}) \circ \tau \vartriangle \mathsf{snd}) \\
= & \quad \{\ \mathsf{Functor\text{-}Comp},\ \mathsf{Accum\text{-}Cata}\ \} \\
 & g \circ (F\ \{\!|g, \tau|\!\} \circ \tau \vartriangle \mathsf{snd})
\end{array}
$$

For this recursion pattern it is more interesting to study left-strictness, because the recursive parameter is the left element of the input pair. The following result holds.

$$\{\!|g, \tau|\!\}_{\mu F}, \tau \text{ left-strict} \quad \Leftrightarrow \quad g, \tau \text{ left-strict} \qquad \mathsf{Accum\text{-}Strict}$$

Since this law is needed to prove uniqueness, the following proof is based on definition Accum-Cata.

$$\begin{array}{ll}
 & g \text{ left-strict} \\
\Rightarrow & \quad \{\, \mathsf{Lstrict\text{-}Def} \,\} \\
 & g \circ (\underline{\bot} \times \mathsf{id}) = \underline{\bot} \circ \mathsf{fst} \\
\Rightarrow & \quad \{\, \mathsf{Ap\text{-}Lstrict},\ \mathsf{Lstrict\text{-}Strict} \,\} \\
 & g \circ (F\ \mathsf{ap} \circ \underline{\bot} \times \mathsf{id}) = \underline{\bot} \circ \mathsf{fst} \\
\Rightarrow & \quad \{\, \mathsf{Prod\text{-}Def},\ \tau \text{ left-strict} \,\} \\
 & g \circ (F\ \mathsf{ap} \circ \tau \circ (\underline{\bot} \times \mathsf{id}) \vartriangle \mathsf{snd}) = \underline{\bot} \circ \mathsf{fst} \\
\Rightarrow & \quad \{\, \mathsf{Prod\text{-}Cancel},\ \mathsf{Prod\text{-}Fusion} \,\} \\
 & g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd}) \circ (\underline{\bot} \times \mathsf{id}) = \underline{\bot} \circ \mathsf{fst} \\
\Rightarrow & \quad \{\, \mathsf{Lstrict\text{-}Def},\ \mathsf{Exp\text{-}Strict} \,\} \\
 & \overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})} \text{ strict} \\
\Rightarrow & \quad \{\, \mathsf{Cata\text{-}Strict} \,\} \\
 & (\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!) \text{ strict} \\
\Rightarrow & \quad \{\, \mathsf{Lstrict\text{-}Comp\text{-}Left},\ \mathsf{Ap\text{-}Lstrict} \,\} \\
 & \mathsf{ap} \circ ((\!|\overline{g \circ (F\ \mathsf{ap} \circ \tau \vartriangle \mathsf{snd})}|\!) \times \mathsf{id}) \text{ left-strict} \\
\Rightarrow & \quad \{\, \mathsf{Accum\text{-}Cata} \,\} \\
 & \{\!|g, \tau|\!\} \text{ left-strict}
\end{array}$$

$$\begin{array}{ll}
 & \{\!|g, \tau|\!\} \text{ left-strict} \\
\Rightarrow & \quad \{\, \mathsf{In\text{-}Strict},\ \mathsf{Lstrict\text{-}Comp\text{-}Left} \,\} \\
 & \{\!|g, \tau|\!\} \text{ left-strict} \circ (\mathsf{in} \times \mathsf{id}) \\
\Rightarrow & \quad \{\, \mathsf{Accum\text{-}Cancel} \,\} \\
 & g \circ (F\ \{\!|g, \tau|\!\} \circ \tau \vartriangle \mathsf{snd}) \text{ left-strict} \\
\Rightarrow & \quad \{\, \mathsf{Lstrict\text{-}Def},\ \mathsf{Prod\text{-}Fusion} \,\} \\
 & g \circ (F\ \{\!|g, \tau|\!\} \circ \tau \circ (\underline{\bot} \times \mathsf{id}) \vartriangle \mathsf{snd} \circ (\underline{\bot} \times \mathsf{id})) = \underline{\bot} \circ \mathsf{fst} \\
\Rightarrow & \quad \{\, \mathsf{Prod\text{-}Cancel},\ \tau \text{ left-strict} \,\} \\
 & g \circ (F\ \{\!|g, \tau|\!\} \circ \underline{\bot} \circ \mathsf{fst} \vartriangle \mathsf{snd}) = \underline{\bot} \circ \mathsf{fst} \\
\Rightarrow & \quad \{\, \mathsf{Prod\text{-}Def},\ \mathsf{Prod\text{-}Functor\text{-}Comp} \,\} \\
 & g \circ (F\ \{\!|g, \tau|\!\} \times \mathsf{id}) \circ (\underline{\bot} \times \mathsf{id}) = \underline{\bot} \circ \mathsf{fst} \\
\Rightarrow & \quad \{\, \mathsf{Lstrict\text{-}Def} \,\} \\
 & g \circ (F\ \{\!|g, \tau|\!\} \times \mathsf{id}) \text{ left-strict} \\
\Rightarrow & \quad \{\, \mathsf{Comp\text{-}Lstrict} \,\} \\
 & g \text{ left-strict}
\end{array}$$

In practice, verifying these strictness conditions is easy. As seen in the examples, most parameters of accumulations are of the form $(\cdot \triangledown \cdot) \circ \mathsf{distl}$. By definition any either is strict, and distl is a left-strict function, which by Lstrict-Comp-Right leads necessarily to a left-strict parameter.

The uniqueness law for accumulations is specified as follows.

$$\begin{array}{c}
f = \{\!|g, \tau|\!\}_{\mu F} \ \wedge\ g, \tau \text{ left-strict} \\
\Leftrightarrow \\
f \circ (\mathsf{in}_{\mu F} \times \mathsf{id}) = g \circ (F f \circ \tau \vartriangle \mathsf{snd}) \ \wedge\ f, \tau \text{ left-strict}
\end{array} \qquad \mathsf{Accum\text{-}Uniq}$$

Similarly to paramorphisms, this law can be proved using the definition Accum-Cata. However, since accumulations defined using Accum-Def also satisfy Accum-Cancel, both definitions are equivalent provided that both g and τ are left-strict. The implication $\Rightarrow$ and the strictness side conditions are equivalent, respectively, to Accum-Cancel and Accum-Strict, already proved using that definition. For the $\Leftarrow$ implication the uniqueness law of catamorphisms is used, as shown in the following calculation.

$$
\begin{array}{ll}
 & f \\
= & \{\ \mathsf{Exp\text{-}Cancel}\ \} \\
 & \mathsf{ap} \circ (\overline{f} \times \mathsf{id}) \\
= & \{\ \mathsf{Cata\text{-}Uniq},\ \mathsf{Exp\text{-}Strict},\ f \text{ left-strict}\ \} \\
 & \left[\begin{array}{ll}
 & \overline{f} \circ \mathsf{in}_{\mu F} \\
 = & \{\ \mathsf{Exp\text{-}Fusion}\ \} \\
 & \overline{f \circ (\mathsf{in}_{\mu F} \times \mathsf{id})} \\
 = & \{\ f \circ (\mathsf{in}_{\mu F} \times \mathsf{id}) = g \circ (F\ f \circ \tau \triangle \mathsf{snd})\ \} \\
 & \overline{g \circ (F\ f \circ \tau \triangle \mathsf{snd})} \\
 = & \{\ \mathsf{Functor\text{-}Id},\ \mathsf{Exp\text{-}Cancel},\ \mathsf{Functor\text{-}Comp}\ \} \\
 & \overline{g \circ (F\ \mathsf{ap} \circ F\ (\overline{f} \times \mathsf{id}) \circ \tau \triangle \mathsf{snd})} \\
 = & \{\ \mathsf{Tau\text{-}Nat}\ \} \\
 & \overline{g \circ (F\ \mathsf{ap} \circ \tau \circ (F\ \overline{f} \times \mathsf{id}) \triangle \mathsf{snd})} \\
 = & \{\ \mathsf{Prod\text{-}Cancel},\ \mathsf{Prod\text{-}Def}\ \} \\
 & \overline{g \circ (F\ \mathsf{ap} \circ \tau \circ (F\ \overline{f} \times \mathsf{id}) \triangle \mathsf{snd} \circ (F\ \overline{f} \times \mathsf{id}))} \\
 = & \{\ \mathsf{Prod\text{-}Fusion},\ \mathsf{Exp\text{-}Fusion}\ \} \\
 & \overline{g \circ (F\ \mathsf{ap} \circ \tau \triangle \mathsf{snd})} \circ F\ \overline{f}
 \end{array}\right. \\
 & \mathsf{ap} \circ (\llparenthesis \overline{g \circ (F\ \mathsf{ap} \circ \tau \triangle \mathsf{snd})} \rrparenthesis \times \mathsf{id}) \\
= & \{\ \mathsf{Accum\text{-}Cata}\ \} \\
 & \{\!| g, \tau |\!\}
\end{array}
$$

Besides Accum-Fusion, Pardo presents other specific fusion laws for this recursion pattern. For example, a change in the accumulating parameter, by means of a function $f : X \to Y$, can be fused in the following way.

$$
\{\!| g, \tau |\!\}_{\mu F} \circ (\mathsf{id} \times f) = \{\!| g \circ (\mathsf{id} \times f), \phi |\!\}_{\mu F} \quad \Leftarrow \quad \tau \circ (\mathsf{id} \times f) = F\ (\mathsf{id} \times f) \circ \phi
$$

The proof uses the definition Accum-Def.

$$
\begin{array}{ll}
 & \{\!| g, \tau |\!\}_{\mu F} \circ (\mathsf{id} \times f) \\
= & \{\ \mathsf{Accum\text{-}Def}\ \} \\
 & \llbracket g, (\tau \triangle \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \rrbracket_{\mu(F \hat{\times} \underline{Y})} \circ (\mathsf{id} \times f) \\
= & \{\ \mathsf{Hylo\text{-}Fusion}\ \} \\
 & \left[\begin{array}{ll}
 & (\tau \triangle \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \circ (\mathsf{id} \times f) \\
 = & \{\ \mathsf{Prod\text{-}Functor\text{-}Comp}\ \} \\
 & (\tau \triangle \mathsf{snd}) \circ (\mathsf{id} \times f) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \\
 = & \{\ \mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Def},\ \mathsf{Prod\text{-}Cancel}\ \} \\
 & (\tau \circ (\mathsf{id} \times f) \triangle f \circ \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \\
 = & \{\ \tau \circ (\mathsf{id} \times f) = F\ (\mathsf{id} \times f) \circ \phi\ \} \\
 & (F\ (\mathsf{id} \times f) \circ \phi \triangle f \circ \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \\
 = & \{\ \mathsf{Prod\text{-}Absor},\ \mathsf{Prod\text{-}Functor\text{-}Comp}\ \} \\
 & (F\ (\mathsf{id} \times f) \times \mathsf{id}) \circ (\mathsf{id} \times f) \circ (\phi \triangle \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id})
 \end{array}\right. \\
 & \llbracket g, (\mathsf{id} \times f) \circ (\phi \triangle \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \rrbracket_{\mu(F \hat{\times} \underline{Y})} \\
= & \{\ \mathsf{Hylo\text{-}Shift},\ \mathsf{id} \times f : F \hat{\times} \underline{X} \dot{\to} F \hat{\times} \underline{Y}\ \} \\
 & \llbracket g \circ (\mathsf{id} \times f), (\phi \triangle \mathsf{snd}) \circ (\mathsf{out}_{\mu F} \times \mathsf{id}) \rrbracket_{\mu(F \hat{\times} \underline{X})} \\
= & \{\ \mathsf{Accum\text{-}Def}\ \} \\
 & \{\!| g \circ (\mathsf{id} \times f), \phi |\!\}_{\mu F}
\end{array}
$$

3.6 Summary

In this chapter we have shown how to define some typical recursion patterns using hylomorphisms. This approach has several advantages:

- It avoid proofs by fixpoint induction – all the usual properties of the recursion operators, including uniqueness, can be proved either using just a small set of laws about hylomorphisms, or previously derived laws about other recursion operators.
- It increases the understanding of the recursion patterns – for the more advanced operators, the definition using hylomorphisms is more understandable than the original one using other recursion patterns. This is the case of paramorphisms and accumulations, where the original definition using catamorphisms can be difficult to understand.
- It also systematizes the proof of the strictness side conditions that characterize initial recursion patterns in **CPO** – these can also be derived by calculation from simple laws concerning the strictness of hylomorphisms and of the basic combinators. To our knowledge, this is also the first time that the strictness side conditions associated with generic accumulations are precisely stated.

The same technique can be applied to other recursion patterns. For example, *histomorphisms* and *generalized catamorphisms* can also be defined using hylomorphisms. The histomorphism recursion pattern (together with its dual – *futumorphisms*) was first presented in [UV99]. It generalizes catamorphisms by allowing the result to depend not only on the recursive result of applying the function to the immediate children of the input, but also on the recursive result of any subterm. For example, it can be used to define the Fibonacci function that, for a given n, depends on the value of applying it to $n-1$ and $n-2$. The *generalized catamorphisms* is a "meta recursion pattern", first presented in [UVP01], that is parameterized by a comonad that encodes the recursive call pattern of a specific recursion pattern. By changing the comonad, it can be used to define catamorphisms, paramorphisms, or histomorphisms.

Chapter 4

Calculating Accumulations Using Fusion

The main goal of this chapter is to revisit some classic work in the area of program transformation using pure point-free calculations. The chapter is focused on the transformation of programs by introducing new accumulating parameters, according to the strategy initially proposed by Bird [Bir84], where the transformed programs are seen as higher-order folds calculated systematically from a specification. 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 can be used as shortcut optimization rules in an automatic program transformation system.

Another goal is the improvement of the machinery that is used to perform point-free calculations in a higher-order setting. Quoting Jeremy Gibbons [Gib94],

> *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 reduce the burden of proofs just to the creative parts.

4.1 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 this example will be used to briefly review different transformation techniques for optimizing programs by introducing accumulating parameters. As seen in Section 3.5, the resulting functions are called accumulations. However, the accumulations derived here are quite different from the ones presented in that section, since they are in the curried form – the input data-structure is traversed using a catamorphism, that outputs a function to be later applied to the accumulating parameter.

4.1.1 Transformation With Fold/Unfold Rules

Consider the typical definition of reverse in Haskell.

```
reverse :: List a -> List a
reverse Nil        = Nil
reverse (Cons h t) = cat (reverse t, Cons h Nil)
```

In order to apply a fold/unfold rule-based transformation [BD77, PP96] we will start with an equivalent definition using conditionals. The abbreviation $\mathsf{wrap}\ h = \mathsf{cons}\ (h, \mathsf{nil})$ will also be used. To simplify the presentation, when working in the pointwise style nil will denote the empty list (the exact definition is $\mathsf{nil}\ \bot_1$).

$$\begin{aligned} \mathsf{reverse}\ l &= \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ \mathsf{nil} \\ &\quad\ \mathsf{else}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ l), \mathsf{wrap}\ (\mathsf{head}\ l))) \end{aligned}$$

The transformation steps should be oriented by the so-called *forced folding* (or *need-for-folding*) principle [Dar81], 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. We can try to apply this strategy directly to the above definition.

$$\begin{aligned}
& \mathsf{reverse}\ l \\
= & \quad \{\text{definition of } \mathsf{reverse}\} \\
& \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ \mathsf{nil} \\
& \mathsf{else}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ l), \mathsf{wrap}\ (\mathsf{head}\ l))) \\
= & \quad \{\text{unfold}\} \\
& \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ \mathsf{nil} \\
& \mathsf{else}\ (\mathsf{cat}\ (\mathsf{if}\ (\mathsf{isnil}\ (\mathsf{tail}\ l))\ \mathsf{then}\ \mathsf{nil} \\
& \qquad\qquad \mathsf{else}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ (\mathsf{tail}\ l)), \mathsf{wrap}\ (\mathsf{head}\ (\mathsf{tail}\ l)))), \mathsf{wrap}\ (\mathsf{head}\ l))) \\
= & \quad \{\text{distributing of } \mathsf{cat} \text{ over the conditional}\} \\
& \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ \mathsf{nil} \\
& \mathsf{else}\ (\mathsf{if}\ (\mathsf{isnil}\ (\mathsf{tail}\ l))\ \mathsf{then}\ (\mathsf{cat}\ (\mathsf{nil}, \mathsf{wrap}\ (\mathsf{head}\ l))) \\
& \qquad \mathsf{else}\ (\mathsf{cat}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ (\mathsf{tail}\ l)), \mathsf{wrap}\ (\mathsf{head}\ (\mathsf{tail}\ l))), \mathsf{wrap}\ (\mathsf{head}\ l))) \\
= & \quad \{\text{associativity of } \mathsf{cat}\} \\
& \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ \mathsf{nil} \\
& \mathsf{else}\ (\mathsf{if}\ (\mathsf{isnil}\ (\mathsf{tail}\ l))\ \mathsf{then}\ (\mathsf{cat}\ (\mathsf{nil}, \mathsf{wrap}\ (\mathsf{head}\ l))) \\
& \qquad \mathsf{else}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ (\mathsf{tail}\ l)), \mathsf{cat}(\mathsf{wrap}\ (\mathsf{head}\ (\mathsf{tail}\ l)), \mathsf{wrap}\ (\mathsf{head}\ l)))))
\end{aligned}$$

At this point one would like to be able to fold the expression using the definition of reverse; however, the presence of the expression $\mathsf{wrap}\ (\mathsf{head}\ l)$ in both cases of the conditional prevents this step. We must appeal to the *generalization strategy* [BD77], according to which a new function definition is introduced.

$$\mathsf{aux}\ (l, y) = \mathsf{cat}\ (\mathsf{reverse}\ l, y)$$

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

$$\begin{array}{ll}
 & \mathsf{aux}\ (l, y) \\
= & \quad \{\text{definition of } \mathsf{aux}\} \\
 & \mathsf{cat}\ (\mathsf{reverse}\ l, y) \\
= & \quad \{\text{unfold}\} \\
 & \mathsf{cat}\ (\mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ \mathsf{nil} \\
 & \qquad \mathsf{else}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ l), \mathsf{wrap}\ (\mathsf{head}\ l))), y) \\
= & \quad \{\text{distributing } \mathsf{cat} \text{ over the conditional}\} \\
 & \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ (\mathsf{cat}\ (\mathsf{nil}, y)) \\
 & \mathsf{else}\ (\mathsf{cat}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ l), \mathsf{wrap}\ (\mathsf{head}\ l))), y) \\
= & \quad \{\text{associativity of } \mathsf{cat}\} \\
 & \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ (\mathsf{cat}\ (\mathsf{nil}, y)) \\
 & \mathsf{else}\ (\mathsf{cat}\ (\mathsf{reverse}\ (\mathsf{tail}\ l), \mathsf{cat}\ (\mathsf{wrap}\ (\mathsf{head}\ l), y))) \\
= & \quad \{\text{fold}\} \\
 & \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ (\mathsf{cat}\ (\mathsf{nil}, y)) \\
 & \mathsf{else}\ (\mathsf{aux}\ (\mathsf{tail}\ l, \mathsf{cat}\ (\mathsf{wrap}\ (\mathsf{head}\ l), y)))
\end{array}$$

The definition of cat can be used to simplify the resulting expression, yielding the definition

$$\mathsf{aux}\ (l, y) = \mathsf{if}\ (\mathsf{isnil}\ l)\ \mathsf{then}\ y\ \mathsf{else}\ (\mathsf{aux}\ (\mathsf{tail}\ l, \mathsf{cons}\ (\mathsf{head}\ l, y)))$$

Finally, since cat has a right-identity we have

$$\begin{array}{ll}
 & \mathsf{reverse}\ l \\
= & \quad \{\text{right-identity of } \mathsf{cat}\} \\
 & \mathsf{cat}\ (\mathsf{reverse}\ l, \mathsf{nil}) \\
= & \quad \{\text{definition of } \mathsf{aux}\} \\
 & \mathsf{aux}\ (l, \mathsf{nil})
\end{array}$$

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

```haskell
reverse :: List a -> List a
reverse l = aux (l, Nil)

aux :: (List a, List a) -> List a
aux (Nil, y)      = y
aux (Cons h t, y) = aux (t, Cons h 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 4.2.3 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 4.2.1 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 change the asymptotic execution time; however, it still is a useful transformation since it produces a tail-recursive definition.

4.1.2 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 [Bir84]. In this seminal paper, Bird introduced the fundamental idea behind this transformation method: 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 then 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 technique was later used by several authors [MFP91, HIT99, dMS99, SdM03]. However, none of these works presents these calculations in pure point-free style. Moreover, some of them don't use the generic fusion law for catamorphisms presented in Section 3.1. Instead, they use the pointwise specialization of this law for particular data types, such as the following one for the curried foldr operator defined on lists (with type $(A \to B \to B) \to B \to \mathsf{List}\ A \to B$).

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

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 chapter the name f_t will be used to denote the accumulation obtained by transforming f.

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

Following the approach just described, in order to obtain the desired accumulation the concatenation operator must be fused with the reverse function, using the above law. For that, it is necessary to redefine reverse using the foldr operator.

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

Dropping the accumulating parameter from our specification we get

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

This is a suitable expression to apply fusion, with f instantiated to $\overline{\mathsf{cat}}$, the curried version of the concatenation operator. Notice that, since cat is left-strict, $\overline{\mathsf{cat}}$ is a strict function due to Exp-Strict. In order to apply fusion there still remains two premises to verify, which will in turn allow us to determine c and h.

$$\overline{\mathsf{cat}}\ \mathsf{nil} = c$$
$$\lambda x r.\ \overline{\mathsf{cat}}\ (\overline{\mathsf{cat}}\ r\ (\mathsf{wrap}\ x)) = \lambda x r.\ h\ x\ (\overline{\mathsf{cat}}\ r)$$

Notice the use of λ-abstraction to encode universal quantification.

Since nil is also a left-identity of concatenation, then $c = \mathsf{id}$. In order to determine h the calculation proceeds as follows.

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

It is now fairly 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 get the curried version of the aux function calculated in the previous section. It should be invoked as `reverse l = reverse_t l Nil`.

```
reverse_t  :: List a -> List a -> List a
reverse_t Nil y        = y
reverse_t (Cons h t) y = reverse_t t (Cons h 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, any experienced functional programmer should have no problem in writing them 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 chapter, to define laws that combine in a single *shortcut* step whole sequences of transformation steps.

4.1.3 Transformation in the Point-free Style

The third method, which will be used extensively in the remaining of the chapter, differs from the one presented above in that all the calculations are done in point-free. Before this can be done, the initial specification needs to be written in the point-free style. The reverse function can be defined as

$$\begin{array}{lcl} \mathsf{reverse} & : & \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{reverse} & = & (\!|\mathsf{nil} \triangledown \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})|\!)_{\mathsf{List}\ A} \end{array}$$

and the specification 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 $\mathsf{reverse}_t$, which is a catamorphism $(\!|h|\!)$, we must find a function h such that

$$\overline{\mathsf{cat}} \circ (\mathsf{nil} \triangledown \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})) = h \circ \mathsf{F}_{\mathsf{List}}\ \overline{\mathsf{cat}} = h \circ (\mathsf{id} + \mathsf{id} \times \overline{\mathsf{cat}})$$

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 the 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 which will be fused is in curried form. As such, it is desirable to introduce new combinators particularly tailored to express properties in a higher-order setting. For the particular case of associativity, it suffices to introduce an uncurried composition operator.

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

In the point-free style it can be defined using the exponential combinators.

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

To show that this point-free definition implements the behavior specified in pointwise, the following fact can be proved.

$$\mathsf{comp} \circ (\underline{f} \mathbin{\triangle} \underline{g}) = \underline{f \circ g} \qquad \textsf{Comp-Pnt}$$

$$\begin{array}{ll}
 & \mathsf{comp} \circ (\underline{f} \mathbin{\triangle} \underline{g}) \\
= & \quad \{\, \textsf{Comp-Def},\ \textsf{Exp-Fusion} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr} \circ ((\underline{f} \mathbin{\triangle} \underline{g}) \times \mathsf{id})} \\
= & \quad \{\, \textsf{Prod-Absor},\ \textsf{Prod-Functor-Comp},\ \textsf{Assocr-Nat} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ (\underline{f} \times (\underline{g} \times \mathsf{id})) \circ \mathsf{assocr} \circ ((\mathsf{id} \mathbin{\triangle} \mathsf{id}) \times \mathsf{id})} \\
= & \quad \{\, \textsf{Prod-Functor-Comp},\ \textsf{Pnt-Def} \,\} \\
 & \overline{\mathsf{ap} \circ (\overline{f \circ \mathsf{snd}} \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{ap} \circ (\overline{g \circ \mathsf{snd}} \times \mathsf{id})) \circ \mathsf{assocr} \circ ((\mathsf{id} \mathbin{\triangle} \mathsf{id}) \times \mathsf{id})} \\
= & \quad \{\, \textsf{Exp-Cancel} \,\} \\
 & \overline{f \circ \mathsf{snd} \circ (\mathsf{id} \times g \circ \mathsf{snd}) \circ \mathsf{assocr} \circ ((\mathsf{id} \mathbin{\triangle} \mathsf{id}) \times \mathsf{id})} \\
= & \quad \{\, \textsf{Prod-Def},\ \textsf{Prod-Cancel},\ \textsf{Assocr-Def} \,\} \\
 & \overline{f \circ g \circ \mathsf{snd} \circ \mathsf{snd} \circ (\mathsf{fst} \circ \mathsf{fst} \mathbin{\triangle} (\mathsf{snd} \times \mathsf{id})) \circ ((\mathsf{id} \mathbin{\triangle} \mathsf{id}) \times \mathsf{id})} \\
= & \quad \{\, \textsf{Prod-Cancel},\ \textsf{Prod-Functor-Comp} \,\} \\
 & \overline{f \circ g \circ \mathsf{snd} \circ (\mathsf{snd} \circ (\mathsf{id} \mathbin{\triangle} \mathsf{id}) \times \mathsf{id})} \\
= & \quad \{\, \textsf{Prod-Def},\ \textsf{Prod-Cancel} \,\} \\
 & \overline{f \circ g \circ \mathsf{snd}} \\
= & \quad \{\, \textsf{Pnt-Def} \,\} \\
 & \underline{f \circ g}
\end{array}$$

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

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

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

$$\left[\begin{array}{ll}
@ & \oplus \circ (\mathsf{id} \times \oplus) \circ \mathsf{assocr} = \oplus \circ (\oplus \times \mathsf{id}) \\
 & \\
 & \overline{\oplus} \circ \oplus \\
= & \quad \{\, \text{Exp-Fusion} \,\} \\
 & \overline{\oplus \circ (\oplus \times \mathsf{id})} \\
= & \quad \{\, @ \,\} \\
 & \overline{\oplus \circ (\mathsf{id} \times \oplus) \circ \mathsf{assocr}} \\
= & \quad \{\, \text{Exp-Cancel, Prod-Functor-Comp} \,\} \\
 & \overline{\mathsf{ap} \circ (\overline{\oplus} \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{ap}) \circ (\mathsf{id} \times (\overline{\oplus} \times \mathsf{id})) \circ \mathsf{assocr}} \\
= & \quad \{\, \text{Prod-Functor-Comp} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ (\overline{\oplus} \times (\overline{\oplus} \times \mathsf{id})) \circ \mathsf{assocr}} \\
= & \quad \{\, \text{Assocr-Nat} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr} \circ ((\overline{\oplus} \times \overline{\oplus}) \times \mathsf{id})} \\
= & \quad \{\, \text{Exp-Fusion} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr}} \circ (\overline{\oplus} \times \overline{\oplus}) \\
= & \quad \{\, \text{Comp-Def} \,\} \\
 & \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus})
\end{array}\right.$$

Equipped with this formulation of associativity, calculating the accumulation becomes very simple.

$$\left[\begin{array}{ll}
@ & \overline{\mathsf{cat}} \circ \mathsf{cat} = \mathsf{comp} \circ (\overline{\mathsf{cat}} \times \overline{\mathsf{cat}}) \\
\dagger & \overline{\mathsf{cat}} \circ \mathsf{nil} = \underline{\mathsf{id}} \\
\ddagger & \overline{\mathsf{cat}} \circ \mathsf{wrap} = \overline{\mathsf{cons}} \\
 & \\
 & \overline{\mathsf{cat}} \circ (\mathsf{nil} \triangledown \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})) \\
= & \quad \{\, \text{Sum-Fusion} \,\} \\
 & \overline{\mathsf{cat}} \circ \mathsf{nil} \triangledown \overline{\mathsf{cat}} \circ \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\
= & \quad \{\, \dagger \,\} \\
 & \underline{\mathsf{id}} \triangledown \overline{\mathsf{cat}} \circ \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\
= & \quad \{\, @ \,\} \\
 & \underline{\mathsf{id}} \triangledown \mathsf{comp} \circ (\overline{\mathsf{cat}} \times \overline{\mathsf{cat}}) \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\
= & \quad \{\, \text{Swap-Nat, Prod-Functor-Comp} \,\} \\
 & \underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cat}} \circ \mathsf{wrap} \times \overline{\mathsf{cat}}) \\
= & \quad \{\, \ddagger, \text{Prod-Functor-Comp} \,\} \\
 & \underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id}) \circ (\mathsf{id} \times \overline{\mathsf{cat}}) \\
= & \quad \{\, \text{Sum-Absor} \,\} \\
 & (\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})) \circ (\mathsf{id} + \mathsf{id} \times \overline{\mathsf{cat}})
\end{array}\right.$$

The result of the transformation is thus

$$\begin{array}{lcl}
\mathsf{reverse}_t & : & \mathsf{List}\ A \to (\mathsf{List}\ A \to \mathsf{List}\ A) \\
\mathsf{reverse}_t & = & (\!|\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})|\!)_{\mathsf{List}\ A}
\end{array}$$

To see that this is the expected point-free definition of $\mathsf{reverse}_t$, it can be converted back to pointwise with the following calculation.

$$
\begin{array}{ll}
 & \mathsf{reverse}_t = (\!|\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})|\!)_{\mathsf{List}\ A} \\
= & \quad \{\ \mathsf{Cata\text{-}Cancel}\ \} \\
 & \mathsf{reverse}_t \circ \mathsf{in}_{\mathsf{List}} = (\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})) \circ \mathsf{F}_{\mathsf{List}}\ \mathsf{reverse}_t \\
= & \quad \{\ \text{definitions of } \mathsf{F}_{\mathsf{List}} \text{ and } \mathsf{in}_{\mathsf{List}}\ \} \\
 & \mathsf{reverse}_t \circ (\mathsf{nil} \triangledown \mathsf{cons}) = (\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id})) \circ (\mathsf{id} + \mathsf{id} \times \mathsf{reverse}_t) \\
= & \quad \{\ \mathsf{Sum\text{-}Fusion},\ \mathsf{Sum\text{-}Absor},\ \mathsf{Prod\text{-}Functor\text{-}Comp}\ \} \\
 & \mathsf{reverse}_t \circ \mathsf{nil} \triangledown \mathsf{reverse}_t \circ \mathsf{cons} = \underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{reverse}_t) \\
= & \quad \{\ \mathsf{Sum\text{-}Equal}\ \} \\
 & \mathsf{reverse}_t \circ \mathsf{nil} = \underline{\mathsf{id}} \wedge \mathsf{reverse}_t \circ \mathsf{cons} = \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{reverse}_t) \\
= & \quad \{\ \eta\text{-expansion, definitions of the basic combinators}\ \} \\
 & \mathsf{reverse}_t\ \mathsf{nil} = \mathsf{id} \wedge \mathsf{reverse}_t\ (\mathsf{cons}\ (x, xs)) = (\mathsf{reverse}_t\ xs) \circ (\overline{\mathsf{cons}}\ x) \\
= & \quad \{\ \eta\text{-expansion, definitions of the basic combinators}\ \} \\
 & \mathsf{reverse}_t\ \mathsf{nil}\ y = \mathsf{id}\ y \wedge \mathsf{reverse}_t\ (\mathsf{cons}\ (x, xs))\ y = \mathsf{reverse}_t\ xs\ (\mathsf{cons}\ (x, y))
\end{array}
$$

Remark. In this example, the solution to the premises of the Cata-Fusion law is not unique, as the following calculation shows.

$$
\begin{array}{ll}
\dagger & \overline{\mathsf{cat}} \circ \mathsf{nil} = \underline{\mathsf{id}} \\
 & \\
 & \overline{\mathsf{cat}} \circ (\mathsf{nil} \triangledown \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})) \\
= & \quad \{\ \mathsf{Sum\text{-}Fusion}\ \} \\
 & \overline{\mathsf{cat}} \circ \mathsf{nil} \triangledown \overline{\mathsf{cat}} \circ \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\
= & \quad \{\ \dagger\ \} \\
 & \underline{\mathsf{id}} \triangledown \overline{\mathsf{cat}} \circ \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\
= & \quad \{\ \mathsf{Exp\text{-}Cancel}\ \} \\
 & \underline{\mathsf{id}} \triangledown \overline{\mathsf{cat}} \circ \mathsf{ap} \circ (\overline{\mathsf{cat}}\ \times \mathsf{id}) \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\
= & \quad \{\ \mathsf{Swap\text{-}Nat},\ \mathsf{Prod\text{-}Functor\text{-}Comp}\ \} \\
 & \underline{\mathsf{id}} \triangledown \overline{\mathsf{cat}} \circ \mathsf{ap} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \overline{\mathsf{cat}}) \\
= & \quad \{\ \mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Sum\text{-}Absor}\ \} \\
 & (\underline{\mathsf{id}} \triangledown \overline{\mathsf{cat}} \circ \mathsf{ap} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})) \circ (\mathsf{id} + \mathsf{id} \times \overline{\mathsf{cat}})
\end{array}
$$

This leads to the following definition of the accumulation.

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

After converting it into pointwise we would get.

```
reverse_t :: List a -> List a -> List a
reverse_t Nil y        = y
reverse_t (Cons h t) y = cat (reverse_t t (wrap h), 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.

4.2 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 seen in the last remark, 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), and thus could be used as *shortcut* optimization rules in an automatic transformation system. The application of these rules is demonstrated in a substantial number of examples.

We start by presenting a transformation scheme that encapsulates the methodology for deriving accumulations in the calculational style using fusion. Among others, it is presented also in [BdM97, HIT99], but is adapted here to the **CPO** setting. Given a binary operator $\oplus : A \times B \to B$ it is defined as follows.

$$\begin{array}{c} (\!|f|\!)_{\mu F}\ x = (\!|g|\!)_{\mu F}\ x\ e \\ \Leftarrow \\ \overline{\oplus} \circ f = g \circ F\ \overline{\oplus}\ \wedge\ \oplus\ (x, e) = x\ \wedge\ \oplus \text{ left-strict} \end{array} \qquad \textsf{Cata-Accum}$$

This scheme is too general to be useful, but will later be instantiated to more concrete rules, to be applied in specific data types. In order to prove Cata-Accum, it should first to converted into the point-free notation. The conclusion can be specified as

$$(\!|f|\!) = \mathsf{ap} \circ ((\!|g|\!) \vartriangle \underline{e} \circ !)$$

Given that $\oplus$ has a right-identity e if $\oplus \circ (\mathsf{id} \vartriangle \underline{e} \circ !) = \mathsf{id}$, the calculation is rather trivial.

$$\begin{array}{ll}
\dagger & \oplus \circ (\mathsf{id} \vartriangle \underline{e} \circ !) = \mathsf{id} \\
\ddagger & \overline{\oplus} \circ f = g \circ F\ \overline{\oplus} \\
 & \\
 & (\!|f|\!) \\
= & \quad \{\ \dagger\ \} \\
 & \oplus \circ (\mathsf{id} \vartriangle \underline{e} \circ !) \circ (\!|f|\!) \\
= & \quad \{\ \textsf{Prod-Fusion},\ \textsf{Bang-Fusion}\ \} \\
 & \oplus \circ ((\!|f|\!) \vartriangle \underline{e} \circ !) \\
= & \quad \{\ \textsf{Exp-Cancel}\ \} \\
 & \mathsf{ap} \circ (\overline{\oplus} \times \mathsf{id}) \circ ((\!|f|\!) \vartriangle \underline{e} \circ !) \\
= & \quad \{\ \textsf{Prod-Absor}\ \} \\
 & \mathsf{ap} \circ (\overline{\oplus} \circ (\!|f|\!) \vartriangle \underline{e} \circ !) \\
= & \quad \{\ \textsf{Cata-Fusion},\ \ddagger,\ \textsf{Exp-Strict},\ \oplus \text{ left-strict}\ \} \\
 & \mathsf{ap} \circ ((\!|g|\!) \vartriangle \underline{e} \circ !)
\end{array}$$

4.2.1 Tail-recursive Accumulations over Lists

Associative Operators. We start with the most classic example of applying the accumulation strategy: to optimize the iteration of an associative operator over a list. Given a left-strict associative operator $\oplus : B \times B \to B$ with right identity e, an element $c : 1 \to B$, and a function $f : A \to B$, it is possible to transform a function

$$\begin{array}{lcl} h & : & \mathsf{List}\ A \to B \\ h & = & (\!| c \triangledown \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id}) |\!)_{\mathsf{List}\ A} \end{array}$$

into

$$\begin{array}{lcl} h_t & : & \mathsf{List}\ A \to B \to B \\ h_t & = & (\!| \overline{\oplus} \circ c \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \mathsf{id}) |\!)_{\mathsf{List}\ A} \end{array}$$

and replace every $h\ l$ by $h_t\ l\ e$.

This transformation is a consequence of Cata-Accum and the following calculation.

$$\left[\begin{array}{ll} @ & \overline{\oplus} \circ \oplus = \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \\ & \\ & \overline{\oplus} \circ (c \triangledown \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id})) \\ = & \quad \{ \mathsf{Sum\text{-}Fusion},\ \oplus \text{ left-strict},\ \mathsf{Exp\text{-}Strict} \} \\ & \overline{\oplus} \circ c \triangledown \overline{\oplus} \circ \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id}) \\ = & \quad \{ @ \} \\ & \overline{\oplus} \circ c \triangledown \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \circ \mathsf{swap} \circ (f \times \mathsf{id}) \\ = & \quad \{ \mathsf{Swap\text{-}Nat} \} \\ & \overline{\oplus} \circ c \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \times \overline{\oplus}) \circ (f \times \mathsf{id}) \\ = & \quad \{ \mathsf{Prod\text{-}Functor\text{-}Comp} \} \\ & \overline{\oplus} \circ c \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \overline{\oplus}) \\ = & \quad \{ \mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Sum\text{-}Absor} \} \\ & (\overline{\oplus} \circ c \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \mathsf{id})) \circ (\mathsf{id} + \mathsf{id} \times \overline{\oplus}) \end{array}\right.$$

In order to see what is going on, both functions can be translated into the pointwise style. It becomes clear that this transformation rule allows to convert 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

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

Example 4.1 (Reverse). It is immediate to see that the reverse function of Section 4.1.3 can be transformed using this rule, with the expected result.

Example 4.2 (Product). Suppose we want to derive a tail-recursive implementation of the following function that multiplies all the numbers in a list.

```
product :: List Nat -> Nat
product Nil        = Succ Zero
product (Cons h t) = mult (h, product t)
```

A straightforward implementation of this function in the point-free style is

$$\begin{aligned} \mathsf{product} &: \mathsf{List\ Nat} \to \mathsf{Nat} \\ \mathsf{product} &= (\!|\mathsf{one} \triangledown \mathsf{mult}|\!)_{\mathsf{List\ Nat}} \end{aligned}$$

The above transformation cannot be applied directly. However, since mult is a commutative operator and given Prod-Functor-Id, the above definition is equivalent to the following.

$$\mathsf{product} = (\!|\mathsf{one} \triangledown \mathsf{mult} \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{id})|\!)_{\mathsf{List\ Nat}}$$

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

$$\begin{aligned} \mathsf{product}_t &: \mathsf{List\ Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{product}_t &= (\!|\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{mult}} \times \mathsf{id})|\!)_{\mathsf{List\ Nat}} \end{aligned}$$

The final tail-recursive implementation in Haskell is

```
product :: List Nat -> Nat
product l = product_t l (Succ Zero)

product_t :: List Nat -> Nat -> Nat
product_t Nil y        = y
product_t (Cons h t) y = product_t t (mult (h,y))
```

This example shows that it is not always immediate to apply the transformation rule to catamorphisms with associative operators. For many operators (namely for commutative ones) the redefinition of the initial specification is trivial. The next example presents a situation where this is not so obvious.

Example 4.3 (Insertion Sort). Consider the insertion sort defined in Example 3.3

$$\begin{aligned} \mathsf{isort} &: \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{isort} &= (\!|\mathsf{nil} \triangledown \mathsf{insert}|\!)_{\mathsf{List}\ A} \end{aligned}$$

where the type of insert is $A \times \mathsf{List}\ A \to \mathsf{List}\ A$. This is clearly not an associative operator. In order to apply the transformation scheme, the definition has to be considerably modified. First, 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.

$$\mathsf{isort} = (\!|\mathsf{nil} \triangledown \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})|\!)_{\mathsf{List}\ A}$$

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

$$\begin{array}{lcl} \mathsf{isort}_t & : & \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{isort}_t & = & (\!|\mathsf{id} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{insert}} \times \mathsf{id})|\!)_{\mathsf{List}\ A} \end{array}$$

This means that insertion sort can be implemented as a tail-recursive definition.

```
isort :: (Ord a) => List a -> List a
isort l = isort_t l Nil

isort_t :: (Ord a) => List a -> List a -> List a
isort_t Nil y        = y
isort_t (Cons h t) y = isort_t t (insert h y)
```

This example illustrates that the application of a transformation may require the introduction of new functions (as in the generalization strategy). These may eventually be eliminated after the shortcut is applied.

Operators With Associative Duals. 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 $\odot$ [BW82] if

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

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

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

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 $c : 1 \to B$, and a function $f : A \to C$, then a function

$$\begin{array}{lcl} h & : & \mathsf{List}\ A \to B \\ h & = & (\!|c \triangledown \oplus \circ \mathsf{swap} \circ (f \times \mathsf{id})|\!) \end{array}$$

can be transformed into

$$\begin{array}{lcl} h_t & : & \mathsf{List}\ A \to C \to B \\ h_t & = & (\!|\overline{\oplus} \circ c \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\odot} \circ f \times \mathsf{id})|\!) \end{array}$$

by replacing every occurrence of $h\ l$ by $h_t\ l\ e$. The proof of this transformation rule is similar to the previous one, because the associative dual law has a similar formulation to associativity.

Example 4.4 (Tree Sorts). Consider again the following definition of insertion sort, taken from Example 4.3.

$$\mathsf{isort} = (\!|\mathsf{nil} \triangledown \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id})|\!)_{\mathsf{List}\ A}$$

There is no reason which prevents an accumulator of a different type to be used, such as, a binary search tree. Instead of using a particular type of tree, lets assume the existence of an abstract type $\mathsf{Tree}\ A$, that comes equipped with the following functions.

$$\mathsf{treeToList} : \mathsf{Tree}\ A \to \mathsf{List}\ A$$

that produces the sorted list of the elements stored in the tree,

$$\mathsf{mkTree} : A \to \mathsf{Tree}\ A$$

that generates a tree with a single element, and

$$\mathsf{mergeTree} : \mathsf{Tree}\ A \times \mathsf{Tree}\ A \to \mathsf{Tree}\ A$$

that merges two trees. The function that inserts an element in a tree can be defined as follows.

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

In order to use trees as accumulators, we now simply try to rewrite the definition of isort in such a way that wrap can be replaced by mkTree. The following calculation uses as hypothesis a property that we can reasonably expect to be verified by an implementation of trees.

$$\left[\begin{array}{ll} \dagger & \mathsf{treeToList} \circ \mathsf{mkTree} = \mathsf{wrap} \\ & \\ & \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}) \\ = & \quad \{\,\dagger\,\} \\ & \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{treeToList} \circ \mathsf{mkTree} \times \mathsf{id}) \\ = & \quad \{\,\text{Prod-Functor-Comp}\,\} \\ & \mathsf{merge} \circ \mathsf{swap} \circ (\mathsf{treeToList} \times \mathsf{id}) \circ (\mathsf{mkTree} \times \mathsf{id}) \\ = & \quad \{\,\text{Swap-Nat}\,\} \\ & \mathsf{merge} \circ (\mathsf{id} \times \mathsf{treeToList}) \circ \mathsf{swap} \circ (\mathsf{mkTree} \times \mathsf{id}) \end{array}\right.$$

By defining $\oplus = \mathsf{merge} \circ (\mathsf{id} \times \mathsf{treeToList})$ the sorting function can be defined as

$$\mathsf{sort} = (\!|\mathsf{nil} \triangledown \oplus \circ \mathsf{swap} \circ (\mathsf{mkTree} \times \mathsf{id})|\!)_{\mathsf{List}\ A}$$

Moreover, the reader will have no difficulty in accepting that the intended implementation of trees should be such that $\mathsf{mergeTree}$ 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 for applying the transformation are now verified, yielding the tail-recursive function

$$\begin{array}{lcl} \mathsf{sort}_t & : & \mathsf{List}\ A \to \mathsf{Tree}\ A \to \mathsf{List}\ A \\ \mathsf{sort}_t & = & (\!|\overline{\oplus} \circ \mathsf{nil} \bigtriangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{mergeTree}} \circ \mathsf{mkTree} \times \mathsf{id})|\!)_{\mathsf{List}\ A} \end{array}$$

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

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

The pointwise implementation of this accumulation in Haskell is

```
sort_t :: (Ord a) => List a -> Tree a -> List a
sort_t Nil y        = treeToList y
sort_t (Cons h t) y = sort_t t (insertTree h y)
```

Possible implementations of trees include ordinary binary search trees (with the obvious ordered insertion operation and $\mathsf{treeToList}$ implemented by an inorder traversal) and leaf trees (with $\mathsf{treeToList}$ 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.

4.2.2 Other Accumulations over Lists

The above transformation scheme can be further generalized to allow for transformations that, while still based on compositions with associative operators (or having an associative dual), do not result in tail-recursive functions.

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 $c : 1 \to B$, and a function $f : A \to C$, it is possible to transform a function

$$\begin{array}{lcl} h & : & \mathsf{List}\ A \to B \\ h & = & (\!|c \bigtriangledown \oplus \circ \mathsf{swap} \circ (f \times g)|\!) \end{array}$$

into

$$\begin{array}{lcl} h_t & : & \mathsf{List}\ A \to C \to B \\ h_t & = & (\!|\overline{\oplus} \circ c \bigtriangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\odot} \circ f \times k)|\!) \end{array}$$

and replace $h\ l$ by $h_t\ l\ e$ if g and k are functions such that $\overline{\oplus} \circ g = k \circ \overline{\oplus}$. Again, the proof is omitted because it is similar to the previous ones.

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 [HIT99]) that computes the initial sums of a list.

```
isums :: List Nat -> List Nat
isums Nil        = Nil
isums (Cons h t) = map ((curry plus) h) (Cons Zero (isums t))
```

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

$$\begin{array}{rcl} \oplus & : & \mathsf{List\ Nat} \times \mathsf{Nat} \to \mathsf{List\ Nat} \\ \oplus\ (l, h) & = & \mathsf{List}\ (\overline{\mathsf{plus}}\ h)\ l \end{array}$$

Instead of applying fusion directly, the above transformation rule will be applied. First, isums is defined in the point-free style as a catamorphism using the operator $\oplus$.

$$\begin{array}{rcl} \mathsf{isums} & : & \mathsf{List\ Nat} \to \mathsf{List\ Nat} \\ \mathsf{isums} & = & (\!|\mathsf{nil} \nabla \oplus \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{cons} \circ (\mathsf{zero} \circ ! \vartriangle \mathsf{id}))|\!)_{\mathsf{List\ Nat}} \end{array}$$

It is also necessary to 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.

$$\begin{array}{c} (l \oplus x) \oplus y = \mathsf{List}\ (\overline{\mathsf{plus}}\ y)\ (\mathsf{List}\ (\overline{\mathsf{plus}}\ x)\ l) \\ = \\ \mathsf{List}\ (\overline{\mathsf{plus}}\ (\mathsf{plus}\ (x, y)))\ l = l \oplus (x \odot y) \end{array}$$

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 interesting part of the point-free proof. In order to apply the transformation a function k must be identified, such that

$$k \circ \overline{\oplus} = \overline{\oplus} \circ \mathsf{cons} \circ (\mathsf{zero} \circ ! \vartriangle \mathsf{id})$$

For that it is convenient to express the following fact about $\oplus$ in the point-free calculus.

$$(\mathsf{cons}\ (x,l)) \oplus y = \mathsf{cons}\ (\mathsf{plus}\ (x,y), l \oplus y)$$

The obvious choice is

$$\oplus \circ (\mathsf{cons} \times \mathsf{id}) = \mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))$$

However, likewise to associativity, a formulation of this property involving the curried version of the operator would simplify calculations. This implies internalizing the split combinator, which can be done in pointwise as $\mathsf{split}\ (f,g) = f \vartriangle g$, and in point-free by the following equation.

$$\begin{array}{lcl} \mathsf{split} & : & (B^A \times C^A) \rightarrow (B \times C)^A \\ \mathsf{split} & = & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))} \end{array} \qquad \mathsf{Split\text{-}Def}$$

In Appendix A some properties about this function are presented, namely its interaction with points, and the proof that it testifies the isomorphism $B^A \times C^A \cong (B \times C)^A$ with inverse $\mathsf{fst}^\bullet \vartriangle \mathsf{snd}^\bullet$. With this definition, the above property can be expressed by the equation

$$\overline{\oplus} \circ \mathsf{cons} = \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus})$$

as the following calculation shows.

$$\left[\begin{array}{ll}
\dagger & \oplus \circ (\mathsf{cons} \times \mathsf{id}) = \mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id})) \\
 & \\
 & \overline{\oplus} \circ \mathsf{cons} \\
= & \quad \{\, \mathsf{Exp\text{-}Fusion} \,\} \\
 & \overline{\oplus \circ (\mathsf{cons} \times \mathsf{id})} \\
= & \quad \{\, \dagger \,\} \\
 & \overline{\mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))} \\
= & \quad \{\, \mathsf{Exp\text{-}Cancel,\ Prod\text{-}Functor\text{-}Comp,\ Prod\text{-}Absor} \,\} \\
 & \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ ((\overline{\mathsf{plus}} \circ \mathsf{fst} \times \mathsf{id}) \vartriangle (\overline{\oplus} \circ \mathsf{snd} \times \mathsf{id}))} \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel,\ Prod\text{-}Def} \,\} \\
 & \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id}) \vartriangle (\mathsf{snd} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id}))} \\
= & \quad \{\, \mathsf{Prod\text{-}Functor\text{-}Comp,\ Prod\text{-}Fusion} \,\} \\
 & \overline{\mathsf{cons} \circ (\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id})) \circ ((\overline{\mathsf{plus}} \times \overline{\oplus}) \times \mathsf{id})} \\
= & \quad \{\, \mathsf{Exp\text{-}Fusion,\ Exp\text{-}Absor} \,\} \\
 & \mathsf{cons}^\bullet \circ \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))} \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}\right.$$

It is now very easy to show that $k = \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\underline{\mathsf{id}} \circ ! \vartriangle \mathsf{id})$.

$$\begin{array}{ll}
\dagger & \overline{\oplus} \circ \mathsf{cons} = \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \\
\S & \overline{\mathsf{plus}} \circ \mathsf{zero} = \underline{\mathsf{id}} \\
 & \\
 & \overline{\oplus} \circ \mathsf{cons} \circ (\mathsf{zero} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{\, \dagger \,\} \\
 & \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \circ (\mathsf{zero} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Prod-Absor} \,\} \\
 & \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \circ \mathsf{zero} \circ ! \vartriangle \overline{\oplus}) \\
= & \quad \{\, \S \,\} \\
 & \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\underline{\mathsf{id}} \circ ! \vartriangle \overline{\oplus}) \\
= & \quad \{\, \text{Bang-Fusion} \,\} \\
 & \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\underline{\mathsf{id}} \circ ! \circ \overline{\oplus} \vartriangle \overline{\oplus}) \\
= & \quad \{\, \text{Prod-Fusion} \,\} \\
 & \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\underline{\mathsf{id}} \circ ! \vartriangle \mathsf{id}) \circ \overline{\oplus}
\end{array}$$

Finally, the transformation can be applied in order to get the desired accumulation. Notice that $\overline{\oplus} \circ \mathsf{nil} = \underline{\mathsf{nil} \circ !}$.

$$\begin{array}{lcl}
\mathsf{isums}_t & : & \mathsf{List\ Nat} \to \mathsf{Nat} \to \mathsf{List\ Nat} \\
\mathsf{isums}_t & = & (\!|\, \underline{\mathsf{nil} \circ !} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{plus}} \times \mathsf{cons}^{\bullet} \circ \mathsf{split} \circ (\underline{\mathsf{id}} \circ ! \vartriangle \mathsf{id}))\,|\!)_{\mathsf{List\ Nat}}
\end{array}$$

After converting this definition to pointwise Haskell we get the following implementation.

```
isums :: List Nat -> List Nat
isums l = isums_t l Zero

isums_t :: List Nat -> Nat -> List Nat
isums_t Nil y        = Nil
isums_t (Cons h t) y = Cons (plus (h,y)) (isums_t t (plus (h,y)))
```

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.

4.2.3 Accumulations over Leaf-labeled 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 data type of leaf-labeled binary trees can be defined by the following fixed point.

$$\begin{array}{rcl}
\mathsf{LTree}\ A & = & \mu(\underline{A} \mathbin{\hat{+}} \mathsf{Id} \mathbin{\hat{\times}} \mathsf{Id}) \\
\mathsf{in}_{\mathsf{LTree}} & = & \mathsf{leaf} \triangledown \mathsf{branch}
\end{array}$$

In Haskell it can be defined as follows.

```
data LTree a = Leaf a | Branch (LTree a) (LTree a)
```

Given a left-strict associative operator $\oplus : B \times B \to B$ with right identity e, and a function $f : A \to B$, it is possible to transform a function

$$\begin{array}{lcl} h & : & \mathsf{LTree}\ A \to B \\ h & = & (\!|f \triangledown \oplus|\!) \end{array}$$

into

$$\begin{array}{lcl} h_t & : & \mathsf{LTree}\ A \to B \to B \\ h_t & = & (\!|\overline{\oplus} \circ f \triangledown \mathsf{comp}|\!) \end{array}$$

with the guarantee that $h\ t = h_t\ t\ e$.

This transformation rule is a direct consequence of Cata-Accum and the following calculation.

$$\left[\begin{array}{ll} @ & \overline{\oplus} \circ \oplus = \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \\ & \\ & \overline{\oplus} \circ (f \triangledown \oplus) \\ = & \quad \{\, \mathsf{Sum\text{-}Fusion},\ \oplus \text{ left-strict},\ \mathsf{Exp\text{-}Strict} \,\} \\ & \overline{\oplus} \circ f \triangledown \overline{\oplus} \circ \oplus \\ = & \quad \{\, @ \,\} \\ & \overline{\oplus} \circ f \triangledown \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \\ = & \quad \{\, \mathsf{Sum\text{-}Absor} \,\} \\ & (\overline{\oplus} \circ f \triangledown \mathsf{comp}) \circ (\mathsf{id} + \overline{\oplus} \times \overline{\oplus}) \end{array}\right.$$

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

```
leaves :: LTree a -> List a
leaves (Leaf x)     = wrap x
leaves (Branch l r) = cat (leaves l, leaves r)
```

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

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

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

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

The implementation of this optimized version with explicit recursion is

```
leaves :: LTree a -> List a
leaves t = leaves_t t Nil

leaves_t :: LTree a -> List a -> List a
leaves_t (Leaf x) y     = Cons x y
leaves_t (Branch l r) y = leaves_t l (leaves_t r y)
```

Remark. This example had already been presented by Bird and de Moor in [BdM97], using exactly the same point-free specification and result. However, their derivation is mainly done in pointwise because the comp combinator was defined using this style, preventing them to reason about associativity with pure point-free calculations.

4.2.4 Accumulations over Rose Trees

It is also possible to define rules for transforming folds over rose trees into accumulations. Rose trees were introduced in Section 3.1.1, and can be defined in Haskell as follows.

```
data Rose a = Forest a (List (Rose a))
```

Given a left-strict associative operator $\oplus : B \times B \to B$ with right identity e, an element $c : 1 \to B$, a function $f : A \to B$, then it is possible to transform

$$\begin{array}{rcl} h & : & \mathsf{Rose}\ A \to B \\ h & = & (\!| \oplus \circ \mathsf{swap} \circ (f \times (\!| c \triangledown \oplus |\!)_{\mathsf{List}\ B}) |\!)_{\mathsf{Rose}\ A} \end{array}$$

into

$$\begin{array}{rcl} h_t & : & \mathsf{Rose}\ A \to B \to B \\ h_t & = & (\!| \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times (\!| \overline{\oplus} \circ c \triangledown \mathsf{comp} |\!)_{\mathsf{List}\ (B \to B)}) |\!)_{\mathsf{Rose}\ A} \end{array}$$

such that $h\ t = h_t\ t\ e$.

This is a consequence of Cata-Accum and the following calculation.

$$
\left[
\begin{array}{ll}
@ & \overline{\oplus} \circ \oplus = \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \\
 & \\
 & \overline{\oplus} \circ \oplus \circ \mathsf{swap} \circ (f \times \llparenthesis c \triangledown \oplus \rrparenthesis) \\
= & \quad \{ @ \} \\
 & \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \circ \mathsf{swap} \circ (f \times \llparenthesis c \triangledown \oplus \rrparenthesis) \\
= & \quad \{ \text{Swap-Nat, Prod-Functor-Comp} \} \\
 & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \overline{\oplus} \circ \llparenthesis c \triangledown \oplus \rrparenthesis) \\
= & \quad \{ \text{Cata-Fusion}, \oplus \text{ left-strict, Exp-Strict} \} \\
 & \left[
 \begin{array}{ll}
  & \overline{\oplus} \circ (c \triangledown \oplus) \\
 = & \quad \{ \text{Sum-Fusion}, \oplus \text{ left-strict, Exp-Strict} \} \\
  & \overline{\oplus} \circ c \triangledown \overline{\oplus} \circ \oplus \\
 = & \quad \{ @ \} \\
  & \overline{\oplus} \circ c \triangledown \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus}) \\
 = & \quad \{ \text{Prod-Functor-Comp, Sum-Absor} \} \\
  & (\overline{\oplus} \circ c \triangledown \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 \llparenthesis \overline{\oplus} \circ c \triangledown \mathsf{comp} \circ (\overline{\oplus} \times \mathsf{id}) \rrparenthesis) \\
= & \quad \{ \text{Sum-Absor} \} \\
 & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \llparenthesis (\overline{\oplus} \circ c \triangledown \mathsf{comp}) \circ (\mathsf{id} + \overline{\oplus} \times \mathsf{id}) \rrparenthesis) \\
= & \quad \{ \text{Cata-Map-Fusion} \} \\
 & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \llparenthesis \overline{\oplus} \circ c \triangledown \mathsf{comp} \rrparenthesis \circ \mathsf{List}\ \overline{\oplus}) \\
= & \quad \{ \text{Prod-Functor-Comp} \} \\
 & \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\oplus} \circ f \times \llparenthesis \overline{\oplus} \circ c \triangledown \mathsf{comp} \rrparenthesis) \circ (\mathsf{id} \times \mathsf{List}\ \overline{\oplus})
\end{array}
\right.
$$

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

```
post :: Rose a -> List a
post (Forest x l) = cat (aux l, wrap x)
    where aux Nil        = Nil
          aux (Cons h t) = cat (post h, aux t)
```

Notice that aux is the flatten function defined in the Example 3.4. The traversal can be expressed in point-free style as follows.

$$
\begin{array}{lcl}
\mathsf{post} & : & \mathsf{Rose}\ A \to \mathsf{List}\ A \\
\mathsf{post} & = & \llparenthesis \mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \llparenthesis \mathsf{nil} \triangledown \mathsf{cat} \rrparenthesis_{\mathsf{List}\ (\mathsf{List}\ A)}) \rrparenthesis_{\mathsf{Rose}\ A}
\end{array}
$$

This definition can be transformed into the following linear time accumulation. This result was simplified assuming the usual facts about cat.

$$
\begin{array}{lcl}
\mathsf{post}_t & : & \mathsf{Rose}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A \\
\mathsf{post}_t & = & \llparenthesis \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \llparenthesis \underline{\mathsf{id}} \triangledown \mathsf{comp} \rrparenthesis_{\mathsf{List}\ (\mathsf{List}\ A \to \mathsf{List}\ A)}) \rrparenthesis_{\mathsf{Rose}\ A}
\end{array}
$$

The optimized version can be implemented in Haskell as follows.

```
post :: Rose a -> List a
post r = post_t r Nil
```

```
post_t :: Rose a -> List a -> List a
post_t (Forest x l) y = aux l (Cons x y)
    where aux Nil y        = y
          aux (Cons h t) y = post_t h (aux t y)
```

4.3 Functions with more than one accumulator

Recall the accumulation pattern presented in Section 4.2.3 for leaf-trees. 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 was 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 is also received 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 ${}^A f\ g = g \circ f$.

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

$$\begin{aligned} {}^A f &: A^C \to A^B \\ {}^A f &= \overline{\mathsf{ap} \circ (\mathsf{id} \times f)} \end{aligned} \qquad \text{Pxe-Def}$$

Likewise to the normal exponentiation, $\bullet$ will be used in superscript when the information about the type is not relevant. Points can be used to see that this definition implements the expected pointwise behavior.

$${}^\bullet f \circ \underline{g} = \underline{g \circ f} \qquad \text{Pxe-Pnt}$$

$$\begin{aligned}
& {}^\bullet f \circ \underline{g} \\
= & \quad \{\,\text{Pxe-Def, Pnt-Def}\,\} \\
& \overline{\mathsf{ap} \circ (\mathsf{id} \times f)} \circ \overline{g \circ \mathsf{snd}} \\
= & \quad \{\,\text{Exp-Fusion}\,\} \\
& \overline{\mathsf{ap} \circ (\mathsf{id} \times f) \circ (\overline{g \circ \mathsf{snd}} \times \mathsf{id})} \\
= & \quad \{\,\text{Prod-Functor-Comp}\,\} \\
& \overline{\mathsf{ap} \circ (\overline{g \circ \mathsf{snd}} \times \mathsf{id}) \circ (\mathsf{id} \times f)} \\
= & \quad \{\,\text{Exp-Cancel}\,\} \\
& \overline{g \circ \mathsf{snd} \circ (\mathsf{id} \times f)} \\
= & \quad \{\,\text{Prod-Def, Prod-Cancel}\,\} \\
& \overline{g \circ f \circ \mathsf{snd}} \\
= & \quad \{\,\text{Pnt-Def}\,\} \\
& \underline{g \circ f}
\end{aligned}$$

This operator determines a contravariant functor, i.e.

$$^{\bullet}\mathsf{id} = \mathsf{id} \qquad ^{\bullet}(f \circ g) = {^{\bullet}g} \circ {^{\bullet}f}$$

The following law relates it with the normal exponentiation.

$$f^{\bullet} \circ {^{\bullet}g} = {^{\bullet}g} \circ f^{\bullet} \tag*{Pxe-Exp}$$

$$\begin{array}{cl}
 & f^{\bullet} \circ {^{\bullet}g} \\
= & \quad \{\,\text{Pxe-Def}\,\} \\
 & f^{\bullet} \circ \overline{\mathsf{ap} \circ (\mathsf{id} \times g)} \\
= & \quad \{\,\text{Exp-Absor}\,\} \\
 & \overline{f \circ \mathsf{ap} \circ (\mathsf{id} \times g)} \\
= & \quad \{\,\text{Exp-Cancel}\,\} \\
 & \overline{\mathsf{ap} \circ (\overline{f \circ \mathsf{ap}} \times \mathsf{id}) \circ (\mathsf{id} \times g)} \\
= & \quad \{\,\text{Prod-Functor-Comp,Exp-Def}\,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times g) \circ (f^{\bullet} \times \mathsf{id})} \\
= & \quad \{\,\text{Exp-Fusion}\,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times g)} \circ f^{\bullet} \\
= & \quad \{\,\text{Pxe-Def}\,\} \\
 & {^{\bullet}g} \circ f^{\bullet}
\end{array}$$

Example 4.8 (Height). Consider the following straightforward implementation of the height function.

```
height :: LTree a -> Nat
height (Leaf x)     = Zero
height (Branch l r) = Succ (max (height l, height r))
```

This can be written as the following catamorphism, where $\mathsf{max} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ computes the maximum of two naturals.

$$\begin{array}{lcl}
\mathsf{height} & : & \mathsf{LTree}\ A \to \mathsf{Nat} \\
\mathsf{height} & = & (\!|\mathsf{zero} \circ ! \triangledown \mathsf{succ} \circ \mathsf{max}|\!)_{\mathsf{LTree}\ A}
\end{array}$$

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

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

The following calculation shows how to derive the point-free specification.

$$
\begin{array}{ll}
& \mathsf{height}_t\ t\ d\ m = \mathsf{max}\ (\mathsf{plus}\ (\mathsf{height}\ t, d), m) \\
\Leftrightarrow & \quad \{\text{definition of curry}\} \\
& \mathsf{height}_t\ t\ d\ m = \overline{\mathsf{max}}\ (\mathsf{plus}\ (\mathsf{height}\ t, d))\ m \\
\Leftrightarrow & \quad \{\eta\text{-reduction, definition of curry}\} \\
& \mathsf{height}_t\ t\ d = \overline{\mathsf{max}}\ (\overline{\mathsf{plus}}\ (\mathsf{height}\ t)\ d) \\
\Leftrightarrow & \quad \{\text{definition of composition}\} \\
& \mathsf{height}_t\ t\ d = (\overline{\mathsf{max}} \circ (\overline{\mathsf{plus}}\ (\mathsf{height}\ t)))\ d \\
\Leftrightarrow & \quad \{\eta\text{-reduction, definition of exponentiation}\} \\
& \mathsf{height}_t\ t = \overline{\mathsf{max}}^{\bullet}\ (\overline{\mathsf{plus}}\ (\mathsf{height}\ t)) \\
\Leftrightarrow & \quad \{\text{definition of composition}\} \\
& \mathsf{height}_t\ t = (\overline{\mathsf{max}}^{\bullet} \circ \overline{\mathsf{plus}} \circ \mathsf{height})\ t \\
\Leftrightarrow & \quad \{\eta\text{-reduction}\} \\
& \mathsf{height}_t = \overline{\mathsf{max}}^{\bullet} \circ \overline{\mathsf{plus}} \circ \mathsf{height}
\end{array}
$$

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

$$
\begin{aligned}
\mathsf{plus}\ (\mathsf{max}\ (x, y), z) &= \mathsf{max}\ (\mathsf{plus}\ (x, z), \mathsf{plus}\ (y, z)) \\
\mathsf{plus}\ (\mathsf{succ}\ x, y) &= \mathsf{plus}\ (x, \mathsf{succ}\ y)
\end{aligned}
$$

The first is similar to the one that motivated the introduction of the split combinator in Example 4.5, and can be written as follows.

$$\overline{\mathsf{plus}} \circ \mathsf{max} = \mathsf{max}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\mathsf{plus}})$$

For the second the new exponentiation combinator can be used.

$$\overline{\mathsf{plus}} \circ \mathsf{succ} = {}^{\bullet}\mathsf{succ} \circ \overline{\mathsf{plus}}$$

A simple calculation allows to obtain this as a consequence of a more conventional point-free specification.

$$
\begin{array}{ll}
\dagger & \mathsf{plus} \circ (\mathsf{succ} \times \mathsf{id}) = \mathsf{plus} \circ (\mathsf{id} \times \mathsf{succ}) \\
\\
& \overline{\mathsf{plus}} \circ \mathsf{succ} \\
= & \quad \{\mathsf{Exp\text{-}Fusion}\} \\
& \overline{\mathsf{plus} \circ (\mathsf{succ} \times \mathsf{id})} \\
= & \quad \{\dagger\} \\
& \overline{\mathsf{plus} \circ (\mathsf{id} \times \mathsf{succ})} \\
= & \quad \{\mathsf{Exp\text{-}Cancel},\ \mathsf{Prod\text{-}Functor\text{-}Comp}\} \\
& \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{succ}) \circ (\overline{\mathsf{plus}} \times \mathsf{id})} \\
= & \quad \{\mathsf{Exp\text{-}Fusion}, \mathsf{Pxe\text{-}Def}\} \\
& {}^{\bullet}\mathsf{succ} \circ \overline{\mathsf{plus}}
\end{array}
$$

For the fusion of $\overline{\mathsf{plus}}$ the calculation proceeds as follows. Notice that this operator is strict due to Exp-Strict and the left strictness of plus.

$$
\left[\begin{array}{ll}
\dagger & \overline{\mathsf{plus}} \circ \mathsf{succ} = {}^{\bullet}\mathsf{succ} \circ \overline{\mathsf{plus}} \\
\ddagger & \overline{\mathsf{plus}} \circ \mathsf{max} = \mathsf{max}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\mathsf{plus}}) \\
\S & \overline{\mathsf{plus}} \circ \mathsf{zero} = \underline{\mathsf{id}} \\
 & \\
 & \overline{\mathsf{plus}} \circ (\mathsf{zero} \circ ! \triangledown \mathsf{succ} \circ \mathsf{max}) \\
= & \quad \{ \mathsf{Sum\text{-}Fusion}, \overline{\mathsf{plus}} \text{ strict} \} \\
 & \overline{\mathsf{plus}} \circ \mathsf{zero} \circ ! \triangledown \overline{\mathsf{plus}} \circ \mathsf{succ} \circ \mathsf{max} \\
= & \quad \{ \S, \dagger \} \\
 & \underline{\mathsf{id}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \overline{\mathsf{plus}} \circ \mathsf{max} \\
= & \quad \{ \ddagger \} \\
 & \underline{\mathsf{id}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \mathsf{max}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\mathsf{plus}}) \\
= & \quad \{ \mathsf{Sum\text{-}Absor} \} \\
 & (\underline{\mathsf{id}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \mathsf{max}^{\bullet} \circ \mathsf{split}) \circ (\mathsf{id} + \overline{\mathsf{plus}} \times \overline{\mathsf{plus}})
\end{array}\right.
$$

The result of the first fusion is then

$$\mathsf{height}_t = \overline{\mathsf{max}^{\bullet}} \circ \llparenthesis \underline{\mathsf{id}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \mathsf{max}^{\bullet} \circ \mathsf{split} \rrparenthesis_{\mathsf{LTree}\ A}$$

The second derivation makes use of the associativity of max and some laws about the basic combinators, namely the following property relating the split function with exponentiation.

$$\mathsf{split} \circ (f^{\bullet} \times g^{\bullet}) = (f \times g)^{\bullet} \circ \mathsf{split} \qquad \mathsf{Split\text{-}Exp}$$

$$
\left[\begin{array}{ll}
 & \mathsf{split} \circ (f^{\bullet} \times g^{\bullet}) \\
= & \quad \{ \mathsf{Split\text{-}Def} \} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))} \circ (f^{\bullet} \times g^{\bullet}) \\
= & \quad \{ \mathsf{Exp\text{-}Fusion} \} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id})) \circ ((f^{\bullet} \times g^{\bullet}) \times \mathsf{id})} \\
= & \quad \{ \mathsf{Prod\text{-}Fusion}, \mathsf{Prod\text{-}Functor\text{-}Comp} \} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \circ (f^{\bullet} \times g^{\bullet}) \times \mathsf{id}) \vartriangle (\mathsf{snd} \circ (f^{\bullet} \times g^{\bullet}) \times \mathsf{id}))} \\
= & \quad \{ \mathsf{Prod\text{-}Def}, \mathsf{Prod\text{-}Cancel} \} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((f^{\bullet} \circ \mathsf{fst} \times \mathsf{id}) \vartriangle (g^{\bullet} \circ \mathsf{snd} \times \mathsf{id}))} \\
= & \quad \{ \mathsf{Prod\text{-}Functor\text{-}Comp}, \mathsf{Prod\text{-}Cancel}, \mathsf{Exp\text{-}Def} \} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((\overline{f \circ \mathsf{ap}} \times \mathsf{id}) \times (\overline{g \circ \mathsf{ap}} \times \mathsf{id})) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))} \\
= & \quad \{ \mathsf{Prod\text{-}Functor\text{-}Comp}, \mathsf{Exp\text{-}Cancel} \} \\
 & \overline{(f \times g) \circ (\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))} \\
= & \quad \{ \mathsf{Exp\text{-}Absor}, \mathsf{Split\text{-}Def} \} \\
 & (f \times g)^{\bullet} \circ \mathsf{split}
\end{array}\right.
$$

Given this property, the calculation proceeds as follows. Notice that $\overline{\mathsf{max}^{\bullet}}$ is strict due to the left-strictness of max, Exp-Strict, and the definition of the exponentiation operator.

$$
\begin{array}{ll}
@ & \overline{\mathsf{max}} \circ \mathsf{max} = \mathsf{comp} \circ (\overline{\mathsf{max}} \times \overline{\mathsf{max}}) \\
\\
 & \overline{\mathsf{max}}^{\bullet} \circ (\underline{\mathsf{id}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \mathsf{max}^{\bullet} \circ \mathsf{split}) \\
= & \quad \{ \text{Sum-Fusion}, \overline{\mathsf{max}}^{\bullet} \text{ strict} \} \\
 & \overline{\mathsf{max}}^{\bullet} \circ \underline{\mathsf{id}} \circ ! \triangledown \overline{\mathsf{max}}^{\bullet} \circ {}^{\bullet}\mathsf{succ} \circ \mathsf{max}^{\bullet} \circ \mathsf{split} \\
= & \quad \{ \text{Exp-Pnt, Pxe-Exp} \} \\
 & \underline{\overline{\mathsf{max}}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \overline{\mathsf{max}}^{\bullet} \circ \mathsf{max}^{\bullet} \circ \mathsf{split} \\
= & \quad \{ \text{Exp-Functor-Comp} \} \\
 & \underline{\overline{\mathsf{max}}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ (\overline{\mathsf{max}} \circ \mathsf{max})^{\bullet} \circ \mathsf{split} \\
= & \quad \{ @ \} \\
 & \underline{\overline{\mathsf{max}}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ (\mathsf{comp} \circ (\overline{\mathsf{max}} \times \overline{\mathsf{max}}))^{\bullet} \circ \mathsf{split} \\
= & \quad \{ \text{Exp-Functor-Comp} \} \\
 & \underline{\overline{\mathsf{max}}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \mathsf{comp}^{\bullet} \circ (\overline{\mathsf{max}} \times \overline{\mathsf{max}})^{\bullet} \circ \mathsf{split} \\
= & \quad \{ \text{Split-Exp} \} \\
 & \underline{\overline{\mathsf{max}}} \circ ! \triangledown {}^{\bullet}\mathsf{succ} \circ \mathsf{comp}^{\bullet} \circ \mathsf{split} \circ (\overline{\mathsf{max}}^{\bullet} \times \overline{\mathsf{max}}^{\bullet}) \\
= & \quad \{ \text{Sum-Absor} \} \\
 & (\underline{\overline{\mathsf{max}}} \circ ! \triangledown {}^{\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 the following accumulation.

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

After expanding the definitions of the combinators, the following implementation is obtained, where it is clear that one of the recursive calls has been made tail-recursive.

```
height :: LTree a -> Nat
height t = height_t t Zero Zero

height_t :: LTree a -> Nat -> Nat -> Nat
height_t (Leaf x) d m     = max (d, m)
height_t (Branch l r) d m = height_t l (Succ d) (height_t r (Succ d) m)
```

Remark. The notion of post composition already appeared in [Gib99] as a means to express some properties about higher-order functions. However, similarly to comp in [BdM97], it was defined in the pointwise style, preventing pure point-free calculations.

4.4 Transforming Hylomorphisms into Accumulations

The goal of this section is to show that the application of the technique presented in this chapter is not restricted to catamorphisms.

Example 4.9 (Factorial). To exemplify the application of the accumulation technique to a function that cannot be directly defined as a catamorphism, consider the definition of factorial. As seen in Example 2.1, it can be defined as the following hylomorphism.

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

Due to Hylo-Split this definition can be decomposed into

$$\mathsf{fact} = (\!|\mathsf{one} \triangledown \mathsf{mult}|\!) \circ [\!((\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}})\!]$$

Since the catamorphism is the product function defined in the Example 4.2, it can be transformed into the tail recursive function $\mathsf{product}_t$ by introducing an accumulating parameter. After applying Hylo-Split in the reverse order the following hylomorphism is obtained.

$$\begin{aligned} \mathsf{fact}_t &: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{fact}_t &= [\![\underline{\mathsf{id}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{mult}} \times \mathsf{id}), (\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}]\!]_{\mathsf{List\ Nat}} \end{aligned}$$

This definition is necessarily tail-recursive because the catamorphism, that encodes all computations done after recursion, is tail-recursive. In pointwise Haskell it corresponds to the following implementation.

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

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

4.5 Related Work

Hu, Iwasaki, and Takeichi have used a calculational approach to several program transformation techniques, including deforestation [HIT96], tupling [HITT97], and accumulations [HIT99]. In this latter work, the authors present a methodology for deriving accumulations using fusion, where the expected structure of the catamorphisms parameter is used in order to facilitate the derivation. However, 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.

Meijer, Fokkinga, and Paterson have introduced in [MFP91] a transformation rule 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. 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 transformation rule for lists and associative operators is strongly related to the *first duality theorem* [Bir98], 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 can be defined as follows.

```
foldl_List :: (b -> a -> b) -> b -> List a -> b
foldl_List f z Nil        = z
foldl_List f z (Cons h t) = foldl_List f (f z h) t
```

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

$$\mathsf{foldr}\ \oplus\ e = \mathsf{foldl}\ \oplus\ e$$

From this theorem and some properties about maps, it is possible to derive the following equivalent formulation of the transformation rule presented in Section 4.2.1.

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

Notice that swap is not used in this rule because the binary operator in foldl takes its parameters in the reverse order.

The work of Sheard and Fegaras on the derivation of accumulations [SF93] 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 respective higher-order folds. For the particular case of lists, the transformation is similar to the one defined in Section 4.2.1, 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 the rule presented in Section 4.2.3 for transformation of functions over leaf trees.

Accumulations are usually defined as higher-order catamorphisms, and that was also the approach followed in this chapter. However, using different recursion patterns it is possible to define some accumulations without going into the higher-order setting, with the advantage that some calculations may become simpler. 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 [Mal90], and later, Gibbons presented a generic definition that works for any regular data type [Gib00]. The generic accumulation recursion pattern presented in Section 3.5 further generalizes this notion: Pardo defined a general transformation rule similar to proposition Cata-Accum using this operator [Par03]. The problem with all these definitions of accumulations is that none is as expressive as the one used here – for example, it is not possible to define the accumulation of Example 4.6, where the value of the accumulator passed to one of the recursive calls depends on the result of another recursive call.

Finally, there is also some research work in program transformation with accumulations that is not concerned with their derivation from inefficient specifications, but rather with studying fusion of functions already defined as accumulations [HIT99, Gib00, Par03, Sve02].

4.6 Summary

In this chapter 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 calculations. 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, that exposed 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 applied to other transformation techniques, such as tupling.

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 a major simplification in the calculations. We have also introduced a new point-free exponentiation operator, equivalent to the right-sectioning of the composition combinator.

As shown in Example 4.3, the main limitation of our approach is that sometimes it is necessary to (non-trivially) change the initial definition of a program to enable the application of the transformation rules.

Remark. It turns out that the point-free definitions of both the uncurried composition operator and the new exponentiation have already been proposed by McLarty in an introductory book to category theory [McL95]. Our results were obtained independently, and it was only close to the end of writing this thesis that we came to the knowledge of this fact.

Chapter 5

Mechanizing Fusion

After the seminal work of Burstall and Darlington on program transformation using fold/unfold rules [BD77], Wadler proposed an algorithm that used some of their ideas to perform *deforestation* of functional programs [Wad88]. The core of this algorithm is a set of transformation rules that manipulate the programs, in search of potential locations where elimination of intermediate data structures can be performed. Naturally, some trickery has to be used in order to avoid infinite unfolding of recursive definitions. As seen in the last chapter, fusion rules can be used precisely to shortcut the fold/unfold cycle, thus avoiding the need to track such infinite unfoldings. With this insight in mind, a number of researchers tried to develop deforestation systems using shortcut fusion to reproduce Wadler's results. This chapter reviews some of these systems.

As presented in the first chapters, fusion rules come in two flavors: calculational fusion, expressed in rules such as Cata-Fusion, where in order to fuse f with $(\!|g|\!)$ one must guess h such that some equation is verified; and acid rain fusion, where, provided both the producer and the consumer are defined according to specific patterns, one can immediately produce the result without any extra guessing. The first style is usually called *cold fusion*, while the second one is called *warm fusion*, evoking the fact that it is somehow easier to apply.

5.1 Warm Fusion

Although all calculations performed in this thesis are based on cold fusion, for a long time warm fusion proved more effective for mechanical transformation. A restriction of the acid rain rule to lists, the so called foldr/build rule, was the starting point for the first successful deforestation system based on shortcut fusion, developed by Gill, Launchbury, and Peyton Jones [GLJ93]. Due to its effectiveness and simplicity it is still used today as an optimization in the *Glasgow Haskell Compiler* (GHC). The major drawback of shortcut fusion is the need to define functions using specific recursion patterns. In the case of acid rain fusion the problem is even worst, since the parameters of some of the recursion patterns must themselves be defined according to a specific pattern: less work in guessing the result is achieved at the cost of more work in structuring the input. For the foldr/build rule this problem was alleviated by Launchbury and Sheard, who developed a mechanism to transform most useful recursive definitions into the appropriate form [LS95].

Takano and Meijer used hylomorphisms to generalize and dualize the foldr/build rule to work on any regular data type, originating the acid rain theorem [TM95] presented in Section 2.4. Shortly after, Hu, Iwasaki, and Takeichi used this theorem to implement a more generic system to perform deforestation using warm fusion [HIT96]. In the context of this system, they developed one algorithm to derive hylomorphisms, to be presented in Section 7.4, and another to find the polymorphic function transformers that enable the application of the theorem. This system was later improved and baptized as the HYLO system [OHIT97].

5.2 Cold Fusion: First Steps

To our knowledge, the first successful attempt to develop a mechanism for performing cold fusion is due to Sheard and Fegaras [SF93]. Likewise to all systems presented so far, their algorithm also operates at the pointwise level. In order to give an overview of this work, we exemplify its use with the application to the fusion law for foldr, the curried pointwise instance of catamorphisms for lists presented in Section 4.1.2. Notice however that the method generalizes to folds over any regular data type (in fact, it also covers mutually recursive types). Recall the type of this recursion pattern

$$\mathsf{foldr} : (A \to B \to B) \to B \to \mathsf{List}\ A \to B$$

and the respective fusion law. As seen in that section, universal quantification can be replaced by lambda abstraction. The strictness side conditions will be ignored in this chapter because none of the systems handles them.

$$\begin{gathered} f\ (\mathsf{foldr}\ g\ e\ l) = \mathsf{foldr}\ h\ c\ l \\ \Leftarrow \\ c = f\ e \wedge \lambda xr.h\ x\ (f\ r) = \lambda xr.f\ (g\ x\ r) \end{gathered}$$

The problem of using this law as a transformation rule, that is, as a rewrite rule oriented from left to right, is that it contains variables in the right-hand side that are not present in the left-hand side. This implies that one has to devise a method for guessing their values, using the side condition to guide the process. Determining c is not a problem because the side condition says exactly what its value is. The real challenge is guessing h, since it appears applied to $f\ r$. If instead of $f\ r$, h was applied just to r, by η-simplification we would also immediately get its definition.

The solution proposed by Sheard and Fegaras is very simple and quite ingenious. First, the variables x and r are replaced in the abstraction of the side-condition by a couple of fresh variables, for example y and t. Then the bodies are transformed using the rewriting system that characterizes the λ-calculus in question, but temporarily augmented with two new rules, namely $x \leadsto y$, and $f\ r \leadsto t$. The left-hand side immediately rewrites to $\lambda yt\,.\,h\ y\ t$ which by η-simplification is equal to h. If using this augmented rewrite system all references to the old variables x and r are eliminated from $\lambda xr.f\ (g\ x\ r)$, the desired definition for h is achieved. Otherwise, the conditions cannot be verified, and fusion fails.

The algorithm used in [LS95] to derive a fold from a recursive definition is based on the clever idea of trying to fuse it with the identity fold, and uses the above rewriting technique to implement fusion. For example, the recursive definition of the length function verifies the following equations, that will be oriented from left to right as rewrite rules.

$$\begin{aligned}\mathsf{length}(\mathsf{nil}) &= \mathsf{zero}\\ \mathsf{length}(\overline{\mathsf{cons}}\ x\ t) &= \mathsf{succ}(\mathsf{length}\ t)\end{aligned}$$

In order to derive the fold that implements this definition fusion can be applied to the right-hand side of the following equation. Notice that $\mathsf{foldr}\ \overline{\mathsf{cons}}\ \mathsf{nil}$ is equal to the identity on lists.

$$\mathsf{length} = \lambda l.\mathsf{length}(\mathsf{foldr}\ \overline{\mathsf{cons}}\ \mathsf{nil}\ l)$$

The side condition of the fusion law immediately gives the value to return when the list is empty, namely $\mathsf{length}(\mathsf{nil})$ (zero after rewriting). To determine the function h, the body of the abstraction $\lambda yt\,.\,\mathsf{length}(\overline{\mathsf{cons}}\ x\ r)$ is transformed using the rewrite system augmented with the rules $x \leadsto y$ and $\mathsf{length}\ r \leadsto t$. The following rewrite sequence occurs.

$$\mathsf{length}(\overline{\mathsf{cons}}\ x\ r) \leadsto \mathsf{succ}(\mathsf{length}\ r) \leadsto \mathsf{succ}\ t$$

Since none of the original variables remains in the final term, fusion succeeds with the following result.

$$\mathsf{length} = \mathsf{foldr}\ (\lambda yt\,.\,\mathsf{succ}\ t)\ \mathsf{zero}$$

The main problem with this technique is that no completeness results are presented, and it is difficult to understand exactly the class of expressions on which it will succeed.

5.3 Higher-order Matching: the MAG System

Some of these ideas were later improved by Sittampalam and de Moor in MAG, a system for program transformation based on cold shortcut fusion [dMS99, SdM03]. The system is tailored to the Haskell language, and is powerful enough to automate in the pointwise setting, for example, all the calculations used in Chapter 4. It has also been used with success to implement optimizations using the tupling strategy.

This system is not fully automatic, but relies on the notion of *active source*: the original (inefficient) Haskell definitions are stored together with sufficient hints (namely, the specification that results from the generalization strategy, and the rewriting rules that capture the creative steps of the derivation) that enable the system to derive the efficient version. Basically MAG implements a term rewriting mechanism that, given a set of transformation rules, tries to apply them in the order in which they appear in the active source, repeating this process until no rule can be applied. For each rule it tries to find a matching subexpression by searching from left-to-right and from largest-to-smaller. MAG does not require the original functions to be defined directly as folds, and instead uses the technique of trying to fuse the original definition with the identity fold, as seen before.

In order to cope with the side conditions of the fusion rule, in particular the condition

$$\lambda xr.h\ x\ (f\ r) = \lambda xr.f\ (g\ x\ r)$$

one has to find a substitution for h such that both sides are equal. Since h is of functional type, this problem is generically known as *higher-order matching*, and is by itself a large research theme. To be more precise, the goal is to find a substitution for h such that both sides are equal after $\beta\eta$-normalization. Last section presented a trick that manages to reduce this particular instance of the problem to rewriting, but whose effectiveness is difficult to access. On the contrary, Sittampalam and de Moor designed a new mechanism to perform generic higher-order matching [dMS01], specifically tailored to the context of program transformation.

Formally, given a pattern P without η-redexes, and a closed normal form term M, we want to find a substitution ϕ such that

$$\mathsf{beta}(P\phi) \simeq M$$

where $\simeq$ represents equality modulo α-equivalence and η-reduction, and $\mathsf{beta}(t)$ denotes the β-normal form of t. The definition of beta for the case of application is

$$\mathsf{beta}(MN) = \begin{cases} \mathsf{beta}(L[\mathsf{beta}(N)/x]) & \text{if} \quad \mathsf{beta}(M) = \lambda x.L \\ (\mathsf{beta}(M))(\mathsf{beta}(N)) & \text{otherwise} \end{cases}$$

If such a substitution exists, it is called a *match*. The biggest problem with higher-order matching is that, in general, an infinite set of matches may exist. For example, matching the pattern $p\ x$ against a constant a leads to the following set of valid matches, where none is an instance of another.

$$\{[(\lambda y.a)/p], [(\lambda y.y)/p, a/x], [(\lambda g.g\ a)/p, (\lambda y.y)/x], [(\lambda g.g(g\ a))/p, (\lambda y.y)/x], \ldots\}$$

The first successful algorithm to solve this problem was defined by Huet and Lang [HL78], by restricting the set of possible matches to *second order* terms. Terms of a base type are of first order, and the order of a term of functional type is one plus the order of the argument. This guarantees that there exists only a finite number of incomparable matches. In the example above, only the first two substitutions fall under this category. Unfortunately, this restriction is not reasonable in the context of program transformation, namely when dealing with higher-order functions. For example, recall the classic example of deriving the efficient, accumulator-based, version of the length function. The parameter of the resulting fold is a function that takes another function as argument. To guess this parameter one must find a third order match, that would not be found by the algorithm of Huet and Lang.

The restriction imposed by de Moor and Sittampalam consists in replacing, in the specification of the problem given above, beta by step, a function that approximates β-normalization by applying β-reduction to all possible subterms that may occur in a single bottom-up traversal of the pattern. Its definition for the case of application is

$$\mathsf{step}(MN) = \begin{cases} L[\mathsf{step}(N)/x] & \text{if} \quad \mathsf{step}(M) = \lambda x.L \\ (\mathsf{step}(M))(\mathsf{step}(N)) & \text{otherwise} \end{cases}$$

The difference between the two functions is that beta is applied again to the result of the substitution, since new β-redexes may result from it.

With this restriction the algorithm always returns a finite set of matches that includes all second order matches, but possibly also some of order higher than two. In the example given above, it would behave similarly to Huet and Lang's algorithm and only return the first two matches, since $\mathsf{step}((\lambda g.g\ a)(\lambda y.y)) \neq a$, but in the derivation of the efficient length it would output the desired third order match.

Apart from efficiency, which is always a problem in higher-order matching (even its restriction to second order terms is known to be NP-hard), the main problem of this algorithm is that, although there is a formal characterization of the set of matches it returns, in practice it is difficult to understand why a particular match is or is not returned. The reason for this is the unintuitive behavior of the step function. On the contrary, with Huet and Lang's algorithm the class of second-order terms is defined in a precise and clear way.

Later, the same authors developed a slightly different version of the matching algorithm, tailored to find all third-order matches that occur in practical problems of program transformation, in particular all of those involving the introduction of accumulation parameters [SdM01]. As we have seen, this set is in general infinite, but by imposing some syntactic restrictions on the patterns it is possible to guarantee that the number of possible matches is finite. When these restrictions are not verified it is still possible to resort to the first version of the algorithm.

In practice, likewise to Bird's functional calculator, the main limitation of MAG is precisely its rewriting system: since no completion, or termination checking, is performed, the user must be careful about the order in which the transformation rules are stated in the active source. As the authors put it [SdM03],

> *... 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.*

A limitation of both systems that use cold fusion, when compared for example with the HYLO system, is that they only deal with folds, thus limiting their applicability.

5.4 Fusion in the Point-free Style

All these systems were developed for the pointwise setting. To our knowledge, the only attempt to implement short-cut fusion in the point-free setting is due to Ross Paterson [Pat], with a re-implementation of Bird's functional calculator [Bir98]. The latter system, to be presented in Section 9.1, is a very simple point-free calculator, that does not support conditional laws and, as such, cannot handle fusion.

Paterson extended Bird's calculator to handle conditional laws, and to give a distinguished treatment to identity and functors. If the user signals one of the constants as a functor, the respective equations Functor-Id and Functor-Comp will be automatically added as contractions to the system when performing simplifications. Although the system is largely undocumented, some reverse engineering revealed a very sophisticated pattern matching algorithm, that not only operates modulo associativity and identity, but also modulo the functor laws. In particular, it uses the functor laws as expansions rules when searching for a matching substitution. This implies using heuristics to avoid expansion loops and trivial matchings, and some form of backtracking in order to explore different expansion alternatives. For example, suppose one adds Distl-Nat as the rewrite rule

$$\mathsf{distl} \circ ((f + g) \times h) \rightsquigarrow (f \times h + g \times h) \circ \mathsf{distl}$$

then the system is able to produce the following reduction, where a and b are arbitrary functions.

$$\mathsf{distl} \circ (a \times b) \rightsquigarrow ((\mathsf{id} \times b) + (\mathsf{id} \times b)) \circ \mathsf{distl} \circ (a \times \mathsf{id})$$

In order to do so, the matching algorithm had to expand a to $(\mathsf{id} + \mathsf{id}) \circ a$ using Sum-Functor-Id, and then expand $((\mathsf{id}+\mathsf{id}) \circ a) \times b$ to $((\mathsf{id}+\mathsf{id}) \times b) \circ (a \times \mathsf{id})$ using Prod-Functor-Comp. It also had to discard trivial matchings, such as replacing all three variables by id after expanding $a \times b$ to $((\mathsf{id} + \mathsf{id}) \times \mathsf{id}) \circ (a \times b)$, since this yields a reduction that does not change the expression.

In order to apply a law like Cata-Fusion as the rewrite rule

$$f \circ (\!| g |\!)_{\mu F} \rightsquigarrow (\!| h |\!)_{\mu F} \quad \Leftarrow \quad f \circ g = h \circ F\ f$$

it must guess a value for h that satisfies the side condition. Given its sophisticated matching algorithm, for some cases it succeeds in doing so, which means that, in practice, it implements a point-free fusion algorithm.

We now give some examples of the abilities and limitations of this system concerning fusion. Since it does not support variables of functor type, all the laws have to be specialized to a specific type. The following examples all use lists. Consider the proof of the Cata-Map-Fusion law specialized to lists. After applying Cata-Fusion to the left-hand side of the equation

$$(\!| f |\!) \circ (\!| \mathsf{in}_{\mathsf{List}} \circ (\mathsf{id} + g \times \mathsf{id}) |\!) = (\!| h |\!)$$

and rewriting with Cata-Cancel and functor laws, we get the following side condition.

$$f \circ (\mathsf{id} + g \times (\!| f |\!)) = h \circ (\mathsf{id} + \mathsf{id} \times (\!| f |\!))$$

Due to its ability to use Functor-Comp as an expansion during matching, the system will temporarily expand the left-hand side to

$$f \circ (\mathsf{id} + g \times \mathsf{id}) \circ (\mathsf{id} + \mathsf{id} \times (\!| f |\!))$$

and trivially instantiate h with $f \circ (\mathsf{id} + g \times \mathsf{id})$, thus yielding the desired result. Unfortunately, since the only expansions performed concern the functor and identity laws, sometimes this system cannot handle some very simple examples. Consider the proof of the fact that the length of a list is not affected by mapping.

$$\mathsf{length} \circ (\!| \mathsf{in}_{\mathsf{List}} \circ (\mathsf{id} + g \times \mathsf{id}) |\!) = \mathsf{length}$$

length is defined as in Example 3.2, that is

$$\mathsf{length} = (\!| \mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{snd}) |\!)$$

After applying fusion, and rewriting the side condition, we get the following matching problem.

$$\mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{length} \circ \mathsf{snd}) = h \circ (\mathsf{id} + \mathsf{id} \times \mathsf{length})$$

In this case the matching algorithm of Paterson would fail because it does not temporarily use Prod-Cancel as an expansion rule, in order to rewrite $\mathsf{length} \circ \mathsf{snd}$ into $\mathsf{snd} \circ (\mathsf{id} \times \mathsf{length})$. Of course, the user is free to remove this law from the rewriting system, or adding it as the expansion $f \circ \mathsf{snd} \rightsquigarrow \mathsf{snd} \circ (\mathsf{id} \times f)$, but that would still not yield the desired result. Fusion would be applied with success, since the matching in the side condition would now be

$$\mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{snd} \circ (g \times \mathsf{length})) = h \circ (\mathsf{id} + \mathsf{id} \times \mathsf{length})$$

but then the system would not be able to prove the trivial fact that the resulting catamorphism is equal to length.

$$(\!|\mathsf{in}_{\mathsf{Nat}} \circ (\mathsf{id} + \mathsf{snd} \circ (g \times \mathsf{id}))|\!)$$

5.5 Summary

This chapter presented a brief review of fusion mechanization. We have seen that almost all systems implemented so far use the pointwise style. Some of them are very sophisticated: for example the MAG system can derive, given appropriate hints by the user, all efficient accumulations presented in Chapter 4. To our knowledge, the only system that implements fusion directly in the point-free style is due to Ross Paterson, but is rather ad-hoc and largely undocumented. None of the systems handles strictness side conditions.

Chapter 6

Pointless Haskell

This chapter describes a Haskell library for point-free programming with recursion patterns. Following a typical joke about point-free programming, the library is called *Pointless Haskell*. The implementation uses some non-standard extensions of the language, but enables the use of a syntax almost identical to the standard theoretical one presented in Chapter 2. This is achieved by introducing a restricted notion of structural equivalence between types. It also provides support for defining polytypic recursion operators, by improving the standard technique of defining data types explicitly as fixed points of functors. A polytypic (or generic) definition denotes a function that accepts a parameter of any regular data-type. It differs from a polymorphic one, because the behavior of the function is not uniform, but depends on the structure of the input's type. The library also includes a generic mechanism to visualize the intermediate data structures of hylomorphisms.

Pointless Haskell can be seen as a *domain specific language*, in the sense that it provides an [vDKV00]

> *[...] executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*

In fact, it is a domain specific *embedded* language [Hud96]. By embedding it in Haskell we avoid the need to develop a new compiler or interpreter, and inherit all the infrastructure of this language. There are many advantages of using Haskell as a host language [Ell99]. In our implementation, characteristics like higher-order features, parametric polymorphism, laziness, and type classes played an essential role. The main disadvantages are the need to sometimes compromise on the notation, the inability to provide good "domain specific error" messages, and some little discrepancies in the semantic domain.

All the Haskell code presented in this chapter was tested using version 6.2 of GHC. Whenever we mention Haskell with extensions we mean the set of extensions switched on by the compiler's flag `-fglasgow-exts`.

6.1 Implementing the Basic Concepts

Most of the primitive functions, basic combinators, and types presented in Chapter 2 are already part of the Haskell 98 standard prelude [Jon03]. For example, the fundamental categorical concepts of composition and identity are predefined as follows.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)

id :: a -> a
id x = x
```

The definition of the identity function stresses a subtle difference in the presentation of the basic constants of the point-free style. In Chapter 2, functions like id or fst were defined as families of monomorphic functions, indexed by the specific type they operate on. Since most of the times this type can be easily inferred from the context, the indexes are usually omitted. The Haskell type system is polymorphic. This means that a single definition can be given for these functions, with the advantage that the specific types are derived by the type inference mechanism.

The notion of functor is captured by a type constructor, and its action on arrows is encoded in a type class. The verification of the functor laws cannot be guaranteed by the language, and it is left to the programmer.

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

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

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

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

```
data One
```

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

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

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

Products. The main problem of using Haskell to host this library is that all its data types are by default pointed and lifted, that is, every type has a distinct bottom element. As noticed by several authors, this means that Haskell does not have true categorical products because $(\bot, \bot) \neq \bot$, neither true categorical exponentials because $\lambda x.\bot \neq \bot$. As discussed in [DJ04], this fact complicates equational reasoning because many standard laws about products and functions no longer hold.

Let us consider products for a moment. It is possible to write a Haskell function that distinguishes `_L` from `(_L,_L)`. For example, if the following function is applied to `(_L,_L)` it returns 0, but when applied to `_L` it returns `_L`.

```
f :: (a,b) -> Int
f (_,_) = 0
```

If *irrefutable matching* was used instead of regular pattern matching, this distinction would not be possible, because operationally no matching would be carried out until one of the variables in the pattern was used. Irrefutable matching is a standard Haskell feature activated by appending `~` to a pattern. The same applies to let expressions, where the matching is always irrefutable. In short, none of the following functions can distinguish `_L` from `(_L,_L)`.

```
g :: (a,b) -> Int
g ~(_,_) = 0

h :: (a,b) -> Int
h x = let (_,_) = x in 0
```

As John Hughes notices in [Hug03], by restricting pattern matching over products to be irrefutable and prohibiting the use of function `seq` (to be presented in Section 7.2), it is possible to "pretend" that these values are equal. In point-free programming the only way to inspect products is by using the destructors, that are predefined in Haskell as follows. Since they can also not distinguish `_L` from `(_L,_L)`, Haskell pairs can "safely" be used to model products.

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) =  y
```

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

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

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

To finalize the discussion about products, we would like to stress that, to our knowledge, even with extensions it is not possible to define a Haskell data type that correctly models the cartesian product. For example, neither *strictness annotations*, neither `newtype` solve the problem. A strictness annotation, with the flag `!`, next to an argument in a data type declaration forces its evaluation prior to the construction of values of that type. Consider the following declaration.

```
data Pair a b = Pair !a !b
```

This data type does not implement cartesian product, but *smashed product*, where, given any x, both $(\perp, x)$ and $(x, \perp)$ are identified with $\perp$. The `newtype` keyword allows us to define unlifted types. Unfortunately, it can only be used to "rename" an already defined type, and thus its single constructor can only receive a single type as argument. If syntactically possible, the following declaration would implement cartesian product.

```
newtype Pair a b = Pair a b
```

Sums. Unlike products, since sums are by definition lifted there is no problem in representing them by a Haskell data type. The predefined `Either` data type is used.

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

New aliases are defined for the constructors.

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

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

The either combinator is also predefined.

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)  = f x
either _ g (Right y) = g y
```

Together with a more convenient infix alias for `either`, the infix sum combinator is also defined.

```
infix 4 \/
(\/) :: (b -> a) -> (c -> a) -> Either b c -> a
(\/) = either

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

Exponentials. Although Haskell does not have true categorical exponentials, the Haskell functional type `->` will be used to model them. Similarly to products, this decision is harmless because the point-free combinators cannot distinguish between `_L` and `\_ -> _L`. Again, the curry combinator is predefined

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y =  f (x, y)
```

and the application combinator can be defined as follows.

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

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

Finally we have a guard combinator that operates on Haskell booleans. In order to simulate the postfix syntax it must be used in a left-sectioning.

```
(?) :: (a -> Bool) -> a -> Either a a
p ? x = if p x then inl x else inr x
```

Examples. Equipped with these combinators, some useful isomorphisms can be defined exactly as in Chapter 2.

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

assocr :: ((a,b),c) -> (a,(b,c))
assocr = fst . fst  /\  snd >< id

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

It is also possible to define in the point-free style an explicitly recursive function, such as the length function.

```
length :: [a] -> Int
length = (zero . bang \/ succ . length . tail) . (null?)
```

6.2 Programming with Explicit Functors

At least since [MH95], it is known how to implement generic versions of the recursion patterns in Haskell by defining data types explicitly as fixed points of functors. The implementation follows directly from the theoretical concepts presented in Section 2.3. This style of programming was used to implement generic recursion patterns by several authors [UVP01, Gib02], and is also followed, with some improvements, in our library.

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

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

The context of the definition restricts the application of `Mu` to members of the `Functor` class. The use of `newtype` guarantees the strictness of `Mu`, and thus enforces the isomorphism between `Mu f` and `f (Mu f)`. `inn` and `out` are aliases to the constructor and the destructor of `Mu`.

```
inn :: Functor f => f (Mu f) -> Mu f
inn = Mu

out :: Functor f => Mu f -> f (Mu f)
out = unMu
```

As seen in Section 2.3, the data type of natural numbers is isomorphic to the fixed point of the base functor $\underline{1} \hat{+} \mathsf{Id}$. Using `Mu` it can be defined in Haskell as follows.

```
newtype FNat x = FNat {unFNat :: Either One x}

instance Functor FNat
    where fmap f = FNat . (id -|- f) . unFNat

type Nat = Mu FNat
```

The zero and successor constructors can then be implemented as expected.

```
zero :: One -> Nat
zero = inn . FNat . inl

sukc :: Nat -> Nat
sukc = inn . FNat . inr
```

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

```
newtype FList a x = FList {unFList :: Either () (a,x)}

instance Functor (FList a)
    where fmap f = FList . (id -|- id >< f) . unFList

type List a = Mu (FList a)
```

The constructors of `List` can be implemented as follows.

```
nil :: One -> List a
nil = inn . FList . inl

cons :: (a, List a) -> List a
cons = inn . FList . inr
```

Using this style of programming polytypism comes for free, since the fundamental recursion operators given in Chapter 2 and Chapter 3 can be generically defined as follows.

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

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

ana :: Functor f => (a -> f a) -> a -> Mu f
ana h = hylo inn h
```

Given these operators, the length and factorial functions can be defined as follows.

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

one :: One -> Nat
one = sukc . zero

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

These functions were presented in examples 3.2 and 2.1, respectively. Notice that, apart from the constructors and destructors of the functor data types, the definitions are exactly the same as in the examples.

In order to define the remaining recursion patterns as hylomorphisms, the concept of *functor transformer* can be used. A functor transformer is a type constructor with kind $(\star \to \star) \to (\star \to \star)$, that given a functor returns another functor. It will be used to capture the functor change that occurs, for example, in the definition of paramorphisms using a hylomorphism. It was used in a similar context in [UVP01]. Recall the definition of paramorphisms: for a data type μF, the functor that captures the shape of recursion in the hylomorphism that implements it is $F \circ (Id \mathbin{\hat{\times}} \underline{\mu F})$. This specific functor change can be captured by the following transformer, defined in the pointwise style as a new data type.

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

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

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

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

Using `para`, the factorial can be defined according to Example 3.15.

```
fact :: Nat -> Nat
fact = para g
    where g = (one \/ mult . (id >< sukc)) . unFNat
```

This approach has some disadvantages. First, since Haskell does not have structural type equivalence, coercing constructors and destructors are used often. Sometimes, this makes it difficult to translate a point-free definition to Haskell. To overcome this problem, one could define tailored instances of the recursion operators for each data type, as proposed in [Gib02]. However, this would preclude polytypism, one of the main advantages of this approach. Second, it is impossible to use the recursion operators with the standard Haskell types, such as lists or integers. Finally, the `Functor` instances must be defined explicitly for every data type, when it is well known that the map function can be easily defined generically by induction on the structure of the type.

6.3 A Point-free Programming Library

This section describes a solution to the problems of the initial approach presented above. The solution is based on ideas developed previously in the context of polytypic programming, namely in the PolyP library [NJ03]. It comprises a restricted notion of structural equivalence between Haskell types.

6.3.1 The PolyP Approach to Recursive Data Types

PolyP is a library that allows polytypic programming in Haskell [NJ03]. Other approaches to generic programming in Haskell have been proposed, such as derivable type classes [HJ01], Generic Haskell [CHJ$^+$01], "lightweight generics"[CH02] (all three built upon Ralph Hinze's approach to generic functional programming [Hin00]), or the "scrap your boilerplate" technique [LJ03]. In the context of this library, the main advantage of PolyP is its view of data types as fixed points of functors. However, instead of using an explicit fixpoint operator, a multi-parameter type class [JJM97] with a functional dependency [Jon00] is used to relate a data type `d` with its base functor `f`. We remark that this is a non-standard Haskell feature provided as an extension. This class can be defined as follows. The dependency means that different data types can have the same base functor, but one data type can have at most one base functor. The use of the primes will be justified later.

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

We would like to stress that PolyP is not directly used in the implementation of Pointless Haskell. Some of its design choices would prevent a syntax similar to the one described in the first chapters. For example, in PolyP all data types are parameterized, which means that they are always generated from bifunctors and always determine a type functor, as described in Section 3.1.1. Given the objectives of our library, that would originate an unnecessarily complicated syntax, and as such we decided to reimplement the subset of PolyP we needed according to our own design principles. For example, the `FunctorOf` class was simplified by restricting base functors to monofunctors (a parameterized type can still be defined using the left-sectioning of a bifunctor). Its methods were also reduced to the essential `in` and `out` functions.

The main advantage of using the `FunctorOf` class is that predefined Haskell types can also be viewed as fixed points of functors. For integers and lists the following instances can be defined.

```
instance FunctorOf FNat Int
    where inn' = (pnt 0 \/ succ) . unFNat
          out' = FNat . (bang -|- pred) . ((==0)?)

instance FunctorOf (FList a) [a]
    where inn' = (pnt [] \/ uncurry (:)) . unFList
          out' = FList . (bang -|- head /\ tail) . (null?)
```

With the exception of the hylomorphism, the definition of the recursion operators must change slightly. For example, catamorphisms and anamorphisms must be redefined as follows, since the relation between a data type and its base functor must be explicitly stated using the `FunctorOf` class.

```
cata :: FunctorOf f d => (f a -> a) -> d -> a
cata g = hylo g out'

ana :: FunctorOf f d => (a -> f a) -> a -> d
ana h = hylo inn' h
```

With this approach it is possible to define, for example, the length function for Haskell lists using anamorphisms (according to Example 3.10).

```
length :: [a] -> Int
length = ana h
    where h = FNat . (id -|- snd) . unFList . out'
```

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

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

6.3.2 Polytypic `Functor` Instances

The PolyP approach to polytypic programming can also be used to avoid the explicit definition of the map functions. The general idea is to describe functors using a fixed set of combinators instead of arbitrary data types, and for these define the appropriate instances of the `Functor` class.

The combinators follow directly from the definition of regular functors presented in Chapter 2: we have the identity and constant functors, the lifting of the sum and product bifunctors, and also the application of a functor to another functor. To allow for a similar notation infix constructors will be used.

```
infixr 5 :+:
infixr 6 :*:
infixr 9 :@:

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

The `Functor` instances are trivial. Notice the use of class constraints to guarantee that only instances of the `Functor` class are lifted.

```
instance Functor Id
    where fmap f (Id x) = Id (f x)

instance Functor (Const t)
    where fmap f (Const x) = Const x

instance (Functor g, Functor h) => Functor (g :+: h)
    where fmap f (Inl x) = Inl (fmap f x)
          fmap f (Inr x) = Inr (fmap f x)

instance (Functor g, Functor h) => Functor (g :*: h)
    where fmap f (x :*: y) = (fmap f x) :*: (fmap f y)

instance (Functor g, Functor h) => Functor (g :@: h)
    where fmap f (Comp x) = Comp (fmap (fmap f) x)
```

Given this set of basic functors and functor combinators, there is no need to declare new functor data types to capture the recursive structure of a data type. Instead, they are declared using this basic set. For example, for integers and lists, the instances of the `FunctorOf` class could be something like

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

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

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

6.3.3 Implicit Coercion

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

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

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

The instances of `Rep` are also rather trivial. For example the identity and constant functors can be evaluated as follows.

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

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

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

```
instance (Rep (g a) b, Rep (h a) c) => Rep ((g :+: h) a) (Either b c)
    where to (Inl x) = Left (to x)
          to (Inr x) = Right (to x)
          from (Left x) = Inl (from x)
          from (Right x) = Inr (from x)

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

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

The most difficult case is that of the composition combinator. To compute the type that implements $(G \circ H)\ A$ we first determine the type B that implements $H\ A$, and then return the type that implements $G\ B$. At the value level, we first apply the translation at the inner location (i.e, the value with type $H\ A$) using the map function for G, and then translate the resulting value of type $G\ B$.

```
instance (Functor g, Rep (h a) b, Rep (g b) c) => Rep ((g :@: h) a) c
    where to (Comp x) = to (fmap to x)
          from y = Comp (fmap from (from y))
```

If G is a type functor instead of an expression involving functor combinators, the user must provide an instance of `Rep` that explicitly tells the compiler how to implement a value of type $G\ B$. In this cases the result is the value itself, and the instances are trivial. For example, the following instance can be defined for lists.

```
instance Rep [a] [a]
    where to = id
          from = id
```

This situation could be avoided by using a different composition combinator for the situation where G is a type functor. However, taking into account that, in practice, the type functor of lists is usually the only one used to declare data types, that would unnecessarily complicate the syntax.

A possible interaction with a Haskell interpreter could now be

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

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

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

Since this type-checking problem would occur frequently, we decided to annotate most of the polytypic functions with the functor to which they should be specialized. Types cannot be passed as arguments to functions, and so this is achieved indirectly through the use of a "dummy" argument and another non-standard Haskell feature, namely scoped type variables [JS02]. Since in Haskell only values of a concrete type (that is, of kind $\star$) can be passed as arguments, it is not possible to state directly the functor to which a function should be specialized. However, by using the type class `FunctorOf`, together with its functional dependency, it suffices to pass as argument a value of a data type that is the fixed point of the desired functor. Since recursive data types can still be defined explicitly using `Mu`, there is always a convenient choice for this parameter.

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

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

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

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

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

Since each data type has a unique base functor the following interaction is valid.

```
> out []
Left _L
> out [1,2,3]
Right (1,[2,3])
```

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

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

Notice that this type annotation is essentially the same that was stated using a subscript in the theoretical notation. It is now possible to program with hylomorphisms in a truly point-free style. For example, the definition of factorial given in Example 2.1 can be transcribed directly to Haskell.

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

Since explicit fixed points can be used, it is possible to use `hylo` without having to declare the type of intermediate data structure, nor the respective instance of the `FunctorOf` class. For example, the Fibonacci function can be defined using a binary shape tree as intermediate structure.

```
fib :: Int -> Int
fib = hylo (_L :: Mu (Const One :+: (Id :*: Id))) f g
    where g = (bang -|- pred /\ pred . pred) . ((<=1)?)
          f = one \/ plus
```

Other recursion patterns can now be defined exactly as presented in Chapter 3, with the functor that generates the intermediate data structure explicitly stated.

```
cata :: (FunctorOf f d, Rep (f a) fa, Rep (f d) fd) =>
        d -> (fa -> a) -> d -> a
cata (_::d) g = hylo (_L::d) g out

ana :: (FunctorOf f d, Rep (f a) fa, Rep (f d) fd) =>
       d -> (a -> fa) -> a -> d
ana (_::d) h = hylo (_L::d) inn h
```

This approach works equally well with data types defined using type functors, such as rose trees. Given the expected declaration of a data type `Rose a`, with an appropriate instance of `FunctorOf`

```
data Rose a = Forest a [Rose a]

instance FunctorOf (Const a :*: ([] :@: Id)) (Rose a)
    where inn' (Const x :*: Comp l) = Forest x (map unId l)
          out' (Forest x l) = Const x :*: Comp (map Id l)
```

the preorder traversal of rose trees can be defined exactly as in Example 3.8.

```
preorder :: Rose a -> [a]
preorder = cata (_L::Rose a) (cons . (id >< flatten))

flatten :: [[a]] -> [a]
flatten = cata (_L::[[a]]) (nil \/ cat)
```

One of the advantages of this approach is that, in order to declare the more advanced recursion patterns, we no longer need to define the functor transformers – this results from the ability to explicitly declare the intermediate data type as the fixed point of a functor. For example, the implementations of paramorphisms and accumulations can now be transcribed from their definitions.

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

accum (_::d) g t =
    hylo (_L :: FunctorOf f d => Mu (f :*: (Const a)))
         g ((t /\ snd) . (out >< id))
```

The types of these recursion operators are omitted because of their increasing complexity. Finally, we present the definitions of the factorial as a paramorphism, and of addition as an accumulation. Again, these are exactly the same as the ones presented in examples 3.15 and 3.20, respectively.

```
fact :: Int -> Int
fact = para (_L::Int) g
    where g = one \/ mult . (id >< succ)

plus :: (Int,Int) -> Int
plus = accum (_L::Int) g t
    where t = (fst -|- id >< succ) . distl
          g = (snd \/ fst) . distl
```

6.4 Visualization of Intermediate Data Structures

This section describes how to incorporate in Pointless Haskell a generic visualization mechanism for the intermediate data structures of hylomorphisms. This mechanism is based on GHood [Rei01], a graphical animation tool built on top of Hood [Gil00], the *Haskell Object Observation Debugger*. This feature is based on previous work on the visualization of recursion trees of Haskell functions [Cun03]. The section starts with a brief review of Hood and GHood.

6.4.1 Hood and GHood

Hood is a portable debugger for full Haskell, based on the concept of observation of intermediate data structures as they are passed between functions. Its author argues that this model is the analog, in the functional paradigm, to the traditional debugging method of breakpointing and variable examination used in imperative programming.

The starting point to the implementation of Hood was the non-standard unsafe function `trace`, included in all major Haskell distributions. This function has the type `String -> a -> a`, and its semantics is to print the first argument as a side effect and then return the second argument. There are some problems with using `trace` for debugging purposes [Gil00]:

- The biggest problem is that `trace` is strict in its first argument, and if we want to output some of the terms being evaluated the strictness properties of the observed program will inevitably change.
- The output tends to be incomprehensible partly due to the unintuitive ordering of lazy evaluation, and partly to the strictness on the first argument that might sometimes trigger the evaluation of other traces.
- Its insertion on the code tends to be invasive, due to the need to convey explicitly the outputed data in the first argument.

In order to overcome these problems, a new combinator with a similar type was developed:

```
observe :: (Observable a) => String -> a -> a
```

However, its semantics is considerably different from `trace`. The label is no longer used to output the observations, but only for identification purposes when multiple observation points coexist in the same program. Instead of just returning the second argument, it stores it into some persistent structure for later rendering. It behaves like an `id` function that somehow remembers its argument. The class restriction will be explained later. For now it suffices to know that there are predefined instances to most standard types.

Consider the following example presented in [Gil00]. In order to trace the execution of the following function that returns the digits of an integer

```
digits :: Int -> [Int]
digits = reverse .
         map (‘mod‘ 10) .
         takeWhile (/= 0) .
         iterate (‘div‘ 10)
```

`observe` can be inserted at the places where inspection of the intermediate data structures is required.

```
digits :: Int -> [Int]
digits = observe "after reverse" . reverse .
         observe "after map" . map (‘mod‘ 10) .
         observe "after takeWhile" . takeWhile (/= 0) .
         observe "after iterate" . iterate (‘div‘ 10)
```

If this function is called with the parameter `1234`, the following trace results.

```
-- after iterate
  1234 : 123 : 12 : 1 : 0 : _
-- after map
  4 : 3 : 2 : 1 : []
-- after reverse
  1 : 2 : 3 : 4 : []
-- after takeWhile
  1234 : 123 : 12 : 1 : []
```

Notice that unevaluated subexpressions are represented by underscore. The implementation of the `observe` function manages to eliminate all the major weaknesses of `trace`:

- It guarantees that the strictness properties of the observed program do not change.
- The capture of information (triggered by the evaluation of `observe`) is decoupled from presentation. The observations are post-processed when the program terminates, and are grouped by their labels into a structured presentation.

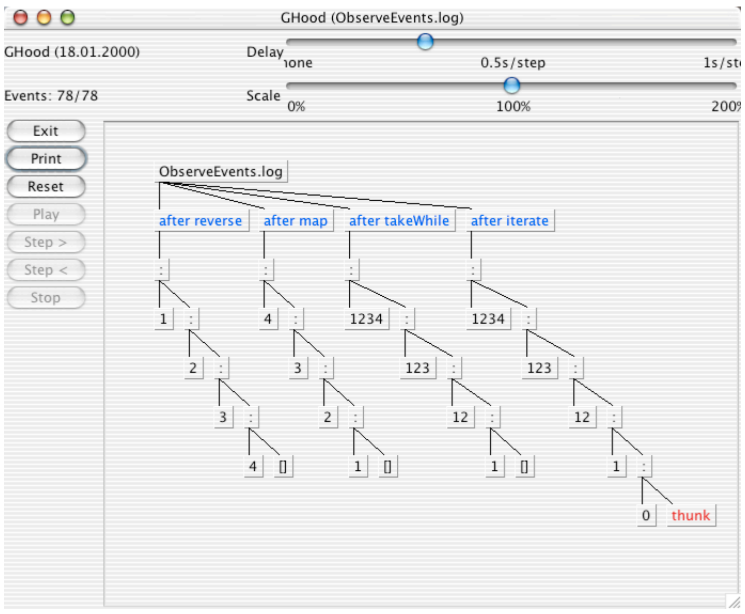

Figure 6.1: GHood screenshot

- Due to the existence of predefined instances for most standard types and a very simple, combinator based, approach to define new instances of the class `Observable`, minimal changes to the program are required.

The implementation of `observe` is based on a helper function `observer`, defined as a method of the class `Observable`. Implementing new instances of the class `Observable` is rather straightforward due to the high-level combinators and monads included in the library. For example, the predefined instance for lists is implemented as follows.

```
instance (Observable a) => Observable [a] where
  observer (a:as) = send ":"  (return (:) << a << as)
  observer []     = send "[]" (return [])
```

Function `send` is responsible for storing the information into some persistent structure for later rendering. Besides collecting information about *what* constructors are being observed, and *where* they are located in the data structure, the `send` function also records *when* the observation was done. This temporal information, that is not used by Hood's rendering, was the departing point to the developing of GHood's animation mechanism. Instead of a text-based visualization, GHood opted for a graphical visualization based on a simple tree-layout algorithm. The major change to the original library consisted of logging the events into a text file, so that they can be processed by an external viewer (developed in Java). For the user there are no changes at all. The final state of the animation for the same example presented above is displayed in Figure 6.1.

6.4.2 Instrumenting Hylomorphisms for Visualization

The key mechanism behind intermediate data structure visualization is the Hylo-Split law, that factorizes a hylomorphism into the composition of a catamorphism with an anamorphism. The application of this law exposes the virtual data structure that represents the recursion tree of the hylomorphism, and allows us to insert an observation point in order to record its runtime evolution. Following this approach, a new version of the hylomorphism can be defined, to be used when visualization is desired.

```
hyloO (_::d) g h = cata (_L::d) g .
                   observe "Recursion Tree" .
                   ana (_L::d) h
```

Notice that this function behaves exactly as the former hylomorphism definition, and can replace it in any definition, as long as the data type `d` belongs to the `Observable` class. For example, in order to observe the recursion tree of the factorial function, it suffices to replace `hylo` by `hyloO` in the previous definition. The list that results from observing `fact 5` is shown in Figure 6.2, and as expected corresponds to the list of all numbers from `5` downto `1`.

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

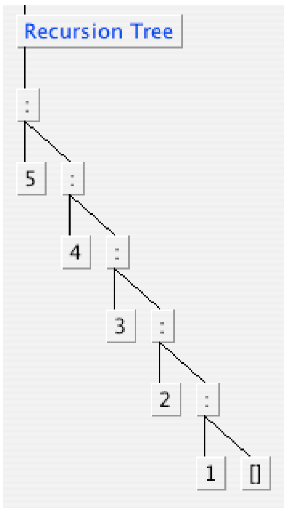

Figure 6.2: Recursion tree of `fact 5`.

It is also possible to define observed versions of other recursion patterns. For example, the definition of the observed catamorphism can be

```
cataO (_::d) f = hyloO (_L::d) f out
```

In order to observe a hylomorphism whose intermediate data structure is given by an explicit fixed point some additional work is required. Since `Mu` can be applied to any arrangement of the functor combinators, a polytypic instance of the `Observable` class must be defined. Such definition is the subject of the next section. Assuming its existence, the recursion tree of the Fibonacci function can be visualized as follows.

```
fib :: Int -> Int
fib = hyloO (_L :: Mu (Const One :+: Id :*: Id)) f g
    where g = (bang -|- pred /\ pred . pred) . ((<=1)?)
          f = one \/ plus
```

The binary shape tree that results from observing `fib 5` is presented in Figure 6.3. Notice that the number of leaves corresponds to the output of the function.

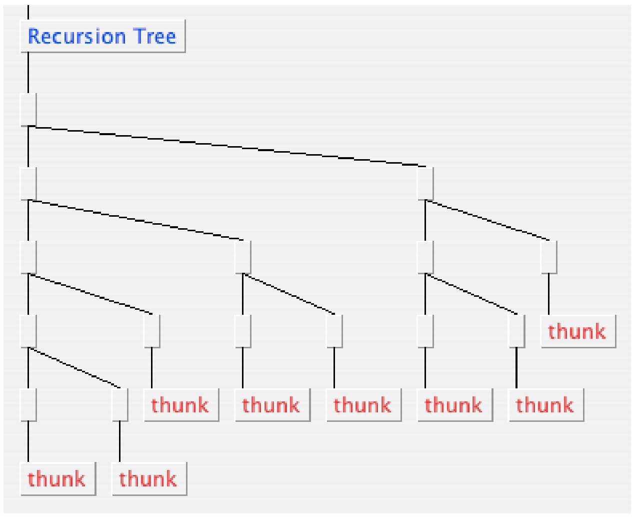

Figure 6.3: Recursion tree of `fib 5`.

A couple of visualization related remarks should be made about this tree. First, unlike the factorial example, the nodes of the tree are not tagged with constructor names because in the fixpoint view they simply do not exist. Second, the thunks that appear on the leaves correspond to unevaluated parts of the data structure. In this particular case, they occur because the leaves contain values of type `One`, that cannot be observed without raising an exception since they are `undefined`. This exposes a minor drawback of the polytypic observation function, namely that it will be impossible to distinguish an observation of the `One` data type from observations of unevaluated structures. For example, consider the following definition of the length function.

```
length :: Observable a => [a] -> Int
length = cataO (_L::Mu (Const One :+: Const a :*: Id)) g
    where g = zero \/ succ . snd
```

The recursion tree of `len [1,2,3]` is shown in Figure 6.4. Notice that, since this function is polymorphic and does not need to evaluate the elements of the list, they are represented by unevaluated thunks. But the same happens with the empty list (the last thunk in the sequence), since it is defined as the left injection of the bottom element.

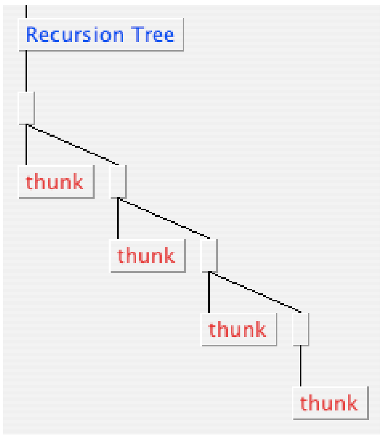

Figure 6.4: Recursion tree of `length [1,2,3]`.

The above definition of `hyloO` still suffers from an annoying limitation. Suppose that one wants to observe a user-defined data type, for which an instance of the `FunctorOf` class has already been defined. With the present definition of `hyloO`, the respective instance of `Observable` must be defined, even if a generic observation for the isomorphic fixed point already exists. In order to redirect the visualization to that instance, and thus avoid the need to define a new one, the definition of `hyloO` is changed as follows. Notice the use of the explicit type parameter, and the `FunctorOf` class, to force the hylomorphism to build as intermediate structure an element of type `Mu f` instead of the isomorphic element of type `d`. This is possible because `f` is simultaneously the base functor of `Mu f` and `d`.

```
hyloO (_::d) g h = cata (_L::FunctorOf f d => Mu f) g .
                   observe "Recursion Tree" .
                   ana (_L::FunctorOf f d => Mu f) h
```

The utility of this trick is illustrated with a final example. Suppose that the instance of `FunctorOf` for the `Tree` data type is already defined. It is well known that the quicksort algorithm can be defined as a hylomorphism using a binary tree as an intermediate structure [Aug99].

```
qsort :: (Observable a, Ord a) => [a] -> [a]
qsort = hyloO (_L::Tree a) f g
    where g = (id -|- fst /\ part) . out
          f = nil \/ cat . (id >< cons) . assocr . (swap >< id) . assocl
```

Most of the work is done by the `part` function, of type `(a,[a]) -> ([a],[a])`, that divides the tail into two lists: one keeps the elements that are smaller than the head of the input list, and the other the remaining elements. Even without declaring the `Observable` instance for `Tree`, the binary search tree that models the recursion tree of this definition can be visualized. Figure 6.5 shows the recursion tree of `qsort [3,2,4,3,1]`.

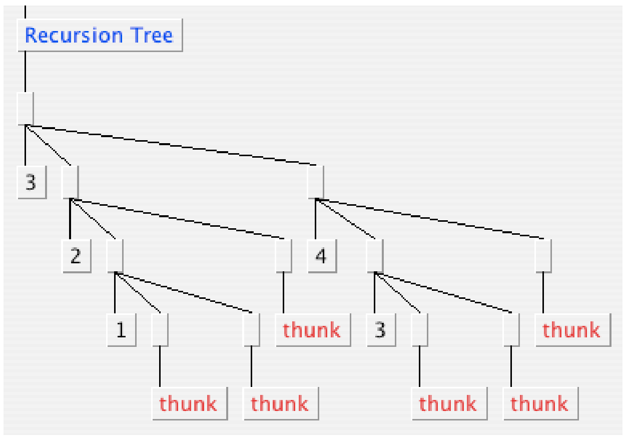

Figure 6.5: Recursion tree of `qsort [3,2,4,3,1]`.

6.4.3 Polytypic `Observable` Instances

In order to explain how a generic observer can be defined, the definition of `<<` is expanded in the following definition of the `Observable` instance for lists.

```
instance (Observable a) => Observable [a] where
  observer (a:as) = send ":"  (do {a'  <- thunk a
                                    as' <- thunk as
                                    return (a':as')})
  observer []     = send "[]" (return [])
```

The first parameter of the send function is the tag that labels the intermediate nodes of the tree. As already mentioned, this tag will be ignored in the visualization of values declared explicitly using the fixpoint operator. The second parameter is a value in the state `ObserverM`, that is executed by the `send` function in order to evaluate a term and simultaneously collect information for the renderer. Function `thunk` is invoked for each child in a node, and graphically it will trigger a new branch in the tree. Even without presenting more details, it is clear that the monadic code follows the structure of the type, and so it should be possible to define generically a function for embedding a value into `ObserverM`. In order to make this idea more precise, the desired instance for the fixpoint view of the list data type is presented.

```
instance Observable a => Observable (Mu (Const One :+: Const a :*: Id))
    where observer (Mu (Inl (Const x))) =
              send "" (do x' <- thunk x
                        return (Mu (Inl (Const x'))))
          observer (Mu (Inr (Const x :*: Id y))) =
              send "" (do x' <- thunk x
                          y' <- thunk y
                          return (Mu (Inr (Const x' :*: Id y'))))
```

Notice that the thunk function is invoked for every recursive occurrence of the data type and, in case of parameterized types, also at the content nodes (signaled by a constant in the base functor).

Following the PolyP approach, a class to contain the new polytypic function is first defined. For a data type μF, this function will be responsible for embedding a value of type F (μF) into the state monad ObserverM. It receives as parameter a monadic function that should be applied when the functor is the identity. The need for this parameter will be justified later.

```
class FunctorO f
    where fmapO :: (a -> ObserverM b) -> f a -> ObserverM (f b)
```

The instances for the basic set of functor combinators can be defined as follows.

```
instance FunctorO Id
    where fmapO f (Id x) = do x' <- f x
                              return (Id x')

instance Observable a => FunctorO (Const a)
    where fmapO f (Const x) = do x' <- thunk x
                                 return (Const x')

instance (FunctorO f, FunctorO g) => FunctorO (f :+: g)
    where fmapO f (Inl x) = do x' <- fmapO f x
                               return (Inl x')
          fmapO f (Inr x) = do x' <- fmapO f  x
                               return (Inr x')

instance (FunctorO f, FunctorO g) => FunctorO (f :*: g)
    where fmapO f (x :*: y) = do x' <- fmapO f x
                                 y' <- fmapO f y
                                 return (x' :*: y')

instance (FunctorO g, FunctorO h) => FunctorO (g :@: h)
    where fmapO f (Comp x) = do x' <- fmapO (fmapO f) x
                                return (Comp x')
```

Since thunk should only be applied at recursive values, it cannot be blindly applied at each occurrence of the identity functor. Due to the composition combinator, the identity functor may also occur in other positions. For example, the following functor can also be used to model lists.

```
Id :@: (Const One :+: Const a :*: Id)
```

The parameterization with a monadic function and the instance implementation for the composition guarantees that, if we invoke `fmapO` with `thunk` as parameter to an element of this type, it will only be applied to the rightmost identity (the one that denotes the recursive occurrence).

The name `fmapO` comes from the fact that this function closely resembles a monadic map [Fok94]. Given a monad M and a functor F, the monadic map is a function of type $(A \to M\ B) \to (F\ A \to M\ (F\ B))$ that distributes a monadic operation through the functor. Likewise to ordinary map, this is a well known polytypic function (over the structure of F), and is offered in the libraries of some generic programming languages, namely Generic Haskell [CHJ+01]. The only place where `fmapO` differs from a monadic map is in the treatment of the constant functor, where it applies `thunk` instead of the `return` function.

Given `fmapO`, the implementation of the `Observable` instance for `Mu` is very simple.

```
instance (Functor f, FunctorO f) => Observable (Mu f)
    where observer (Mu x) = send "" (do x' <- fmapO thunk x
                                        return (Mu x'))
```

Finally, the trivial instance of `Observable` for the terminal data type is defined. Remember that it is not possible to observe the only value of this type, unless an exception is raised. In practice that will never happen, and an unevaluated thunk will be displayed instead.

```
instance Observable One
    where observer _ = _L
```

6.5 Summary

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

We have also shown how to implement a generic mechanism for visualizing intermediate data structures of hylomorphisms. This feature is very useful for program understanding, since it graphically exposes the intermediate data structure that results from the factorization of a hylomorphism using the Hylo-Split law. Since it is based on Hood, the implementation preserves the strictness properties of the original program. This feature is also very useful to identify opportunities to apply the Hylo-Shift law. For example, in the visualization of the `length` function, the existence of unevaluated thunks means that, in fact, the type that parameterizes the hylomorphism can be simplified into a natural number, by shifting all work into the anamorphism side. Notice that the visualized tree looks like an element of type List 1 that is isomorphic to Nat.

The Pointless Haskell library is available for download from the following web page.

`http://wiki.di.uminho.pt/bin/view/Alcino/PointlessHaskell`

Chapter 7

Deriving Point-free Definitions

This thesis advocates the use of the point-free style and recursion patterns for program calculation and transformation. However, it is a fact that most programmers use explicit recursion instead of recursion patterns, and probably no one uses a pure point-free style. In this chapter we show how pointwise definitions can be converted into the point-free style, and how to replace explicit recursion by hylomorphisms. The idea is to enable one to program in one style, and automatically move to the other in order to perform proofs and calculations. A useful comparison here is that of mathematical transforms such as the Fourier transform or the Laplace transform, which allow to express functions in different domains in which certain manipulations are easier to perform.

The well-known equivalence between simply-typed λ-calculus (with pairs and terminal object) and cartesian closed categories was first suggested by Lambek [Lam80]. This equivalence includes a translation from pointwise terms to categorical combinators, that was later used by Curien to define a new implementation technique for functional languages – the *categorical abstract machine* [Cur93]. This translation is also the starting point to our point-free derivation mechanism. We show how it can be extended to handle sums and recursion. For recursion two different techniques will be used: a direct encoding of the fixpoint operator using hylomorphisms; and the hylomorphism derivation algorithm of Hu, Iwasaki, and Takeichi [HIT96]. We also show how this core λ-calculus can be used to encode structured types and pattern matching. In practice, this means that the translation can be applied to a reasonable subset of most modern functional programming languages, namely Haskell. A standard denotational semantics of λ-calculus based on pcpos is assumed.

7.1 Typed λ-Calculi and Cartesian Closed Categories

This section briefly presents a typed λ-calculus with products and a terminal object, and its translation to cartesian closed categories. Given a set of base types Σ, types are determined by the following grammar, where $\alpha \in \Sigma$.

$$\begin{array}{rcll} A & ::= & \alpha & \text{(Base types)} \\ & | & \mathsf{1} & \text{(Terminal type)} \\ & | & A \times A & \text{(Pairs)} \\ & | & A \to A & \text{(Functions)} \end{array}$$

The set of raw typed λ-terms is given by the following grammar, were c denotes a constant with type $\Delta(c)$, and x ranges over variables.

$$\begin{array}{rcll} M & ::= & \star & \text{(Unit)} \\ & | & c & \text{(Constant)} \\ & | & x & \text{(Variable)} \\ & | & \lambda x : A.M & \text{(Abstraction)} \\ & | & (MM) & \text{(Application)} \\ & | & \langle M, M \rangle & \text{(Pairing)} \\ & | & \mathsf{fst}(M) & \text{(First projection)} \\ & | & \mathsf{snd}(M) & \text{(Second projection)} \end{array}$$

The usual notions of free and bound variable are defined on terms. $\mathsf{FV}(M)$ denotes the set of free variables in M, and $M[N/x]$ the capture-avoiding substitution of N for the free occurrences of x in M. Typing rules are expressed in terms of typing judgments $\Gamma \vdash M : A$, where M is a term, A a type, and Γ is a typing context – a list of type declarations for the free variables in M. All variables in a context are assumed to be different, and can be reordered implicitly. The valid type judgments of this λ-calculus are those that can be derived from the following set of typing rules. A term in a valid type judgment is said to be well typed.

$$\frac{}{\Gamma \vdash \star : \mathsf{1}} \qquad \frac{}{\Gamma \vdash c : \Delta(c)} \qquad \frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}(M) : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}(M) : B}$$

The equational theory of this typed λ-calculus is the least typed congruence generated by the following equations (modulo α-equivalence). It will be denoted by $=_\lambda$. Typing information is given in the η-equations since they can only be applied to terms of a specific type.

$$\begin{array}{lr}
M =_\lambda \star \quad \Leftarrow \quad M : 1 & \textsf{Eta-Unit} \\
(\lambda x : A.M)N =_\lambda M[N/x] & \textsf{Beta-Func} \\
M =_\lambda \lambda x : A.Mx \quad \Leftarrow \quad M : A \to B \wedge x \notin \textsf{FV}(M) & \textsf{Eta-Func} \\
\textsf{fst}(\langle M, N\rangle) =_\lambda M \wedge \textsf{snd}(\langle M, N\rangle) = N & \textsf{Beta-Pair} \\
M =_\lambda \langle \textsf{fst}(M), \textsf{snd}(M)\rangle \quad \Leftarrow \quad M : A \times B & \textsf{Eta-Pair}
\end{array}$$

Example 7.1 (Swap). To exemplify the use of $=_\lambda$, the isomorphism between $A \times B$ and $B \times A$ is proved. The same proof was given in the point-free style in Chapter 2.

$$\begin{array}{lcl}
\textsf{swap} & : & A \times B \to B \times A \\
\textsf{swap} & = & \lambda x.\langle \textsf{snd}(x), \textsf{fst}(x)\rangle
\end{array}$$

$$\begin{array}{ll}
 & \textsf{swap}\ (\textsf{swap}\ M) \\
=_\lambda & \quad \{\text{ Definition of } \textsf{swap} \} \\
 & (\lambda x.\langle \textsf{snd}(x), \textsf{fst}(x)\rangle)\ ((\lambda x.\langle \textsf{snd}(x), \textsf{fst}(x)\rangle)\ M) \\
=_\lambda & \quad \{\ \textsf{Beta-Func}\ \} \\
 & (\lambda x.\langle \textsf{snd}(x), \textsf{fst}(x)\rangle)\ \langle \textsf{snd}(M), \textsf{fst}(M)\rangle \\
=_\lambda & \quad \{\ \textsf{Beta-Func}\ \} \\
 & \langle \textsf{snd}(\langle \textsf{snd}(M), \textsf{fst}(M)\rangle), \textsf{fst}(\langle \textsf{snd}(M), \textsf{fst}(M)\rangle)\rangle \\
=_\lambda & \quad \{\ \textsf{Beta-Pair}\ \} \\
 & \langle \textsf{fst}(M), \textsf{snd}(M)\rangle \\
=_\lambda & \quad \{\ \textsf{Eta-Pair}\ \} \\
 & M
\end{array}$$

Translation

The translation from the typed λ-calculus (pointwise) to the internal language of a cartesian closed category (point-free) is rather ingenious. It is detailed in many text books on the subject [LS86, Pie91, AL91, Cro93]. The way variables are handled resembles the translation of the lambda calculus into the *de Bruijn notation* [dB72], where variables are represented by integers that “measure” the distance to the abstraction were they where bound. In the present case, the typing context that contains the bound variables is represented by a left-nested pair, and a variable will be replaced by the path to its position in that tuple.

We begin with the interpretation of types. The function that translated types and typing contexts to objects will be denoted by $\mathcal{O}(\,\cdot\,)$. It is assumed that each base type A in Σ is represented by an object in the category, denoted by $\mathcal{O}(A)$. For the remaining types the translation is defined as expected.

$$\begin{array}{lcl}\mathcal{O}(1) & = & 1\\ \mathcal{O}(A \times B) & = & \mathcal{O}(A) \times \mathcal{O}(B)\\ \mathcal{O}(A \to B) & = & \mathcal{O}(B)^{\mathcal{O}(A)}\end{array}$$

Assuming that ϵ denotes the empty context, the object that represents a typing context is generated as follows.

$$\begin{array}{lcl}\mathcal{O}(\epsilon) & = & 1\\ \mathcal{O}(\Gamma, x : A) & = & \mathcal{O}(\Gamma) \times \mathcal{O}(A)\end{array}$$

The translation from λ-terms to morphisms in the category will be denoted by the function $\mathcal{M}(\,\cdot\,)$. To be more precise, this function operates on typing judgments. A judgment $\Gamma \vdash M : A$ will be interpreted as a morphism

$$\mathcal{M}(\Gamma \vdash M : A) : \mathcal{O}(\Gamma) \to \mathcal{O}(A)$$

according to the following rules. In order to simplify the presentation, type annotations are omitted from the basic categorical combinators.

$$\begin{array}{lcl}\mathcal{M}(\Gamma \vdash \star : 1) & = & !\\ \mathcal{M}(\Gamma \vdash c : A) & = & \underline{c} \circ !\\ \mathcal{M}(\Gamma, x : A \vdash x : A) & = & \mathsf{snd}\\ \mathcal{M}(\Gamma, y : A \vdash x : B) & = & \mathcal{M}(\Gamma \vdash x : B) \circ \mathsf{fst} \quad \Leftarrow \quad x \neq y\\ \mathcal{M}(\Gamma \vdash \lambda x : A.M : A \to B) & = & \overline{\mathcal{M}(\Gamma, x : A \vdash M : B)}\\ \mathcal{M}(\Gamma \vdash MN : A) & = & \mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash M : B \to A) \vartriangle \mathcal{M}(\Gamma \vdash N : B))\\ \mathcal{M}(\Gamma \vdash \langle M, N\rangle : A \times B) & = & \mathcal{M}(\Gamma \vdash M : A) \vartriangle \mathcal{M}(\Gamma \vdash N : B)\\ \mathcal{M}(\Gamma \vdash \mathsf{fst}(M) : A) & = & \mathsf{fst} \circ \mathcal{M}(\Gamma \vdash M : A \times B)\\ \mathcal{M}(\Gamma \vdash \mathsf{snd}(M) : B) & = & \mathsf{snd} \circ \mathcal{M}(\Gamma \vdash M : A \times B)\end{array}$$

Notice that in the rules for application and projections, the type of M can be obtained by type inference using the rules presented before.

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

$$\mathsf{ap} \circ (\mathcal{M}(\vdash M : A \to B) \circ ! \vartriangle \mathsf{id})$$

Example 7.2 (Identity). For example, the identity function $\lambda x : A.x$ is translated into

$$\overline{\mathsf{snd}} : 1 \to A^A$$

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

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

Example 7.3 (Swap). For the swap function defined in Example 7.1 we get the following translation.

$$\overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} : 1 \to B \times A^{A \times B}$$

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

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

Soundness

It can be shown that this translation is sound, i.e, that all equivalences proved using $=_\lambda$ can also be proved in the categorical setting using the equational theory presented in Chapter 2. Because of its length we omit the full proof of this fact. The interested reader can check it, for example, in [Cur93, Mar96]. Since the equational laws related to products have a direct correspondence in the categorical side, the most interesting part of the proof concerns Beta-Func and Eta-Func. The fundamental result is that the concept of substitution is replaced by that of composition. Consider the following typing rule for substitution.

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash M[N/x] : B}$$

Assuming that the object that results from the translation of a context or a type is denoted by the same identifier, we have that

$$\begin{array}{rcl} \mathcal{M}(\Gamma, x : A \vdash M : B) & : & \Gamma \times A \to B \\ \mathcal{M}(\Gamma \vdash N : A) & : & \Gamma \to A \\ \mathcal{M}(\Gamma \vdash M[N/x] : B) & : & \Gamma \to B \end{array}$$

Then, it can be proved by structural induction on M that

$$\mathcal{M}(\Gamma \vdash M[N/x] : B) = \mathcal{M}(\Gamma, x : A \vdash M : B) \circ (\mathsf{id} \vartriangle \mathcal{M}(\Gamma \vdash N : A))$$

The soundness of the translation concerning Beta-Func follows trivially from this lemma.

$$\begin{array}{cl} & \mathcal{M}(\Gamma \vdash (\lambda x : A.M)N : B) \\ = & \quad \{\text{ Translation of application }\} \\ & \mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash (\lambda x : A.M) : A \to B) \vartriangle \mathcal{M}(\Gamma \vdash N : A)) \\ = & \quad \{\text{ Translation of abstraction }\} \\ & \mathsf{ap} \circ (\overline{\mathcal{M}(\Gamma, x : A \vdash M : B)} \vartriangle \mathcal{M}(\Gamma \vdash N : A)) \\ = & \quad \{\text{ Prod-Absor, Exp-Cancel }\} \\ & \mathcal{M}(\Gamma, x : A \vdash M : B) \circ (\mathsf{id} \vartriangle \mathcal{M}(\Gamma \vdash N : A)) \\ = & \quad \{\text{ Above lemma }\} \\ & \mathcal{M}(\Gamma \vdash M[N/x] : B) \end{array}$$

7.2 Sums

We will now enrich this λ-calculus with sums.

$$\begin{array}{rcll} A & ::= & \ldots & \\ & | & A + A & \text{(Sums)} \end{array}$$

At the term level, it is augmented with case analysis and injections.

$$\begin{array}{rcll} M & ::= & \ldots & \\ & | & \mathsf{case}(M, M, M) & \text{(Case analysis)} \\ & | & \mathsf{inl}_A(M) & \text{(First injection)} \\ & | & \mathsf{inr}_A(M) & \text{(Second injection)} \end{array}$$

The typing rules for these constructs are as follows.

$$\frac{\Gamma \vdash L : A + B \quad \Gamma \vdash M : A \to C \quad \Gamma \vdash N : B \to C}{\Gamma \vdash \mathsf{case}(L, M, N) : C}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl}_A(M) : A + B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{inr}_B(M) : A + B}$$

Typically, the case expression will be used together with lambda abstractions, in the form $\mathsf{case}(L, \lambda x : A.M, \lambda y : B.N)$. In Haskell, and assuming that sum is implemented by the `Either` data type, this expression corresponds to

```
case L of (Left x) -> M; (Right y) -> N
```

Concerning the equational theory, the following equivalences hold.

$$\mathsf{case}(\mathsf{inl}(L), M, N) =_\lambda ML \;\wedge\; \mathsf{case}(\mathsf{inr}(L), M, N) =_\lambda NL \qquad \textsf{Beta-Case}$$

$$\begin{array}{c} MN =_\lambda \mathsf{case}(N, \lambda x : A.M(\mathsf{inl}(x)), \lambda y : B.M(\mathsf{inr}(y))) \\ \Leftarrow \\ x, y \notin \mathsf{FV}(M) \;\wedge\; N : A + B \;\wedge\; M \text{ strict} \end{array} \qquad \textsf{Eta-Case}$$

Given the pcpo semantics, a case expression diverges when its first argument diverges. This fact justifies the strictness side condition in Eta-Case.

Translation

Our approach to the translation of sums is slightly different from other published works, such as [Cro93]. The injections are handled straightforwardly.

$$\begin{array}{rcl} \mathcal{M}(\Gamma \vdash \mathsf{inl}(M) : A + B) & = & \mathsf{inl} \circ \mathcal{M}(\Gamma \vdash M : A) \\ \mathcal{M}(\Gamma \vdash \mathsf{inr}(M) : A + B) & = & \mathsf{inr} \circ \mathcal{M}(\Gamma \vdash M : B) \end{array}$$

The case translation is more difficult. We will first give an intuition of how it works. Notice that $\mathsf{case}(L, M, N) : C$ is equivalent to $(M \triangledown N)L$ (defined in a mixed, pointwise and point-free, style). This equivalence exposes the fact that a case is just an instance of application, and as such its translation exhibits the same top level structure:

$$\mathcal{O}(\Gamma) \xrightarrow{\mathcal{M}(\Gamma \vdash M \triangledown N) \vartriangle \mathcal{M}(\Gamma \vdash L)} \mathcal{O}(C^{A+B} \times (A + B)) \xrightarrow{\mathsf{ap}} \mathcal{O}(C)$$

The question remains of how to combine $\mathcal{M}(\Gamma \vdash M : A \to C)$ and $\mathcal{M}(\Gamma \vdash N : B \to C)$ in order to obtain the equivalent of $\mathcal{M}(\Gamma \vdash M \triangledown N : A + B \to C)$. Our solution is based on the internalization of the either combinator, that can be defined as follows.

$$\begin{array}{rcl} \mathsf{either} & : & C^A \times C^B \to C^{A+B} \\ \hline \mathsf{either} & = & (\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \end{array} \qquad \textsf{Either-Def}$$

To show that this definition is correct we show that its action on a pair of points yields the expected result. This calculation uses some properties about distr that are stated and proved in Appendix A.

$$\mathsf{either} \circ (\underline{f} \vartriangle \underline{g}) = \underline{f \triangledown g} \qquad \textsf{Either-Pnt}$$

$$
\begin{array}{ll}
 & \mathsf{either} \circ (\underline{f} \vartriangle \underline{g}) \\
= & \quad \{\, \mathsf{Either\text{-}Def} \,\} \\
 & \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr}} \circ (\underline{f} \vartriangle \underline{g}) \\
= & \quad \{\, \mathsf{Exp\text{-}Fusion},\ \mathsf{Sum\text{-}Functor\text{-}Id} \,\} \\
 & \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ ((\underline{f} \vartriangle \underline{g}) \times (\mathsf{id} + \mathsf{id}))} \\
= & \quad \{\, \mathsf{Distr\text{-}Nat} \,\} \\
 & \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ ((\underline{f} \vartriangle \underline{g}) \times \mathsf{id} + (\underline{f} \vartriangle \underline{g}) \times \mathsf{id}) \circ \mathsf{distr}} \\
= & \quad \{\, \mathsf{Sum\text{-}Functor\text{-}Comp},\ \mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Prod\text{-}Cancel} \,\} \\
 & \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\underline{f} \times \mathsf{id} + \underline{g} \times \mathsf{id}) \circ \mathsf{distr}} \\
= & \quad \{\, \mathsf{Pnt\text{-}Def},\ \mathsf{Sum\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\} \\
 & \overline{(f \circ \mathsf{snd} \triangledown g \circ \mathsf{snd}) \circ \mathsf{distr}} \\
= & \quad \{\, \mathsf{Sum\text{-}Absor},\ \mathsf{Distr\text{-}Snd} \,\} \\
 & \overline{(f \triangledown g) \circ \mathsf{snd}} \\
= & \quad \{\, \mathsf{Pnt\text{-}Def} \,\} \\
 & \underline{f \triangledown g}
\end{array}
$$

Notice that, given $f : A \to B$, in the definition $\underline{f} = \overline{f \circ \mathsf{snd}}$ the projection snd has type $1 \times A \to A$. However, the above proof is independent of this fact, and thus the following more general law is also valid.

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

Given this combinator, the translation of the case is defined as

$$
\begin{array}{c}
\mathcal{M}(\Gamma \vdash \mathsf{case}(L, M, N) : C) \\
= \\
\mathsf{ap} \circ (\mathsf{either} \circ (\mathcal{M}(\Gamma \vdash M : A \to C) \vartriangle \mathcal{M}(\Gamma \vdash N : B \to C)) \vartriangle \mathcal{M}(\Gamma \vdash L : A + B))
\end{array}
$$

Example 7.4 (Coswap). In order to exemplify the translation of sums, consider its application to the pointwise definition of coswap.

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

The following result is obtained.

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

The following calculation shows that this expression corresponds to the definition Coswap-Def.

$$\begin{array}{ll}
 & \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inr} \circ \mathsf{snd}} \mathbin{\triangle} \overline{\mathsf{inl} \circ \mathsf{snd}}) \mathbin{\triangle} \mathsf{snd})} \circ ! \mathbin{\triangle} \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\} \\
 & \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inr} \circ \mathsf{snd}} \mathbin{\triangle} \overline{\mathsf{inl} \circ \mathsf{snd}}) \mathbin{\triangle} \mathsf{snd}) \circ (! \mathbin{\triangle} \mathsf{id}) \\
= & \quad \{\, \mathsf{Either\text{-}Const} \,\} \\
 & \mathsf{ap} \circ (\overline{(\mathsf{inr} \mathbin{\triangledown} \mathsf{inl}) \circ \mathsf{snd}} \mathbin{\triangle} \mathsf{snd}) \circ (! \mathbin{\triangle} \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\} \\
 & (\mathsf{inr} \mathbin{\triangledown} \mathsf{inl}) \circ \mathsf{snd} \circ (\mathsf{id} \mathbin{\triangle} \mathsf{snd}) \circ (! \mathbin{\triangle} \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\} \\
 & \mathsf{inr} \mathbin{\triangledown} \mathsf{inl}
\end{array}$$

Soundness

To prove that this translation is sound concerning Beta-Case and Eta-Case, the following facts should be proved using the equational theory of categorical combinators.

$$\mathcal{M}(\Gamma \vdash \mathsf{case}(\mathsf{inl}(L), M, N)) = \mathcal{M}(\Gamma \vdash ML)$$
$$\mathcal{M}(\Gamma \vdash \mathsf{case}(\mathsf{inr}(L), M, N)) = \mathcal{M}(\Gamma \vdash NL)$$
$$\mathcal{M}(\Gamma \vdash MN) = \mathcal{M}(\Gamma \vdash \mathsf{case}(N, \lambda x : A.M(\mathsf{inl}(x)), \lambda y : B.M(\mathsf{inr}(y))))$$
$$\Leftarrow$$
$$x, y \notin \mathsf{FV}(M) \ \wedge\ N : A + B \ \wedge\ M \text{ strict}$$

To simplify the proof it is convenient to restate the laws for sums using the internalized version of the either combinator. Sum-Cancel can be defined and proved as follows.

$$^\bullet\mathsf{inl} \circ \mathsf{either} = \mathsf{fst} \ \wedge\ ^\bullet\mathsf{inr} \circ \mathsf{either} = \mathsf{snd} \qquad \mathsf{Either\text{-}Cancel}$$

$$\begin{array}{ll}
 & ^\bullet\mathsf{inl} \circ \mathsf{either} \\
= & \quad \{\, \mathsf{Pxe\text{-}Def} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{inl})} \circ \mathsf{either} \\
= & \quad \{\, \mathsf{Exp\text{-}Fusion},\ \mathsf{Prod\text{-}Functor\text{-}Comp} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{either} \times \mathsf{inl})} \\
= & \quad \{\, \mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Either\text{-}Def},\ \mathsf{Exp\text{-}Cancel} \,\} \\
 & \overline{(\mathsf{ap} \mathbin{\triangledown} \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ (\mathsf{id} \times \mathsf{inl})} \\
= & \quad \{\, \mathsf{Distr\text{-}Cancel} \,\} \\
 & \overline{(\mathsf{ap} \mathbin{\triangledown} \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{inl}} \\
= & \quad \{\, \mathsf{Sum\text{-}Absor},\ \mathsf{Sum\text{-}Cancel} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{fst} \times \mathsf{id})} \\
= & \quad \{\, \mathsf{Exp\text{-}Fusion},\ \mathsf{Exp\text{-}Reflex} \,\} \\
 & \mathsf{fst}
\end{array}$$

As usual, we omit the similar proof for right injection. Notice that, due to Exp-Equal, Either-Cancel can alternatively be defined in uncurried form as follows.

$$\begin{array}{c}
\mathsf{ap} \circ (\mathsf{either} \times \mathsf{inl}) = \mathsf{ap} \circ (\mathsf{fst} \times \mathsf{id}) \\
\mathsf{ap} \circ (\mathsf{either} \times \mathsf{inr}) = \mathsf{ap} \circ (\mathsf{snd} \times \mathsf{id})
\end{array} \qquad \mathsf{Either\text{-}Cancel\text{-}Alt}$$

For the remaining laws, namely Sum-Reflex and Sum-Fusion, a formulation using either is more difficult due to the strictness side conditions. First, notice that both these laws can be replaced by the following, that resembles Eta-Case at the point-free level. The proof of this equivalence is quite easy and is omitted.

$$f \circ \mathsf{inl} \triangledown f \circ \mathsf{inr} = f \quad \Leftarrow \quad f \text{ strict} \qquad \textsf{Eta-Sum}$$

Strict application. Consider a strict application operator $\mathsf{sap} : B^A \times A \to B$ defined by the following equations.

$$\mathsf{sap}\ (f, x) = \begin{cases} \bot & \text{if } x = \bot \\ f\ x & \text{otherwise} \end{cases}$$

Then $\overline{\mathsf{sap}} : B^A \to B^A$ is a strictify operator, that transforms any function into its strict version. The experienced Haskell programmer certainly recognizes this function, since it is predefined in the language as follows, and is widely used to avoid unneeded laziness.

```
($!) :: (a -> b) -> (a -> b)
f $! x = x ‘seq‘ f x
```

This implementation uses the standard seq function, that verifies the following equations. Notice that this function cannot be defined within the language, but must be provided by all compilers and interpreters.

$$\mathsf{seq}\ x\ y = \begin{cases} \bot & \text{if } x = \bot \\ y & \text{otherwise} \end{cases}$$

Given strict f it is obvious that $\overline{\mathsf{sap}}\ f = f$. In general, given a function $f : A \to B^C$ that returns a strict function, $\overline{\mathsf{sap}} \circ f = f$ holds. A function of this type returns a strict function if its uncurried version is right-strict. This concept is formalized as follows.

$$f \text{ right-strict} \quad \Leftrightarrow \quad f \circ (\mathsf{id} \times \underline{\bot}) = \underline{\bot} \circ \mathsf{snd} \qquad \textsf{Rstrict-Def}$$

$$\overline{\mathsf{sap}} \circ f = f \quad \Leftarrow \quad \mathsf{ap} \circ (f \times \mathsf{id}) \text{ right-strict} \qquad \textsf{Sap-Cancel}$$

Likewise to seq, it is not possible to define sap in general using the basic set of categorical combinators. However, for the particular case of arguments of sum type it can be defined. In Section 2.2 we have seen that $\mathsf{distr} \circ \mathsf{undistr} = \mathsf{id}$, but the same is not true for $\mathsf{undistr} \circ \mathsf{distr} : A \times (B + C) \to A \times (B + C)$. Given a value $(x, \bot)$, distr returns $\bot$ and thus undistr cannot recover the initial value, and also returns $\bot$. This fact, that makes **CPO** a non-distributive category, can be used to define the following instance of sap for sums.

$$\begin{array}{lcl} \mathsf{sap} & : & C^{(A+B)} \times (A + B) \to C \\ \mathsf{sap} & = & \mathsf{ap} \circ \mathsf{undistr} \circ \mathsf{distr} \end{array} \qquad \textsf{Sap-Def}$$

If the second component of the input pair is not $\bot$, the function in the first component is applied to it. Otherwise, $\mathsf{undistr} \circ \mathsf{distr}$ returns $\bot_{C^{(A+B)} \times (A+B)} = (\bot_{C^{(A+B)}}, \bot_{A+B})$, and since $\bot_{C^{(A+B)}}$ is the function that always returns bottom, the application yields $\bot_C$, as desired.

With the strictify operator, $\mathsf{Eta\text{-}Sum}$ can be restated, without an explicit strictness side-condition, as $f \circ \mathsf{inl} \triangledown f \circ \mathsf{inr} = \overline{\mathsf{sap}}\ f$. This law can now be internalized using the either combinator.

$$\mathsf{either} \circ ({}^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) = \overline{\mathsf{sap}} \qquad \mathsf{Eta\text{-}Either}$$

$$\begin{aligned}
& \mathsf{either} \circ ({}^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \\
= \quad & \{\, \mathsf{Either\text{-}Def} \,\} \\
& \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr}} \circ ({}^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \\
= \quad & \{\, \mathsf{Exp\text{-}Fusion},\ \mathsf{Sum\text{-}Functor\text{-}Id} \,\} \\
& \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ (({}^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \times (\mathsf{id} + \mathsf{id}))} \\
= \quad & \{\, \mathsf{Distr\text{-}Nat} \,\} \\
& \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ ((({}^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \times \mathsf{id}) + (({}^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \times \mathsf{id})) \circ \mathsf{distr}} \\
= \quad & \{\, \mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Prod\text{-}Cancel} \,\} \\
& \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ ({}^{\bullet}\mathsf{inl} \times \mathsf{id} + {}^{\bullet}\mathsf{inr} \times \mathsf{id}) \circ \mathsf{distr}} \\
= \quad & \{\, \mathsf{Pxe\text{-}Def},\ \mathsf{Sum\text{-}Absor} \,\} \\
& \overline{(\mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{inl})} \times \mathsf{id}) \triangledown \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{inr})} \times \mathsf{id})) \circ \mathsf{distr}} \\
= \quad & \{\, \mathsf{Exp\text{-}Cancel} \,\} \\
& \overline{(\mathsf{ap} \circ (\mathsf{id} \times \mathsf{inl}) \triangledown \mathsf{ap} \circ (\mathsf{id} \times \mathsf{inr})) \circ \mathsf{distr}} \\
= \quad & \{\, \mathsf{Sum\text{-}Fusion},\ \mathsf{Ap\text{-}Lstrict},\ \mathsf{Lstrict\text{-}Strict} \,\} \\
& \overline{\mathsf{ap} \circ ((\mathsf{id} \times \mathsf{inl}) \triangledown (\mathsf{id} \times \mathsf{inr})) \circ \mathsf{distr}} \\
= \quad & \{\, \mathsf{Undistr\text{-}Def} \,\} \\
& \overline{\mathsf{ap} \circ \mathsf{undistr} \circ \mathsf{distr}} \\
= \quad & \{\, \mathsf{Sap\text{-}Def} \,\} \\
& \overline{\mathsf{sap}}
\end{aligned}$$

Given this basic set of laws the soundness proof is much easier. As expected, the proof concerning the $\mathsf{Beta\text{-}Case}$ equations uses the cancellation law of either.

$$\begin{aligned}
& \mathcal{M}(\Gamma \vdash \mathsf{case}(\mathsf{inl}(L), M, N)) \\
= \quad & \{\, \text{Translation of case} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash N)) \vartriangle \mathcal{M}(\Gamma \vdash \mathsf{inl}(L))) \\
= \quad & \{\, \text{Translation of injection} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash N)) \vartriangle \mathsf{inl} \circ \mathcal{M}(\Gamma \vdash L)) \\
= \quad & \{\, \mathsf{Prod\text{-}Absor} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \times \mathsf{inl}) \circ ((\mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash N)) \vartriangle \mathcal{M}(\Gamma \vdash L)) \\
= \quad & \{\, \mathsf{Either\text{-}Cancel\text{-}Alt} \,\} \\
& \mathsf{ap} \circ (\mathsf{fst} \times \mathsf{id}) \circ ((\mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash N)) \vartriangle \mathcal{M}(\Gamma \vdash L)) \\
= \quad & \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Prod\text{-}Cancel} \,\} \\
& \mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash L)) \\
= \quad & \{\, \text{Translation of application} \,\} \\
& \mathcal{M}(\Gamma \vdash ML)
\end{aligned}$$

The soundness proof concerning Eta-Case uses the following lemma, stating that a variable that is not used in a term can be removed from the context.

$$\mathcal{M}(\Gamma, x : A \vdash M) = \mathcal{M}(\Gamma \vdash M) \circ \mathsf{fst} \quad \Leftarrow \quad x \notin \mathsf{FV}(M) \qquad \text{Drop-Var}$$

This can be proved by structural induction on the shape of M, as shown in [Mar96] for the cartesian closed subset. The extension of this proof to cover sums is trivial.

Another fact about the translation is that a strict function of type $A \to B$ is converted into a function $\Gamma \to B^A$ that returns a strict function. As such, due to Sap-Cancel the following lemma holds.

$$\overline{\mathsf{sap}} \circ \mathcal{M}(\Gamma \vdash M) = \mathcal{M}(\Gamma \vdash M) \quad \Leftarrow \quad M \text{ strict} \qquad \text{Strict-Trans}$$

We first prove that

$$\left.\begin{array}{l} \mathcal{M}(\gamma \vdash \lambda x : A.M(\mathsf{inl}(x))) = {}^{\bullet}\mathsf{inl} \circ \mathcal{M}(\Gamma \vdash M) \\ \mathcal{M}(\gamma \vdash \lambda x : A.M(\mathsf{inr}(x))) = {}^{\bullet}\mathsf{inr} \circ \mathcal{M}(\Gamma \vdash M) \end{array}\right. \quad \Leftarrow \quad x \notin \mathsf{FV}(M) \qquad \text{Aux}$$

$$\begin{array}{cl}
 & \mathcal{M}(\Gamma \vdash \lambda x : A.M(\mathsf{inl}(x))) \\
= & \quad \{\text{Translation of abstraction}\} \\
 & \overline{\mathcal{M}(\Gamma, x : A \vdash M(\mathsf{inl}(x)))} \\
= & \quad \{\text{Translation of application}\} \\
 & \overline{\mathsf{ap} \circ (\mathcal{M}(\Gamma, x : A \vdash M) \vartriangle \mathcal{M}(\Gamma, x : A \vdash \mathsf{inl}(x)))} \\
= & \quad \{\text{Drop-Var},\ x \notin \mathsf{FV}(M)\} \\
 & \overline{\mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash M) \circ \mathsf{fst} \vartriangle \mathcal{M}(\Gamma, x : A \vdash \mathsf{inl}(x)))} \\
= & \quad \{\text{Injection translation}\} \\
 & \overline{\mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash M) \circ \mathsf{fst} \vartriangle \mathsf{inl} \circ \mathcal{M}(\Gamma, x : A \vdash x))} \\
= & \quad \{\text{Variable translation}\} \\
 & \overline{\mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash M) \circ \mathsf{fst} \vartriangle \mathsf{inl} \circ \mathsf{snd})} \\
= & \quad \{\text{Prod-Def}\} \\
 & \overline{\mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash M) \times \mathsf{inl})} \\
= & \quad \{\text{Prod-Functor-Comp, Exp-Fusion}\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{inl})} \circ \mathcal{M}(\Gamma \vdash M) \\
= & \quad \{\text{Pxe-Def}\} \\
 & {}^{\bullet}\mathsf{inl} \circ \mathcal{M}(\Gamma \vdash M)
\end{array}$$

Soundness concerning Eta-Case can then be proved by the following calculation.

$$
\begin{array}{cl}
 & \mathcal{M}(\Gamma \vdash \mathsf{case}(N, \lambda x : A.M(\mathsf{inl}(x)), \lambda y : B.M(\mathsf{inr}(y)))) \\
= & \quad \{\, \text{Translation of case} \,\} \\
 & \mathsf{ap} \circ (\mathsf{either} \circ (\mathcal{M}(\Gamma \vdash \lambda x : A.M(\mathsf{inl}(x))) \vartriangle \mathcal{M}(\Gamma \vdash \lambda y : B.M(\mathsf{inr}(y)))) \vartriangle \mathcal{M}(\Gamma \vdash N)) \\
= & \quad \{\, \mathsf{Aux},\ x \notin \mathsf{FV}(M),\ y \notin \mathsf{FV}(M) \,\} \\
 & \mathsf{ap} \circ (\mathsf{either} \circ ({}^{\bullet}\mathsf{inl} \circ \mathcal{M}(\Gamma \vdash M) \vartriangle {}^{\bullet}\mathsf{inr} \circ \mathcal{M}(\Gamma \vdash M)) \vartriangle \mathcal{M}(\Gamma \vdash N)) \\
= & \quad \{\, \mathsf{Prod\text{-}Fusion} \,\} \\
 & \mathsf{ap} \circ (\mathsf{either} \circ ({}^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \circ \mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash N)) \\
= & \quad \{\, \mathsf{Eta\text{-}Either} \,\} \\
 & \mathsf{ap} \circ (\overline{\mathsf{sap}} \circ \mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash N)) \\
= & \quad \{\, \mathsf{Strict\text{-}Trans} \,\} \\
 & \mathsf{ap} \circ (\mathcal{M}(\Gamma \vdash M) \vartriangle \mathcal{M}(\Gamma \vdash N)) \\
= & \quad \{\, \text{Translation of application} \,\} \\
 & \mathcal{M}(\Gamma \vdash MN)
\end{array}
$$

7.3 Explicit Recursion

Instead of working with a fixed set of base types and a set of constants to manipulate them, we will incorporate in our λ-calculus the theory of recursive data types presented in Section 2.3. To be more specific, at the type level we have

$$
\begin{array}{rcll}
A & ::= & \ldots & \\
 & | & \mu(F) & \text{(Recursive data type)}
\end{array}
$$

where F is a regular functor, whose operation on types is defined as expected. At the term level the generic constructor and destructor, and the fixpoint operator are introduced.

$$
\begin{array}{rcll}
M & ::= & \ldots & \\
 & | & \mathsf{in}(M) & \text{(Constructor)} \\
 & | & \mathsf{out}(M) & \text{(Destructor)} \\
 & | & \mathsf{fix}(M) & \text{(Fixpoint)}
\end{array}
$$

The typing rules are

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

$$
\frac{\Gamma \vdash M : A \to A}{\Gamma \vdash \mathsf{fix}(M) : A}
$$

The characterization of the fixpoint operator includes the following equation (for a complete theory see [SP00]).

$$
\mathsf{fix}(M) =_{\lambda} M(\mathsf{fix}(M)) \qquad \mathsf{Beta\text{-}Fix}
$$

Translation

The translation of the constructors and destructors is trivial.

$$\begin{array}{lcl}\mathcal{M}(\Gamma \vdash \mathsf{in}(M) : \mu(F)) & = & \mathsf{in} \circ \mathcal{M}(\Gamma \vdash M : F(\mu(F))) \\ \mathcal{M}(\Gamma \vdash \mathsf{out}(M) : F(\mu(F))) & = & \mathsf{out} \circ \mathcal{M}(\Gamma \vdash M : \mu(F))\end{array}$$

For the fixpoint operator the translation uses the definition of fix using a hylomorphism, presented in section 2.4. The intermediate data structure of the hylomorphism is a stream of functions of type $A \rightarrow A$.

$$\mathcal{M}(\Gamma \vdash \mathsf{fix}(M) : A) = [\![\mathsf{ap}, \mathsf{id} \triangle \mathsf{id}]\!] \circ \mathcal{M}(\Gamma \vdash M : A \rightarrow A)$$

Example 7.5 (Repeat). We now give a simple example of translating a recursive function defined over a recursive data type. Consider the function that given an element generates an infinite stream with copies of that value. It is quite similar to the one defined in Example 3.12 for lists. As seen in Section 2.4, streams of elements of type A are defined as the fixed point of $\underline{A} \hat{\times} \mathsf{Id}$. In the point-free style this function could be defined as follows.

$$\begin{array}{lcl}\mathsf{repeat} & : & A \rightarrow \mathsf{Stream}\ A \\ \mathsf{repeat} & = & [\!(\mathsf{id} \triangle \mathsf{id})\!]_{\mathsf{Stream}\ A}\end{array}$$

Using the λ-calculus presented this chapter, a possible definition is

$$\begin{array}{lcl}\mathsf{repeat} & : & A \rightarrow \mathsf{Stream}\ A \\ \mathsf{repeat} & = & \lambda x.\mathsf{fix}(\lambda y.\mathsf{in}(\langle x, y\rangle))\end{array}$$

By applying the translation rules to this definition we get the following point-free expression. In this case, the intermediate data structure of the hylomorphism that encodes fix is a stream of functions of type $\mathsf{Stream}\ A \rightarrow \mathsf{Stream}\ A$.

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

The following calculation shows that this expression is indeed equivalent to the above anamorphism.

$$\left[\begin{array}{ll}
 & \mathsf{ap} \circ (\overline{\mathsf{fix} \circ \overline{\mathsf{in} \circ (\mathsf{snd} \circ \mathsf{fst} \vartriangle \mathsf{snd})}} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Prod-Absor, Exp-Cancel} \,\} \\
 & \mathsf{fix} \circ \overline{\mathsf{in} \circ (\mathsf{snd} \circ \mathsf{fst} \vartriangle \mathsf{snd})} \circ (! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Prod-Absor, Prod-Reflex} \,\} \\
 & \mathsf{fix} \circ \overline{\mathsf{in} \circ (\mathsf{snd} \times \mathsf{id})} \circ (! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Exp-Fusion, Prod-Cancel} \,\} \\
 & \mathsf{fix} \circ \overline{\mathsf{in}} \\
= & \quad \{\, \text{Fix-Def, Hylo-Fusion} \,\} \\
 & \left[\begin{array}{ll}
 & (\mathsf{id} \vartriangle \mathsf{id}) \circ \overline{\mathsf{in}} = (\mathsf{id} \times \overline{\mathsf{in}}) \circ (\overline{\mathsf{in}} \vartriangle \mathsf{id}) \\
 = & \quad \{\, \text{Prod-Fusion, Prod-Absor} \,\} \\
 & \overline{\mathsf{in}} \vartriangle \overline{\mathsf{in}} = \overline{\mathsf{in}} \vartriangle \overline{\mathsf{in}}
 \end{array}\right. \\
 & \llbracket \mathsf{ap}, \overline{\mathsf{in}} \vartriangle \mathsf{id} \rrbracket \\
= & \quad \{\, \text{Prod-Absor}, \overline{\mathsf{in}} \times \mathsf{id} : \underline{A} \times \mathsf{Id} \dot{\to} \underline{(\mathsf{Stream}\ A \to \mathsf{Stream}\ A)} \times \mathsf{Id} \,\} \\
 & \llbracket \mathsf{ap} \circ (\overline{\mathsf{in}} \times \mathsf{id}), \mathsf{id} \vartriangle \mathsf{id} \rrbracket \\
= & \quad \{\, \text{Exp-Cancel, Ana-Def} \,\} \\
 & [\!(\mathsf{id} \vartriangle \mathsf{id})\!]
\end{array}\right.$$

Example 7.6 (Repeat). Notice that the repeat function can also be defined as follows.

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

In this case, the translation rules yield the following point-free expression.

$$\mathsf{fix} \circ \overline{\overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap} \circ (\mathsf{snd} \circ \mathsf{fst} \vartriangle \mathsf{snd}))}}$$

To prove that this is also equal to the expected anamorphism, some simplifications are first performed.

$$\left[\begin{array}{ll}
 & \mathsf{ap} \circ (\mathsf{fix} \circ \overline{\overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap} \circ (\mathsf{snd} \circ \mathsf{fst} \vartriangle \mathsf{snd}))}} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Prod-Def} \,\} \\
 & \mathsf{ap} \circ (\mathsf{fix} \circ \overline{\overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap} \circ (\mathsf{snd} \times \mathsf{id}))}} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Prod-Def, Prod-Cancel} \,\} \\
 & \mathsf{ap} \circ (\mathsf{fix} \circ \overline{\overline{\mathsf{in} \circ (\mathsf{snd} \circ (\mathsf{snd} \times \mathsf{id}) \vartriangle \mathsf{ap} \circ (\mathsf{snd} \times \mathsf{id}))}} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Prod-Fusion, Exp-Fusion} \,\} \\
 & \mathsf{ap} \circ (\mathsf{fix} \circ \overline{\overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap})} \circ \mathsf{snd}} \circ ! \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Exp-Fusion,Prod-Def, Prod-Cancel} \,\} \\
 & \mathsf{ap} \circ (\mathsf{fix} \circ \overline{\overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap})} \circ \mathsf{snd}} \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Fix-Def, Hylo-Fusion} \,\} \\
 & \mathsf{ap} \circ (\llbracket \mathsf{ap}, \overline{\overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap})} \circ \mathsf{snd}} \vartriangle \mathsf{id} \rrbracket \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Prod-Absor, Hylo-Shift}, \dagger \,\} \\
 & \mathsf{ap} \circ (\llbracket \mathsf{ap} \circ (\overline{\overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap})} \circ \mathsf{snd}} \times \mathsf{id}), \mathsf{id} \vartriangle \mathsf{id} \rrbracket \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Exp-Cancel} \,\} \\
 & \mathsf{ap} \circ (\llbracket \overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap})} \circ \mathsf{snd}, \mathsf{id} \vartriangle \mathsf{id} \rrbracket \vartriangle \mathsf{id}) \\
= & \quad \{\, \text{Hylo-Shift}, \mathsf{snd} : \underline{A} \times \mathsf{Id} \dot{\to} \mathsf{Id}, \text{Prod-Cancel} \,\} \\
 & \mathsf{ap} \circ (\llbracket \overline{\mathsf{in} \circ (\mathsf{snd} \vartriangle \mathsf{ap})}, \mathsf{id} \rrbracket \vartriangle \mathsf{id})
\end{array}\right.$$

Concerning †, notice that

$$\overline{\overline{\mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap})} \circ \mathsf{snd}} \times \mathsf{id} : \underline{((\mathsf{Stream}\ A)^A \dot{\to} (\mathsf{Stream}\ A)^A)} \times \mathsf{Id} \to \underline{A} \times \mathsf{Id}$$

and notice also that, in the final hylomorphism, the intermediate data structure is $\mu(\mathsf{Id})$. Ana-Uniq can be used to prove that this strange expression is indeed equivalent to the expected anamorphism.

$$\begin{aligned}
& \mathsf{out} \circ \mathsf{ap} \circ ([\![\overline{\mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap})}, \mathsf{id}]\!] \triangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Hylo\text{-}Cancel} \,\} \\
& \mathsf{out} \circ \mathsf{ap} \circ (\overline{\mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap})} \circ [\![\overline{\mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap})}, \mathsf{id}]\!] \triangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\} \\
& \mathsf{out} \circ \mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap}) \circ ([\![\overline{\mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap})}, \mathsf{id}]\!] \triangle \mathsf{id}) \\
= & \quad \{\, \mathsf{In\text{-}Out\text{-}Iso},\ \mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Cancel} \,\} \\
& \mathsf{id} \triangle \mathsf{ap} \circ ([\![\overline{\mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap})}, \mathsf{id}]\!] \triangle \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Absor} \,\} \\
& (\mathsf{id} \times \mathsf{ap} \circ ([\![\overline{\mathsf{in} \circ (\mathsf{snd} \triangle \mathsf{ap})}, \mathsf{id}]\!] \triangle \mathsf{id})) \circ (\mathsf{id} \triangle \mathsf{id})
\end{aligned}$$

7.4 Deriving Hylomorphisms from Recursive Definitions

This last example shows that calculating with the derived point-free expressions can sometimes be quite difficult. This is largely due to the fact that hylomorphisms are introduced only to encode the fixpoint operator, yielding definitions very different from those one would get if making the derivation by hand. Ideally, one would like the resulting hylomorphisms to be more informative about the original function definition, in the sense that the intermediate data structure should model its recursion tree.

Hu, Iwasaki, and Takeichi have defined an algorithm that derives such hylomorphisms from an explicitly recursive definition [HIT96]. This algorithm was developed to be used in the fusion system HYLO [OHIT97], that uses the acid rain laws presented in Section 2.4 to perform deforestation. Basically, the algorithm mechanizes the informal process that has been used to derive hylomorphisms in Section 2.4. It has several limitations, namely it can not handle mutual or nested recursion, but it covers most of the useful function definitions.

In the present context, the idea is to use this algorithm in a stage prior to the point-free translation defined in the previous sections. First, a pointwise hylomorphism is derived, and then the translation is applied to the algebra and the coalgebra that parameterize it. The main difference between the presentation given in [HIT96] and the one given here lies in the underlying λ-calculus. While the original formulation allowed for user defined types *a la* Haskell and general pattern matching, in our λ-calculus data types are declared as fixed points, and pattern matching is restricted to sums. In [OHIT97] some extensions were made in order to cover more language constructs, but since we found some errors in those extensions the presentation will be restricted to the basic algorithm presented in [HIT96].

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

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

we will define three transformations: one to derive the functor that generates the intermediate data type ($\mathcal{F}$), a second one to derive the algebra ($\mathcal{A}$), and another one for the coalgebra ($\mathcal{C}$). The above function will be translated into the following hylomorphism, where a dot on top of an identifier signals the creation of a fresh variable.

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

Some restrictions are imposed on the syntax used to define recursive functions. The first is that the definition must be a closed expression, that is $\mathsf{FV}(L) = \{f, x\}$. This restriction guarantees that the algebra and the coalgebra are also closed. If that was not the case, it would be necessary to propagate the typing context inside the hylomorphism. This can only be achieved by changing the intermediate data structure, in a similar way to the accumulation recursion pattern presented in Section 3.5. Of course, the algebra and the coalgebra would also need to be modified, and the translation would produce hylomorphisms as unmanageable as the ones obtained by direct encoding of the fixpoint operator.

The second restriction is that the body L of the function $\mathsf{fix}(\lambda f.\lambda x.L)$ to be translated must be defined in two stages, according to the following grammar.

$$\begin{array}{rcll}
L & ::= & \mathsf{case}(M, \lambda x.L, \lambda x.L) & \Leftarrow f \notin \mathsf{FV}(M) \\
 & | & N & \\
N & ::= & \star & \\
 & | & c & \\
 & | & x & \Leftarrow x \neq f \\
 & | & (f\ N) & \Leftarrow f \notin \mathsf{FV}(N) \\
 & | & (N\ N) & \\
 & | & \langle N, N \rangle & \\
 & | & g(N) & \Leftarrow g \in \{\mathsf{fst}, \mathsf{snd}, \mathsf{inl}, \mathsf{inr}, \mathsf{in}, \mathsf{out}\}
\end{array}$$

Notice that M denotes the grammar of the full λ-calculus defined in the previous sections, and that x ranges over variables. Throughout this section, we also assume that f is the distinguished identifier of the variable in the first abstraction of the fixpoint.

A function is thus defined by a decision tree, implemented by case analysis on the input, whose leaves are the different possible outputs. Although it may seem very restrictive, this grammar covers a lot of interesting functions, as we will shortly see. One of the main limitations in expressiveness is the absence of abstractions, which prevents the definition of some higher-order functions, like the accumulations defined in Chapter 4.

The three transformations are now presented in turn. To be precise, they should be defined over typing judgments, but to simplify the presentation the contexts and types will be omitted. The transformation that generates the functor is defined as follows.

$$\begin{array}{lcl}
\mathcal{F}(\mathsf{case}(L, \lambda x.M, \lambda y.N)) & = & \mathcal{F}(M) \mathbin{\hat{+}} \mathcal{F}(N) \\
\mathcal{F}(\star) & = & \underline{1} \\
\mathcal{F}(c) & = & \underline{1} \\
\mathcal{F}(x : A) & = & \underline{A} \\
\mathcal{F}(f\ M) & = & \mathsf{Id} \\
\mathcal{F}(M\ N) & = & \mathcal{F}(M) \mathbin{\hat{\times}} \mathcal{F}(N) \\
\mathcal{F}(\langle M, N\rangle) & = & \mathcal{F}(M) \mathbin{\hat{\times}} \mathcal{F}(N) \\
\mathcal{F}(g(M)) & = & \mathcal{F}(M) \quad \Leftarrow \quad g \in \{\mathsf{fst}, \mathsf{snd}, \mathsf{inl}, \mathsf{inr}, \mathsf{in}, \mathsf{out}\}
\end{array}$$

This transformation yields a summand for each path along the decision tree. Each summand signals the presence of a recursive invocation using the identity functor. It also has place-holders for the input-dependent information that should be propagated unchanged from the coalgebra to the algebra. Since abstractions are not allowed, a constant functor of appropriate type is introduced for every variable outside a recursive invocation.

The definition of the coalgebra is very simple. It uses the same case analysis pattern of the original function in order to put all the relevant information in the correct places.

$$\begin{array}{lcl}
\mathcal{C}(\mathsf{case}(L, \lambda x.M, \lambda y.N)) & = & \mathsf{case}(L, \lambda x.\mathsf{inl}(\mathcal{C}(M)), \lambda y.\mathsf{inr}(\mathcal{C}(N))) \\
\mathcal{C}(\star) & = & \star \\
\mathcal{C}(c) & = & \star \\
\mathcal{C}(x) & = & x \\
\mathcal{C}(f\ M) & = & M \\
\mathcal{C}(M\ N) & = & \langle \mathcal{C}(M), \mathcal{C}(N)\rangle \\
\mathcal{C}(\langle M, N\rangle) & = & \langle \mathcal{C}(M), \mathcal{C}(N)\rangle \\
\mathcal{C}(g(M)) & = & \mathcal{C}(M) \quad \Leftarrow \quad g \in \{\mathsf{fst}, \mathsf{snd}, \mathsf{inl}, \mathsf{inr}, \mathsf{in}, \mathsf{out}\}
\end{array}$$

The generation of the algebra is a little more complicated, because each variable and recursive call must be replaced by the path to the place where it was stored by the coalgebra. In order to achieve this, an accumulator with the current path is used.

$$\begin{array}{lcl}
\mathcal{A}(\mathsf{case}(L, \lambda x.M, \lambda y.N), E) & = & \mathsf{case}(E, \lambda\dot{x}.\mathcal{A}(M, x), \lambda\dot{y}.\mathcal{A}(N, y)) \\
\mathcal{A}(\star, E) & = & \star \\
\mathcal{A}(c, E) & = & c \\
\mathcal{A}(x, E) & = & E \\
\mathcal{A}(f\ M, E) & = & E \\
\mathcal{A}(M\ N, E) & = & (\mathcal{A}(M, \mathsf{fst}(E))\ \mathcal{A}(N, \mathsf{snd}(E))) \\
\mathcal{A}(\langle M, N\rangle, E) & = & \langle \mathcal{A}(M, \mathsf{fst}(E)), \mathcal{A}(N, \mathsf{snd}(E))\rangle \\
\mathcal{A}(g(M), E) & = & g(\mathcal{A}(M), E) \quad \Leftarrow \quad g \in \{\mathsf{fst}, \mathsf{snd}, \mathsf{inl}, \mathsf{inr}, \mathsf{in}, \mathsf{out}\}
\end{array}$$

Without further optimizations, these transformations generate some redundancy in the intermediate data structure of the hylomorphism. In fact, it is not necessary to store elements of type 1 for all constants. A useful simplification that can be implemented over pairing and application, is to create a tuple only if variables or recursive calls appear on both sub-terms. We briefly show how to implement this simplification on pairs. The definition for application is similar. Notice that the occurrence of a recursive call is signaled by the presence of the free variable f. For the intermediate data type we have

$$\mathcal{F}(\langle M, N\rangle) = \begin{cases} \mathcal{F}(M) & \text{if } \ \mathsf{FV}(N) = \emptyset \\ \mathcal{F}(N) & \text{if } \ \mathsf{FV}(M) = \emptyset \\ \mathcal{F}(M) \hat{\times} \mathcal{F}(N) & \text{otherwise} \end{cases}$$

and for the coalgebra and algebra:

$$\mathcal{C}(\langle M, N\rangle) = \begin{cases} \mathcal{C}(M) & \text{if } \ \mathsf{FV}(N) = \emptyset \\ \mathcal{C}(N) & \text{if } \ \mathsf{FV}(M) = \emptyset \\ \langle \mathcal{C}(M), \mathcal{C}(N)\rangle & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\langle M, N\rangle, E) = \begin{cases} \langle \mathcal{A}(M, E), \mathcal{A}(N, E)\rangle & \text{if } \ \mathsf{FV}(N) = \emptyset \vee \mathsf{FV}(M) = \emptyset \\ \langle \mathcal{A}(M, \mathsf{fst}(E)), \mathcal{A}(N, \mathsf{snd}(E))\rangle & \text{otherwise} \end{cases}$$

We now give three examples of using this translation.

Example 7.7 (Repeat). By applying this algorithm to the fixpoint definition of repeat

$$\mathsf{fix}(\lambda f.\lambda x.\mathsf{in}(\langle x, f\ x\rangle))$$

previously given in Example 7.6, we get

$$[\![\lambda z.\mathcal{A}(\mathsf{in}(\langle x, f\ x\rangle), z), \lambda x.\mathcal{C}(\mathsf{in}(\langle x, f\ x\rangle))]\!]_{\mu(\mathcal{F}(\mathsf{in}(\langle x, f\ x\rangle)))}$$

After evaluating the three transformations

$$\begin{aligned}
\mathcal{F}(\mathsf{in}(\langle x, f\ x\rangle)) &= \mathcal{F}(\langle x, f\ x\rangle)\\
&= \mathcal{F}(x)\ \hat{\times}\ \mathcal{F}(f\ x)\\
&= \underline{A}\ \hat{\times}\ \mathsf{id}\\
\mathcal{C}(\mathsf{in}(\langle x, f\ x\rangle)) &= \mathcal{C}(\langle x, f\ x\rangle)\\
&= \langle \mathcal{C}(x), \mathcal{C}(f\ x)\rangle\\
&= \langle x, x\rangle\\
\mathcal{A}(\mathsf{in}(\langle x, f\ x\rangle), z) &= \mathsf{in}(\mathcal{A}(\langle x, f\ x\rangle, z))\\
&= \mathsf{in}(\langle \mathcal{A}(x, \mathsf{fst}(z)), \mathcal{A}(f\ x, \mathsf{snd}(z))\rangle)\\
&= \mathsf{in}(\langle \mathsf{fst}(z), \mathsf{snd}(z)\rangle)
\end{aligned}$$

the following hylomorphism is obtained.

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

This pointwise expression can be converted into the following point-free definition.

$$\mathsf{repeat} = [\![\mathsf{ap} \circ (\overline{\mathsf{in} \circ (\mathsf{fst} \circ \mathsf{snd} \vartriangle \mathsf{snd} \circ \mathsf{snd})} \circ ! \vartriangle \mathsf{id}), \mathsf{ap} \circ (\overline{\mathsf{snd} \vartriangle \mathsf{snd}} \circ ! \vartriangle \mathsf{id})]\!]_{\mu(\underline{A}\hat{\times}\mathsf{Id})}$$

Unlike with the direct translation of the fixpoint operator, it is now very easy to prove that this hylomorphism is equivalent to the anamorphism $[\!(\mathsf{id} \vartriangle \mathsf{id})\!]$, as the following calculations shows.

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

$$\begin{aligned}
&\mathsf{ap} \circ (\overline{\mathsf{snd} \vartriangle \mathsf{snd}} \circ ! \vartriangle \mathsf{id})\\
=&\quad \{\ \text{Prod-Absor, Exp-Cancel}\ \}\\
&(\mathsf{snd} \vartriangle \mathsf{snd}) \circ (! \vartriangle \mathsf{id})\\
=&\quad \{\ \text{Prod-Fusion, Prod-Cancel}\ \}\\
&\mathsf{id} \vartriangle \mathsf{id}
\end{aligned}$$

Example 7.8 (Length). Consider now the pointwise definition of length.

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

The hylomorphism derivation algorithm yields the following definition.

$$[\![\lambda z.\mathsf{case}(z, \lambda x.\mathsf{in}(\mathsf{inl}(\star)), \lambda y.\mathsf{in}(\mathsf{inr}(y))), \lambda l.\mathsf{case}(\mathsf{out}(l), \lambda x.\mathsf{inl}(\star), \lambda y.\mathsf{inr}(\mathsf{snd}(y)))]\!]_{\mu(\underline{1}\hat{+}\mathsf{Id})}$$

After converting the algebra and the coalgebra into the point-free style we get

$$\mathsf{length} = [\![\beta, \gamma]\!]_{\mu(\underline{1}\hat{+}\mathsf{Id})}$$

where

$$\begin{aligned}
\beta &= \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{in} \circ \mathsf{inl} \circ !} \triangle \overline{\mathsf{in} \circ \mathsf{inr} \circ \mathsf{snd}}) \triangle \mathsf{snd})} \circ ! \triangle \mathsf{id})\\
\gamma &= \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl} \circ !} \triangle \overline{\mathsf{inr} \circ \mathsf{snd} \circ \mathsf{snd}}) \triangle \mathsf{out} \circ \mathsf{snd})} \circ ! \triangle \mathsf{id})
\end{aligned}$$

The following calculations show that this hylomorphism is equivalent to the anamorphism defined in the Example 3.10.

$$\begin{array}{ll}
 & \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl} \circ !} \triangle \overline{\mathsf{inr} \circ \mathsf{snd} \circ \mathsf{snd}}) \triangle \mathsf{out} \circ \mathsf{snd})} \circ ! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\}\\
 & \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl} \circ !} \triangle \overline{\mathsf{inr} \circ \mathsf{snd} \circ \mathsf{snd}}) \triangle \mathsf{out} \circ \mathsf{snd}) \circ (! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Bang\text{-}Fusion},\ \mathsf{Either\text{-}Const} \,\}\\
 & \mathsf{ap} \circ (\overline{(\mathsf{inl} \circ ! \triangledown \mathsf{inr} \circ \mathsf{snd}) \circ \mathsf{snd}} \triangle \mathsf{out} \circ \mathsf{snd}) \circ (! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\}\\
 & (\mathsf{inl} \circ ! \triangledown \mathsf{inr} \circ \mathsf{snd}) \circ \mathsf{snd} \circ (\mathsf{id} \times \mathsf{out} \circ \mathsf{snd}) \circ (! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Prod\text{-}Cancel} \,\}\\
 & (\mathsf{inl} \circ ! \triangledown \mathsf{inr} \circ \mathsf{snd}) \circ \mathsf{out}\\
= & \quad \{\, \mathsf{Sum\text{-}Def},\ \mathsf{Bang\text{-}Reflex} \,\}\\
 & (\mathsf{id} + \mathsf{snd}) \circ \mathsf{out}
\end{array}$$

$$\begin{array}{ll}
 & \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{in} \circ \mathsf{inl} \circ !} \triangle \overline{\mathsf{in} \circ \mathsf{inr} \circ \mathsf{snd}}) \triangle \mathsf{snd})} \circ ! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\}\\
 & \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{in} \circ \mathsf{inl} \circ !} \triangle \overline{\mathsf{in} \circ \mathsf{inr} \circ \mathsf{snd}}) \triangle \mathsf{snd}) \circ (! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Bang\text{-}Fusion},\ \mathsf{Either\text{-}Const} \,\}\\
 & \mathsf{ap} \circ (\overline{(\mathsf{in} \circ \mathsf{inl} \circ ! \triangledown \mathsf{in} \circ \mathsf{inr}) \circ \mathsf{snd}} \triangle \mathsf{snd}) \circ (! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel} \,\}\\
 & (\mathsf{in} \circ \mathsf{inl} \circ ! \triangledown \mathsf{in} \circ \mathsf{inr}) \circ \mathsf{snd} \circ (\mathsf{id} \triangle \mathsf{snd}) \circ (! \triangle \mathsf{id})\\
= & \quad \{\, \mathsf{Prod\text{-}Cancel},\ \mathsf{Sum\text{-}Fusion} \,\}\\
 & \mathsf{in} \circ (\mathsf{inl} \circ ! \triangledown \mathsf{inr})\\
= & \quad \{\, \mathsf{Sum\text{-}Def},\ \mathsf{Bang\text{-}Reflex} \,\}\\
 & \mathsf{in} \circ (\mathsf{id} + \mathsf{id})\\
= & \quad \{\, \mathsf{Sum\text{-}Functor\text{-}Id} \,\}\\
 & \mathsf{in}
\end{array}$$

Example 7.9 (Map). In order to apply the hylomorphism derivation to the definition of the map function for lists, the first parameter should be treated as a constant, since the derivation algorithm can only handle functions with one parameter. Given a function $g : A \to B$, map can be defined as follows.

$$\begin{array}{lcl}
\mathsf{map}\ g & : & \mathsf{List}\ A \to \mathsf{List}\ B\\
\mathsf{map}\ g & = & \mathsf{fix}(\lambda f.\lambda x.\mathsf{case}(\mathsf{out}(x), \lambda y.\mathsf{in}(\mathsf{inl}(\star)), \lambda z.\mathsf{in}(\mathsf{inr}(\langle g(\mathsf{fst}(z)), f(\mathsf{snd}(z))\rangle))))
\end{array}$$

The derived pointwise hylomorphism uses a redundant intermediate data structure, because the derivation algorithm is blind to the fact that only the first projection of z must be propagated.

$$\mathsf{map}\ g = [\![\beta, \gamma]\!]_{\mu(1 \hat{+} \underline{A \times \mathsf{List}\ A} \hat{\times} \mathsf{Id})}$$

where

$$\beta = \lambda x.\mathsf{case}(x, \lambda y.\mathsf{in}(\mathsf{inl}(\star)), \lambda z.\mathsf{in}(\mathsf{inr}(\langle g(\mathsf{fst}(\mathsf{fst}(z))), \mathsf{snd}(z)\rangle)))$$
$$\gamma = \lambda x.\mathsf{case}(\mathsf{out}(x), \lambda y.\mathsf{inl}(\star), \lambda z.\mathsf{inr}(\langle z, \mathsf{snd}(z)\rangle))$$

This fact does not complicate the calculations because the expected intermediate data structure is easily obtained by shifting a natural transformation. After converting the hylomorphism parameters into the point-free style, and performing simplifications similar to the ones in the previous examples, we get the following definition.

$$[\![\mathsf{in} \circ (\mathsf{id} + g \circ \mathsf{fst} \times \mathsf{id}), (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{snd}) \circ \mathsf{out}]\!]_{\mu(1 \hat{+} \underline{A \times \mathsf{List}\ A} \hat{\times} \mathsf{Id})}$$

Using Prod-Functor-Comp, Sum-Functor-Comp, Hylo-Shift and the fact

$$\mathsf{id} + \mathsf{fst} \times \mathsf{id} : 1 \hat{+} \underline{A \times \mathsf{List}\ A} \hat{\times} \mathsf{Id} \stackrel{.}{\to} 1 \hat{+} \underline{A} \hat{\times} \mathsf{Id}$$

it can be transformed into

$$[\![\mathsf{in} \circ (\mathsf{id} + g \times \mathsf{id}), (\mathsf{id} + \mathsf{fst} \times \mathsf{id}) \circ (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{snd}) \circ \mathsf{out}]\!]_{\mu(1 \hat{+} \underline{A} \hat{\times} \mathsf{Id})}$$

Finally, by applying Sum-Functor-Comp, Prod-Absor, Prod-Reflex, and Cata-Def, the expected catamorphism is obtained.

$$\mathsf{map}\ g = (\!|\mathsf{in} \circ (\mathsf{id} + g \times \mathsf{id})|\!)_{\mathsf{List}\ A}$$

7.5 Pattern Matching and Structured Types

An important feature of most modern functional programming languages is pattern matching. Usually it is used together with *structured types*, that is, types declared as collections of constructors, in the style allowed by the Haskell `data` keyword [Jon87]. It is well-known how to implement an algorithm for calculating the isomorphic fixpoint of a structured type [NJ03]. In Section 2.3 we informally described how to determine the base functor and the isomorphism in from the constructors. The inverse is also easy to implement: a constructor can be defined in terms of in, the injections, and pairs, using the λ-calculus defined in this chapter. For example, the constructors of the Nat and List data types can be defined as follows.

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{zero} & : & \mathsf{Nat} & \mathsf{succ} & : & \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{zero} & = & \mathsf{in}(\mathsf{inl}(\star)) & \mathsf{succ} & = & \lambda x.\mathsf{in}(\mathsf{inr}(x)) \end{array}$$

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{nil} & : & \mathsf{List}\ A & \mathsf{cons} & : & A \times \mathsf{List}\ A \to \mathsf{List}\ A \\ \mathsf{nil} & = & \mathsf{in}(\mathsf{inl}(\star)) & \mathsf{cons} & = & \lambda x.\mathsf{in}(\mathsf{inr}(\langle \mathsf{fst}(x), \mathsf{snd}(x) \rangle)) \end{array}$$

In practice, this means that by replacing every occurrence of a constructor by its "fixpoint definition", it suffices to have pattern matching over the generic constructor in, sums, pairs, and the constant $\star$. In the remaining of this section we show how to implement such a mechanism, but with some limitations: there can be no repeated variables in the patterns, no overlapping, and the patterns must be exhaustive. Syntactically, the following construct is introduced. It matches an expression against a set of patterns, binds all the variables on the matching one, and returns the respective right-hand side.

$$\begin{array}{rcl} M & ::= & \ldots \\ & | & \mathsf{match}(M, \{P \to M, \ldots, P \to M\}) \end{array}$$

The syntax of patterns is determined by the following grammar. Notice that patterns are equivalent up to α-conversion.

$$\begin{array}{rcl} P & ::= & \star \\ & | & x \\ & | & \langle P, P \rangle \\ & | & \mathsf{in}(P) \\ & | & \mathsf{inl}(P) \\ & | & \mathsf{inr}(P) \end{array}$$

A matching is well-typed if, for each pattern, there exists a typing context containing all its variables, such that the type of the pattern is the same as that of the matching expression. This construct is merely syntactic sugar, and it is possible to translate a well-typed matching into the core λ-calculus previously defined using a term rewriting system.

The rudiments of rewriting theory will be presented with a little more detail in the next chapter, but to understand the translation the following concepts suffice. The application of the substitution θ to M is denoted by $M\theta$. A term L is an instance of M if there exists a substitution θ such that $L = M\theta$. A *term rewriting system* R is a set of rewrite rules with shape $M \leadsto N$, indicating that instances of M may be replaced in any context by instances of N. This means that the final rewrite relation extends R through the following inference rules, where f is any function symbol.

$$\frac{M \leadsto N}{M\theta \leadsto N\theta} \qquad \frac{M \leadsto N}{f(\ldots M \ldots) \leadsto f(\ldots N \ldots)}$$

We will now give the rules of the term rewriting system that encodes the required translation. Matching over $\star$ succeeds trivially. There can only be one pattern in the set due to the non-overlapping constraint.

$$\mathsf{match}(M, \{\star \to N\}) \leadsto N$$

Matching over a variable, binds the variable and triggers a substitution in the right-hand side. Again, due to the non-overlapping constraint the set of patterns must be a singleton.

$$\mathsf{match}(M, \{x \to N\}) \leadsto N[M/x]$$

For pairs, components are matched in turns. The chosen order is irrelevant, but after matching one projection with a specific pattern, the other one must only be matched against the pairing patterns.

$$\begin{array}{lcl}
\mathsf{match}(M, \{\langle P_1, Q_{1,1}\rangle \to N_{1,1} & & \mathsf{match}(\mathsf{fst}(M), \{P_1 \to \mathsf{match}(\mathsf{snd}(M), \{Q_{1,1} \to N_{1,1} \\
\quad \ldots & & \qquad \ldots \\
\quad \langle P_1, Q_{1,j}\rangle \to N_{1,j} & & \qquad Q_{1,j} \to N_{1,j}\}) \\
\quad \ldots & \leadsto & \quad \ldots \\
\quad \langle P_i, Q_{i,1}\rangle \to N_{i,1} & & \quad P_i \to \mathsf{match}(\mathsf{snd}(M), \{Q_{i,1} \to N_{i,1} \\
\quad \ldots & & \qquad \ldots \\
\quad \langle P_i, Q_{i,k}\rangle \to N_{i,k}\}) & & \qquad Q_{i,k} \to N_{i,k}\})\})
\end{array}$$

To match over a sum type case analysis is used. Due to the exhaustiveness requirement, the set of patterns can be partitioned into two disjoint sets, containing terms whose outermost constructor is inl and inr, respectively.

$$\begin{array}{lcl}
\mathsf{match}(M, \{\mathsf{inl}(P_{1,1}) \to N_{1,1}, & & \mathsf{case}(M, \lambda\dot{x}.\mathsf{match}(x, \{P_{1,1} \to N_{1,1}, \\
\quad \ldots, & & \qquad \ldots, \\
\quad \mathsf{inl}(P_{1,i}) \to N_{1,i}, & \leadsto & \qquad P_{1,i} \to N_{1,i}\}), \\
\quad \mathsf{inr}(P_{2,1}) \to N_{2,1}, & & \quad \lambda\dot{y}.\mathsf{match}(y, \{P_{2,1} \to N_{2,1}, \\
\quad \ldots, & & \qquad \ldots, \\
\quad \mathsf{inr}(P_{2,j}) \to N_{2,j}\}) & & \qquad P_{2,j} \to N_{2,j}\}))
\end{array}$$

Finally, when matching a value of a recursive type, the out function is used in order to expose its top level structure.

$$\begin{array}{lcl}
\mathsf{match}(M, \{\mathsf{in}(P_1) \to N_1, & & \mathsf{match}(\mathsf{out}(M), \{P_1 \to N_1, \\
\quad \ldots, & \leadsto & \quad \ldots, \\
\quad \mathsf{in}(P_i) \to N_i\}) & & \quad P_i \to N_i\})
\end{array}$$

Notice that this rewrite relation is guaranteed to terminate because the patterns always get smaller. We now give some examples of how to use this construct.

Example 7.10 (Assocr). Pattern matching is particularly useful to implement bookkeeping functions, namely when one needs to rearrange information in tuples. For example associativity for products can now be implemented as follows.

$$\begin{aligned}\mathsf{assocr} &: (A\times B)\times C\to A\times(B\times C)\\ \mathsf{assocr} &= \lambda t.\mathsf{match}(t,\{\langle\langle x,y\rangle,z\rangle\to\langle x,\langle y,z\rangle\rangle\})\end{aligned}$$

The above rewrite rules allow us to derive the following, not so intuitive, definition in the core λ-calculus.

$$\begin{aligned}
&\lambda t.\mathsf{match}(t,\{\langle\langle x,y\rangle,z\rangle\to\langle x,\langle y,z\rangle\rangle\})\\
\leadsto\ &\lambda t.\mathsf{match}(\mathsf{fst}(t),\{\langle x,y\rangle\to\mathsf{match}(\mathsf{snd}(t),\{z\to\langle x,\langle y,z\rangle\rangle\})\})\\
\leadsto\ &\lambda t.\mathsf{match}(\mathsf{fst}(t),\{\langle x,y\rangle\to\langle x,\langle y,\mathsf{snd}(t)\rangle\rangle\})\\
\leadsto\ &\lambda t.\mathsf{match}(\mathsf{fst}(\mathsf{fst}(t)),\{x\to\mathsf{match}(\mathsf{snd}(\mathsf{fst}(t)),\{y\to\langle x,\langle y,\mathsf{snd}(t)\rangle\rangle\})\})\\
\leadsto\ &\lambda t.\mathsf{match}(\mathsf{fst}(\mathsf{fst}(t)),\{x\to\langle x,\langle\mathsf{snd}(\mathsf{fst}(t)),\mathsf{snd}(t)\rangle\rangle\})\\
\leadsto\ &\lambda t.\langle\mathsf{fst}(\mathsf{fst}(t)),\langle\mathsf{snd}(\mathsf{fst}(t)),\mathsf{snd}(t)\rangle\rangle
\end{aligned}$$

By translating this definition into the point-free style and performing some routine calculations, the expected definition Assocr-Def is obtained.

Example 7.11 (Distr). Another interesting example is the implementation of distributivity.

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

The translation of the matching clause proceeds as follows.

$$\begin{aligned}
&\lambda x.\mathsf{match}(\mathsf{fst}(x),\{y\to\mathsf{match}(\mathsf{snd}(x),\{\mathsf{inl}(z)\to\mathsf{inl}(\langle y,z\rangle)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{inr}(z)\to\mathsf{inr}(\langle y,z\rangle)\})\})\\
\leadsto\ &\lambda x.\mathsf{match}(\mathsf{snd}(x),\{\mathsf{inl}(z)\to\mathsf{inl}(\langle\mathsf{fst}(x),z\rangle)\\
&\qquad\qquad\qquad\qquad \mathsf{inr}(z)\to\mathsf{inr}(\langle\mathsf{fst}(x),z\rangle)\})\\
\leadsto\ &\lambda x.\mathsf{case}(\mathsf{snd}(x),\lambda w.\mathsf{match}(w,\{z\to\mathsf{inl}(\langle\mathsf{fst}(x),z\rangle)\})\\
&\qquad\qquad\qquad\quad \lambda v.\mathsf{match}(v,\{z\to\mathsf{inr}(\langle\mathsf{fst}(x),z\rangle)\}))\\
\leadsto\ &\lambda x.\mathsf{case}(\mathsf{snd}(x),\lambda w.\mathsf{inl}(\langle\mathsf{fst}(x),w\rangle)\\
&\qquad\qquad\qquad\quad \lambda v.\mathsf{match}(v,\{z\to\mathsf{inr}(\langle\mathsf{fst}(x),z\rangle)\}))\\
\leadsto\ &\lambda x.\mathsf{case}(\mathsf{snd}(x),\lambda w.\mathsf{inl}(\langle\mathsf{fst}(x),w\rangle)\\
&\qquad\qquad\qquad\quad \lambda v.\mathsf{inr}(\langle\mathsf{fst}(x),v\rangle))
\end{aligned}$$

After translating this expression into the point-free style we get

$$\mathsf{ap}\circ(\overline{\mathsf{ap}\circ(\mathsf{either}\circ(\overline{\mathsf{inl}\circ(\mathsf{fst}\circ\mathsf{snd}\circ\mathsf{fst}\,\triangle\,\mathsf{snd})}\,\triangle\,\overline{\mathsf{inl}\circ(\mathsf{fst}\circ\mathsf{snd}\circ\mathsf{fst}\,\triangle\,\mathsf{snd})})\,\triangle\,\mathsf{snd}\circ\mathsf{snd})}\circ!\triangle\mathsf{id})$$

The following calculation shows that it is indeed equivalent to distr.

$$
\begin{array}{ll}
& \mathsf{ap} \circ (\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \circ \mathsf{fst} \bigtriangleup \mathsf{snd})} \bigtriangleup \overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \circ \mathsf{fst} \bigtriangleup \mathsf{snd})}) \bigtriangleup \mathsf{snd} \circ \mathsf{snd})} \circ ! \bigtriangleup \mathsf{id}) \\
= & \quad \{\, \text{Prod-Absor, Exp-Cancel, Prod-Def} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \times \mathsf{id})} \bigtriangleup \overline{\mathsf{inl} \circ (\mathsf{fst} \circ \mathsf{snd} \times \mathsf{id})}) \bigtriangleup \mathsf{snd} \circ \mathsf{snd}) \circ (! \bigtriangleup \mathsf{id}) \\
= & \quad \{\, \text{Exp-Fusion, Prod-Fusion} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl}} \bigtriangleup \overline{\mathsf{inr}}) \circ \mathsf{fst} \circ \mathsf{snd} \bigtriangleup \mathsf{snd} \circ \mathsf{snd}) \circ (! \bigtriangleup \mathsf{id}) \\
= & \quad \{\, \text{Prod-Fusion, Prod-Cancel, Prod-Def} \,\} \\
& \mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inl}} \bigtriangleup \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \text{Prod-Functor-Id, Either-Def, Exp-Cancel} \,\} \\
& (\mathsf{ap} \bigtriangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ ((\overline{\mathsf{inl}} \bigtriangleup \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \text{Sum-Functor-Id, Distr-Nat} \,\} \\
& (\mathsf{ap} \bigtriangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ ((\overline{\mathsf{inl}} \bigtriangleup \overline{\mathsf{inr}}) \times \mathsf{id} + (\overline{\mathsf{inl}} \bigtriangleup \overline{\mathsf{inr}}) \times \mathsf{id}) \circ \mathsf{distr} \\
= & \quad \{\, \text{Sum-Functor-Comp, Prod-Functor-Comp, Prod-Cancel} \,\} \\
& (\mathsf{ap} \bigtriangledown \mathsf{ap}) \circ (\overline{\mathsf{inl}} \times \mathsf{id} + \overline{\mathsf{inr}} \times \mathsf{id}) \circ \mathsf{distr} \\
= & \quad \{\, \text{Sum-Absor, Exp-Cancel} \,\} \\
& (\mathsf{inl} \bigtriangledown \mathsf{inr}) \circ \mathsf{distr} \\
= & \quad \{\, \text{Sum-Reflex} \,\} \\
& \mathsf{distr}
\end{array}
$$

Example 7.12 (Length). Assume that lists are declared using constructors nil and cons, the length function can be defined in a high-level, Haskell-like, style.

$$
\begin{array}{lcl}
\mathsf{length} & : & \mathsf{List}\ A \to \mathsf{Nat} \\
\mathsf{length} & = & \mathsf{fix}(\lambda f.\lambda l.\mathsf{match}(l, \{\mathsf{nil} \qquad\qquad \to \mathsf{zero}, \\
& & \qquad\qquad\qquad\qquad\quad \mathsf{cons}\ \langle h, t\rangle \to \mathsf{succ}(f(t))\}))
\end{array}
$$

By replacing the constructors with the definition given above, and applying some β-reductions we get the following definition

$$
\begin{array}{l}
\mathsf{fix}(\lambda f.\lambda l.\mathsf{match}(l, \{\mathsf{in}(\mathsf{inl}(\star)) \qquad \to \mathsf{in}(\mathsf{inl}(\star)), \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}(\mathsf{inr}(\langle h, t\rangle)) \to \mathsf{in}(\mathsf{inr}(f(t)))\}))
\end{array}
$$

After rewriting this into the core λ-calculus the definition given in Example 7.8 is obtained.

Example 7.13 (Fibonacci). The advantage of using pattern matching together with the structured view of data types is even more evident when nested matching is needed. One such example is the fibonacci definition.

$$
\begin{array}{lcl}
\mathsf{fib} & : & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{fib} & = & \mathsf{fix}(\lambda f.\lambda n.\mathsf{match}(n, \{\mathsf{zero} \qquad\qquad \to \mathsf{succ}(\mathsf{zero}), \\
& & \qquad\qquad\qquad\qquad \mathsf{succ}(\mathsf{zero}) \qquad \to \mathsf{succ}(\mathsf{zero}), \\
& & \qquad\qquad\qquad\qquad \mathsf{succ}(\mathsf{succ}(y)) \to \mathsf{plus}(\langle f(y), f(\mathsf{succ}(y))\rangle)\}))
\end{array}
$$

By expanding the constructors and eliminating the pattern matching, we get the following less intuitive definition.

$$\begin{aligned}
&\mathsf{fix}(\lambda f.\lambda n.\mathsf{case}(\mathsf{out}(n),\lambda x.\mathsf{in}(\mathsf{inr}(\mathsf{in}(\mathsf{inl}(\star)))),\\
&\qquad\qquad\qquad \lambda y.\mathsf{case}(\mathsf{out}(y),\lambda z.\mathsf{in}(\mathsf{inr}(\mathsf{in}(\mathsf{inl}(\star)))),\\
&\qquad\qquad\qquad\qquad\qquad \lambda w.\mathsf{plus}(\langle f(w), f(\mathsf{in}(\mathsf{inr}(w)))\rangle)))))
\end{aligned}$$

Finally, after transforming this definition into the point-free style and simplifying the result, the following definition of fibonacci is obtained.

$$\begin{aligned}
\mathsf{fib} &: \mathsf{Nat} \to \mathsf{Nat}\\
\mathsf{fib} &= [\![\mathsf{one} \triangledown (\mathsf{one} \triangledown \mathsf{plus}), (\mathsf{id} + (\mathsf{id} + \mathsf{id} \vartriangle \mathsf{succ}) \circ \mathsf{out}) \circ \mathsf{out}]\!]_{\mu(\underline{1} \hat{+} (\underline{1} \hat{+} \mathsf{Id} \hat{\times} \mathsf{Id}))}
\end{aligned}$$

7.6 Summary

In this chapter we have developed a mechanism to translate a function defined in a core functional programming language into the programming style presented in Chapter 2. Starting from the standard translation of the simply typed λ-calculus into cartesian closed categories, we have shown how to enrich it with case analysis over sums, and generalized recursion. The translation of the former is based on the internalization of the either combinator, and for the latter the hylomorphism encoding of the fixpoint operator was used. Although the resulting expressions are quite verbose, and sometimes quite intricate, they can be simplified by calculation. We have also shown how to enrich the core λ-calculus with a limited form of pattern matching that simplifies the definition of functions in many cases.

We have shown how to adapt the hylomorphism derivation algorithm first presented in [HIT96] to our λ-calculus. This algorithm enables the derivation of more tractable hylomorphisms, provided that the functions are defined with a special restricted syntax. When combined with pattern matching, this syntax corresponds to the one typically used to define most recursive functions in languages like Haskell.

Chapter 8

Decidability of Equality

One of the initial claims of this thesis, and one that is usually referred in the literature, is that point-free reasoning is more amenable to mechanization when compared to the pointwise style. The goal of this chapter is to shed some light on this question. We point to some relevant theoretical achievements and discuss some of the difficulties in implementing a decision procedure for point-free equations.

To be more specific, the chapter addresses the problem of deciding equality in *almost bicartesian closed categories* – categories with terminal object, products, exponentials, and non-empty coproducts – thus covering the set of basic categorical combinators presented in Chapter 2. Notice that we mention coproducts and not sums, which means that for this chapter all the issues concerning strictness will be set aside in the presentation. The subject already presents enough difficulties, even with this simplification. However, for the sake of coherence with the remaining chapters of the thesis, coproducts will still be referred as sums.

Due to the equivalence between the typed λ-calculus (with pairs, sums, and terminal object) and almost bicartesian closed categories, it is indifferent to study decidability in either formalism. In Chapter 7 only the translation from pointwise to point-free terms was presented, but it is straightforward to define and prove the soundness of the converse translation. The focus of the chapter will be on term rewriting systems, since this is a natural and well-established way of implementing both program transformation and equational reasoning.

In fact, almost all the research concerning decidability of equality in almost bicartesian closed categories has been carried out in the equivalent typed λ-calculus. The chapter starts by presenting some of problems that occur when using standard rewriting to implement a decision procedure in this setting. We then present an overview of the first decidability proof, that uses an expansionary rewrite system to normalize pointwise terms. An equivalent solution based on *normalization by evaluation* is also briefly discussed. Finally, the few known results about point-free normalization are presented.

8.1 Equational Reasoning and Term Rewriting Systems

Given an equational theory E, we say that the equality $M = N$ is valid in E (denoted $E \vdash M = N$) iff $M = N$ is derivable from the set of equations E using the following rules, where f is any function symbol and θ is a substitution.

$$\frac{}{M = M} \quad \frac{M = N}{N = M} \quad \frac{L = M \quad M = N}{L = N} \quad \frac{M = N}{M\theta = N\theta} \quad \frac{M = N}{f(\ldots M \ldots) = f(\ldots N \ldots)}$$

A theory E is decidable iff there is an algorithm which can determine if $E \vdash M = N$ for any M and N. One of the classic ways of deciding equality is through rewriting [Klo92, Pla93]. The basic idea is to orient equations in E into rules $M \rightsquigarrow N$, in order to obtain a term rewriting system R, which hopefully can be used to decide equality in E. As mentioned in Section 7.5, the rewrite relation extends the set R through the following inference rules.

$$\frac{M \rightsquigarrow N}{M\theta \rightsquigarrow N\theta} \quad \frac{M \rightsquigarrow N}{f(\ldots M \ldots) \rightsquigarrow f(\ldots N \ldots)}$$

A *redex* of L is a subterm that is an instance of the left-hand side of a rewrite rule. $\rightsquigarrow^*$ is defined as the reflexive transitive closure of $\rightsquigarrow$. A term M is *reducible* if there is a term N such that $M \rightsquigarrow N$. If that is not the case, then M is *irreducible.* If $M \rightsquigarrow^* N$ and N is irreducible then N is a *normal form* of M. A term rewriting system is *terminating* if has no infinite rewriting sequences.

It is convenient to define the relation $M \leftrightsquigarrow N$ to mean $M \rightsquigarrow N$ or $N \rightsquigarrow M$, and its reflexive transitive closure $\leftrightsquigarrow^*$. We also write $M \downarrow N$ if there exists a term L such that $M \rightsquigarrow^* L$ and $N \rightsquigarrow^* L$, and $M \uparrow N$ if there exists a L such that $L \rightsquigarrow^* M$ and $L \rightsquigarrow^* N$. A term rewriting system is *confluent* if, for all terms M and N, if $M \uparrow N$ then $M \downarrow N$. It is *locally confluent* if for all terms L, M, and N, if $L \rightsquigarrow M$ and $L \rightsquigarrow N$ then $M \downarrow N$. One of the easiest ways to prove confluence is through Newman's lemma, that states that if a term rewriting system is locally confluent and terminating then it is confluent.

Suppose that a term L can be rewritten using two different rules $M_1 \rightsquigarrow N_1$ and $M_2 \rightsquigarrow N_2$. This means that L has two redexes $R_1 = M_1\theta_1$ and $R_2 = M_2\theta_2$. Moreover, suppose that R_1 is a subterm of R_2 but does not occur in a variable position of M_2, that is, θ_2 does not contain the substitution R_1/x for any variable x. The two terms that result from rewriting L in this situation are called a *critical pair.* To prove local confluence it suffices to show that all critical pairs have common normal forms.

A term rewriting system is *Church-Rosser* if, for all terms M and N, $M \leftrightsquigarrow^* N$ iff $M \downarrow N$. It is a well-known result that a term rewriting system is Church-Rosser iff it is confluent. This result gives us a procedure for decidability in the equational theory that originated R. Since $M = N$ iff $M \leftrightsquigarrow^* N$, it is only necessary to show that M and N have a common normal form in order to decide if they are equal. However, in order to get an effective equality decision procedure, it is also convenient that the rewriting system is *terminating*.

8.2 Normalization by Rewriting

Recall the equational theory of the core λ-calculus presented in the previous chapter. Consider how these equations should be oriented to produce a term rewriting system. β-equations can be oriented as contraction rules, i.e. rules that reduce the size of the terms. From that theory we get the following set of rules.

$$\begin{aligned}
&(\lambda x : A.M)N \rightsquigarrow M[N/x] \\
&\mathsf{fst}(\langle M, N\rangle) \rightsquigarrow M \;\wedge\; \mathsf{snd}(\langle M, N\rangle) \rightsquigarrow N \\
&\mathsf{case}(\mathsf{inl}(L), M, N) \rightsquigarrow ML \;\wedge\; \mathsf{case}(\mathsf{inr}(L), M, N) \rightsquigarrow NL
\end{aligned}$$

It is not so obvious how η-equations should be treated. These equalities can be interpreted in two different ways, leading to different orientations as rewrite rules [dCK93]:

- In an operational way, stating possible optimizations in a program. For instance, if Eta-Pair is oriented as a contraction rule, it can be used to replace the term $\langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle$ by the more efficient M.

- In a theoretical way, making explicit the relation between a term and its type. For example, if Eta-Func is oriented as an expansion rule

  $$M \rightsquigarrow \lambda x : A.Mx \quad \Leftarrow \quad M : A \to B \;\wedge\; x \notin \mathsf{FV}(M)$$

  a term $M : A \to B$ will be embedded in an abstraction, making clear that it is of functional type, even without type inference.

Due to Eta-equations, decidability of equality in the presence of products and terminal object was for a long time an open problem. The first equation to raise problems was Eta-Pair, also known as *surjective pairing*. After initial evidence by Klop [Klo80] that using this equation as a contraction rule breaks confluence of an untyped λ-calculus with products and fixpoints, Pottinger proved confluence and termination in the typed scenario [Pot81]. However, when tackling the decision problem for cartesian closed categories, Lambek and Scott [LS86] showed that adding the terminal object, together with Eta-Unit, to a rewriting system containing either Eta-Func or Eta-Pair as contraction rules breaks confluence. Adam Obtulowicz is credited by these authors as being the first to point out this problem. Given any type A, the critical pairs that arise in this rewriting system are the following.

$$\begin{tikzcd}
 & \langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle : 1 \times A \arrow[dl, squiggly, "\mathsf{Eta\text{-}Unit}"'] \arrow[dr, squiggly, "\mathsf{Eta\text{-}Pair}"] & \\
\langle \star, \mathsf{snd}(M)\rangle & & M
\end{tikzcd}$$

$$\begin{tikzcd}
 & \langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle : A \times 1 \arrow[dl, squiggly, "\mathsf{Eta\text{-}Unit}"'] \arrow[dr, squiggly, "\mathsf{Eta\text{-}Pair}"] & \\
\langle \mathsf{fst}(M), \star\rangle & & M
\end{tikzcd}$$

$$\begin{tikzcd}
 & (\lambda x : 1.Mx) : 1 \to A \arrow[dl, squiggly, "\mathsf{Eta\text{-}Unit}"'] \arrow[dr, squiggly, "\mathsf{Eta\text{-}Func}"] & \\
(\lambda x : 1.M\star) & & M
\end{tikzcd}$$

$$\begin{tikzcd}
 & (\lambda x : A.Mx) : A \to 1 \arrow[dl, squiggly, "\mathsf{Eta\text{-}Unit}"'] \arrow[dr, squiggly, "\mathsf{Eta\text{-}Func}"] & \\
\lambda x : A.\star & & M
\end{tikzcd}$$

One had to wait until the nineties to see a proof of decidability for cartesian closed categories. Curien and Di Cosmo showed that it is possible to guarantee that all the offending critical pairs have common normal forms by extending, in a non-trivial way, the contractive reduction system [CdC91]. By non-trivial we mean using an infinite set of reduction rules that can be described in a finite way.

A few years later a much simpler solution, without any additional rules, was achieved just by using the η-equations as expansionary rewrite rules. Although in the past several authors had suggested reversing the traditional order of these equations, it was only in the beginning of the last decade that a renewed and sustained interest in expansions occurred, followed by (more or less independent) results showing confluence and termination for the λ-calculus with pairs and terminal object [dCK93, Dou93, JG95].

8.2.1 Expansionary Systems

The expansionary term rewriting system for deciding equality in a cartesian closed category can be obtained by orienting the η-equations as follows.

$$\begin{aligned}
M &\rightsquigarrow \star &&\Leftarrow\quad M : 1\\
M &\rightsquigarrow \lambda x : A.Mx &&\Leftarrow\quad M : A \to B \wedge x \notin \mathsf{FV}(M)\\
M &\rightsquigarrow \langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle &&\Leftarrow\quad M : A \times B
\end{aligned}$$

However, this is not enough to ensure termination because the following reduction loops may occur. Notice that in the first reduction sequence the last term is equal to the first by α-equivalence.

$$\begin{aligned}
&\lambda x : A.M \leadsto \lambda y : A.(\lambda x : A.M)y \leadsto \lambda y : A.M[y/x] \\
&MN \leadsto (\lambda x : A.Mx)N \leadsto MN \\
&\langle M, N\rangle \leadsto \langle \mathsf{fst}(\langle M, N\rangle), \mathsf{snd}(\langle M, N\rangle)\rangle \leadsto \langle M, \mathsf{snd}(\langle M, N\rangle)\rangle \leadsto \langle M, N\rangle \\
&\mathsf{fst}(M) \leadsto \mathsf{fst}(\langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle) \leadsto \mathsf{fst}(M) \\
&\mathsf{snd}(M) \leadsto \mathsf{snd}(\langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle) \leadsto \mathsf{snd}(M)
\end{aligned}$$

In order to avoid these loops one must avoid expansion of terms that are already abstractions or pairs, and that are either applied or projected. As such, the correct definition of the rewrite rules for functions and pairs is:

$$M \leadsto \lambda x : A.Mx \quad \Leftarrow \quad \begin{cases} M : A \to B \\ x \notin \mathsf{FV}(M) \\ M \text{ is not a } \lambda\text{-abstraction} \\ M \text{ is not applied} \end{cases}$$

$$M \leadsto \langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle \quad \Leftarrow \quad \begin{cases} M : A \times B \\ M \text{ is not a pair} \\ M \text{ is not projected} \end{cases}$$

There are some disadvantages in using expansions. Besides being type directed, the last side conditions mean that the rewrite relation is no longer a congruence, which makes difficult both the proofs of confluence and the implementation of the rewriting system. We will now present some examples of derivations (and normal forms) using this rewrite system, including some of the critical pairs that arose in the contractive system. Notice that the normal forms obtained with this system correspond to the long $\beta\eta$-normal forms of Huet [Hue76].

$$\begin{aligned}
&M : A \to 1 \leadsto \lambda x : A.Mx \leadsto \lambda x : A.\star \\
&M : A \to (B \to C) \leadsto \lambda x : A.Mx \leadsto \lambda x : A.\lambda y : B.(Mx)y \\
&M : (A \to B) \to C \leadsto \lambda x : A \to B.Mx \leadsto \lambda x : A \to B.M(\lambda y : A.xy) \\
&M : 1 \times A \leadsto \langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle \leadsto \langle \star, \mathsf{snd}(M)\rangle \\
&M : A \times (B \times C) \leadsto \langle \mathsf{fst}(M), \mathsf{snd}(M)\rangle \leadsto \langle \mathsf{fst}(M), \langle \mathsf{fst}(\mathsf{snd}(M)), \mathsf{snd}(\mathsf{snd}(M))\rangle\rangle
\end{aligned}$$

8.2.2 Coping With Sums.

Both Dougherty [Dou93], and Di Cosmo and Kesner [dCK93] showed that it is possible to also add Beta-Case as a contraction to this rewriting system without breaking confluence.

The latter authors went a little further by showing how to handle *weak sums* [dCK94]. Weak sums are characterized by the following restriction of Eta-Case.

$$M = \mathsf{case}(M, \lambda x : A.\mathsf{inl}(x), \lambda y : B.\mathsf{inr}(y)) \quad \Leftarrow \quad M : A + B \qquad\qquad \mathsf{Eta\text{-}Wsum}$$

Comparing with the laws that characterize sums in the point-free setting, this law corresponds just to the reflexivity law Sum-Reflex, while Eta-Case also captures Sum-Fusion. Adding Eta-Wsum as a contraction breaks confluence, as the following critical pair shows. Notice that, unlike for the cartesian fragment, this example does not involve the terminal object.

$$\begin{tikzcd}
\mathsf{case}(M, \lambda x : A \to B.\mathsf{inl}(x), \lambda y : C.\mathsf{inr}(y)) \arrow[d, rightsquigarrow, "\mathsf{Eta\text{-}Func}"'] \arrow[dr, rightsquigarrow, "\mathsf{Eta\text{-}Wsum}"] & \\
\mathsf{case}(M, \lambda x : A \to B.\mathsf{inl}(\lambda z : A.xz), \lambda y : C.\mathsf{inr}(y)) & M
\end{tikzcd}$$

The problem with this critical pair is that $\mathsf{inl}(x)$ is not a normal form with respect to the expansionary cartesian subset of the rewriting rules. The solution of Di Cosmo and Kesner was to postpone the rewriting until the normal form of the case analysis argument(s) is reached. The following rule, where $\|M\|$ denotes the normal form of M w.r.t. expansion rules Eta-Func, Eta-Pair, and Eta-Unit, can thus be added to the system without breaking confluence.

$$\mathsf{case}(M, \|\lambda x : A.\mathsf{inl}(x)\|, \|\lambda y : B.\mathsf{inr}(y)\|) \rightsquigarrow M$$

Unfortunately, this simple trick cannot be used for the more general Eta-Case.

The first proof of decidability for almost bicartesian closed categories was achieved by Neil Ghani, by orienting Eta-Case as an expansion [Gha95]. In order to give an overview of his solution, we will first present the problems that arise from using Eta-Case as an expansionary rewrite rule. Notice that Eta-Case can alternatively be defined as follows.

$$\begin{gathered}
M[N/z] =_\lambda \mathsf{case}(N, \lambda x : A.M[\mathsf{inl}(x)/z], \lambda y : B.M[\mathsf{inr}(y)/z]) \\
\Leftarrow \\
x, y \notin \mathsf{FV}(M) \ \wedge \ N : A + B
\end{gathered}$$

By orienting this equation from left to right we get an expansionary rewrite rule. First, notice that this rule can only be applied to free subterms of sum type. N is a *free subterm* of M if none of its free variables is bound in M. Notice also that the rule is highly non-local, in the sense that N is expanded to the head of the term. If N is the first argument of a case expression new redexes will be generated that must be rewritten using Beta-Case. Take, for example, the following term.

$$\lambda x : A + A.\langle\mathsf{case}(x, \lambda y : A.y, \lambda z : A.z), \star\rangle$$

The whole term is not a redex because it does not contain any free subterm of sum type. In this case, a possible redex is the pair because $x : A + A$ is a free subterm of sum type. After the expansion we get the term

$$\begin{aligned}\lambda x : A + A.\mathsf{case}(x, \quad & \lambda w : A.\langle \mathsf{case}(\mathsf{inl}(w), \lambda y : A.y, \lambda z : A.z), \star\rangle, \\ & \lambda k : A.\langle \mathsf{case}(\mathsf{inr}(k), \lambda y : A.y, \lambda z : A.z), \star\rangle)\end{aligned}$$

which in turn, after rewriting with Beta-Case, leads to the following normal form. Notice how the case expression was expanded into the head of the reduct, by exchange with the pair constructor.

$$\lambda x : A + A.\mathsf{case}(x, \lambda w : A.\langle w, \star\rangle, \lambda k : A.\langle k, \star\rangle)$$

Per se, this rule is not enough to guarantee confluence. When a redex contains several free subterms of sum type we get a set of normal forms that only differ in the order of eliminating the subterms through case analysis. For example, the term

$$\langle \mathsf{case}(M, \lambda x : A.x, \lambda y : A.y), \mathsf{case}(N, \lambda z : B.z, \lambda w : B.w)\rangle$$

where $M : A + A$ and $N : B + B$, originates the following critical pair.

$$\begin{aligned}\mathsf{case}(M, \quad & \lambda x : A.\mathsf{case}(N, \lambda z : B.\langle x, z\rangle, \lambda w : B.\langle x, w\rangle), \\ & \lambda y : A.\mathsf{case}(N, \lambda z : B.\langle y, z\rangle, \lambda w : B.\langle y, w\rangle)) \\ \mathsf{case}(N, \quad & \lambda x : A.\mathsf{case}(M, \lambda z : B.\langle z, x\rangle, \lambda w : B.\langle w, x\rangle), \\ & \lambda y : A.\mathsf{case}(M, \lambda z : B.\langle z, y\rangle, \lambda w : B.\langle w, z\rangle))\end{aligned}$$

Ghani's solution to this problem consisted in decomposing the full expansionary equational theory into two different relations: the first is a terminating rewrite relation that adds to the cartesian subset just the equation Eta-Wsum as an expansion, together with a set of contraction rules to simplify combinations of case with all the constructors; the second is a *conversion* relation that implements an algebra of case analysis, which identifies terms that only differ in the order of the elimination of sums. By embedding the rewrite relation into the conversion relation, proved decidable by showing that each term has a (finite, enumerable) set of related terms, the full expansionary rewrite relation is shown to be decidable.

To be more specific, the rewrite relation contains the following expansions that, depending on the type of the redex, are disallowed for unit, pairs, abstractions, case expressions, projected and applied terms, and terms subject to case analysis.

$$\begin{array}{l}
M \rightsquigarrow \star \quad \Leftarrow \quad M : 1 \\
M \rightsquigarrow \lambda x : A.Mx \quad \Leftarrow \quad M : A \to B \wedge x \notin \mathsf{FV}(M) \\
M \rightsquigarrow \langle \mathsf{fst}(M), \mathsf{snd}(M) \rangle \quad \Leftarrow \quad M : A \times B \\
M \rightsquigarrow \mathsf{case}(M, \lambda x : A.\mathsf{inl}(x), \lambda y : B.\mathsf{inr}(y)) \quad \Leftarrow \quad M : A + B
\end{array}$$

The rewriting relation contains also the following contractions. Notice that the set of rules capturing the interaction of case with all the constructors (including case itself), implements Sum-Fusion in pointwise terms.

$$\begin{array}{l}
\mathsf{fst}(\langle M, N \rangle) \rightsquigarrow M \\
\mathsf{snd}(\langle M, N \rangle) \rightsquigarrow N \\
\mathsf{fst}(\mathsf{case}(L, \lambda x : A.M, \lambda y : B.N)) \rightsquigarrow \mathsf{case}(L, \lambda x : A.\mathsf{fst}(M), \lambda y : B.\mathsf{fst}(N)) \\
\mathsf{snd}(\mathsf{case}(L, \lambda x : A.M, \lambda y : B.N)) \rightsquigarrow \mathsf{case}(L, \lambda x : A.\mathsf{snd}(M), \lambda y : B.\mathsf{snd}(N)) \\
(\lambda x : A.M)N \rightsquigarrow M[N/x] \\
\mathsf{case}(L, \lambda x : A.M, \lambda y : B.N)P \rightsquigarrow \mathsf{case}(L, \lambda x : A.MP, \lambda y : B.NP) \\
\mathsf{case}(\mathsf{inl}(L), \lambda x : A.M, \lambda y : B.N) \rightsquigarrow M[L/x] \\
\mathsf{case}(\mathsf{inr}(L), \lambda x : A.M, \lambda y : B.N) \rightsquigarrow N[L/y] \\
\mathsf{case}(\mathsf{case}(L, \lambda x : A.M, \lambda y : B.N), \lambda z : C.P, \lambda w : D.Q) \rightsquigarrow \\
\qquad\qquad\qquad \mathsf{case}(L, \quad \lambda x : A.\mathsf{case}(M, \lambda z : C.P, \lambda w : D.Q), \\
\qquad\qquad\qquad\qquad\qquad \lambda y : B.\mathsf{case}(N, \lambda z : C.P, \lambda w : D.Q))
\end{array}$$

The role of the conversion relation is to handle the equalities generated by equation Eta-Case for the particular case when the free subterm N is the first argument of a case analysis, taking into account the simplification of the new redexes using Beta-Case. Thus, deciding the equality of the critical pair given above will now be done by the conversion relation. The author defines a function to compute equivalent terms under the conversion relation, but we omit its presentation here. To show that two terms are equal, their normal forms according to the rewrite relation are first computed. Then, the respective sets of equivalent terms under the conversion relation are compared.

8.3 Normalization by Evaluation

Although rewriting is the most well-known technique to perform normalization and decide equality of λ-terms other approaches exist. Rewriting is intensional in the sense that it always operates at the syntactic level. In this section we briefly present *normalization by evaluation*, an extensional and reduction-free technique to normalize closed terms, first introduced by Berger and Schwichtenberg [BS91]. By extensional we mean that it operates on "semantic" values of a specific meta-language and returns a syntactic representation of their normal form (initially it was even presented as an inverse of the evaluation functional). It is reduction-free because all needed β-reductions will be implicitly carried out by the concrete implementation of the meta-language. This typically allows substantial performance gains when compared to rewriting.

The implementation of normalization by evaluation uses two type-dependent and mutually recursive functions. The first, typically called *reify* and denoted by $\downarrow$, maps values to syntactic terms, and the second, called *reflect* and denoted by $\uparrow$, converts terms back into values. Normalization of a term is achieved by reifying its value. We will now present the definition of these functions for the simply typed λ-calculus with products and terminal object introduced in Section 7.1. Syntactic terms will be denoted by the grammar introduced in that section. For values bold font will be used, with the keywords $\mathbf{pair}(\cdot,\cdot)$, $\mathbf{fst}$, and $\mathbf{snd}$ for pairs and $\mathbf{@}$ for application. α denotes an arbitrary base type.

$$\begin{array}{rcll}
\downarrow^{\alpha}\ v & = & v & \\
\downarrow^{1}\ v & = & \star & \\
\downarrow^{A\to B}\ v & = & \lambda x.\downarrow^{B}(v\mathbf{@}(\uparrow^{A}(x))) & \quad\text{with } x \text{ fresh}\\
\downarrow^{A\times B}\ v & = & \langle\downarrow^{A}(\mathbf{fst}(v)),\downarrow^{B}(\mathbf{snd}(v))\rangle &
\end{array}$$

$$\begin{array}{rcl}
\uparrow^{\alpha}\ t & = & t\\
\uparrow^{1}\ t & = & \boldsymbol{\star}\\
\uparrow^{A\to B}\ t & = & \boldsymbol{\lambda x.}\uparrow^{B}(t(\downarrow^{A}(\boldsymbol{x})))\\
\uparrow^{A\times B}\ t & = & \mathbf{pair}(\uparrow^{A}(\mathsf{fst}(t)),\uparrow^{B}(\mathsf{snd}(t)))
\end{array}$$

To exemplify the normalization process, consider an arbitrary value f of type $A\times A\to A\times A$, with A a base type. This value is reified as follows.

$$\begin{array}{rcl}
\downarrow^{A\times A\to A\times A}\ f & = & \lambda x.\downarrow^{A\times A}(\boldsymbol{f}\mathbf{@}\uparrow^{A\times A}(x))\\
& = & \lambda x.\downarrow^{A\times A}(\boldsymbol{f}\mathbf{@pair}(\uparrow^{A}(\mathsf{fst}(x)),\uparrow^{A}(\mathsf{snd}(x))))\\
& = & \lambda x.\langle\downarrow^{A}(\mathbf{fst}(\boldsymbol{f}\mathbf{@pair}(\mathsf{fst}(x),\mathsf{snd}(x)))),\downarrow^{A}(\mathbf{snd}(\boldsymbol{f}\mathbf{@pair}(\mathsf{fst}(x),\mathsf{snd}(x))))\rangle\\
& = & \lambda x.\langle\mathbf{fst}(\boldsymbol{f}\mathbf{@pair}(\mathsf{fst}(x),\mathsf{snd}(x))),\mathbf{snd}(\boldsymbol{f}\mathbf{@pair}(\mathsf{fst}(x),\mathsf{snd}(x)))\rangle
\end{array}$$

Consider now that the concrete value f to be normalized is the identity function. Then the meta-level interpreter would reduce the resulting term to the expected normal form $\lambda x.\langle\mathsf{fst}(x),\mathsf{snd}(x)\rangle$. But if f is the swap function then the normal form is $\lambda x.\langle\mathsf{snd}(x),\mathsf{fst}(x)\rangle$. To give another example, consider the normalization of the identity function of type $B^A\to B^A$. The meta-level reductions are performed as the calculation proceeds.

$$\begin{aligned}
\downarrow^{(A\to B)\to(A\to B)}(\boldsymbol{\lambda z.z}) &= \lambda x.\downarrow^{A\to B}((\boldsymbol{\lambda z.z})@\uparrow^{A\to B}(x)) \\
&= \lambda x.\downarrow^{A\to B}(\uparrow^{A\to B}(x)) \\
&= \lambda x.\downarrow^{A\to B}(\boldsymbol{\lambda z.}\uparrow^{B}(x\downarrow^{A}(\boldsymbol{z}))) \\
&= \lambda x.\downarrow^{A\to B}(\boldsymbol{\lambda z.}x\boldsymbol{z}) \\
&= \lambda x.\lambda y.\downarrow^{B}(\boldsymbol{\lambda z.}x\boldsymbol{z})@\uparrow^{A}(y)) \\
&= \lambda x.\lambda y.(\boldsymbol{\lambda z.}x\boldsymbol{z})@y \\
&= \lambda x.\lambda y.xy
\end{aligned}$$

Notice that the normal forms obtained with normalization by evaluation are exactly the same produced by the expansionary rewrite system presented in the last section.

The most well known application of normalization by evaluation is *type-directed partial evaluation* [Dan96], where this technique is used to efficiently decompile specialized programs. In this paper, Danvy already describes an extension to handle sum types, which unfortunately does not guarantee unique normal forms. The first proof of decidability for almost bicartesian closed categories using normalization by evaluation is due to Altenkirch, Dybjer, Hofmann, and Scott[ADHS01]. To guarantee unique normal forms, they defined a generalization of the case construct that allows simultaneous analysis of several terms. Interestingly, this parallel case had already been suggest by Ghani [Gha95] as a way to eliminate the conversion relation and directly obtain unique normal forms.

Unfortunately, although the proof presented in [ADHS01] is clearly constructive, the functions reflect and reify are presented in such an abstract way that makes the implementation of a concrete algorithm for normalization not obvious. Only very recently, Balat, Di Cosmo, and Fiore managed to present an extension to Danvy's original algorithm that performs extensional normalization with sums [BdCF04]. Although the definition of their normal forms was guided by [ADHS01], they use the standard case analysis construct. In practice this means that their algorithm only computes normal forms modulo a conversion relation, a situation that already arose in the expansionary rewrite system proposed by Ghani [Gha95].

Implementing normalization by evaluation in a statically typed language like Haskell is not trivial, due to the type-dependence of both reify and reflect. Some implementations have been proposed, namely in [Ros98] where the type-dependence is tackled by using type classes, and in [Rhi99] where an explicit syntactic characterization of types parameterizes both functions. This last technique is also used in the implementation proposed in [BdCF04] for OCaml [Rém02]. But in order to handle sums and to fix the relative positions between the case construct and lambda abstractions (the first should be lifted as high as possible), some sophisticated control operators had to be used. These operators, initially proposed in [GRR95], generalize both exceptions and continuations.

8.4 Normalization in the Point-Free Setting

An excellent study about rewriting in the point-free setting is due to Thérèse Hardin [Har89]. This subsumes another study published at the same time by Hirofumi Yokouchi [Yok89]. The main objective of this study was to provide a theoretical basis for the implementation of the categorical abstract machine, and it was thus mainly concerned with simulating β-reduction in the point-free setting. Its main result was that indeed it can be simulated by the application of the law

$$\mathsf{ap} \circ (\overline{f} \triangle g) = f \circ (\mathsf{id} \triangle g) \qquad \mathsf{Beta}$$

oriented from left to right, followed by rewriting with the following confluent system (denoted by Subst).

$$\begin{aligned}
&(f \circ g) \circ h \rightsquigarrow f \circ (g \circ h)\\
&\mathsf{id} \circ f \rightsquigarrow f\\
&f \circ \mathsf{id} \rightsquigarrow f\\
&\mathsf{fst} \circ (f \triangle g) \rightsquigarrow f\\
&\mathsf{snd} \circ (f \triangle g) \rightsquigarrow g\\
&(f \triangle g) \circ h \rightsquigarrow f \circ h \triangle g \circ h\\
&\mathsf{fst} \triangle \mathsf{snd} \rightsquigarrow \mathsf{id}\\
&\mathsf{fst} \circ f \triangle \mathsf{snd} \circ f \rightsquigarrow f\\
&\overline{f} \circ h \rightsquigarrow \overline{f \circ (h \circ \mathsf{fst} \triangle \mathsf{snd})}
\end{aligned}$$

Notice that, within the equational theory of cartesian closed categories, Beta is equivalent to Exp-Cancel, as the following calculations show.

$$\left[\begin{array}{ll}
 & \mathsf{ap} \circ (\overline{f} \triangle g)\\
= & \quad \{\, \mathsf{Prod\text{-}Absor} \,\}\\
 & \mathsf{ap} \circ (\overline{f} \times \mathsf{id}) \circ (\mathsf{id} \triangle g)\\
= & \quad \{\, \mathsf{Exp\text{-}Cancel} \,\}\\
 & f \circ (\mathsf{id} \triangle g)
\end{array}\right.
\qquad
\left[\begin{array}{ll}
 & \mathsf{ap} \circ (\overline{f} \times \mathsf{id})\\
= & \quad \{\, \mathsf{Prod\text{-}Def} \,\}\\
 & \mathsf{ap} \circ (\overline{f} \circ \mathsf{fst} \triangle \mathsf{snd})\\
= & \quad \{\, \mathsf{Exp\text{-}Fusion} \,\}\\
 & \mathsf{ap} \circ (\overline{f \circ (\mathsf{fst} \times \mathsf{id})} \triangle \mathsf{snd})\\
= & \quad \{\, \mathsf{Beta} \,\}\\
 & f \circ (\mathsf{fst} \times \mathsf{id}) \circ (\mathsf{id} \triangle \mathsf{snd})\\
= & \quad \{\, \mathsf{Prod\text{-}Absor},\ \mathsf{Prod\text{-}Reflex} \,\}\\
 & f
\end{array}\right.$$

Unfortunately, this study didn't take into account neither sums nor terminal object, and was carried out in an untyped setting. For example, Klop's counterexample is pointed out as a proof of non-confluence when Beta is added to Subst. To be a complete theory of cartesian closed categories, Subst still misses the equations that characterize the terminal object, namely Bang-Fusion and Bang-Reflex, and the remaining exponential laws, namely Beta and Exp-Reflex. Notice that, although both Prod-Reflex and Prod-Fusion can be replaced just by the following law (that corresponds to the pointwise Eta-Pair), all three must be added to the rewriting system in order to recover confluence.

$$\mathsf{fst} \circ f \vartriangle \mathsf{snd} \circ f = f \quad \Leftarrow \quad f : A \to B \times C \qquad \textsf{Eta-Prod}$$

Hardin achieves better confluence results by restricting the analysis to the subset of categorical terms that results from the translation, presented in Section 7.1, from a λ-calculus with products into cartesian closed categories. For this subset the confluence of Subst augmented with Beta was proved.

Interestingly, when trying to get a confluent system with similar power to Subst+Beta for unrestricted terms, Hardin suggests using Prod-Reflex as an expansion rule, but discards this hypothesis immediately because it has no operational sense (remember that the goal of the study was to provide a theoretical basis for the implementation of an abstract machine). A similar rule was also suggested by Yakouchi, but only to be applied to a curried function (in this case it is guaranteed that the domain is a pair).

$$\overline{f} \rightsquigarrow \overline{f \circ (\mathsf{fst} \vartriangle \mathsf{snd})}$$

Although Klop's counterexample is not valid in a typed setting, the critical pairs presented in Section 8.2, involving the terminal object, can be easily adapted to prove non-confluence of a contractive rewrite system for point-free combinators. First we have to be clear about what we mean by contractive. As seen above, products can be characterized either by Prod-Reflex and Prod-Fusion, or just by Eta-Prod. The same applies to exponentiation and to the terminal object, which means that Exp-Reflex, Exp-Fusion, Bang-Reflex, and Bang-Fusion can be replace by the following equations in a complete theory of cartesian closed categories.

$$\overline{\mathsf{ap} \circ (f \times \mathsf{id})} = f \quad \Leftarrow \quad f : A \to C^B \qquad \textsf{Eta-Exp}$$
$$f = !_A \quad \Leftarrow \quad f : A \to 1 \qquad \textsf{Eta-Bang}$$

A contractive system for point-free combinators results from orienting η-equations and all cancellation laws from left to right. Among the critical pairs that exist in this system we have, for example,

$$! \vartriangle \mathsf{snd} \circ f \xleftarrow{\textsf{Eta-Bang}} \mathsf{fst} \circ f \vartriangle \mathsf{snd} \circ f \xrightarrow{\textsf{Eta-Prod}} f$$

given $f : A \to 1 \times B$, and

$$\overline{!} \xleftarrow{\textsf{Eta-Bang}} \overline{\mathsf{ap} \circ (f \times \mathsf{id})} \xrightarrow{\textsf{Eta-Exp}} f$$

for $f : A \to 1^B$. Notice that in the first case $\mathsf{fst} \circ f : A \to \mathsf{1}$ and in the second $\mathsf{ap} \circ (f \times \mathsf{id}) : A \times B \to \mathsf{1}$. One could think that by using the reflex and fusion laws instead of these, a finer grain of control can be achieved, and these critical pairs avoided. For example, for products we can try to use a partially contractive system by orienting Prod-Reflex and Prod-Fusion as follows.

$$\begin{aligned}&(f \vartriangle g) \circ h \rightsquigarrow f \circ h \vartriangle g \circ h\\ &\mathsf{fst} \vartriangle \mathsf{snd} \rightsquigarrow \mathsf{id}\end{aligned}$$

Notice that the fusion law is being used as a safe expansion rule, since this orientation guarantees termination. Unfortunately, as already mentioned for the Subst system, this orientation creates the critical pair

$$\mathsf{fst} \circ f \vartriangle \mathsf{snd} \circ f \xleftarrow{\text{Prod-Fusion}} (\mathsf{fst} \vartriangle \mathsf{snd}) \circ f \xrightarrow{\text{Prod-Reflex}} f$$

whose elimination implies adding again Eta-Prod as a contraction to the system, making impossible to avoid the offending critical pairs involving the terminal object. Similar problems exist with exponentiation.

It seems reasonable to suggest that using Eta-Prod and Eta-Exp as expansions rules would result, similarly to what happens in the pointwise setting, in a decision procedure for cartesian closed categories. The critical pairs mentioned above would indeed disappear, but unfortunately new ones would appear. For example, given a function $f : A \times B \to C$ we would have

$$f \circ (\mathsf{fst} \vartriangle \mathsf{snd}) \xleftarrow{\text{Eta-Prod}} f \circ \mathsf{id} \xrightarrow{\text{Id-Nat}} f$$

The problem is that now we are dealing with functions, which are characterized not only by their range type but also by their domain type. As mentioned before, the role of expansionary rewrite rules is to make explicit the relation between a term and its type. But when using Eta-Prod and Eta-Exp as expansions only the type of the range becomes explicit. For instance, in a correct expansionary system, a function of type $f : A \times B \to C \times D$ should be rewritten into

$$(\mathsf{fst} \circ f \vartriangle \mathsf{snd} \circ f) \circ (\mathsf{fst} \vartriangle \mathsf{snd})$$

However, it is not clear how to define the rules of such a system, and even less clear how to state precisely the side conditions that would ensure termination. As expected, additional problems would arise from trying to handle sums. Let us recall the exchange law Abides.

$$(f \vartriangle g) \triangledown (h \vartriangle i) = (f \triangledown h) \vartriangle (g \triangledown i)$$

A correct derivation in an expansionary rewrite system would transform the left-hand side as follows.

$$
\begin{array}{ll}
 & (f \vartriangle g) \triangledown (h \vartriangle i) \\
\rightsquigarrow & \mathsf{fst} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \vartriangle \mathsf{snd} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \\
\rightsquigarrow & (\mathsf{fst} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \circ \mathsf{inl} \triangledown \mathsf{fst} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \circ \mathsf{inr}) \vartriangle \mathsf{snd} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \\
\rightsquigarrow & (f \triangledown h) \vartriangle \mathsf{snd} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \\
\rightsquigarrow & (f \triangledown h) \vartriangle (\mathsf{snd} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \circ \mathsf{inl} \triangledown \mathsf{snd} \circ ((f \vartriangle g) \triangledown (h \vartriangle i)) \circ \mathsf{inr}) \\
\rightsquigarrow & (f \triangledown h) \vartriangle (g \triangledown i)
\end{array}
$$

Unfortunately, by applying the same rewriting strategy to the right-hand side of the equation it would yield as normal form the left-hand side. This example seems to suggest that, perhaps unsurprisingly, in the presence of sums, an expansionary rewrite system would also not produce unique normal forms in the point-free setting. Probably, some kind of conversion relation is again needed, in which we guess the Abides law itself would play a key role.

In all this discussion the problems raised by the associativity of composition were ignored. For the cartesian subset, Hardin's research suggests that by orienting Comp-Assoc as

$$(f \circ g) \circ h \rightsquigarrow f \circ (g \circ h)$$

confluence is preserved. In fact, the offending critical pairs presented above are independent of this law. But in the presence of sums, this orientation would prevent some necessary applications of Sum-Fusion, thus leading to new critical pairs. In an expansionary system, treating composition as a binary combinator would, for example, also difficult the detection of projected or applied terms. In both cases, a better approach would be to use equational rewriting modulo associativity. Typically, this involves flattening the composition operator, and developing special matching algorithms. A possible implementation is described in Section 9.1, when presenting Bird's functional calculator.

8.5 Summary

In this chapter we have presented some theoretical results concerning the decidability of equality in almost bicartesian closed categories, both in the point-free and pointwise styles. The main conclusion of this study is that, so far, there is no evidence supporting the commonly accepted claim that the point-free style is more amenable to mechanization. In fact, the only known decision procedures are defined at the pointwise level, and we have shown some of the difficulties that arise when trying to implement a rewriting system to decide equality directly in the point-free style.

Chapter 9

Towards the Mechanization of Point-free Calculations

Very few theorem provers have been developed with the specific aim of reasoning about functional programs. Most never left the prototype stage, and practically all of them are tailored to reason about programs written in the pointwise style, and to interactive induction-based proofs. Among these we point out Sparkle [dMvEP01], a semi-automatic tactic-based theorem prover for Clean [PvE98], a lazy functional language very similar to Haskell. This is a full-featured tool, with a sophisticated graphical user-interface and a powerful hint mechanism to suggest tactics to the user, that can automatically prove most trivial goals. The reasoning process takes place in a subset of Clean, that shares the same lazy graph-rewriting semantics of the full language.

Although many authors argue that the point-free equational reasoning style is more amenable to mechanization, even fewer systems seem to have been developed using this approach. The development of a generic interactive *Equational Reasoning Assistant* for Haskell was initiated by Andy Gill [Gil95], and later continued by Noel Winstanley [Win98], but the project seems to have been abandoned. Anyway, its only goal was to assist the user during the interactive construction of the proof, and no attempt was made towards mechanization. In fact, to our knowledge, the only attempt to develop such a tool is due to Richard Bird, and is described as a programming project in the last chapter of his introductory book on Haskell [Bir98]. This system is based on a previous undocumented implementation by Mike Spivey.

This chapter starts precisely by presenting this system. We will then show how it can be modified in order to automate some of the calculations performed in the thesis. In particular, we show how a simple rewriting system can be used to simplify most of the point-free expressions that result from the derivation algorithm presented in Chapter 7. Our implementation differs from Bird's since we reuse Haskell pattern matching abilities, together with the *Scrap Your Boilerplate* generic programming library [LJ03] to simplify the implementation of rewriting strategies.

9.1 Birds's Functional Calculator

In Bird's functional calculator expressions can be variables (to be used in rewrite rules), constants, or can be built from simpler expressions using composition as the only distinguished combinator. No commitment is made to a particular set of constants, like the ones characterizing almost bicartesian closed categories. Not even identity gets a special treatment. All expressions are of functional type, and constants can be higher-order functions taking any number of (uncurried) arguments. However, no type-inference is ever performed, and thus the calculator cannot use type information to guide the rewriting process. This excludes, for example, any attempt to encode Eta-Prod as an expansion rule. The data type of expressions is defined as follows.

```
data Expr     = Var VarName | Con ConName [Expr] | Compose [Expr]
                deriving (Eq, Show)
type VarName = Char
type ConName = String
```

Notice the use of lists to encode constants with an arbitrary number of arguments. The intention to define a rewriting system modulo Comp-Assoc results in the use of lists to represent compositions of expressions. Of course, this data type *per se* does not guarantee the expected behavior of composition, and a couple of invariants must be imposed on it: the argument of `Compose` must be of length at least two, and it cannot contain an element also built with `Compose`. In order to guarantee this invariant, composition of expressions must always be defined using the following function instead of the constructor.

```
compose :: [Expr] -> Expr
compose xs = if null (tail xs) then head xs
             else Compose (concat (map decompose xs))


decompose :: Expr -> [Expr]
decompose (Var v)      = [Var v]
decompose (Con f xs)   = [Con f xs]
decompose (Compose xs) = xs
```

The equational theory that models the constants of the system will be captured by a set of (named) laws. A law is always oriented from left to right as a rewriting rule. Bird defined a parser that converts laws expressed in a user-friendly syntax, similar to the one used throughout this thesis, into the following abstract data type.

```
type Law     = (LawName, Expr, Expr)
type LawName = String
```

Typically, one is only interested in the final result of the rewriting process, but in order to inform the user of its progress, calculations are modeled by the following type that stores all intermediate steps.

```
type Calculation = (Expr, [Step])
type Step        = (LawName, Expr)
```

When rewriting finishes, calculations are rendered in a style similar to the one used in the proofs of this thesis, by interleaving expressions with the name of the appropriate law that allowed the calculation to progress.

The rewriting engine receives a list of laws and an expression in order to produce a calculation. The orientation of the laws is fixed by the user, but there are still some design decisions to be made: first, if two laws are applicable, which one will be chosen; second, in which order subexpressions will be matched against the laws; and third, should preference be given to laws or subexpressions when trying to progress with the calculation (i.e., should a law be tried exhaustively with every possible subexpression or vice-versa). The first decision is indirectly left to the user, since the system simply picks the first in the list. The searching of matching subexpressions is fixed from left-to-right (both in compositions and the arguments of constants) and from largest-to-smaller. This means that all subsegments of a composition are tried before the individual subexpressions. Concerning the last decision, Bird decided to partition the list of laws into two lists: the laws in the first will be tried in sequence with all possible subexpressions, and afterwards each subexpression is tried in sequence with each law of the second. Thus, the final type of the calculation function is

```
calculate :: ([Law], [Law]) -> Expr -> Calculation
```

The global list of laws provided by the user is automatically partitioned into the two lists. The first list will contain the laws that reduce the complexity of the expression, the so-called basic laws. As explained above, these will always be tried first during the rewriting process. The complexity measure is given by the number of subexpressions, as encoded by the following definition.

```
complexity :: Expr -> Int
complexity (Var v)      = 1
complexity (Con f xs)   = 1
complexity (Compose xs) = length xs
```

Function `calculate` is used both for the simplification of expressions and for proving equalities. In the later case, it is just applied to both expressions, and the final result compared with syntactic equality. The key part of the rewriting engine is an algorithm for matching modulo associativity. Most of it is a fairly standard accumulator-based matching algorithm, except for the interesting case of matching two compositions. Given two lists of lengths m and n respectively, if $m > n$ then they cannot be matched. If matching was modulo associativity and identity then this would not be the case, since some of the variables in the first list could be substituted by the identity function. If $m \leq n$ then each element of the first has to be matched with a non-empty segment of the second. The following function generates all possible pairs of expressions to be matched. If $m > n$ it returns an empty list.

```
align :: [Expr] -> [Expr] -> [[(Expr, Expr)]]
align xs ys = [zip xs (map compose zs) | zs <- partitions (length xs) ys]
```

Function `partitions` generates all possible partitions of the second list into m expressions, and has the following type.

```
partitions :: Int -> [a] -> [[[a]]]
```

Although simple, given the appropriate list of laws this calculator is powerful enough to automate many calculations, as the author demonstrates in [Bir98]. Of course, the user must be very careful with the order and the orientation of laws given to the calculator, since the tool does not attempt to perform any completion or termination checking. One of its major limitations is the absence of conditional rules, since it is not possible to express laws like Cata-Fusion.

As seen in Chapter 7, the expressions that result from the pointwise to point-free translation are quite complex. Naturally, some experiments were made to see if this calculator could be used to mechanize their simplification. Unfortunately, such attempts were not successful due to the absence of typing information and the fixed rewriting strategy. In the remaining of this chapter we show how to implement a prototype calculator without such limitations. We also show how it can be used to perform the desired simplifications. The generic programming library used in the implementation is first briefly reviewed.

9.2 Scrap Your Boilerplate

Most of the functional programs that operate on large data types typically perform relevant operations only at specific locations, but must have a lot of "boilerplate" code in order to handle the traversal through the data structure. *Scrap Your Boilerplate* (SyB) is a generic programming library developed by Lämmel and Peyton Jones [LJ03], whose goal is precisely to allow the specification of this boilerplate code once and for all, leaving for the programmer the implementation of just the interesting parts.

The generic programming technique supported by SyB has some nice properties:

- It is simple, in the sense that very few theoretical concepts have to be mastered in order to use it. The library includes also very few functions.

- It is general, since it can handle arbitrary data types, namely parameterized, mutually-recursive, and nested ones. It also subsumes other programming paradigms, such as term rewriting.
- It supports data type evolution, because very little code must be modified if a data type changes. In the Haskell distributions where it is natively supported, like GHC, no code has to be changed at all.

The implementation of this library required two Haskell extensions, both provided by most popular Haskell distributions. The first is rank-2 polymorphism, that will be used to implement generic traversals whose arguments must be polymorphic functions. The second is a type-safe cast with signature

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

that given an argument `x` of type `a`, compares the types `a` and `b` and returns `Just x` if they are equal, or `Nothing` otherwise. This function cannot be defined in standard Haskell. Class `Typeable` contains types for which a type representation can be obtained (to be compared for equality in `cast`), and its implementation can be automatically derived in GHC at least for all monomorphic types.

In order to understand precisely what is meant by boilerplate we will present a very simple example. Consider the following data type of geometrical figures, that allows figures to be built by grouping other figures.

```
data Point  = Point Int Int
data Figure = Circle Point Int
            | Rectangle Point Point
            | Group Name [Figure]
data Name   = Name String
```

A function that displaces a picture by a given vector can be defined as follows.

```
move :: Point -> Figure -> Figure
move d (Circle p r)    = Circle (moveP d p) r
move d (Rectangle p q) = Rectangle (moveP d p) (moveP d q)
move d (Group n l)     = Group n (map (move d) l)

moveP :: Point -> Point -> Point
moveP (Point dx dy) (Point x y) = Point (x+dx) (y+dy)
```

The only interesting part of this definition corresponds to function `moveP`, where the displacement of a point is defined. On the contrary, `move` is pure boilerplate code since it just encodes a standard traversal of the data type `Figure`.

9.2.1 Implementing Generic Transformations

In order to avoid the explicit definition of such boilerplate code, SyB provides a set of generic traversal combinators, such as

```
everywhere :: Data a => (forall b . Data b => b -> b) -> a -> a
```

that given a polymorphic transformation, applies it to every node in a data structure while traversing it in a bottom-up manner. The need for the `Data` constraint will be explained later. Notice the use of a rank-2 type to guarantee that the parameter is indeed polymorphic.

Taken alone, this function is not very useful. The only true polymorphic function of type `forall b . b -> b` is the identity function, which in turn implies that this traversal could only behave as the identity. However, using the `cast` function it is possible to define the following function, that transforms a monomorphic function of type `b -> b` into a full polymorphic one, that operates as the original when its argument is of type `b` and as the identity otherwise.

```
mkT :: (Typeable a, Typeable b) => (b -> b) -> a -> a
mkT f = case (cast f) of Just g  -> g
                         Nothing -> id
```

Equipped with this function it is now possible to define `move` just as follows.

```
move :: Point -> Figure -> Figure
move d = everywhere (mkT (moveP d))
```

When a node of type `Point` is found `moveP` is applied. In the remaining nodes no transformation is applied, because `mkT (moveP d)` behaves as the identity.

The question remains of how to define generically recursive traversals like `everywhere`. These will be defined using a small set of non-recursive one-layer traversals belonging to the class `Data`. One such traversal is `gmapT`, that given a polymorphic transformation applies it to all the immediate children of a value.

```
class Typeable a => Data a where
    gmapT :: (forall b . Data b => b -> b) -> a -> a
```

If one wants to use SyB with a particular type, it is necessary that this type belongs to class `Data`. For example, the instance declaration for `Figure` is as follows.

```
instance Data Figure where
    gmapT f (Circle p r)    = Circle (f p) (f r)
    gmapT f (Rectangle p q) = Rectangle (f p) (f q)
    gmapT f (Group n l)     = Group (f n) (f l)
```

For lists we have a similar definition.

```
instance Data a => Data [a] where
    gmapT f []     = []
    gmapT f (x:xs) = f x : f xs
```

From these examples, it is clear what was meant by one-layer and non-recursive – just compare this definition with that of the `map` function.

Using `gmapT` it is now possible to define different generic traversals. `everywhere` is defined as follows.

```
everywhere :: Data a => (forall b . Data b => b -> b) -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

For example, a top-down traversal could also be defined.

```
everywhere' :: Data a => (forall b . Data b => b -> b) -> a -> a
everywhere' f x = gmapT (everywhere' f) (f x)
```

Interestingly, this technique of defining recursive traversals in two steps, first a one-layer map and then tying the recursive knot, is similar to that of modeling a recursive data type as a fixed point of a functor, as presented in Chapter 2.

9.2.2 Generic Queries

The same design pattern can be applied to other kinds of boilerplate code. Consider, for example, a function that collects all names of grouped figures in a list. In this case, a generic query should be used instead of a generic transformation. First, a function to extend monomorphic queries into polymorphic ones is defined. It is necessary to specify a default value to return when the input is not of type `b`.

```
mkQ :: (Typeable a, Typeable b) => r -> (b -> r) -> a -> r
mkQ d q a = case (cast a) of Just b  -> q b
                             Nothing -> d
```

The class `Data` is then extended with a new non-recursive one-layer map, that collects in a list the results of querying all children.

```
class Typeable a => Data a where
    gmapT :: (forall b . Data b => b -> b) -> a -> a
    gmapQ :: (forall b . Data b => a -> r) -> a -> [r]
```

The definition of `gmapQ` is also very easy. For example, for figures the instance of the `Data` class can be extended as follows.

```
    gmapT f (Circle p r)    = [f p, f r]
    gmapT f (Rectangle p q) = [f p, f q]
    gmapT f (Group n l)     = [f n, f l]
```

Recursive querying traversals can be defined using gmapT. For example, the following combines the recursive results at each node using the first argument.

```
everything :: (r -> r -> r) -> (forall b . Data b => b -> r) -> a -> r
everything k f x = foldl k (f x) (gmapQ (everything k f) x)
```

The function that collects all names of grouped figures can now be defined as follows. Notice that only the interesting case of querying a value of type Name is specified.

```
names :: Figure -> [String]
names = everything (++) ([] ‘mkQ‘ aux)
    where aux (Name s) = [s]
```

9.2.3 Monadic Transformations

For the implementation of our point-free calculator SyB will be used to get rid of yet another kind of boilerplate code, where transformations are parameterized by a monad. To be more specific we are interested in a special kind of monad, whose interface is captured by the MonadPlus class.

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

There is no agreement about the laws these functions should verify, but to our purposes they should be interpreted as follows: mzero denotes failure or no result in a computation, and mplus a non-deterministic choice between two computations. A particular instance of this class – the data type Maybe – will be used to represent partial computations. In this case, mplus is biased towards the first argument.

```
instance MonadPlus Maybe where
    mzero             = Nothing
    Nothing ‘mplus‘ x = x
    x ‘mplus‘ _       = x
```

For this monad some additional functions can be defined, such as the guard function that, given a predicate, returns mzero if it is not verified.

```
guard :: MonadPlus m => Bool -> m ()
```

For example, suppose that we add squares to our figure data type. Assuming that a square is represented by a point and the size of its side, the new declaration is

```
data Figure = Circle Point Int
            | Rectangle Point Point
            | Square Point Int
            | Group Name [Figure]
```

Notice that previously defined functions using the traversal combinators, such as `move` and `names`, do not need to be changed in order to handle this new declaration. Suppose now that we want to define a function that traverses this data type and transforms one square `Rectangle` into a `Square`. Again, this function has a lot of boilerplate but the only interesting part can be encoded in the following monadic transformation. `fail` is a standard function of the `Monad` class, that for `MonadPlus` monads just returns `mzero`.

```
square :: Figure -> Maybe Figure
square (Rectangle (Point x1 y1) (Point x2 y2)) =
    do guard (abs (x2-x1) == abs (y2-y1))
       return (Square (Point (min x1 x2) (min y1 y2)) (abs (x2-x1)))
square _ = fail "Not a square!"
```

The following function will be used to convert a monomorphic transformation into a polymorphic one. In this case, `mzero` is returned if the type of the input value is different from `b`.

```
mkMp :: (MonadPlus m, Typeable a, Typeable b) => (b -> m b) -> a -> m a
```

A new one-layer map must also be added to the `Data` class.

```
gmapMo :: MonadPlus m => (forall b . Data b => b -> m b) -> a -> m a
```

This function applies a transformation to at most one immediate subterm. It returns `mzero` if the transformation can not be applied to any of them. `gmapMo` can be used to define a generic traversal that applies a transformation to at most one subterm of a data structure.

```
once :: (MonadPlus m, Data a) => (forall b . Data b => b -> m b) -> a -> m a
once f x = f x `mplus` (gmapMo (once f) x)
```

To square a single rectangle we can now use the following expression. If there are no square rectangles inside a figure it returns `Nothing`.

```
once (mkMp square)
```

It may seem that in order to benefit from SyB one has to implement a lot of one-layer maps for each data type. As seen in the examples, these are also pure boilerplate and, in fact, they can be defined as instances of a more fundamental traversal scheme, whose definition is quite intricate and will be omitted. As such, it suffices to implement this single traversal. Furthermore, in GHC this definition can be automatically derived, and thus SyB can be used for free.

9.3 A Typed Point-free Calculator

In this section we present a prototype rewriting system for typed point-free combinators. Although its design is inspired in Bird's functional calculator, presented in Section 9.1, the implementation is quite different and simpler due to the use of the SyB library. Our goal is to show that in practice, and in spite of the theoretical limitations pointed out in the previous chapter, it is very easy to define a working system that mechanizes some point-free calculations.

We begin by defining the abstract syntax data type that models our set of combinators. `BANG` represents !, and `ARROW` can be used to declare additional constants. Its usage will become clear later. Following Bird's design principle, composition is encoded using lists.

```
data Term = ID Type  | Comp [Term]
          | BANG Type
          | FST Type | SND Type | Term :/\: Term
          | INL Type | INR Type | Term :\/: Term
          | AP Type  | Curry Term
          | IN Type  | OUT Type
          | Hylo Type Term Term
          | ARROW Name Type
```

Since we are interested in typed terms, constants are tagged with their type. Types are represented by the following abstract syntax, where recursive data types are restricted to polynomial functors.

```
data Type = One | Type :*: Type | Type :+: Type | Type :-> Type | Mu Funct

data Funct = Id | Const Type | Funct :**: Funct | Funct :++: Funct
```

Type inference for point-free terms is encoded in the function

```
infer :: Term -> Maybe Type
```

The implementation of this function becomes very easy by using some special features of the `Maybe` monad, namely guards and pattern matching inside `do` expressions (when matching fails `Nothing` is immediately returned). For example the implementation for the split combinator is defined as follows.

```
infer (f :/\: g) = do (a :-> b) <- infer f
                      (c :-> d) <- infer g
                      guard (a == c)
                      return (a :-> (b :*: d))
```

Notice that variables are absent from the data type `Term`. The reason is that we will not develop a matching algorithm from scratch, but instead reuse the native matching mechanism of Haskell. Suppose that the type of rewriting rules was defined as

```
type Rule = Term -> Maybe Term
```

where `Maybe` is used to capture the fact that applying a rule may fail. One example of such a rule could be

```
prod_reflex :: Rule
prod_reflex (FST a :/\: SND _) = return (ID a)
prod_reflex _ = fail "Product reflexivity not applicable"
```

This definition assumes that input terms are correctly typed, and as such it is not necessary to check that the types of `FST` and `SND` are the same.

Strategies are defined as functions that combine and transform rules into new rules. Function `once` defined in Section 9.2.3 using SyB is an example of a fundamental strategy. Similarly to the figure example presented in that section, it is possible to define a rewrite rule to apply Prod-Reflex at most once inside a given term simply as follows.

```
once (mkMp prod_reflex)
```

Another useful strategy consists in trying to repeatedly apply a rule until it fails.

```
many :: Rule -> Rule
many r = (r ‘andthen‘ many r) ‘orelse‘ return
```

This definition uses the following basic combinators.

```
orelse :: Rule -> Rule -> Rule
orelse f g x = f x ‘mplus‘ g x


andthen :: Rule -> Rule -> Rule
andthen r g e = r e >>= g
```

The first tries to apply the first rule, and in case it fails tries to apply the second. The second applies the first rule, and then feeds its result into the second.

For example, using `many` it is possible to define a rewriting system that applies Prod-Reflex exhaustively inside a term.

```
many (once (mkMP prod_reflex))
```

Unfortunately, this approach does not allow a correct implementation of rules involving composition. For example, suppose Prod-Fusion was implemented as follows.

```
prod_fusion :: Rule
prod_fusion (Comp ((f :/\: g) : h : l)) =
    return (Comp ((Comp [f, h] :/\: Comp [g, h]) : l))
prod_fusion _ = fail "Product fusion not applicable"
```

In a rule like `once (mkMp prod_fusion)` fusion would only be tried at the beginning of compositions. In order to solve this problem, and still be able to reuse matching, the definition of `Rule` is generalized as follows.

```
type Rule = Data a => a -> Maybe a
```

Now it is possible to have rewrite rules with different input types. Of course, mkMp must be used in order to get the desired polymorphic type.

```
prod_reflex :: Rule
prod_reflex = mkMp aux
    where aux (FST a :/\: SND _) = return (ID a)
          aux _ = fail "Product reflexivity not applicable"

prod_fusion :: Rule
prod_fusion = mkMp aux
    where aux ((f :/\: g) : h : l) =
              return ((Comp [f, h] :/\: Comp [g, h]) : l)
          aux _ = fail "Product fusion not applicable"
```

By invoking once prod_reflex on a term, at most one subterm of type Term will be transformed. But if once prod_fusion is invoked, fusion will be tried in a subterm of type [Term]. Since the subterms of a list are its head and its tail, once will also try to apply its argument to all suffixes of a list. This means that prod_fusion will be attempted not only at the beginning of a composition, but also at arbitrary positions inside the list.

It may seem that, using the boilerplate library, we manage to reimplement Bird's calculator in a couple of lines, but a closer inspection reveals that the functionality we get is not quite the same. First, the matching algorithm is not as powerful as Bird's. For example, in the implementation of Prod-Fusion variable h will only be matched against a single term, while with Bird's matching algorithm it could be matched against any subsequence of the remaining list. In practice, for this particular equational theory, this is not a problem. This feature is only relevant for the fusion laws, and for those a single application to a sequence of functions can always be replaced by a sequence of applications to one function at the time.

Unlike Bird's calculator, our prototype does not store the intermediate steps of calculation, returning to the user just the final result. This a drawback when defining interactively the rewriting strategy to be used.

The more serious limitation is that the invariant of composition is not automatically guaranteed. Recall that the argument of Comp must be of length at least two, and it cannot contain an element also built with Comp. Given the rewrite rule

```
comp_inv :: Rule
comp_inv = (mkMp aux1) ‘orelse‘ (mkMp aux2)
    where aux1 (Comp [x]) = return x
          aux1 _          = fail "Not applicable"
          aux2 (Comp l : m) = return (l++m)
          aux2 _            = fail "Not applicable"
```

the following rule can be used to repair a term in order to satisfy the invariant.

```
repair :: Rule
repair = many (once comp_inv)
```

But when implementing an arbitrary rewriting strategy, namely one that uses rules that change the cardinality of a composition, the user must carefully introduce applications of `comp_inv` in order to maintain the invariant.

The major advantage of our prototype, besides typing information, is the ability to define flexible rewriting strategies, instead of having a fixed top-down left-to-right as in Bird's calculator. This advantage came for free from using the SyB library, and it was fundamental to implement the rewriting system presented in the next section.

9.4 Simplification of Terms Translated from Pointwise

As an example of using the above prototype calculator, we will show how to simplify terms obtained with the translation defined in Chapter 7. Since the goal is to simplify expressions, the fundamental technique will be a contractive rewriting system that covers the equational theory of almost bicartesian closed categories, with the exception of the equations characterizing the terminal object. This is a terminating, but not confluent system. However, since we are not interested in proving equalities, confluence is not a very relevant property.

Previous section already presented the implementation of some rewrite rules, namely Prod-Reflex and Prod-Fusion. The remaining can be implemented in a similar fashion. For example, Prod-Cancel and Eta-Prod are defined as follows.

```
prod_cancel :: Rule
prod_cancel = mkMp aux
    where aux (FST _ : (f :/\: g) : l) = return (f:l)
          aux (SND _ : (f :/\: g) : l) = return (g:l)
          aux _ = fail "Product cancellation not applicable"

eta_prod :: Rule
eta_prod = mkMp aux
    where aux (Comp (FST _ : f) :/\: Comp (SND _ : g)) =
              do guard (f == g)
                 return (Comp f)
          aux _ = fail "Surjective pairing not applicable"
```

The last rule shows another limitation of reusing the Haskell pattern matching mechanism. Since no repeated variables can appear in the left-hand side of a definition, it is necessary to use a guard to ensure that the rule can be applied.

The full contractive rewriting system can be implemented as follows. Notice that `comp_inv` is given the highest application priority so that the composition invariant is correctly maintained.

```
simplify :: Rule
simplify = many (once comp_inv    `orelse` once id_nat      `orelse`
                 once beta        `orelse` once exp_reflex  `orelse`
                 once exp_fusion  `orelse` once eta_exp     `orelse`
                 once prod_cancel `orelse` once prod_reflex `orelse`
                 once prod_fusion `orelse` once eta_prod    `orelse`
                 once sum_cancel  `orelse` once sum_reflex  `orelse`
                 once sum_fusion  `orelse` once eta_sum)
```

By careful inspection of the examples presented Chapter 7, namely Example 7.8 concerning the length function, it can be seen that the key move in the simplification process is the application of law Either-Const as the following rewrite rule.

$$\mathsf{either} \circ (\overline{f \circ \mathsf{snd}} \bigtriangleup \overline{g \circ \mathsf{snd}}) \rightsquigarrow \overline{(f \bigtriangledown g) \circ \mathsf{snd}}$$

To identify possible applications of this rule it is better to consider either as constant. In fact, its definition was only expanded in more contrived examples, namely in Example 7.11 concerning distr. As such, the `ARROW` constructor will be used to encode this function, which means that the above rewrite rule is implemented as follows.

```
either_const :: Rule
either_const = mkMp aux
    where aux (ARROW "either" _ : (Curry (Comp f) :/\: Curry (Comp g)) : l) =
              do (SND (c :*: a)) <- return (last f)
                 (SND (_ :*: b)) <- return (last g)
                 let expr = Comp (init f) :\/: Comp (init g)
                 return (Curry (Comp [expr, SND (c :*: (a :+: b))]) : l)
          aux _ = fail "Either const not applicable"
```

This implementation assumes that terms type-check correctly before being subject to simplification, and so the type of either is not verified. Again, due to pattern matching limitations, it is necessary to explicitly access the last expression in a composition in the right-hand side of the rule definition.

Unfortunately, after translating pointwise definitions into the point-free style it is not usually possible to apply this rule immediately. One of the reasons is the existence of "uninformative" arrows to the terminal object, that using Eta-Bang can be replaced by more useful ones. In particular, this is the case of arrows $!_{A\times 1}$ that can be replaced by snd, thus increasing the chances to apply Either-Const. The problem of implementing Eta-Bang as an expansion is that a useful function has to be guessed by looking at the type. This cannot be done in general, but for this particular application the following implementation suffices. Notice that this is the only rule for which type information is necessary.

```
eta_bang :: Rule
eta_bang = mkMp aux
    where aux (BANG One)         = return (ID One)
          aux (BANG (a :*: One)) = return (SND (a :*: One))
          aux (BANG (One :*: a)) = return (FST (One :*: a))
          aux _ = fail "Bang expansion not possible"

bangs :: Rule
bangs = many (once eta_bang)
```

In some examples it is still necessary to perform some additional simplifications in order to be able to apply Either-Const: cancellation of products and exponentials, and application of Prod-Fusion as a rewrite rule oriented from right to left. Since this last rule tries to break a split into a composition of smaller expressions it is denoted as product fission. These simplifications are encoded in the following rewriting system.

```
reduce :: Rule
reduce = many (once comp_inv    `orelse` once beta `orelse`
               once prod_cancel `orelse` once prod_fission)
```

The correct implementation of product fission is a bit tedious because it is necessary to implement matching modulo identity. Notice the insertion of a stop condition to avoid infinite applications of the rule.

```
prod_fission :: Rule
prod_fission = mkMp aux
    where aux (Comp f :/\: Comp g) =
              do guard (last f == last g)
                 return (Comp [Comp (init f) :/\: Comp (init g), last f])
          aux (ID _ :/\: ID _) = fail "Cannot perform fission for ever"
          aux (Comp f :/\: g) =
              do guard (last f == g)
                 (_ :-> a) <-  infer g
                 return (Comp [Comp (init f) :/\: ID a, g])
          aux (f :/\: Comp g) =
              do guard (last g == f)
                 (_ :-> a) <-  infer f
                 return (Comp [ID a :/\: Comp (init g), f])
          aux (f :/\: g) =
              do guard (f == g)
                 (_ :-> a) <-  infer f
                 return (Comp [ID a :/\: ID a, f])
          aux _ = fail "Product fission not applicable"
```

After one application of Either-Const it may be necessary to simplify the term before trying to apply it again. As such, the strategy that was used to eliminate either interleaves rewriting with the system `reduce` with applications of `either-const`.

```
eithers :: Rule
eithers = many (reduce ‘andthen‘ once either_const)
```

Finally, all these systems are combined in order to define the following rewriting system.

```
bangs ‘andthen‘ eithers ‘andthen‘ simplify
```

This system simplifies correctly all the examples presented in Chapter 7, with the exception of Example 7.11 (which implies expanding the definition of either). Notice that the abstract syntax defined above does not include the derived combinators for products, sums, and exponentials. It is very simple to extend it, and to define a strategy to insert them in the resulting expressions. By doing so, we would get exactly the same final definitions as in the mentioned examples.

9.5 Summary

This chapter shows that, in practice, despite all the negative results pointed out in Chapter 8, it is possible to implement useful rewriting systems for point-free expressions. For example, given some user guidance, Bird's functional calculator can automate many of the proofs presented in this thesis, provided no type information and conditional rules (such as fusion) are needed. Building on this experience, we developed a prototype of a typed functional calculator, on which we managed to implement a rewriting system that, thanks to type information, is able to effectively simplify most of the point-free expressions obtained with the translation defined in Chapter 7. The implementation of this prototype was very simple thanks to the SyB generic programming library.

Chapter 10

Conclusions and Future Work

This thesis advocates a methodology for equational reasoning about functional programs, where functions should first be converted into a point-free style of programming with recursion patterns before being subject to calculations. Since recursion patterns are already well studied, we have focused on issues related to the point-free style. Within this context, our main contributions are:

- [Chapter 4] A framework for performing point-free calculations with higher-order functions based on the internalization of some basic combinators. This framework was used to implement Bird's accumulation strategy using a pure point-free style. Without internalizations, such point-free calculations would be considerably longer, and less attractive when compared to the equivalent pointwise ones.

- [Chapters 7, 9] A mechanism to translate pointwise code into the point-free style, that can be applied to a λ-calculus rich enough to represent the core functionality of a real functional programming language. The translation is based on the well-known equivalence between simply typed λ-calculi and cartesian closed categories suggested by Lambek. This equivalence was extended in order to cover sums and the fixpoint operator. Although none of its components is completely new, it is the first time they are put together in order to build a complete translation. In Chapter 9 it is also shown how to use rewriting to simplify the expressions that result from this translation. Interestingly, the effectiveness of this translation results from the use of internalizations in order to give a more concise encoding for sums.

- [Chapter 6] Pointless Haskell – a library that enables programming in a pure point-free style within Haskell. The utility of having such a library can be questioned given that we have proposed a methodology based on using point-free exclusively for calculations, but when proving by hand it is very useful to be able to understand or type check the expressions being manipulated. We have used it precisely for that purpose in some calculations in the thesis. Since it is embedded in Haskell, it also allows one to take any existent program and translate into point-free just the definitions that will be subject to calculation. When combined with the mechanism for deriving point-free hylomorphisms, this library can also be used for program understanding purposes: given a recursive definition it allows one to graphically visualize its recursion tree, since it corresponds to the intermediate data structure of the derived hylomorphism. It is also useful to identify possible applications of the Hylo-Shift law since it shows unevaluated thunks: these usually suggest possible simplifications of the intermediate data structure, by shifting work from the catamorphism into the anamorphism.

A final contribution of the thesis is an accurate picture of the state of the art of reasoning by calculation in the point-free style, which hopefully clarifies some common misconceptions. Particularly relevant is the study presented in Chapter 8, that somehow contradicts the general assumption that calculations in this style are more amenable to mechanization than those in pointwise. As seen in Chapter 9 with Bird's functional calculator, and our own prototype system, it is indeed quite easy to implement a rewrite system for point-free expressions. However, the only known procedures to decide equality in almost bicartesian closed categories (whose equational theory equals that of our basic set of combinators, if strictness is not taken into account) are defined for pointwise terms. Similarly, as shown in Chapter 5, in the context of program transformation using short-cut fusion almost all the implemented systems operate on pointwise definitions, and the only known system designed for point-free is a prototype and is not based on solid theoretical foundations.

On Strictness

Although some progress has been made in this work with the use of the left-strictness concept, one problem of our approach is that the treatment of strictness issues is still not satisfactory. This problem is particularly evident in the uniqueness laws. The strictness side conditions that characterize some of these laws restrict their application to strict functions, which means that reasoning about full **CPO** must be taken with special care. In practice, this means that uniqueness laws may not give a precise characterization of a recursion pattern, and proofs using these laws may not exist. In Section 3.1 we gave an example of this problem, by showing that the fusion law for catamorphisms cannot be derived from its uniqueness. Similar problems occur with other laws. For example, in Section 3.1.1, we presented a proof of Cata-Map-Fusion that was based on the hylomorphism definition of catamorphisms and the Hylo-Shift law. The traditional proof of Cata-Map-Fusion in **Set** is based on Cata-Fusion. If the same tactic was attempted in **CPO**, unnecessarily strictness side conditions would be required, as the following calculation shows.

$$
\begin{array}{ll}
 & \llparenthesis f \rrparenthesis_{TB} \circ T\ g \\
= & \quad \{\ \mathsf{Map\text{-}Def}\ \} \\
 & \llparenthesis f \rrparenthesis_{TB} \circ \llparenthesis \mathsf{in}_T \circ (g \star_T \mathsf{id}) \rrparenthesis_{TA} \\
= & \quad \{\ \mathsf{Cata\text{-}Fusion},\ \mathsf{Cata\text{-}Strict},\ f \text{ strict}\ \} \\
 & \left[\begin{array}{ll}
 & \llparenthesis f \rrparenthesis_{TB} \circ \mathsf{in}_T \circ (g \star_T \mathsf{id}) \\
= & \quad \{\ \mathsf{Cata\text{-}Cancel}\ \} \\
 & f \circ (\mathsf{id} \star_T \llparenthesis f \rrparenthesis_{TB}) \circ (g \star_T \mathsf{id}) \\
= & \quad \{\ \mathsf{Bifunctor\text{-}Comp}\ \} \\
 & f \circ (g \star_T \mathsf{id}) \circ (\mathsf{id} \star_T \llparenthesis f \rrparenthesis_{TB})
\end{array}\right. \\
 & \llparenthesis f \circ (g \star_T \mathsf{id}) \rrparenthesis_{TA}
\end{array}
$$

This problem happens in many other laws like, for example, Cata-Split, where the traditional proof using Cata-Uniq would imply the strictness of both catamorphisms, or the proof of the last fusion law concerning accumulations (presented in Section 3.5).

But always relying on the hylomorphism definition may also not be a good proof strategy. For example, although the definition using hylomorphisms is more understandable, for paramorphisms and accumulations we had to resort to their definition using catamorphisms in order to prove uniqueness. Since no direct proof was found in terms of the definition and the basic set of laws about hylomorphisms, this seems to suggest that this set may not be a complete characterization of this recursion pattern, and should be extended. However, when trying to do the same proofs using directly the fixpoint definition and fixpoint induction we faced similar problems: for example, the suggestion given in [Mei92] for proving paramorphism uniqueness using direct fixpoint induction clearly does not work.

Future Work

Naturally, the problems identified above point to some of the most interesting theoretical tasks to pursue in the future.

Mechanization. In the context of the proposed reasoning methodology, the most important task concerns mechanization. First, it would be desirable to find a direct decision procedure for almost bicartesian closed categories using point-free combinators. Chapter 8 raises some problems that occur when trying to define an expansionary rewrite system in this setting, but also gives some suggestions that deserve further investigation: to expand based on both the domain and range of an arrow, to define normal forms up to the Abides law, and to use rewriting modulo associativity of the composition operator. Second, in order to implement fusion, it is necessary to define automatic procedures to solve for unknowns in equations. As seen in Chapter 5, at the pointwise level, the fusion side-condition determines a higher-order matching problem, that has already been shown to be decidable under restrictions broad enough to cover most interesting problems. Unfortunately, this approach does not make sense at the point-free level. We intend to explore the use of *narrowing* to tackle this problem [Pla93]: narrowing extends rewriting precisely by allowing variables in the terms subject to manipulation, and is implemented using unification instead of pattern matching.

Strictness. Concerning strictness, it is important to study changes in the base category in order to better suit the Haskell semantic domain, where all types are lifted. But prior to this, it is necessary to untangle the mess concerning the strictness side-conditions in the laws that characterize recursion patterns. Specially useful would be the existence of uniqueness laws that can be used to reason about non-strict functions as well. In the seminal "bananas paper" [MFP91] that pioneered the point-free calculus in **CPO**, Meijer, Fokkinga, and Patterson defined the uniqueness law for catamorphisms as follows.

$$f = (\!|g|\!)_{\mu F} \quad \Leftrightarrow \quad f \circ \mathsf{in} = g \circ F\ f \ \wedge\ f \circ \bot = (\!|g|\!)_{\mu F} \circ \bot$$

Although this version of the law covers non-strict functions, it has been subsequently discarded by most authors in favor of the one presented in this thesis, because it is very difficult to verify in practice the side-condition concerning bottom, unless the functions are indeed strict. We intend to study if this version of the law can help with the above mentioned problems. For example, so far it is not clear how it can be used to prove Cata-Map-Fusion without imposing strictness side conditions.

Internalizations. The use of internalizations turned out to be a key aspect of the thesis. They were used in different contexts, namely to simplify proofs about higher-order features and to simplify the translation from pointwise to point-free. Since we are now using internalized versions of the basic combinators, it is also useful to have "internalized" versions of the laws that characterize them, in order to avoid working directly with the definitions. This was the case in Chapter 7, where we used this technique in order to prove the soundness of the pointwise to point-free translation concerning sums. We think this approach can be a step forward towards an even purer point-free calculus, in the sense that variables are also eliminated from laws, and as such it deserves further investigation. Going in the opposite direction, we also think that the concept of point can be used to smoothly integrate pointwise features in our calculus. For example, they can be used to show (using point-free calculations) that the combinators have the expected pointwise behavior. We intend to explore their usage in proofs where the pointwise style is clearly more practical, namely those concerning bookkeeping functions.

Inwards Fusion. Chapter 4 shows how to apply Bird's accumulation strategy using pure point-free calculations. We intend to apply the same methodology to other program transformation techniques, namely in the elimination of nested recursion patterns. Some progress has already been made in establishing laws for this kind of "inwards fusion". A simple example that is already covered is the transformation of the following specification of the list filtering function, that uses an auxiliary recursive function.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p l = case (until p l) of [] -> []
                                 (x:xs) -> x:(filter p xs)

until :: (a -> Bool) -> [a] -> [a]
until p [] = []
until p (x:xs) | p x       = x:xs
               | otherwise = until p xs
```

This specification can be converted into the following direct recursive function.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x:(filter p xs)
                | otherwise = filter p xs
```

In this example no gains in efficiency are achieved, but we have already studied other examples where that happens, such as the classic example of computing a list of factorials, first optimized by Burstall and Darlington using fold/unfold transformations [BD77].

Tools. On the practical side there are also some tasks to be pursued. First, the prototype implementation [Pro95] of the pointwise to point-free translation mechanism must be improved. Given the λ-calculus defined in Chapter 7, with the extensions to structured types and pattern matching, it will be possible to use this tool to convert a subset of Haskell into the point-free style. The resulting code can be executed using the Pointless Haskell library presented in Chapter 6. Besides deploying it as a stand-alone tool, we intend to include the translation as a refactoring in the *Haskell Refactorer* project [LRT03], so that programmers can interactively convert between both styles within they favorite text editor. Second, we also intend to upgrade our prototype typed point-free calculator presented in Chapter 9 into a rewriting tool that operates on concrete point-free syntax, combining the *active source* concept of MAG with the flexibility to define various rewriting strategies using a simple tactic language.

Visual Programming. In this thesis we do not advocate the direct use of the point-free style in programming, since definitions can easily become very long and impossible to understand. This is a consequence of the need to frequently use bookkeeping expressions to rearrange values, and the "unnatural" use of a textual representation for point-free programs. Both problems could be alleviated by using some sort of visual programming language, since the combinators can be better understood as connectors in a kind of data-flow diagram, and bookkeeping can easily be defined in a "play with re-wiring" style. Preliminary studies, carried out to assess the viability of such a language, suggest promising applications of graph rewriting in the manipulation of such diagrams, for example in the conversion of "pointwise like" re-wiring graphs into pure point-free diagrams. One of the biggest challenges of this task, is how to visually represent in a natural and understandable way the higher-order features of the language.

Appendix A

Additional Laws and Proofs

$$f \circ \mathsf{ap} = \mathsf{ap} \circ (f^{\bullet} \times \mathsf{id}) \qquad \text{Ap-Nat}$$

$$\begin{array}{ll}
 & \mathsf{ap} \circ (f^{\bullet} \times \mathsf{id}) \\
= & \quad \{\, \text{Exp-Def} \,\} \\
 & \mathsf{ap} \circ (\overline{f \circ \mathsf{ap}} \times \mathsf{id}) \\
= & \quad \{\, \text{Exp-Cancel} \,\} \\
 & f \circ \mathsf{ap}
\end{array}$$

$${}^{\bullet}f \circ \overline{g} = \overline{g \circ (\mathsf{id} \times f)} \qquad \text{Pxe-Absor}$$

$$\begin{array}{ll}
 & {}^{\bullet}f \circ \overline{g} \\
= & \quad \{\, \text{Pxe-Def} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times f)} \circ \overline{g} \\
= & \quad \{\, \text{Exp-Fusion} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{id} \times f) \circ (\overline{g} \times \mathsf{id})} \\
= & \quad \{\, \text{Prod-Functor-Comp, Exp-Cancel} \,\} \\
 & \overline{g \circ (\mathsf{id} \times f)}
\end{array}$$

$$\mathsf{split} \circ (\underline{f} \triangle \underline{g}) = \underline{f \triangle g} \qquad \text{Split-Pnt}$$

$$\begin{array}{ll}
 & \mathsf{split} \circ (\underline{f} \triangle \underline{g}) \\
= & \quad \{\, \text{Split-Def} \,\} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} \triangle \mathsf{snd} \times \mathsf{id})} \circ (\underline{f} \triangle \underline{g}) \\
= & \quad \{\, \text{Exp-Fusion} \,\} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} \triangle \mathsf{snd} \times \mathsf{id}) \circ ((\underline{f} \triangle \underline{g}) \times \mathsf{id})} \\
= & \quad \{\, \text{Prod-Fusion, Prod-Functor-Comp, Prod-Cancel} \,\} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ (\underline{f} \times \mathsf{id} \triangle \underline{g} \times \mathsf{id})} \\
= & \quad \{\, \text{Pnt-Def, Prod-Absor, Exp-Cancel} \,\} \\
 & \overline{f \circ \mathsf{snd} \triangle g \circ \mathsf{snd}} \\
= & \quad \{\, \text{Prod-Fusion, Pnt-Def} \,\} \\
 & \underline{f \triangle g}
\end{array}$$

$$\mathsf{split} \circ (\mathsf{fst}^{\bullet} \triangle \mathsf{snd}^{\bullet}) = \mathsf{id} \qquad \text{Split-Iso-Left}$$

$$\begin{array}{ll}
 & \mathsf{split} \circ (\mathsf{fst}^{\bullet} \vartriangle \mathsf{snd}^{\bullet}) \\
= & \quad \{\,\mathsf{Split\text{-}Def}\,\} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} \vartriangle \mathsf{snd} \times \mathsf{id})} \circ (\mathsf{fst}^{\bullet} \vartriangle \mathsf{snd}^{\bullet}) \\
= & \quad \{\,\mathsf{Exp\text{-}Fusion}\,\} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} \vartriangle \mathsf{snd} \times \mathsf{id}) \circ ((\mathsf{fst}^{\bullet} \vartriangle \mathsf{snd}^{\bullet}) \times \mathsf{id})} \\
= & \quad \{\,\mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Prod\text{-}Cancel}\,\} \\
 & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ (\mathsf{fst}^{\bullet} \times \mathsf{id} \vartriangle \mathsf{snd}^{\bullet} \times \mathsf{id})} \\
= & \quad \{\,\mathsf{Exp\text{-}Def},\ \mathsf{Prod\text{-}Absor},\ \mathsf{Exp\text{-}Cancel}\,\} \\
 & \overline{\mathsf{fst} \circ \mathsf{ap} \vartriangle \mathsf{fst} \circ \mathsf{ap}} \\
= & \quad \{\,\mathsf{Prod\text{-}Fusion},\ \mathsf{Prod\text{-}Reflex}\,\} \\
 & \overline{\mathsf{ap}} \\
= & \quad \{\,\mathsf{Exp\text{-}Reflex}\,\} \\
 & \mathsf{id}
\end{array}$$

$$(\mathsf{fst}^{\bullet} \vartriangle \mathsf{snd}^{\bullet}) \circ \mathsf{split} = \mathsf{id} \qquad \mathsf{Split\text{-}Iso\text{-}Right}$$

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

$$\mathsf{distl} \circ (\mathsf{inr} \times \mathsf{id}) = \mathsf{inr} \ \wedge\ \mathsf{distl} \circ (\mathsf{inl} \times \mathsf{id}) = \mathsf{inl} \qquad \mathsf{Distl\text{-}Cancel}$$

$$\begin{array}{ll}
 & \mathsf{distl} \circ (\mathsf{inl} \times \mathsf{id}) \\
= & \quad \{\,\mathsf{Distl\text{-}Def}\,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ (\mathsf{inl} \times \mathsf{id}) \\
= & \quad \{\,\mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Sum\text{-}Cancel}\,\} \\
 & \mathsf{ap} \circ (\overline{\mathsf{inl}} \times \mathsf{id}) \\
= & \quad \{\,\mathsf{Exp\text{-}Cancel}\,\} \\
 & \mathsf{inl}
\end{array}$$

$$\begin{array}{ll}
 & \mathsf{distl} \circ (\mathsf{inr} \times \mathsf{id}) \\
= & \quad \{\,\mathsf{Distl\text{-}Def}\,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ (\mathsf{inr} \times \mathsf{id}) \\
= & \quad \{\,\mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Sum\text{-}Cancel}\,\} \\
 & \mathsf{ap} \circ (\overline{\mathsf{inr}} \times \mathsf{id}) \\
= & \quad \{\,\mathsf{Exp\text{-}Cancel}\,\} \\
 & \mathsf{inr}
\end{array}$$

$$\mathsf{distr} \circ (\mathsf{id} \times \mathsf{inr}) = \mathsf{inr} \ \wedge\ \mathsf{distr} \circ (\mathsf{id} \times \mathsf{inl}) = \mathsf{inl} \qquad \mathsf{Distr\text{-}Cancel}$$

$$\begin{array}{ll}
 & \mathsf{distr} \circ (\mathsf{id} \times \mathsf{inl}) \\
= & \quad \{\, \mathsf{Distr\text{-}Def} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{distl} \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{inl}) \\
= & \quad \{\, \mathsf{Swap\text{-}Nat},\ \mathsf{Distl\text{-}Cancel} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{inl} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Sum\text{-}Def},\ \mathsf{Sum\text{-}Cancel} \,\} \\
 & \mathsf{inl} \circ \mathsf{swap} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Swap\text{-}Iso} \,\} \\
 & \mathsf{inl}
\end{array}$$

$$\begin{array}{ll}
 & \mathsf{distr} \circ (\mathsf{id} \times \mathsf{inr}) \\
= & \quad \{\, \mathsf{Distr\text{-}Def} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{distl} \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{inr}) \\
= & \quad \{\, \mathsf{Swap\text{-}Nat},\ \mathsf{Distl\text{-}Cancel} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{inr} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Sum\text{-}Def},\ \mathsf{Sum\text{-}Cancel} \,\} \\
 & \mathsf{inr} \circ \mathsf{swap} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Swap\text{-}Iso} \,\} \\
 & \mathsf{inr}
\end{array}$$

$$(\mathsf{fst} + \mathsf{fst}) \circ \mathsf{distl} = \mathsf{fst} \qquad \mathsf{Distl\text{-}Fst}$$

$$\begin{array}{ll}
 & (\mathsf{fst} + \mathsf{fst}) \circ \mathsf{distl} \\
= & \quad \{\, \mathsf{Distl\text{-}Def} \,\} \\
 & (\mathsf{fst} + \mathsf{fst}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \mathsf{Ap\text{-}Nat} \,\} \\
 & \mathsf{ap} \circ ((\mathsf{fst} + \mathsf{fst})^{\bullet} \times \mathsf{id}) \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Functor\text{-}Comp},\ \mathsf{Sum\text{-}Fusion},\ (\mathsf{fst} + \mathsf{fst})^{\bullet}\ \mathrm{strict} \,\} \\
 & \mathsf{ap} \circ (((\mathsf{fst} + \mathsf{fst})^{\bullet} \circ \overline{\mathsf{inl}} \triangledown (\mathsf{fst} + \mathsf{fst})^{\bullet} \circ \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \mathsf{Exp\text{-}Absor},\ \mathsf{Sum\text{-}Def},\ \mathsf{Sum\text{-}Cancel} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl} \circ \mathsf{fst}} \triangledown \overline{\mathsf{inr} \circ \mathsf{fst}}) \times \mathsf{id}) \\
= & \quad \{\, \mathsf{Prod\text{-}Cancel},\ \mathsf{Prod\text{-}Def} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{fst} \circ (\mathsf{inl} \times \mathsf{id})} \triangledown \overline{\mathsf{fst} \circ (\mathsf{inr} \times \mathsf{id})}) \times \mathsf{id}) \\
= & \quad \{\, \mathsf{Exp\text{-}Fusion} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{fst}} \circ \mathsf{inl} \triangledown \overline{\mathsf{fst}} \circ \mathsf{inr}) \times \mathsf{id}) \\
= & \quad \{\, \mathsf{Sum\text{-}Fusion},\ \overline{\mathsf{fst}}\ \mathrm{strict},\ \mathsf{Sum\text{-}Reflex} \,\} \\
 & \mathsf{ap} \circ (\overline{\mathsf{fst}} \times \mathsf{id}) \\
= & \quad \{\, \mathsf{Exp\text{-}Cancel} \,\} \\
 & \mathsf{fst}
\end{array}$$

$$(\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distr} = \mathsf{snd} \qquad \mathsf{Distr\text{-}Snd}$$

$$\begin{array}{ll}
 & (\mathsf{snd} + \mathsf{snd}) \circ \mathsf{distr} \\
= & \quad \{\, \mathsf{Distr\text{-}Def} \,\} \\
 & (\mathsf{snd} + \mathsf{snd}) \circ (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{distl} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Sum\text{-}Functor\text{-}Comp},\ \mathsf{Swap\text{-}Def},\ \mathsf{Prod\text{-}Cancel} \,\} \\
 & (\mathsf{fst} + \mathsf{fst}) \circ \mathsf{distl} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Distl\text{-}Fst} \,\} \\
 & \mathsf{fst} \circ \mathsf{swap} \\
= & \quad \{\, \mathsf{Swap\text{-}Def},\ \mathsf{Prod\text{-}Cancel} \,\} \\
 & \mathsf{snd}
\end{array}$$

$$\mathsf{distl} \circ ((f + g) \times h) = (f \times h + g \times h) \circ \mathsf{distl} \qquad \mathsf{Distl\text{-}Nat}$$

$$
\begin{array}{ll}
 & (f \times h + g \times h) \circ \mathsf{distl} \\
= & \quad \{\, \text{Distl-Def, Ap-Nat} \,\} \\
 & \mathsf{ap} \circ ((f \times h + g \times h)^{\bullet} \times \mathsf{id}) \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \text{Prod-Functor-Comp, Sum-Fusion, } (f \times h + g \times h)^{\bullet} \text{ strict} \,\} \\
 & \mathsf{ap} \circ (((f \times h + g \times h)^{\bullet} \circ \overline{\mathsf{inl}} \triangledown (f \times h + g \times h)^{\bullet} \circ \overline{\mathsf{inr}}) \times \mathsf{id}) \\
= & \quad \{\, \text{Exp-Absor, Sum-Def, Sum-Cancel} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl} \circ (f \times h)} \triangledown \overline{\mathsf{inr} \circ (g \times h)}) \times \mathsf{id}) \\
= & \quad \{\, \text{Prod-Functor-Comp, Exp-Fusion} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl} \circ (\mathsf{id} \times h)} \circ f \triangledown \overline{\mathsf{inr} \circ (\mathsf{id} \times h)} \circ g) \times \mathsf{id}) \\
= & \quad \{\, \text{Sum-Absor, Prod-Functor-Comp} \,\} \\
 & \mathsf{ap} \circ ((\overline{\mathsf{inl} \circ (\mathsf{id} \times h)} \triangledown \overline{\mathsf{inr} \circ (\mathsf{id} \times h)}) \times \mathsf{id}) \circ ((f + g) \times \mathsf{id}) \\
= & \quad \{\, \text{Pxe-Absor} \,\} \\
 & \mathsf{ap} \circ ((^{\bullet}h \circ \overline{\mathsf{inl}} \triangledown {}^{\bullet}h \circ \overline{\mathsf{inr}}) \times \mathsf{id}) \circ ((f + g) \times \mathsf{id}) \\
= & \quad \{\, \text{Sum-Fusion, } {}^{\bullet}f \text{ strict, Prod-Functor-Comp} \,\} \\
 & \mathsf{ap} \circ (^{\bullet}h \times \mathsf{id}) \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ ((f + g) \times \mathsf{id}) \\
= & \quad \{\, \text{Pxe-Def, Exp-Cancel} \,\} \\
 & \mathsf{ap} \circ (\mathsf{id} \times h) \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ ((f + g) \times \mathsf{id}) \\
= & \quad \{\, \text{Prod-Functor-Comp, Distl-Def} \,\} \\
 & \mathsf{distl} \circ ((f + g) \times h)
\end{array}
$$

$$\mathsf{distr} \circ (f \times (g + h)) = (f \times g + f \times h) \circ \mathsf{distr} \qquad \text{Distr-Nat}$$

$$
\begin{array}{ll}
 & \mathsf{distr} \circ (f \times (g + h)) \\
= & \quad \{\, \text{Distr-Def} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{distl} \circ \mathsf{swap} \circ (f \times (g + h)) \\
= & \quad \{\, \text{Swap-Nat,Distl-Nat} \,\} \\
 & (\mathsf{swap} + \mathsf{swap}) \circ (g \times f + h \times f) \circ \mathsf{distl} \circ \mathsf{swap} \\
= & \quad \{\, \text{Prod-Functor-Comp, Swap-Nat} \,\} \\
 & (f \times g + f \times h) \circ (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{distl} \circ \mathsf{swap}
\end{array}
$$

$$(^{\bullet}\mathsf{inl} \triangledown {}^{\bullet}\mathsf{inr}) \circ \mathsf{either} = \mathsf{id} \qquad \text{Either-Iso-Right}$$

$$
\begin{array}{ll}
 & (^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \circ \mathsf{either} \\
= & \quad \{\, \text{Either-Def} \,\} \\
 & (^{\bullet}\mathsf{inl} \vartriangle {}^{\bullet}\mathsf{inr}) \circ \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr}} \\
= & \quad \{\, \text{Prod-Fusion, Pxe-Absor} \,\} \\
 & \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ (\mathsf{id} \times \mathsf{inl})} \vartriangle \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr} \circ (\mathsf{id} \times \mathsf{inr})} \\
= & \quad \{\, \text{Distr-Cancel} \,\} \\
 & \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{inl}} \vartriangle \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{inr}} \\
= & \quad \{\, \text{Sum-Absor, Sum-Cancel} \,\} \\
 & \overline{\mathsf{ap} \circ (\mathsf{fst} \times \mathsf{id})} \vartriangle \overline{\mathsf{ap} \circ (\mathsf{snd} \times \mathsf{id})} \\
= & \quad \{\, \text{Exp-Fusion, Exp-Reflex} \,\} \\
 & \mathsf{fst} \vartriangle \mathsf{snd} \\
= & \quad \{\, \text{Prod-Reflex} \,\} \\
 & \mathsf{id}
\end{array}
$$

Bibliography

[ADHS01] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 303–310. IEEE Press, 2001.

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Computer Scientist.* Foundations of Computing Series. The MIT Press, 1991.

[Aug99] Lex Augusteijn. Sorting morphisms. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, 1999.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[Bac89] Roland Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, 1989.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–76, January 1977.

[BdCF04] Vincent Balat, Roberto di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 64–76. ACM Press, 2004.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[Bir84] Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

[Bir87] Richard Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* International Series in Computer Science. Prentice Hall, 1998.

[BP99] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, pages 203–211. IEEE Press, 1991.

[BW82] Friedrich L. Bauer and Hans Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[CdC91] Pierre-Louis Curien and Roberto di Cosmo. A confluent reduction system for the λ-calculus with surjective pairing and terminal object. In Leach, Monien, and Artalejo, editors, *Proceedings of the 18th International Conference on Automata, Languages and Programming (ICALP'91)*, volume 510 of *LNCS*, pages 291–302. Springer-Verlag, 1991.

[CH02] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*, pages 90–104, 2002.

[CHJ$^+$01] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[CP05] Alcino Cunha and Jorge Sousa Pinto. Point-free program transformation. *Fundamenta Informaticae*, 66(4):315–352, 2005. Special Issue on Program Transformation.

[Cro93] Roy Crole. *Categories for Types.* Cambridge University Press, 1993.

[Cun03] Alcino Cunha. Automatic visualization of recursion trees: a case study on generic programming. *Electronic Notes in Theoretical Computer Science*, 86(3), 2003. Selected papers of the 12th International Workshop on Functional and (Constraint) Logic Programming.

[Cur93] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Birkhäuser, 2nd edition, 1993.

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–257. ACM Press, 1996.

[Dar81] John Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.

[dB72] Nicolaas de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[dCK93] Roberto di Cosmo and Delia Kesner. A confluent reduction for the extensional typed λ-calculus with pairs, sums, recursion and terminal object. In Andrzej Lingas, editor, *Proceedings of the 20th International Conference on Automata, Languages and Programming (ICALP'93)*, volume 700 of *LNCS*, pages 645–656. Springer-Verlag, 1993.

[dCK94] Roberto de Cosmo and Delia Kesner. Simulating expansions without expansions. *Mathematical Structures in Computer Science*, 4(3):315–362, 1994.

[DJ04] Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction (MPC'04)*, volume 3125 of *LNCS*. Springer-Verlag, 2004.

[dMS99] Oege de Moor and Ganesh Sittampalam. Generic program transformation. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Proceedings of the 3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 116–149. Springer-Verlag, 1999.

[dMS01] Oege de Moor and Ganesh Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269:135–162, 2001.

[dMvEP01] Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer. Theorem proving for functional programmers – SPARKLE: A functional theorem prover. In T. Arts and M. Mohnen, editors, *In Proceedings of the 13th International Workshop on the Implementation of Functional Languages (IFL'01)*, volume 2312 of *LNCS*, pages 55–71. Springer-Verlag, 2001.

[Dou93] Daniel Dougherty. Some lambda calculi with categorical sums and products. In Claude Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 137–151. Springer-Verlag, 1993.

[Ell99] Conal Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, 1999.

[FM91] Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

[Fok89] Maarten Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4), 1989.

[Fok94] Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94–28, University of Twente, June 1994.

[Gha95] Neil Ghani. $\beta\eta$-equality for coproducts. In *Proceedings of of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA'95)*, volume 902 of *LNCS*, pages 171–185. Springer-Verlag, 1995.

[GHA01] Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1), 2001. Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Science.

[Gib94] Jeremy Gibbons. An introduction to the Bird-Meertens formalism. In *New Zealand Formal Program Development Colloquium Seminar*, Hamilton, November 1994.

[Gib99] Jeremy Gibbons. A pointless derivation of radix sort. *Journal of Functional Programming*, 9(3):339–346, 1999.

[Gib00] Jeremy Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 37(1–3):37–65, 2000.

[Gib02] Jeremy Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer-Verlag, 2002.

[Gil95] Andy Gill. The technology behind a graphical user interface for an equational reasoning assistant. In *Proceedings of the 8th Annual Glasgow Workshop on Functional Programming*, July 1995.

[Gil00] Andy Gill. Debugging Haskell by observing intermediate data structures. In G. Hutton, editor, *Proceedings of the 4th ACM SIGPLAN Haskell Workshop*, 2000.

[GLJ93] Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, June 1993. ACM Press.

[GRR95] Carl Gunter, Didier Rémy, and Jon Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the 7th international conference on Functional Programming Languages and Computer Architecture*, pages 12–23. ACM Press, 1995.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[Har89] Thérèse Hardin. Conflence results for the pure strong categorical logic CCL. λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65(3):291–342, 1989.

[Hin00] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 119–132. ACM Press, 2000.

[HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.

[HIT99] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Calculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 164–175. ACM Press, 1997.

[HJ01] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *ENTCS*. Elsevier, 2001.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), December 1996.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre 1,2,...,ω.* Thèse de doctorat es sciences mathématiques, Université Paris VII, 1976.

[Hug03] John Hughes. Re: Learning to love laziness. Message sent to the Haskell mailing list, September 2003.

[IHT98] Hideya Iwasaki, Zhenjiang Hu, and Masato Takeichi. Towards manipulation of mutually recursive functions. In *Proceedings of the 3rd Fuji International Symposium on Functional and Logic Programming*, pages 61–79. World Scientific, 1998.

[JG95] Barry Jay and Neil Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, 1995. Appeared previously in 1992 as Technical Report ECS-LFCS-92-243 from the University of Edimburgh.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploring of the design space. In *Proceedings of the Haskell Workshop*, 1997.

[Jon87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Jon00] Mark Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.

[JS02] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. To be submitted to The Journal of Functional Programming, March 2002.

[Klo80] Jan Willem Klop. Combinatory reduction systems. Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980.

[Klo92] Jan Willem Klop. Term rewriting systems. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[Kly05] Graham Klyne. Re: [haskell-cafe] point-free style (was: Things to avoid). Message sent to the Haskell-Cafe mailing list, February 2005.

[Lam80] Joachim Lambek. From lambda calculus to cartesian closed categories. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic*, pages 375–402. Academic Press, 1980.

[LJ03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37. ACM Press, 2003.

[LRT03] Huiqina Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 27–38. ACM Press, 2003.

[LS86] Joachim Lambek and Philip Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Series in Advanced Mathematics.* Cambridge University Press, 1986.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, 1995.

[Mal90] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, October 1990.

[Mar96] Alfio Martini. Category theory and the simply-typed lambda calculus. Technical Report 7, Technische Universitaet Berlin, Informatik, 1996.

[McL95] Colin McLarty. *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides.* Oxford University Press, 1995.

[Mee86] Lambert Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[Mee92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[Mei92] Erik Meijer. *Calculating Compilers.* PhD thesis, Nijmegen University, 1992.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS.* Springer-Verlag, 1991.

[MG01] Clare Martin and Jeremy Gibbons. On the semantics of nested datatypes. *Information Processing Letters*, 80(5):233–238, December 2001.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, 1995.

[NJ03] Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Draft proceedings of the 15th International Workshop on the Implementation of Functional Languages (IFL'03)*, 2003.

[OHIT97] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Proceedings of the IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, 1997.

[Par00] Alberto Pardo. Towards merging recursion and comonads. In Johan Jeuring, editor, *Proceedings of the 2nd Workshop on Generic Programming*, pages 50–68, Ponte de Lima, Portugal, 2000. Department of Computer Science, Utrecht University. Technical Report UU-CS-2000-19.

[Par03] Alberto Pardo. Generic accumulations. In J. Gibbons and J. Jeuring, editors, *Proceedings of the 2002 IFIP TC2 Working Conference on Generic Programming*, pages 49–78, Schloss Dagstuhl, Germany, 2003. Kluwer Academic Publishers.

[Pat] Ross Paterson. A simple equational calculator. `http://www.soi.city.ac.uk/~ross/software/calc.html`.

[Pie91] Benjamin Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing Series. MIT Press, 1991.

[Pla93] David Plaisted. *Equational Reasoning and Term Rewriting Systems*, volume 1 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter 5, pages 273–364. Oxford University Press, 1993.

[Pot81] Garrel Pottinger. The Church Rosser theorem for the typed lambda-calculus with surjective pairing. *Notre Dame Journal of Formal Logic*, 22(3):264–268, 1981.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[Pro95] José Proença. Tranformações pointwise - point-free. Technical Report DI-PURe-05.02.01, Departamento de Informática, Universidade do Minho, February 1995.

[PvE98] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean language report – version 1.3. Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, June 1998.

[Rei01] Claus Reinke. GHood - graphical visualisation and animation of Haskell object observations. In Ralf Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, volume 59 of *ENTCS*. Elsevier, 2001.

[Rey77] J.C. Reynolds. Semantics of the domain of flow diagrams. *Journal of the ACM*, 24(3):484–503, July 1977.

[Rhi99] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In O. Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRICS Note Series, pages 25–29, Department of Computer Science, University of Aarhus, 1999.

[Ros98] Kristoffer Rose. Type-directed partial evaluation in haskell. In O. Danvy and P. Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation*, number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, 1998.

[Rém02] Didier Rémy. Using, understanding, and unraveling the OCaml language. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, volume 2395 of *LNCS*, pages 413–536. Springer-Verlag, 2002.

[SdM01] Ganesh Sittampalam and Oege de Moor. Higher-order pattern matching for automatically applying fusion transformations. In *Proceedings of the 2nd Symposium on Programs as Data Objects*, volume 2053 of *LNCS*. Springer-Verlag, 2001.

[SdM03] Ganesh Sittampalam and Oege de Moor. Mechanising fusion. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, chapter 5, pages 79–104. Palgrave Macmillan, 2003.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, 1993.

[SP00] Alex Simpson and Gordon Plotkin. Complete axioms for categorical fixed-point operators. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 30–44. IEEE Press, 2000.

[Sve02] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 124–132. ACM Press, 2002.

[THT98] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 30(3), September 1998.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 306–313. ACM Press, 1995.

[UV99] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *INFORMATICA*, 10(1):5–26, 1999.

[UVP01] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. *Nordic Journal of Computing*, 8(3):366–390, 2001.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6), June 2000.

[Vos95] T. Vos. Program construction and generation based on recursive types. Master's thesis, Utrecht University, 1995.

[VU98] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, number 300 in LNCS, pages 344–358. Springer-Verlag, 1988.

[Win98] Noel Winstanley. *ERA User Manual – version 2.0.* Department of Computer Science, University of Glasgow, March 1998.

[Yok89] Hirofumi Yokouchi. Curch-rosser theorem for a rewriting system on categorical combinators. *Theoretical Computer Science*, 65(3):271–290, 1989.