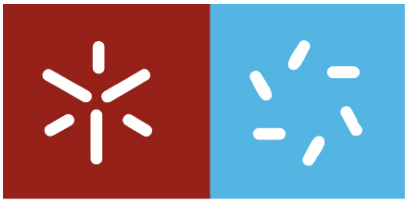
Haskell Manipulation[1]

Licenciatura em Matemática e Ciências de Computação
Internship Report



Universidade do Minho

José Miguel Paiva Proença *n.33715*
jproenca@di.uminho.pt

Supervisors: *Alcino Cunha*
Jorge Sousa Pinto

October, 2005

[1]FCT under contract POSI/CHS/44304/2002.

Resumo

Haskell é uma linguagem tipada, *lazy* e puramente funcional.

Na programação funcional existem vários estilos possíveis, mas nenhum é considerado melhor que os restantes. Dois estilos opostos são o estilo *pointwise* e o estilo *point-free*, que podem ser distinguidos pelo facto de no primeiro serem usadas variáveis, e no segundo a composição de funções.

Actualmente já existe uma ferramenta que permite traduzir código em *pointwise* para *point-free*. Um dos problemas desta tradução é o facto de produzir expressões maiores e mais complexas do que o esperado.

Este trabalho aborda dois *refactorings* (que farão parte de uma ferramenta que está actualmente a ser desenvolvida no projecto HaRe[1]), e a simplificação e transformação de expressões em *point-free*. Para este segundo objectivo foi estudada uma forma de definir regras e estratégias, usando uma notação simples, e de as aplicar a termos *point-free*.

[1] HaRe – *Haskell Refactoring* – é um projecto que está a ser desenvolvido em Canterbury, Inglaterra, que aplica vários *refactorings* a código *Haskell*, preservando a indentação e os comentários (secção 2.3).

Abstract

Haskell is a typed, lazy, **purely functional** language.

In functional programming there are several possible styles, but none is considered better than the others. Two opposite styles are the *pointwise* and the *point-free* programming styles, which can be distinguished by the fact that, in the first case variables are used, and in the second one functional composition is used instead.

In previous work a way of translating *pointwise* code into *point-free* was developed [Cun05, Pro05b]. But the resulting expressions are more complicated and bigger than expected, since they are generated by an automated process.

This work focuses on two *Haskell* refactorings (to be part of the tool beeing developed in the HaRe[2] project), and on the simplification and transformation of *point-free* terms. For the latter goal a new way of defining possible rules and strategies using an easy notation is introduced, and their application to *point-free* terms is then studied in detail.

[2]HaRe – Haskell Refactoring – is a project being developed in Canterbury that applies different refactorings to a *Haskell* source code, preserving comments and indentation (section 2.3).

Acknowledgments

I would like to start by thanking my supervisors, Alcino and Jorge, for all their support during this project. They guided me closely along this project, and promoted my research life by allowing me to participate in a spring school (Birmingham), in an international project (HaRe project, in Canterbury), and in an important *Haskell* workshop (IFL'05, to promote ongoing work in the PURe project).

During my stay for three weeks in Canterbury, where part of this work was developed, everyone was very helpful: Simon Thompson, Huiqing Li, Chris and Tom. Thank you for your support!

I also thank the FCT for supporting me financially with a BIC grant that made this work possible.

I am grateful for my friends, who gave me a fantastic support during these months. Inside the office, Miguel, João Paulo, Tiago, Paulo and Francisco. My closer friends, David, Gonçalo, Jácome, Tércio, Ana and João. You were great!

My very special thanks is definitely going to the one who shared the roof with me for the last months, and who gave my life its meaning. A sweet and thankful kiss to Xana!

Contents

List of Figures

Chapter 1

Introduction

In functional programming two distinguished styles of programming can be found: the *pointwise* and the *point-free* styles. The first one is characterised by the use of variables and the application of functions to other *pointwise* expressions, while in the second programs are written without variables, and the composition of functions is used instead of application. In this work *point-free* terms consist only of categorically-inspired combinators and algebraic data types defined as fixed points of functors. The recursion is made with type-parameterised recursion patterns (implicit recursion).

For some readers the concept of the *point-free* style may not be very clear. To make it easier to understand let us consider a very simple example in *Haskell*.

$$\begin{aligned} &\mathsf{assocr} : (A \times B) \times C \to A \times (B \times C) \\ &\mathsf{assocr}\ ((a, b),\ c) = (a,\ (b, c)) \end{aligned}$$

This function re-associates a nested pair with three elements to the right. Another way of expressing this function, without using pattern matching, is using the fst and snd functions to inspect the values of the argument:

$$\mathsf{assocr}\ x\ =\ \Big(\mathsf{fst}\ (\mathsf{fst}\ x),\ (\mathsf{snd}\ (\mathsf{fst}\ x),\ \mathsf{snd}\ x)\Big)$$

Both these definitions are defined in *pointwise*, since functions are applied to variables. It is also possible to express this function in *point-free*, by combining the fst and snd functions, using special combinators:

$$\mathsf{assocr} = (\mathsf{fst} \circ \mathsf{fst}) \vartriangle (\mathsf{snd} \times \mathsf{id})$$

In this *point-free* expression, the outermost pair is constructed with the split constructor $(\cdot \vartriangle \cdot)$, while the innermost pair is constructed by transforming the original pair with the product constructor $(\cdot \times \cdot)$

Both styles have advantages and disadvantages. The *pointwise* style is usually easier to write and to understand, while the *point-free* allows for algebraic and equational reasoning, studied for a long time in the domains of mathematics and computer science.

In this work the *point-free* language was defined by a simple set of constants and basic combinators that can be used to define more complex macros and expressions. The recursion is

represented by the hylomorphism recursion pattern, that can be used to define other recursion patterns and even any fixed point definition.

In previous work the conversion from a subset of *pointwise Haskell* to *point-free* has been studied, as can be seen in chapter 4. So the next natural steps are:

1. To increase the subset of *pointwise Haskell* that can be converted, by the removal of syntactic sugar
2. To simplify automatically the complex *point-free* expressions produced by DrHylo, a tool described in section 4.2 that translates some *Haskell* code to the *point-free* style, and to manipulate *point-free* expressions in general.

The main emphasis of this work is on how the simplification process can be automated and how the user can define rules and strategies to simplify and transform *point-free* expressions. The removal of guards, which can be seen as a way of increasing the *pointwise* subset of interpreted *Haskell*, was studied as a refactoring in the HaRe project [LRT03].

The manipulation of *point-free* terms is done using generic traversals. The generic libraries in the *Strafunski* software bundle [LV03] were used in the final version of the tool that simplifies and transforms *point-free* terms, but other approaches were also tried.

Report structure

In the first three chapters some previous studies and the state of the art is presented. Some tools related to term rewriting (the Dumatel system, the MAG system, and the HaRe project) and a small review on how to perform generic traversals using the *Scrap your Boilerplate* approach and *Strafunski*, can be found in chapter 2. In the next chapter some syntax definitions are introduced, and the *pointwise* and *point-free* styles are formally introduced. Finally, in chapter 4, a framework for programming in *point-free*, part of the *Uminho Haskell Software*, is described.

Chapter 5 describes the work developed in the context of the HaRe project. Two refactorings for *Haskell* developed in this work can be found in here: the conversion of *pointwise* to *point-free*, and the removal of guards from a function definition.

In chapter 6 the most important part of this work is described, which is a tool that simplifies and transforms *point-free* programs automatically (the SimpliFree tool).

We conclude and give, in chapter 7, some ideas of future work.

Chapter 2

Term rewriting

One of the main goals of this work is to perform rewriting on a specific small language – a *point-free* language. There are also similar approaches to achieve this goal, developed in *Haskell* language, like the Dumatel and MAG systems.

The main objective is not to prove equivalences, but to simplify terms. The first case often requires the simplification, when the most reduced term is unique (*confluence*). Since the transformation studied with *point-free* terms only deals with a small set of operators, there is no need for generalising concepts like associativity (only composition is associative).

In this chapter we will start by looking at the Dumatel system, which uses an algebraic approach to reason about programs and to construct proofs automatically. In the next section the transformation system MAG will be analysed, which implements two novel matching algorithms, and is able to apply several transformations with side conditions. In section 2.3 we will take a look at a the HaRe project, that allows the application of refactorings to *Haskell* projects just by using editors like *gvim* or *emacs*.

In the last section of this chapter two different approaches to term traversal are presented, using, respectively, the *Data.Generics* packages, and using *Strafunski* libraries.

2.1 Dumatel system

The Dumatel system [Mec05] is a term rewriting system written in the *Haskell* language. The name comes from a Russian joke, where it means *thinker*. The implementation language and strategy language are the *Haskell* code, while the object language for this system is a smaller and simpler language. In this section only some characteristics of Dumatel will be presented.

The proofs result from a *first-approach* strategy, that combines several prover parts:

- proof by induction;
- proof by arbitrary constants;
- proof by induction on a variable;

- proof by searching for equational lemma;
- others.

Dumatel uses an algebraic approach, allowing for the definition of operators (that form a total order) and the attribution of properties to these operators. Algorithms (implemented as *Haskell* functions) for substitution, matching and unification (the most general unification – mgu – is obtained by the Martelli and Montanari method) are defined in this tool.

The term ordering is a very important concept, and in this tool it should be a *simplification ordering total on gound terms* (which means that it is a total order for ground terms, subterms are always smaller, it is monotonous by the term structure, and it is invariant under substitutions). Two different algorithms for comparing these kind of terms were implemented: the lexicographic-path ordering (lpo), and the Knuth-Bendix order (kbo). The *kbo* algorithm is currently very slow, and the *lpo* is used instead.

In the *Haskell* file with the property to be proved several properties, operator definitions, and parameters needed to be introduced by the definition of functions. The object language to be parsed and interpreted are strings passed as arguments to the functions with the corresponding definition.

Although the prover can already prove several simple problems, like the double reverse property for lists (in this case by induction) – with the specification `forall [Xs] ((rev rev Xs) = Xs)` – in other apparently simple problems it fails to prove in real time. This is the case of commutativity of the multiplication of naturals, when the multiplication is defined recursively with the plus operator. There are still several bugs that have been reported since the release of Dumatel-1.02, that will probably be solved in the near future. In the current version notes it is said to be "*hardly usable*", so it should be safer to wait for the next release to use it in practise.

To define a proof the user has to write a client program, which uses a complex syntax, and lots of specific functions to set several parameters of the proof, together with the definition of the several operators present in the expression to be proved. In this report no examples will be presented of proofs due to its length and to the fact that there are several functions and details that were not explained here. A detailed description of this tool and some examples can be found in the manual of Dumatel [Mec05].

2.2 MAG System

The MAG system is a program transformation system developed by Sittampalam and de Moor, for a small functional language similar to *Haskell* [dMS99, SdM03].

It implements a new mechanism to perform generic higher-order matching [dMS01], that allows to find new functions by pattern matching. The first successful algorithm to overcome the problem of the infinite solutions when pattern matching a function, was defined by Huet and Lang [HL78]. For that they restricted the set of possible matches to *second order* terms. Since this restriction is not reasonable in the context of program transformation, de Moor and Sittampalam proposed a different approach: instead of that restriction, they substituted the function that applied the β-normalisation by a similar one (less complete). The main problem with this approach is that the behaviour of this normalisation is no longer intuitive.

This system relies on the concept of *active source*: the original source code has instructions on how to optimise it. Basically, the MAG system 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 code, repeating this process until no rule can be applied. To apply a rule the system makes a left-to-right and larger-to-smaller traversal until a matching rule is found. No loop detection is made through this process, so the user has to be careful enough to avoid infinite loops.

An example of a program transformation that this system is capable of doing is the optimisation of the naive version of reverse, using the following specification:

```
fastreverse xs ys = reverse (foldr (:) [] xs) ++ ys = reverse xs []
```

and redefining reverse to: `reverse xs = fastreverse xs []`.

The MAG source code for this transformation is:

```
{- reverse.p -}

reverse [] = [];
reverse (x:xs) = reverse xs ++ [x];

TRANSFORM fastrev
REDEFINE reverse xs = fastreverse xs []
SPECIFYING fastreverse xs ys = reverse (foldr (:) [] xs) ++ ys
USING
      DEFINITION reverse,(++);
      catassoc: (xs ++ ys) ++ zs = xs ++ (ys ++ zs);
      fusion: f (foldr plusl e xs) = foldr crossl e' xs,
                  if {f e = e';
                      \ x y -> f (plusl x y)
                         = \ x y -> crossl x (f y)}
END;

foldr f e [] = e;
foldr f e (x:xs) = f x (foldr f e xs);
```

The syntax description can be found in the system's manual[1]. After the application of the *fastrev* transformation the following code is obtained:

```
   fastreverse
= { fastrev specification of fastreverse }
   (\ a -> (++) (reverse (foldr (:) [] a)))
= { fusion

      (++) (reverse [])
   = { definition of reverse }
      (++) []
   = { definition of (++) }
      (\ a -> a)
```

[1] http://web.comlab.ox.ac.uk/oucl/research/pdt/progtools/mag/index.html

```
      (\ a b -> (++) (reverse (a : b )))
   = { definition of reverse }
      (\ a b -> (++) (reverse b ++ (a : [])))
   = { catassoc }
      (\ a b c -> reverse b ++ ((a : []) ++ c))
   = { definition of (++) }
      (\ a b c -> reverse b ++ (a : ([] ++ c)))
   = { definition of (++) }
      (\ a b c -> reverse b ++ (a : c ))
  }
   foldr (\ c d e -> d (c : e )) (\ f -> f)

reverse xs = fastreverse xs [];
fastreverse = foldr (\ c d e -> d (c : e )) (\ f -> f);
foldr f e [] = e;
foldr f e (x : xs ) = f x (foldr f e xs);
```

As it can be seen, the correct efficient version is successfully calculated, defined as a *fold*. This system is powerful enough to automate several useful transformations, and in some cases it is possible to optimise using the tupling strategy.

2.3 The HaRe Project

A refactoring can be defined as a source-to-source program transformation that only changes the program structure and organization, not program functionality. So the main goal of this project is to promote the improvement of the design of existing code, by the application of structural changes.

Similar projects already exist for other paradigms, but not for the functional paradigm. The *Haskell* language was chosen as the concrete case-study.

2.3.1 Overview of Refactoring

To show some of the advantages of refactoring, we will start by presenting a sequence of transformations to a small part of *Haskell* code, extracted from [LRT03], in figure 2.1.

In the first 3 refactorings in figure 2.1 different ways of lifting the definition of `table` to the top level are shown. The last refactoring exemplifies a typical program development scenario, where new parameters are introduced to make the definition more general.

The application of each transformation step does not only consist of textual search and replace, since the program functionality must be preserved, and there may be certain side-conditions that need to be validated. By program functionality is meant that the semantics of the program is not changed, or at least no *observable* change (from the point of view of a well defined interface) exists, since some functionality may be added (as in the last refactoring of figure 2.1).

There are several issues related to the side-conditions in the application of transformation steps: the validity of an attempted refactoring step may need to be checked before (precon-

The original code:

```
showAll :: Show a => [a] -> String
showAll = table . map show
    where
    format :: [String] -> [String]
    format []     = []
    format [x]    = [x]
    format (x:xs) = (x ++ "\n") : format xs
    table :: [String] -> String
    table = concat . format
```

A simple way to refactor:

```
-- First move format to the top level ...
showAll = table . map show
    where
    table = concat . format
format []     = ...

-- ... then move table to the top level.
showAll = table . map show
table   = concat . format
format []     = ...
```

Another simple way to refactor:

```
-- First make format local to table ...
showAll = table . map show
    where
    table = concat . format
        where
        format []     = ...

-- ... then move table to the top level.
showAll = table . map show
table   = concat . format
    where
    format []     = ...
```

Refactoring with type change:

```
-- Move table to the top level; it needs to
-- take format as a parameter.
showAll = table format . map show
    where
    format :: [String] -> [String]
    format []     = ...
table :: ([String] -> [String]) -> [String]
         -> String
table format = concat . format

--Rename the formal parameter for readability.
showAll = table format . map show
    where
    format []     = ...
table fmt = concat . fmt

-- Move format to the top level.
showAll = table format . map show
format []     = ...
table fmt = concat . fmt

-- Unfold (inline) the definition of table
showAll = (concat . format) . map show
format []     = ...
table fmt = ...

-- Remove the definition of table
showAll = (concat . format) . map show
format []     = ...

-- Define table to be (concat . format)
showAll = table . map show
table = concat . format
format []     = ...
```

Refactoring for generalisation:

```
-- The refactored program, typed.
showAll :: Show a => [a] -> String
showAll = table . map show
format :: [String] -> [String]
format []     = []
format [x]    = [x]
format (x:xs) = (x ++ "\n") : format xs
table :: [String] -> String
table = concat . format

-- Stage 1: make "\n" a parameter of format.
...
format :: [a] -> [[a]] -> [[a]]
format sep []      = []
format sep [x]     = [x]
format sep (x:xs) = (x ++ sep) : format sep xs
table = concat . format "\n"

-- Stage 2: now make "\n" a parameter of table
showAll :: Show a => [a] -> String
showAll = table "\n" . map show
format :: [a] -> [[a]] -> [[a]]
format sep []      = ...
table :: [a] -> [[a]] -> [a]
table sep = concat . format sep

-- Stage 3: copy showAll, calling it makeTable;
-- generalise by making show and "\n" parameters.
...
makeTable :: (a -> [b]) -> [b] -> [a] -> [b]
makeTable trans sep = table sep . map trans

-- Stage 4: make showAll an instance of makeTable.
showAll :: Show a => [a] -> String
showAll = makeTable show "\n"
...
```

Figure 2.1: Refactoring examples

ditions) or after (postcondition) the transformation. In some cases there are several global transformations that need to be applied to compensate for any problem resulting from a local transformation, to ensure validity.

Other important issues covered in this project are: **(i)** the integration of the tools with editors (in this case with the Vi and Emacs families of editors), **(ii)** the preservation of the layout style of the code, and **(iii)** the preservation of all comments.

2.3.2 Basic Refactorings

In the prototype tool developed in this project, several basic refactorings with side-conditions were introduced. Among them we can find:

Renaming - Replace an identifier by other with a different name;

Duplication of a definition, using a given name or a fresh generated name;

Deletion of any unused definition;

Promote one level - Lifting of a definition from a `where` or `let` to the surrounding binding group;

Demote one level - Demotion of a definition to a local scope given by a `where` or a `let`;

Add an argument - Usually to prepare for the generalisation of a definition parameters are added to definitions of constants and functions;

Remove an argument - Removal of unused arguments;

Generalise a definition - By the selection of a sub-expression in the right-hand side of the definition, introducing that sub-expression as a new parameter;

Inline a definition - Unfold of a definition by its corresponding right-hand side;

Introduce a definition to denote an identified expression.

These apparently innocent refactorings require several issues to be taken into account, like polymorphic vs monomorphic bindings, and the monomorphic restriction.

2.3.3 Tools Involved

Before the real work involved in the refactoring begins in the HaRe tool, there is still the need to parse the source texts to extract the abstract syntax trees, that requires large amounts of "boilerplate" code. For these tasks several different tools were analysed and tested, as we will soon see.

To parse, represent the abstract syntax tree, and to print, the Programatica's frontend was used [HHJK04]. When compared with other frontends it includes several other features, like the use of a lexer that preserves more information about source programs, the use of

an abstract syntax with a parameterised version (using mutually recursive types) supporting syntax variants and extensions, among others.

The existence of a large number of algebraic types, each consisting of a large number of constructors, led to the use of a tool that provides a general way to traverse and to specify strategies. The *Strafunski* project [LV03] was chosen, since it combines generic and strategic programming in a pragmatic way, that soon became an indispensable tool for this project's purposes. In section 2.4.2 it is possible to have a better view of how the generic traversals are implemented in *Strafunski*.

The initial plan was to use an IDE built from scratch to interact with the user, but the implementation effort would have distracted from the current project's main research questions, and would also be questionable for pragmatic reasons. So it was decided to design the *Haskell* refactorer to be independent of a specific GUI, and to produce instead a generic textual API and bindings to this API for Vim and Emacs – two of the most popular programmer's editors.

2.3.4 Implementation

To use the tool in Vim or Emacs, the user only has to place the cursor on a part of the code or to select the part to be read, and then select the desired refactoring to be applied from the top menu. In page 47 a screenshot of the application of a refactoring can be found. In some cases the editor will still ask for extra parameters, like the name of an identifier when renaming.

The implementation architecture tries to avoid the parsing after each refactoring when it is not necessary, updating only the position of the needed definitions, and to preserve comments and position information (possible with *Programatica*'s lexer).

The lexer produces a token stream (with comments and white spaces), that is also parsed into an AST. The AST is only used as an auxiliary representation to guide the direct modification of the token stream. The refactorer performs the program analyses and transformations on the AST, but the refactorer will modify the token stream once the AST is changed.

A more detailed description of the implementation of HaRe can be found in [LRT03].

2.4 Generic Traversals

The main goal of this work is the simplification of *point-free* terms, by the application of several rules in a certain order. The application of these rules is usually not applied to the full *point-free* term, but rather to a sub-term. So a term traversal is needed for each rule that is applied.

Term traversals could be done without the use of any generic traversal mechanism, but the reuse of code that already does this can reduce the amount of produced code and make it easier to understand. Two different libraries were studied in the development of this tool:

- *Data.Generics* libraries, already in default GHC libraries, using the *Scrap your Boilerplate* approach (SyB);

- *StrategyLib* libraries, part of *Strafunski* software bundle.

2.4.1 The SyB approach

Scrap your Boilerplate is a generic programming approach for Haskell, developed by Ralph Lämmel and Peyton Jones [LJ03]. It is supported in the GHC ($\geq$ 6.0) implementation of *Haskell*, and can be used by importing the module *Data.Generics*. This approach defines not only traversal schemes, but other generic operations, like read, show and equality.

In this section this approach will not be studied in deep detail. We cover only the necessary material to understand how can it be used to apply rules using traversal schemes defined in SyB.

To apply generic operations to a user defined data type, the instances of some classes need to be defined. This classes will allow for a manipulation of the type and the use of generic folds over the data type:

- `Typeable` - Allows type information relative to a value.
- `Data` - Defines a generic function `gfoldl` for folding over instances of this class.

These classes can be derived automatically using the *deriving* option when defining the data type, in the supported version of GHC.

Note that this approach requires two *Haskell* extensions: rank2-polymorphism, that is used to implement generic traversals (since they receive polymorphic functions as arguments), and a type-safe cast with signature

```
cast ::  (Typeable a, Typeable b) => a -> Maybe b
```

that returns `Nothing` if types are not equal, or the same value otherwise.

With the `cast` function it is possible to define ways of transforming a monomorphic function into a full polymorphic one (with functions like `mkT` and `mkQ`, which create a transformation or a query, respectively). With these functions, together with generic folds defined for the `Data` class, some generic traversals can now be defined, as in the following examples:

```
everywhere :: (forall a. Data a => a -> a)
           -> (forall a. Data a => a -> a)
everywhere f = f . gmapT (everywhere f)
```

```
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 first example performs a transformation to a data type, and the second performs a query. When using the strategies it is important to apply the conversion from a monomorphic type into a full polymorphic one, before passing as an argument, using the correct function.

To better understand this idea a small example will be presented. Let us consider the *Haskell* syntax in the package `Language.Haskell.Syntax` of GHC, and assume that the instances of *Typeable* and *Data* are already defined. The root of a *Haskell* module is

```
data HsModule = HsModule ...
```

Now let us suppose that we want to do two different operations:

1. change the names (of functions, variables, ...) that began with *normalize* to *normali**s**e*;
2. collect all integer literals.

The syntax tree of the *Haskell* code is not very small, so it would require to check for several cases to perform these operations. The first operator can be easily encoded using the `everywhere` strategy.

```
british_norm :: HsModule -> HsModule
british_norm = everywhere (mkT change_string)
    where change_string :: String -> String
          change_string x | "normalize" `isPrefixOf` x
                          = "normalise" ++ drop 9 x
          change_string x = x
```

The second operation requires the use of a query, that can be encoded using the `everything` strategy.

```
collect_ints :: HsModule -> [Int]
collect_ints = everything (++) (mkQ [] getInt)
    where getInt :: HsLiteral -> [Int]
          getInt (HsInt i) = [fromInteger i]
          getInt _ = []
```

So using this approach several lines of boilerplate code can avoided and replaced by very small and easy to read functions. Other common strategies like *somewhere* and *something* can also be found in these libraries.

2.4.2 The Strafunski approach

Strafunski is a software bundle for implementing language processing documents [LV03]. In this work we will focus be on the support provided for generic traversals over typed representations of parse trees, although it also provides means of integrating external components (such as parsers, pretty printers, and graph visualisation tools).

In a similar way to SyB, most functions require the instantiation of the following classes:

- `Typeable` - as in SyB;
- `Term` - where the conversion between a term representation is defined (using the `Dynamic` library).

Although instances for the `Term` class cannot be automatically derived by most compilers (unlike the `Data` class), it is possible to derive the code defining the correct instance (for both `Typeable` and `Term` classes) by using a tool called `DrIFT`, which is part of the *Strafunski* bundle.

The way the `Dynamic` library works will not be explained here, but the mechanisms are also associated with the same *Haskell* extension needed for the definition of the `cast` function, and the rank2-polymorphism extension.

In *Strafunski* a number of functional strategies are defined, that are composed via function combinators, as described in [LV02a]. Each strategy combinator is associated to a *type preserving* or to a *type unifying* strategy (using the postfixes `TP` and `TU`, respectively).

A strategy has type `TP m` or `TU u m`, where `m` is a monad (or a MonadPlus), and `u` is the type being calculated in the type unifying traversal. Some combinators will now be explored in more detail.

After the defining a strategy, to apply it to a given term it is necessary to use one of the following functions:

$$applyTP :: (Monad\ m, Term\ t) \Rightarrow TP\ m \rightarrow t \rightarrow m\ t$$
$$applyTU :: (Monad\ m, Term\ t) \Rightarrow TU\ u\ m \rightarrow t \rightarrow m\ u$$

The strategy is usually defined as a *scheme* applied to *steps*, where:

- the *scheme* defines the type of traversal (responsable for the application of the steps for different places of the term);
- the *steps* are a default strategy (like *id* or *fail*) updated by the functions

$$adhocTP :: (Monad\ m, Term\ t) \Rightarrow TP\ m \rightarrow (t \rightarrow m\ t) \rightarrow TP\ m$$
$$adhocTU :: (Monad\ m, Term\ t) \Rightarrow TU\ a\ m \rightarrow (t \rightarrow m\ u) \rightarrow TU\ u\ m$$

The most common traversals are: *full_td*, *full_bu*, *once_td*, *stop_td*, etc, that also receive the postfix `TU` or `TP`.

Some of the basic combinators can be found in figure 2.2.

$$\begin{array}{rl}
idTP & :: Monad\ m \Rightarrow TP\ m \\
failTP & :: MonadPlus\ m \Rightarrow TP\ m \\
seqTP & :: Monad\ m \Rightarrow TP\ m \rightarrow TP\ m \rightarrow TP\ m \\
passTP & :: Monad\ m \Rightarrow TU\ u\ m \\
 & \rightarrow (u \rightarrow TP\ m) \rightarrow TP\ m \\
choiceTP & :: MonadPlus\ m \Rightarrow TP\ m \\
 & \rightarrow TP\ m \rightarrow TP\ m
\end{array}
\qquad
\begin{array}{rl}
constTU & :: Monad\ m \Rightarrow u \rightarrow TU\ u\ m \\
failTU & :: MonadPlus\ m \Rightarrow TU\ u\ m \\
seqTU & :: Monad\ m \Rightarrow TP\ m \rightarrow TU\ u\ m \rightarrow TU\ u\ m \\
passTU & :: Monad\ m \Rightarrow TU\ u\ m \\
 & \rightarrow (u \rightarrow TU\ u'\ m) \rightarrow TU\ u'\ m \\
choiceTU & :: MonadPlus\ m \Rightarrow TU\ u\ m \\
 & \rightarrow TU\ u\ m \rightarrow TU\ u\ m
\end{array}$$

Figure 2.2: Some basic strategy combinators

For example, to traverse any data type and collect all strings inside that data type, directly or indirectly (a type unifying traversal), it is only necessary to write:

```
collectStr :: (Monad m, Term t) => t -> m [String]
collectStr = applyTU (scheme steps)
  where scheme = full_tdTU
```

```
steps = (constTU []) `adhocTU` getStr
getStr :: String -> m [a]
getStr s = return [s]
```

The collected type must be an instance of *Monoid* (in this case it is a list), which means the functions *mempty* and *mappend* must be defined.

In [LV02b] more details can be found on how to combine strategies and how to define new strategy themes.

Chapter 3

Pointwise and *Point-free* Functional Programming

Two opposite styles can be defined in functional languages: the *pointwise* and the *point-free* style. The first one is based on the λ-calculus, where variables and pattern matching are used to inspect values. The *point-free* style is distinguish by not using variables, and using combination of functions instead. The inspection of values and the application of functions can only be made through the use of special combinators.

In particular, the *point-free* style consists only of a small set of categorically-inspired combinators, where algebraic data types are viewed as fixed-points of functors. Recursion (implicit) is defined by the use of type-parameterised recursion patterns.

In this chapter the formal definitions for types and for the *pointwise* and *point-free* language will be presented.

3.1 Types

In both the *pointwise* and the *point-free* style, types can be defined according to the following syntax:

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

In this work a standard domain-theoretic semantics is assumed, where types are pointed complete partial orders (pcpo), with least element $\perp$. The single type element is represented by $\mathsf{1}$ (its only inhabitant is $\perp$); continuous functions from A to B are represented by $A \to B$; the cartesian product by $A \times B$; the separated sum (with distinguished least element) by $A + B$; and finally a recursive (regular type), defined as a the fixed point of a functor F, is represented by μF.

In the syntax described before, F and G represent functor definitions. A functor can be viewed as a mapping from types to types and from functions to functions. The functors

defined here are the identity functor (Id), the constant functor that always returns the given type A ($\underline{A}$), the lifted product and sum bifunctors ($F \otimes G$ and $F \oplus G$, respectively), and the composition of functors (as $F \circledcirc G$).

Their operation on types can be informally summarised as follows.

$$\begin{array}{lll} \mathsf{Id}\ A & \mapsto A & (F \otimes G)\ A \mapsto (F\ A) \times (G\ A) \\ \underline{A}\ B & \mapsto A & (F \oplus G)\ A \mapsto (F\ A) + (G\ A) \\ & & (F \circledcirc G)\ A \mapsto F\ (G\ A) \end{array}$$

For example, using this definition, booleans, naturals and lists can be easily defined as:

$$\begin{array}{rcl} \mathsf{Bool} & = & 1 + 1 \\ \mathsf{Nat} & = & \mu(\underline{1} \oplus \mathsf{Id}) \\ \mathsf{List}\ A & = & \mu(\underline{1} \oplus \underline{A} \otimes \mathsf{Id}) \end{array}$$

3.2 *Pointwise* Language

The syntax of the *pointwise* language is very similar to the λ-calculus with pairs and sums, together with the fix function and constructors to inspect and construct fixed points, according to the following grammar:

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

Similarly to the traditional λ-calculus, there are variables, abstractions and applications. The symbol $\star$ is the unique inhabitant of the terminal type (therfore, $\star = \bot_1$); $\langle M, N \rangle$ allows to construct pairs; fst and snd are the projections from products; inl and inr are the injection into a sum; $\mathsf{case}\ L\ M\ N$ inspects if L was injected with inl or inr, and applies M or N respectively.

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

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

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

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

Figure 3.1: Typing rules for the *pointwise* language

The typing rules for the *pointwise* language can be found in figure 3.1. Some examples of *pointwise* terms can be seen in figure 3.2, where constructors for booleans, naturals and lists are defined, together with other polymorphic terms.

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

Figure 3.2: Example of basic *pointwise* terms

The definition of recursive functions can be made using the fix function, as can be seen in figure 3.3, where *mult* is the function that multiplies naturals.

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

Figure 3.3: Example of *pointwise* recursive functions

3.3 *Point-free* Language

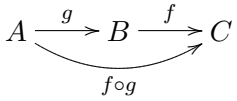
The *point-free* style was introduced by John Backus in 1977, in an ACM Turing Award Communication [Bac78]. It is here defined by a small set of categorical-inspired combinators. Most of the combinators presented in this section were defined by this author.

The main goal of this style is to gain not only the programming power associated to these combinators, but mainly a collection of useful algebraic laws. In some cases it is possible to prove properties in this style by equational reasoning in a much simpler and more concise way than using a *pointwise* style.

In this chapter some basic combinators will be presented, followed by the definition of some recursive patterns, and finally the definition of the *point-free* language that will be used in this work will be shown.

3.3.1 Basic Combinators

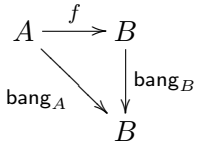
The **composition** operator can be seen as a *point-free* combinator. The composition of two functions $f : A \to B$ and $g : B \to C$ will be represented by $f \circ g$, where $f \circ g : A \to C$. The following diagram illustrates the composition's behaviour.



The **terminal object** is still $\star = \bot_1$. This means that, from any other object A there is a unique function to $\mathsf{1}$, namely the function that always returns $\bot_1$. This function is represented by $\mathsf{bang}_A : A \to \mathsf{1}$. Its uniqueness is captured by the law

$$\mathsf{bang}_B \circ f = \mathsf{bang}_A$$

which is valid for any $f : A \to B$, as can be seen in the following diagram.



Next some type constructors (like products and sums) will be introduced, together with their own combinators and laws.

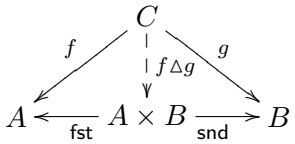
The **product** of two types is defined as the cartesian product:

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

It is possible to define the projection functions and the *split* combinator (denoted by $\cdot \vartriangle \cdot$).

$$\begin{array}{ll} \mathsf{fst}\ (x, y) = x & \\ \mathsf{snd}\ (x, y) = y & \qquad\qquad (f \vartriangle g)\ x = (f\ x, g\ x) \end{array}$$

The universal law of products can be obtained from the diagram below. In this diagram the product of two objects A and B is an object $A \times B$, together with two projections $\mathsf{fst} : A \times B \to A$ and $\mathsf{snd} : A \times B \to B$, such that for every object C, and functions $f : C \to A$ and $g : C \to B$, there exists exactly one function from C to $A \times B$ that makes the diagram commute (denoted by $f \vartriangle g$, represented by a dashed line in the diagram).



The universal law of products can then be defined as:

$$k = (f \vartriangle g) \Leftrightarrow \begin{cases} \mathsf{fst} \circ k = g \\ \mathsf{snd} \circ k = f \end{cases}$$

Another useful combinator is the product of functions (product bifunctor on functions). It will be denoted by $\cdot \times \cdot$, and it is defined as follows (we will assume that this operator binds stronger than any other combinator in order to avoid the proliferation of parentheses).

$$f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd}$$

The *swap* function can now be easily defined with the *split* operator, that swaps the elements of a pair.

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

There are several more laws about products that will not be presented here.

The **sum** will be presented, as in most functional languages, as a tagged union of two sets, together with a new *bottom* element.

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

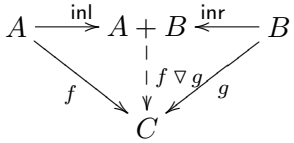
In an similar way to products, there are injection functions and the *either* combinator (denoted by $\cdot \triangledown \cdot$), that are defined as follows:

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

The universal law of sums can be written as:

$$k = f \triangledown g \Leftrightarrow \begin{cases} k \circ \mathsf{inl} = g \\ k \circ \mathsf{inr} = f \end{cases}$$

Strictness conditions will not be mentioned in here (although there are some issues that can be relevant in some cases). This law is illustrated by the existence of a unique function from $A + B$ to C that makes the diagram commute.



It is now possible to turn the sum into a bifunctor by introducing the sum combinator $(\cdot + \cdot)$ defined bellow.

$$f + g = \mathsf{inl} \circ f \triangledown \mathsf{inr} \circ g$$

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

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

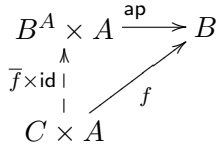
Exponentiation is associated to the *apply* and *curry* functions (denoted by $\overline{\cdot}$)

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

Likewise to products and sums, it is possible to write the universal law of exponentiation:

$$k = \overline{f} \quad \Leftrightarrow \quad \mathsf{ap} \circ (f \times \mathsf{id})$$

Looking at the diagram below, it is possible to verify that from any object C and any function $f : C \times A \to B$, $\overline{f}$ is the only function from C to B^A that makes the diagram commute.



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

$$f^A = \overline{f \circ \mathsf{ap}}$$

It is now possible to define functions to distribute the product and sum, in a *point-free* style.

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

The elements of an object A can be represented by functions with type $1 \to A$, called **points**. Given an element $x \in A$, its point is denoted by $\underline{x}$. Functions also have a point – given a function $f : A \to B$, its point is $\underline{f} : 1 \to B^A$ (and it is also possible to define a function based on its point). Using the previously defined combinators, the *point* and *unpoint* combinators can be defined as follows.

$$\begin{array}{rcl} \underline{f} & = & \overline{f \circ \mathsf{snd}} \\ f & = & \mathsf{ap} \circ (\underline{f} \circ \mathsf{bang} \vartriangle \mathsf{id}) \end{array}$$

3.3.2 Recursive Patterns

We have seen before that recursive data types are here viewed as fixed-points of a base functor. Given a base functor F, there is a unique data type μF, and two unique functions $\mathsf{in}_{\mu F} : F(\mu F) \to \mu F$ and $\mathsf{out}_{\mu F} : \mu F \to F(\mu F)$ that are each other's inverse. The $\mathsf{in}_{\mu F}$ allows the construction of a fixed-point, while the $\mathsf{out}_{\mu F}$ function allows its inspection. The following diagram shows the isomorphism.

$$\mu F \underset{\mathsf{in}_{\mu F}}{\overset{\mathsf{out}_{\mu F}}{\underset{\cong}{\rightleftarrows}}} F(\mu F)$$

It is now possible to define constructors for the boolean, natural and list types, as we have done in *pointwise* style (in figure 3.4).

$$\begin{array}{lcllcl}
\mathsf{true} & : & \mathsf{Bool} & \mathsf{false} & : & \mathsf{Bool} \\
\mathsf{true} & = & \mathsf{inl} & \mathsf{false} & = & \mathsf{inr} \\
\\
\mathsf{zero} & : & \mathsf{Nat} & \mathsf{succ} & : & \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{zero} & = & \mathsf{in} \circ \mathsf{inl}) & \mathsf{succ} & = & \mathsf{in} \circ \mathsf{inr} \\
\\
\mathsf{nil} & : & \mathsf{List}\ A & \mathsf{cons} & : & A \to \mathsf{List}\ A \to \mathsf{List}\ A \\
\mathsf{nil} & = & \mathsf{in} \circ \mathsf{inl} & \mathsf{cons} & = & \overline{\mathsf{in} \circ \mathsf{inr}} \\
\mathsf{null} & : & \mathsf{List}\ A \to \mathsf{Bool} & & & \\
\mathsf{null} & = & (\mathsf{true} \triangledown \mathsf{false} \circ \mathsf{bang}) \circ \mathsf{out} & & &
\end{array}$$

Figure 3.4: Examples of *point-free* terms

The next step is to define recursive patterns to manipulate data types viewed as fixed points. The main advantage of using recursive patterns in the definition of functions instead of explicit recursion is the fact that they allow for the application of several well known calculational laws, thus being more appropriate when equational reasoning is desired. The recursion patterns presented in this section are: the **catamorphism** (also known as fold), the **anamorphism** (also known as unfold), the **hylomorphism**, and the **paramorphism**.

Catamorphisms

A catamorphism, also known as *fold*, is an iteration, where data types are "*consumed*" by substituting their constructors by arbitrary functions. For example, the *Haskell* function `foldr` is a catamorphism for lists.

Given a function $g : F\ A \to A$, it is possible to define the catamorphism of g (denoted by $(\!|g|\!)$), that iterates upon a data type μF. It can be defined as

$$
\begin{array}{lcl}
(\!|g|\!)_{\mu F} & : & \mu F \to A \\
(\!|g|\!)_{\mu F} & = & g \circ F\ (\!|g|\!)_{\mu F} \circ \mathsf{out}_{\mu F}
\end{array}
$$

according to the following diagram.

$$
\begin{array}{ccc}
\mu F & \xrightarrow{\mathsf{out}_{\mu F}} & F\ (\mu F) \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle F\ (\!|g|\!)} \\
A & \xleftarrow[g]{} & F\ A
\end{array}
$$

The function g is called the *gene* of the catamorphism.

Anamorphisms

The dual recursion pattern to iteration is a way of producing elements of some recursive data type. It is also known as *unfold*.

Given a function $h : A \to F\ A$, it is possible to define an anamorphism for any recursive type (denoted as $[\!|h|\!]$) as:

$$
\begin{array}{lcl}
[\!|h|\!]_{\mu F} & : & A \to \mu F \\
[\!|h|\!]_{\mu F} & = & \mathsf{in}_{\mu F} \circ F\ [\!|h|\!] \circ h
\end{array}
$$

according to the following diagram.

$$
\begin{array}{ccc}
A & \xrightarrow{h} & F\ A \\
{\scriptstyle [\!|h|\!]}\downarrow & & \downarrow{\scriptstyle F\ [\!|h|\!]} \\
\mu F & \xleftarrow[\mathsf{in}_{\mu F}]{} & F\ (\mu F)
\end{array}
$$

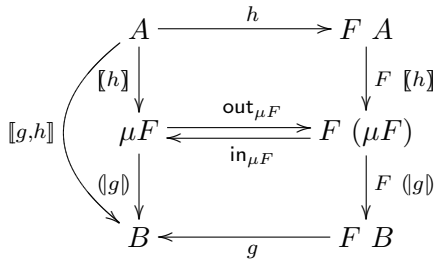
The function h is called the *gene* of the anamorphism.

Hylomorphisms

Hylomorphisms were originally introduced by Fokkinga and Meijer in [FM91]. Given a functor F and functions $g : F\ B \to V$ and $h : A \to F\ A$, it is possible to define a hylomorphism (denoted by $[\![g, h]\!]$) as:

$$
\begin{array}{lcl}
[\![g, h]\!]_{\mu F} & : & A \to B \\
[\![g, h]\!]_{\mu F} & = & (\!|g|\!)_{\mu F} \circ [\!|h|\!]_{\mu F}
\end{array}
$$

according to the following diagram.



Note that the hylomorphism can also be defined as a fixed point as:

$$[\![g, h]\!]_{\mu F} = \mu(\lambda f \ . \ g \circ Ff \circ h)$$

The catamorphisms and anamorphisms introduced before can also be defined using the hylomorphism recursion pattern:

$$\begin{array}{lcl} (\!|g|\!)_{\mu F} & = & [\![g, \mathsf{out}_{\mu F}]\!]_{\mu F} \\ (\![h]\!)_{\mu F} & = & [\![\mathsf{in}_{\mu F}, h]\!]_{\mu F} \end{array}$$

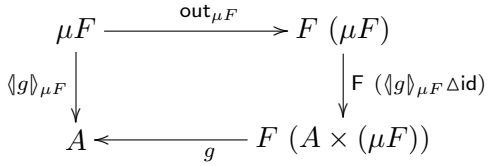
Paramorphisms

Paramorphisms resemble of catamorphisms, since that both performs recursion on the domain recursive type. The main difference is that, while catamorphisms encode iteration, the paramorphisms encode the notion of primary recursion [Mee92]. This means that, unlike catamorphisms, the value to which the recursion is applied is also passed as an argument to its gene (and not only the recursion result).

Given a function $g : F\ (A \times (\mu F)) \to A$, a paramorphism can be defined as:

$$\begin{array}{lcl} \langle\!|g|\!\rangle_{\mu F} & : & \mu F \to A \\ \langle\!|g|\!\rangle_{\mu F} & = & g \circ F\ (\langle\!|g|\!\rangle_{\mu F} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mu F} \end{array}$$

according to the following diagram.



In this case the gene is more complex than in the catamorphism case. Note that paramorphisms can also be defined as hylomorphisms:

$$\langle\!|g|\!\rangle_{\mu F} = [\![g, F\ (\mathsf{id} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mu F}]\!]_{\mu(F \circ (\mathsf{Id} \otimes \underline{\mu F}))}$$

3.3.3 The Language

The syntax of the core *point-free* language consists only of a few basic combinators:

$$\begin{array}{rcl} M, N & ::= & \mathsf{id} \mid \mathsf{bang} \mid \mathsf{ap} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{in} \mid \mathsf{out} \\ & & M \circ N \mid M \vartriangle N \mid M \triangledown N \mid \overline{M} \mid \mathsf{hylo}_{\mu F}\ M\ N \end{array}$$

The F in the hylo constructor represents a functor, as seen before. All the other constructors seen before can be easily derived from this set of *point-free* combinators.

An important issue is the fact that the only way of defining recursion is by the hylomorphism recursion pattern. This is not only because the remaining recursion patterns can be defined as hylomorphisms, but also because it have been proved to be powerful enough to allow for the definition of any fixpoint [MH95].

Recalling some *point-free* definitions introduced in figure 3.4, recursive functions like the factorial and the length of a list can be defined as seen in figure 3.5.

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

Figure 3.5: Example of recursive functions in *point-free*

Chapter 4

The *Point-free* framework

The *point-free* style can be very helpful in proofs and in program transformation. The *Haskell* language seems to be a good choice to study this style, since it is a well known lazy functional language with type checking.

The framework consists of several *Haskell* libraries and tools, part of the *Uminho Haskell Software*. Among them it is possible to find:

- Libraries containing the syntax, parsers and pretty printers of a *pointwise* and *point-free* language.
- The Pointless library, that allows the execution and type-checking of *point-free* code, using a notation very similar to the one used at the theoretical level. It also defines recursion patterns, parameterised by data types.
- The DrHylo tool, that allows programmers to automatically convert *Haskell* code to *point-free* code with recursion patterns, that can be executed with the Pointless library.

In this section only the Pointless library and the DrHylo tool will be presented. The SimpliFree tool developed in this work (to be presented only in chapter 6) will also be included in this framework.

4.1 Pointless *Haskell*

The Pointless library was developed by Alcino Cunha [Cun05], to allow for programmers to execute and type-check their programs in the *point-free* style. Several problems associated with the theoretical characterisation of the *point-free* language, and with the implementation of the recursion patterns parameterised on the data types (views as fixed points of functors), were contemplated in the development of this library.

This section starts by explaining some problems in the implementation of the basic combinators. After that, the implementation of functors and data types, based on the PolyP approach, is described. In the end, a practical implementation of recursion is defined, using the functor definition.

4.1.1 Implementing the Basic Combinators

It is well known that the semantics of a real functional programming language like Haskell differs from the standard domain-theoretic characterisation, since all data types are by default pointed and lifted (every type has a distinct bottom element). This means that *Haskell* does not have true categorical products because $(\bot, \bot) \neq \bot$, nor true categorical exponentials because $(\lambda x.\bot) \neq \bot$. Concerning products, any function defined using pattern matching, such as `\(_,_) -> 0`, can distinguish between $(\bot, \bot)$ and $\bot$. For exponentials, the examples are more subtle and typically involve using the standard `seq` function.

As discussed in [DJ04], this fact complicates equational reasoning because the standard laws about products and functions no longer hold. In point-free however, as will be shown later, pairs can only be inspected using a standard set of combinators that cannot distinguish both elements, and thus *Haskell* pairs can safely be used to model products. If we prohibit the use of `seq`, the same applies to functions. This problem does not occur with sums because the separated sum also has a distinguished least element. Sums are modelled by the standard Haskell data type `Either`.

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

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

```
data One
_L = undefined
```

The only element of this data type is `undefined`, thus it correctly implements 1. The alias `_L = undefined` is defined since this element will be used often, with the advantage that it graphically resembles the mathematical notation.

The definition of the *point-free* combinators in the **Pointless** library is trivial, and some of them can be seen in figure 4.1. Equipped with these definitions, non-recursive point-free expressions can be directly defined in Haskell. For example, the swap and distr functions can be encoded as follows.

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

4.1.2 Implementing Functors and Data Types

```
-- The final object
bang :: a -> One
bang _ = _L

-- Products
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

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

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

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

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

Figure 4.1: *Haskell* definition for basic *point-free* combinators

The implementation of recursive types in **Pointless** is based on the generic programming library PolyP [NJ03]. This library also views data types as fixed points of functors, but instead of using an explicit fixpoint operator, a non-standard multi-parameter type class with a functional dependency [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.

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

The dependency means that different data types can have the same base functor, but each data type can have at most one. The main advantage of using `FunctorOf` is that predefined *Haskell* types can be viewed as fixed points of functors (the use of the primes will be explained later).

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

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

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

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

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

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

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

For these, the instance of the `FunctorOf` class can be defined once and for all.

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

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

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

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

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

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

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

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

A possible interaction with a Haskell interpreter could now be

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

Note the annotations are compulsory since the same standard Haskell type can represent different functor combinations. This type-checking problem can be avoided by annotating the polytypic functions with the functor to which they should be specialized (similarly to the theoretical notation). Types cannot be passed as arguments to functions, and so this is achieved indirectly through the use of a "dummy" argument. By using the type class `FunctorOf`, together with its functional dependency, it suffices to pass as argument a value of a data type that is the fixed point of the desired functor. Since recursive data types can still be defined explicitly using `Mu`, there is always a convenient choice for this parameter.

To achieve an implicit coercion mechanism it suffices to insert the conversions in the functions that refer to functors, namely `inn'`, `out'`, and `fmap`. In fact, this was the reason why the primes were used in the declaration of the `FunctorOf` class. The following functions should be used instead.

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

4.1.3 Implementing Recursion

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

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

Due to the use of implicit coercion it is now possible to program with hylomorphisms in a truly point-free style. For example, the definition of factorial from Section 3.3.3 can now be transcribed directly to Haskell. The same applies to derived recursion patterns. Notice the use of bottom as the dummy argument to indicate the type to which a polytypic function should be instantiated.

```
fact :: Int -> Int
fact = hylo (_L :: [Int]) f g   where g = (id -|- succ /\ id) . out
                                      f = one \/ mult
cata (_::d) g = hylo (_L::d) g out
ana  (_::d) g = hylo (_L::d) inn g
para (_::d) g = hylo (_L::FunctorOf f d => Mu (f :@: (Id :*: Const d)))
                     g
                     (pmap (_L::d) (id /\ id) . out)
```

The functor change that occurs in the definition of paramorphisms can be naturally modeled, due to the ability to use explicit fixed points.

4.2 DrHylo

The DrHylo [Cun05, CPP05] is a tool that converts a subset of *pointwise Haskell* code to *point-free* expressions. The resulting code can be executed and type-checked using the Pointless library.

This translation is based on the well known equivalence introduced by Lambek [Lam80], between simply-typed λ-calculus and Cartesian Closed Categories (CCC). In this work, the

author defined a translation from *pointwise* terms to *point-free* combinators, already used by Curien in the implementation of the *categorical abstract machine* [Cur93]. In the DrHylo tool this translation was also used, and extended to handle sums and recursion.

The three main problems approached by this tool are:

- Translation of non-recursive *pointwise* expressions to *point-free*;
- Translation of explicit recursion to recursion patterns;
- Removal of pattern matching.

4.2.1 *Pointwise* to *Point-free*

The first question on the translation to *point-free* is how to eliminate variables. In this approach, this process resembles the translation of the λ-calculus into the *de Bruijn notation*, where variables are represented by naturals that measure the distance to their binding abstractions. Typing contexts are represented by left-nested pairs, as defined by the grammar $\Gamma ::= \star \mid \langle \Gamma, x : A \rangle$, with x a variable and A a type. Each variable will be replaced by the path to its position in the context tuple, given as follows.

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

The translation, denoted by Φ, operates on typing judgements. A judgement is translated as $\Phi(\Gamma : B \vdash M : A) : B \to A$ according to the rules defined in figure 4.2 (typing information is omitted).

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

Figure 4.2: Translation rules of non-recursive *pointwise*to *point-free*

Note that the translation of a closed term $M : A \to B$ is a an element of type $1 \to (A \to B)$. But it can be easily converted into a *point-free* function of the expected type by the application of a macro after the translation:

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

For example, the swap function is translated as the following closed term of functional type.

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

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

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

In the translation proposed by Lambek the sums were not evaluated. In this translation there were some concerns relative to the case construct. Let us start by noticing that $\mathsf{case}\ L\ M\ N$ is equivalent to $(M \triangledown N)\ L$. This equivalence exposes the fact that a case is just an instance of application, and as such its translation exhibits the same top level structure $\mathsf{ap} \circ (\Phi(\Gamma \vdash M \triangledown N) \vartriangle \Phi(\Gamma \vdash L))$. The question remains of how to combine $\Phi(\Gamma \vdash M) : \Gamma \to (A \to C)$ and $\Phi(\Gamma \vdash N) : \Gamma \to (B \to C)$ into a function of type $\Gamma \to (A + B \to C)$. The proposed solution is based on the internalisation of the uncurried version of the either combinator, that can be defined in *point-free* as follows.

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

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

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

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

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

It can be shown that the translation Φ is sound [Cur93], *i.e.*, all equivalences proved with an equational theory for the λ-calculus can also be proved using the equations that characterise the point-free combinators. Soundness of the translation of sums is proved in [Cun05].

4.2.2 Recursion

Two methods can be used for translating recursive definitions into hylomorphisms. The first is based on the direct encoding of fix by a hylomorphism, first proposed in [MH95]. The insight to this result is that $\mathsf{fix}\ f$ is determined by the infinite application $f\ (f\ (f\ \ldots))$, whose

recursion tree is a stream of functions f, subsequently consumed by application. Streams can be defined as $\mathsf{Stream}\ A = \mu(\underline{A} \otimes \mathsf{Id})$ with a single constructor $\mathsf{in} : A \times \mathsf{Stream}\ A \to \mathsf{Stream}\ A$.

Given a function f, the hylomorphism builds the recursion tree $\mathsf{in}\ (f, \mathsf{in}\ (f, \mathsf{in}\ (f, \ldots)))$, and then just replaces in by ap. The operator and its straightforward translation are given as follows

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

Although complete, this translation yields definitions that are difficult to manipulate by calculation. Ideally, one would like the resulting hylomorphisms to be more informative about the original function definition, in the sense that the intermediate data structure should model its recursion tree. An algorithm that derives such hylomorphisms from explicitly recursive definitions has been proposed [HIT96]. In the present context, the idea is to use this algorithm in a stage prior to the point-free translation: first, a *pointwise* hylomorphism is derived, and then the translation is applied to its parameter functions.

DrHylo incorporates this algorithm, adapted to the present setting where data types are declared as fixed points, and pattern matching is restricted to sums. For the algorithm to work correctly, some restrictions must be imposed on the syntax used to define recursive functions, however these are broad enough to encompass most useful definitions.

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

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

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

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

4.2.3 Pattern Matching

One of the problems with the current translation is the fact that it is restricted to the *pointwise Haskell* code defined before. To allow the translation to realistic *Haskell* code, some extra manipulation was added to the original code:

- An algorithm for defining `FunctorOf` instances (described in [NJ03]) is incorporated in DrHylo.
- Constructors are replaced by their equivalent fixpoint definitions.

- Pattern matching is only performed over the generic constructor in, sums, pairs, and the constant $\star$, due to the previous items.

A new constructor was then added to the *pointwise* definition 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. It matches an expression against a set of patterns, binds all the variables in the matching pattern, and returns the respective right-hand side.

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

Instead of directly translating this new construct to *point-free*, a rewriting system is defined that eliminates generalized pattern-matching, and simplifies expressions back into the core λ-calculus previously defined [Cun05]. We remark that since *Haskell* does not have true products, this rewrite relation can sometimes produce expressions whose semantic behaviour is different from the original. Consider the Haskell function `\(x,y) -> 0`. This function diverges when applied to `_L`, but returns zero if applied to `(_L,_L)`. This function can be directly encoded using match and translated into the core λ-calculus using the following rewrite sequence.

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

Since it no longer has pattern-matching, the resulting function is different from the original since it never diverges. Apart from this problem, with this pattern-matching construct it is now possible to translate into *point-free* many typical *Haskell* functions, such as the ones defined in appendix C.1.

For example, using this constructor distr and the length function can be defined as follows (list constructors are replaced by their *pointwise* definition given in Section 3.2).

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

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

Chapter 5

Haskell refactorings

The HaRe project was described in section 2.3. The main tool allows the application of refactorings to *Haskell* code, using editors like Vim or Emacs. In this chapter two refactorings developed in the context of *point-free* programming will be described:

- Conversion from *pointwise* code to *point-free*. Uses the same theoretical background used by DrHylo (section 4.2). This refactoring was developed under the coordination of the author of DrHylo, and during DrHylo's development, so there are several similarities with this tool. The explicit recursion (`fix` function) is not converted to recursion patterns.

- Removal of guards. The main purpose of this refactoring is to remove syntactic sugar (in this case the guards), to enrich the language recognised by DrHylo, since this tool does not support guards. In this work no other syntactic sugar is tried to be removed, since we decided to put a bigger effort into the simplification of *point-free* terms (chapter 6) rather than in covering a larger subset of *Haskell*.

5.1 *Pointwise* to *point-free*

The first refactoring addresses the conversion of *pointwise* code to *point-free* code, already approached in section 4.2 (according to [Pro05b, Cun05]).

Using HaRe, a user only has to select the expression in *pointwise* that he wants to convert, and then select the conversion option under the HaRe menu of the editor (as shown in figure 5.1). The conversion process consists in:

- Parse of the expression by *Programatica*'s tools (lexer, parser and AST), done automatically by HaRe;

- Conversion (if possible) of the term in *Programatica*'s syntax tree to a *Haskell* data type representing a *pointwise* term (`GLTerm`, in module *GlobalPW*);

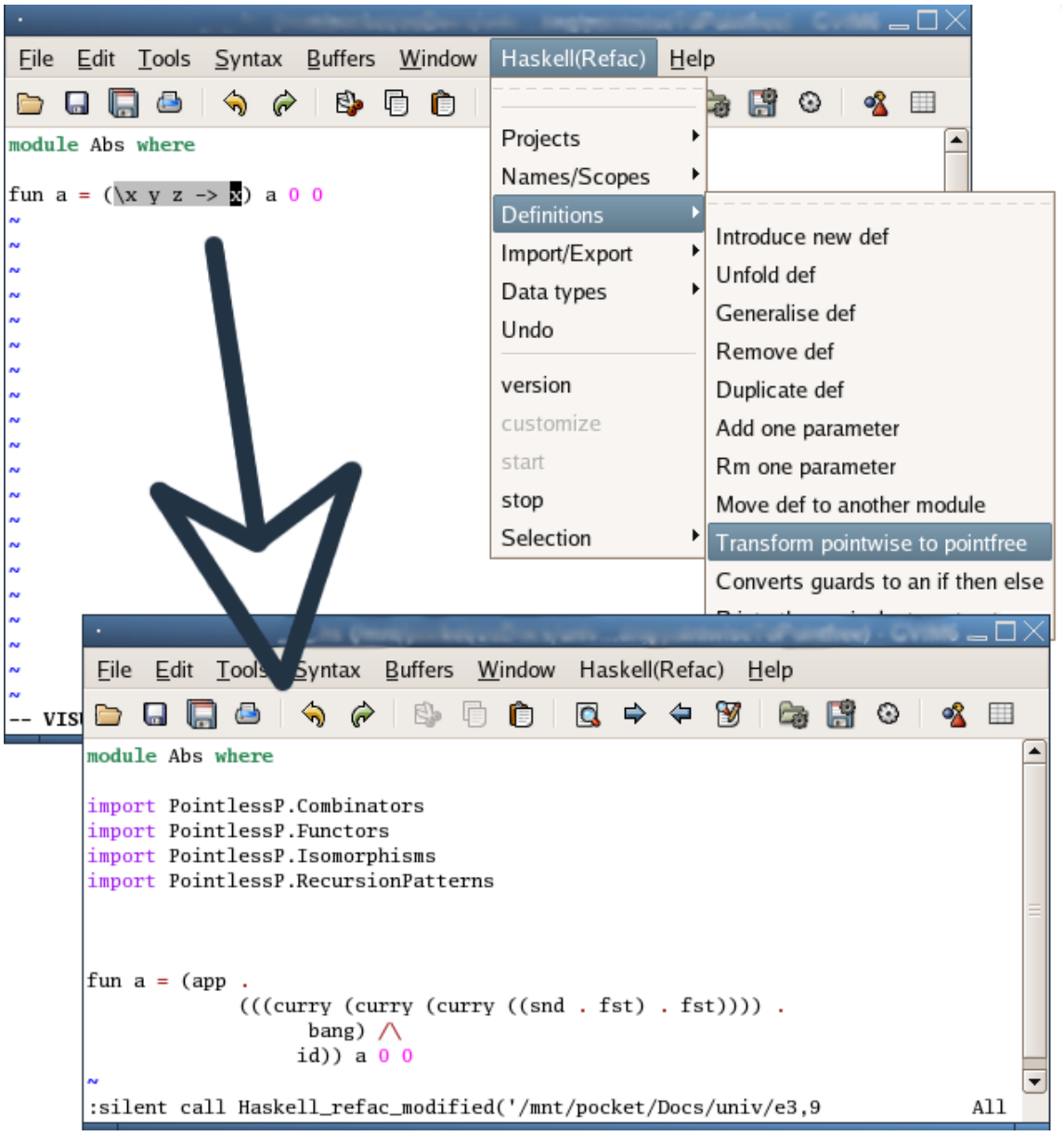


Figure 5.1: Using HaRe to convert from *pointwise* to *point-free*

- Conversion of the `GLTerm` to a simpler *pointwise* term, with less constructors (`PWTerm`, in module *PWCore*);
- Conversion of the *pointwise* term to a *point-free* term (`PFTerm`), using a set of rules specified in 4.2.1 (in module *PwPfConversion*).
- Conversion of the resulting *point-free* term back to a term in *Programatica*'s syntax tree, that is later printed back using the HaRe tool pretty printer.

The diagram in figure 5.2 illustrates the process of conversion (the token stream described in section 2.3 will be ommited here).

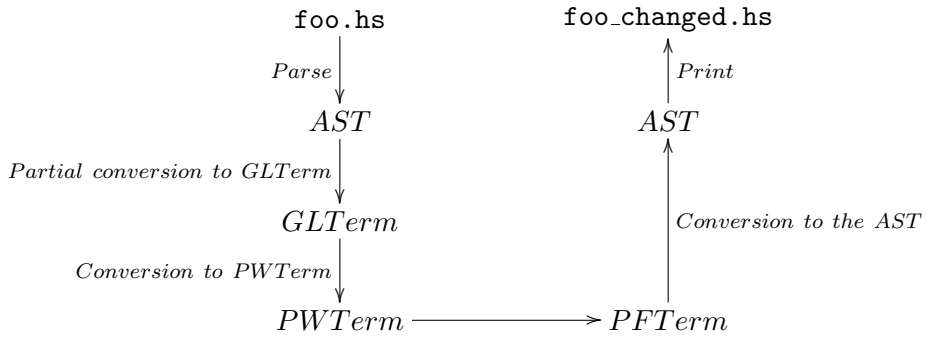


Figure 5.2: Conversion process from *pointwise* to *point-free*

5.1.1 Read *pointwise* expressions

Only a small subset of *pointwise* languages are recognised by the refactoring. The following grammar defines the language that can be converted to *point-free*.

G, G_1, G_2, G_3 ::=
`(`G`)` | `undefined` | `()` | `_L` | `inN (_L::`*type*`)` G | `ouT (_L::`*type*`)`
| `True` | `False` | `0` | `[]` | $n, n > 0$ | `[`G_1`,`G_2`,...]` | G_1`:`G_2 | `succ` G | `pred` G
| `(==0)` G | G`==0` | `head` G | `tail` G | `null` G
| `recNat` G_1 G_2 G_3 | `recList` G_1 G_2 G_3
| `fix` G | `if` G_1 `then` G_2 `else` G_3 | *var* | `(`G_1`,`G_2`)` | `fst` | `snd` | `Left` | `Right`
| G_1 *infixOp* G_2 | `\`*var*$_1$*var*$_2$ `...->` G
| `case` G_1 `of Left` *var*$_1$`->` G_2`;Right` *var*$_2$`->` G_3
| `let` *var* $=$ G_1 `in` G_2

where *var*, *var*$_1$ and *var*$_2$ are variables that are read as strings, *infixOp* is an infix operator, and *type* is also read as a single string.

5.1.2 *Pointwise* definitions

As mentioned before, two different data types were defined to represent *pointwise* terms. *PWCore* is the most general *pointwise* language, that is later transformed into a *point-free* term. *GlobalPW* is a more specific *pointwise* language, with more syntactic sugar than the previous language, but less powerful. The two languages are defined in table 5.1.

PWCore.hs

```
data PWTerm
  = Unit               -- Unit
  | Var' String        -- Variable
  | PWTerm:@:PWTerm    -- Aplication
  | Abstr String PWTerm  -- Abstraction
  | PWTerm:><:PWTerm   -- Pair
  | Fst PWTerm         -- Point-wise first
  | Snd PWTerm         -- Point-wise second
  | Inl PWTerm         -- Point-wise left injection
  | Inr PWTerm         -- Point-wise right injection
  | Case PWTerm (String,PWTerm) (String,PWTerm)
                       -- Case of
  | In  HsExpP PWTerm -- Injection on a specified type
  | Out HsExpP PWTerm -- Extraction of the functor of
                       --   a specified type
  | Fix PWTerm         -- Fixed point
```

GlobalPW.hs

```
data GLTerm
  = Star               -- Unit
  | V String           -- Variable
  | GLTerm:-:GLTerm    -- Aplication
  | Lam String GLTerm  -- Abstraction
  | GLTerm:&:GLTerm    -- Pair
  | Pi1 GLTerm         -- Point-wise first
  | Pi2 GLTerm         -- Point-wise second
  | Inl' GLTerm    -- Point-wise left injection
  | Inr' GLTerm    -- Point-wise right injection
  | Case' GLTerm (String,GLTerm) (String,GLTerm)
                       -- Case of
  | In'  HsExpP GLTerm -- Injection on a specified type
  | Out' HsExpP GLTerm -- Extraction of the functor
                       --  of a specified type
  | Fix' GLTerm        -- Fixed-point
  | T'                 -- Constant True
  | F'                 -- Constant False
  | Z'                 -- Constant Zero
  | Suc GLTerm         -- Successor
  | Pred GLTerm        -- Predecessor
  | IsZ GLTerm         -- is zero?
  | Ite GLTerm GLTerm GLTerm -- ^if then else
  | N'                 -- Empty list
  | GLTerm ::: GLTerm  -- Constructor for lists
  | Hd GLTerm          -- Head of a list
  | Tl GLTerm          -- Tail of a list
  | IsN  GLTerm        -- is the list empty?
  | Letrec String GLTerm GLTerm
                    -- recursive let
  | RecNat  GLTerm GLTerm GLTerm
                    -- Primitive recursion on Nat's
  | RecList GLTerm GLTerm GLTerm
                    -- Primitive recursion on List's
```

Table 5.1: *Pointwise* language definitions

Only a *GLTerm* can be parsed from an expression from *Programatica*'s syntax tree.

5.1.3 Resulting *point-free* expressions

After a successful conversion a new *point-free* term is obtained. The *point-free* terms have the syntax described in section 3.3.3. The main problem with the resulting terms is that, besides the fact that the recursion is still explicit (unlike code produced by DrHylo), the resulting terms are usually too big and complex. The DrHylo tool has the same problem, which we address with the automatic simplification of these terms studied in this work in chapter 6.

The produced expressions are very similar to the resulting code of DrHylo, so only a small example will be presented here.

Consider the *coswap* function written with a lambda expression.

```
coswap = \x -> case x of Left y -> Right y
                         Right y -> Left y
```

After the selection of the lambda expression and the application of this refactoring the following function is obtained:

```
coswap = app .
           (((curry
                (app .
                  ((curry (((Right . snd) \/ (Left . snd)) . distr)) /\
                    snd))) .
              bang) /\
            id)
```

Modules from the Pointless library are also added to the imports list (if necessary), to ensure that the file can still be executed correctly. The imported modules are not from the original Pointless library, because the HaRe parser does not support infix constructors, used by Pointless.

More examples of this refactoring can be found in `http://wiki.di.uminho.pt/wiki/bin/view/Ze/Bic`.

5.2 Removal of guards

Syntactic sugar exists for practical issues, to make a language easier to use, but it does not make the language more powerful. This is why DrHylo only focus on a *Core* language with very little syntactic sugar. On the other way, it is obviously useful to extend it to the complete *Haskell* language, since it is far more expressive. The extension of this *Core* language is the main motivation for the removal of syntactic sugar, in this case of *Haskell* guards.

In this section the removal of guards (which are not part of the recognized *pointwise* grammar) is studied. The removal consists not only in replacing a guard for an *if then else*, but mainly in manipulating the several matches that define a function. This substitution is only applied when possible.

The difficulty of removing guards lies on the following facts:

- it is not always possible to remove a guard (since it may not allow the function to match in further cases);
- it may be necessary to merge matches before removing the guards.

The process of removing guards consists of three stages:

Consistency check – where for each match it is checked if it can be converted and swapped with matches defined below, without changing the function behaviour;

Merge of matches – where similar matches with guards are merged into a single match, renaming the needed variables;

Conversion to *if then elses* – where the guards are finally removed and replaced by nested *if then elses*, and an error is returned if no *otherwise* case is found.

5.2.1 Consistency check

Before any evaluation, each match is compared with the matches below on the same function definition, by the function `isConsist`. Each match is then associated with a boolean stating either that it is consistent or not.

The function `isConsist` returns:

- `Just True` – if disjoint patterns are found;
- `Just False` – if an incongruence is found and there are no disjoint patterns;
- `Nothing` – if no incongruence nor disjoint patterns are found

Note that two patterns are said to be disjoint if they have different constructors or different constant values, and incongruent if one is a variable and the other something else. In this comparison, every sub-pattern is also evaluated, and all possible cases for patterns are contemplated. The value returned by `isConsist` is then converted to a boolean, based on the presence of guards (`Nothing` becomes `True` if there are guards, or `False` otherwise).

5.2.2 Merge of matches

After knowing which matches can be manipulated and swapped, the next step is to try to merge matches with guards and similar patterns, using all possible combinations.

Because there is sometimes the need to create fresh variables, a string that is not the prefix of any of the variable names (in this case the smaller non-empty sequence of x's) is used as the base name, and a counter is added to the variable name. The function `mergeMatches` uses a state monad with partiality, to manage fresh variables with the state, and that fails when the merging of two matches is not possible.

Before trying to merge matches, the declarations of the consistent matches are passed inside the guards and to the main expressions, by converting the *wheres* to *lets*. This is necessary because after the merging of two matches, the declarations inside the *where* clause of each match will now affect the other matches as well. This will duplicate the declarations inside the *where* clause.

In the merge of two matches all the possible patterns were contemplated, as before. In some cases there is more than one way to merge patterns. The merge process of each base case is now explained in more detail.

Similar patterns – If the two patterns are syntactically equivalent, then no transformation is applied. Since this is the first test to be performed, the patterns are considered to be different in the remaining tests;

Variables – When two different variables are found they are replaced by a fresh variable: in the pattern, in the guards and in the main expression;

Parenthesis – They are initially ignored, and after the merging they are placed back;

WildCard (underscore) – Can only be matched with a variable. In this case a fresh variable will replace both the wild card and the variable, in every important place of the match;

var @ pattern – If both matches have this construction, then the variable and the associated pattern are split into two different patterns, and after the recursive evaluation the resulting variable is "glued" back to the resulting pattern. If only one match has this construction, then the variable is replaced by a fresh variable, so that the same variable can also be assigned to the second pattern;

Irrefutable pattern – When an irrefutable pattern is found, it is replaced by a fresh variable, and a *let* constructor is added inside the guards and in the main expression, assigning the irrefutable pattern to take the value of the fresh variable. This way the fact that *Haskell* uses lazy evaluation allows the behaviour of the function to be the same as before;

Constructors with fields – Whenever a constructor with fields is found, it is converted into a pattern where the constructor is applied to the different arguments, using wild cards when a variable name is not assigned;

Application – When, in both matches, an application of a constructor is found, and the constructors have the same name, then the constructor is ignored and arguments are merged (using recursion), and in the end the new arguments are extracted and wrapped with the constructor again:

Infix application – Same approach to normal application was taken.

Tuples and lists – Each pattern inside the tuple or list is added to the remaining patterns to be merged, and in the end they are "glued" again with the tuple or list constructor.

5.2.3 Conversion to *if then elses*

This final stage is the simplest one, since there are not many cases to evaluate. The function `guards2ifs` is applied to each match with a guard and with the consistency check mark. When the *otherwise* guard is not found, the *else* case of the resulting expression issues an error message. The consistency check made previously ensures that no other match would succeed after every guard fails, therefore it is safe to issue the error message.

5.2.4 Tracing an example

To give a better understanding of how this refactoring works, the removal of guards of a simple example will be explored in detail. The example does not do anything in particular, only tries to cover differents problems in the removal of guards.

```
foo (h:t1) x | h == 0 = x
foo []     y | y > 0 = 1
foo (h:t2) z | otherwise = -z
foo []     v | v < 0 = -1
```

The first step is to check, for each match, if any inconsitency is found with the matches below. After this step, each match is associated with a boolean, stating if the it can be swaped with the matches below and the corresponding guards can be removed.

```
True  -->  foo (h:t1) x | h == 0 = x
True  -->  foo []     y | y > 0 = 1
True  -->  foo (h:t2) z | otherwise = -z
True  -->  foo []     v | v < 0 = -1
```

In this case all the matches are consistent with the matches below, which means that, if a pattern matching is found after the merging of matches, no other pattern matching is possible.

In the merging step all the possible combinations to merge matches are perfomed. This is the most complex part of this calculations. After the merging of matches some fresh variables are introduced, as it can be seen below.

```
foo ((h : xx_0)) xx_1
    | h == 0 = xx_1
    | otherwise = -xx_1
foo [] xx_2
    | xx_2 > 0 = 1
    | xx_2 < 0 = -1
    | otherwise = error "UnMatched Pattern"
```

Finally, the replacement of guards by nested *if then elses* is a simple process that produces the following code:

```
foo ((h : xx_0)) xx_1
    = if h == 0 then xx_1 else -xx_1
foo [] xx_2
    =   if xx_2 > 0
        then 1
        else if xx_2 < 0
             then -1
             else error "UnMatched Pattern"
```

This example and others of removal of guards can be found in appendix A.

5.2.5 Future work on the removal of guards

In the removal of guards an effort was made to evaluate every possible case, and to introduce new variables only when really necessary. But even an apparently easy piece of syntactic sugar like the conversion of guards to *if then elses* can get very complicated when all possibilities are analysed.

There are still some cases that could be improved. For example, the following function,

```
f x | x > 0 = 1
f x = 0
```

could be evaluated in a previous stage to:

```
f x | x > 0 = 1
f x | otherwise = 0
```

and only then the matches could be merged, so that the guards can be converted to *if then elses*.

Another possible improvement would be to try to merge the declarations inside the *wheres* when possible, instead of placing them inside *let* expressions. This would require to check if the declared functions or constants in one declaration are not used in the other match, or equally defined in the respective declaration.

Chapter 6

SimpliFree tool

6.1 Introduction

One of the main problems with the conversion to *point-free* code described in section 4.2.1 is the fact that the resulting terms are much more complex than the expected terms. This is due to the automated process that applies the several transformations.

The SimpliFree tool [Pro05a] was developed to simplify the resulting *point-free* terms as much as possible, and in an automated way. The approach taken is based on the notion of *active source*, in the same way as MAG 2.2: the code to be analysed is annotated with special commentated blocks with rules to be used in the transformation of the code.

Unlike MAG, the SimpliFree tool uses the *Haskell* compiler's pattern matching mechanism, and instead of having a fixed strategy, it allows the user to produce new strategies or to adapt existing ones using several strategy combinators, to suit particular cases.

In order to simplify most *point-free* terms, the possibility of importing some strategies from a rules repository (built for the tool) was introduced. This import can be mentioned by a special commentated block or by the use of arguments. In this way the concept of active source can be avoided in some cases, if the user desires so.

To apply the strategies in a more efficient way, the pattern matching of *Haskell* compilers is used. This means that the tool does not translate the original *point-free* terms directly to simpler *point-free* terms. The SimpliFree tool produces a new *Haskell* file with functions that apply the transformations to the *point-free* terms found in the original file. The produced file imports a SimpliFree library which defines the core functions for the traversals on *point-free* terms, taking the maximum advantage possible of the *Haskell* compiler pattern matching.

When the produced file is interpreted, it is possible to follow the intermediate results and the rules applied at each step to every simplification made. The `main` function prints a new *Haskell* program, similar to the original one, where the simplified terms replace the old ones. Diagram 6.1 illustrates the way files are organised when using the tool.

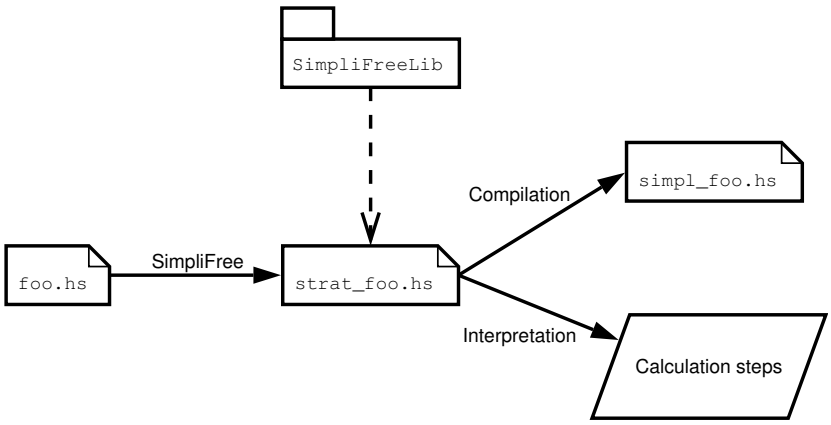


Figure 6.1: Diagram with SimpliFree architecture

6.2 Term traversal

A very important issue is how to traverse a term to apply a transformation. In this work more than one way of traversing the terms were tested, all using generic schemes:

1. Calculation of a simplified term with ***Data.Generics*** libraries (SyB approach);
2. Calculation of a simplified term with *Strafunski*;
3. Calculation of intermediate steps and the simplified term with *Strafunski*;
4. Calculation of intermediate steps that can contain *computations* (to be presented bellow).

The application of a strategy returns a *Computation*, which is defined as a final *point-free* term and a list of intermediate results (pairs of $term \times rule$). But only on the last two generic schemes the list of intermediate results is not empty.

Module *SimpliFreeLib* contains, among others, functions that use the generic libraries to define the traversals. This is the only file that needs to be altered when changing the way traversals are made. In the final version of the SimpliFree tool the *Strafunski* library was used, calculating the intermediate results (scheme 4). The strategy combinators defined in the library are: *rule*, *many*, *or*, *and*, *oneOrMore*, *optional* and *fail*. Other auxiliary functions were also defined inside the library.

6.2.1 Traversal with the *Generics* library

This approach has several advantages:

- It is not very difficult to implement;
- The needed instances can be automatically derived by GHC;

- The needed libraries are already in the default libraries of the current versions of GHC.

The main problem is that the resulting code is not so efficient as the one using the *Strafunski* libraries, and the fact that the instances for the needed class cannot be derived automatically into an external file (without using the *deriving* option).

In this approach rules and strategies have the same type:

```
type Strat m = Pointfree.Term -> m Pointfree.Term
```

And the definition of the strategy that normalises the composition (reassociates composition to the right), the function that builds a rule, and the strategy combinator *and* are as follows:

```
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)

normalise :: MonadPlus m => Strat m
normalise = iteratePF (once (mkMp flat))
  where flat ((x :.: y) :.: z) = return (x :.: (y :.: z))
        flat _ = fail "no need to flat"
        iteratePF :: MonadPlus m => Strat m -> Strat m
        iteratePF strat = (strat `andPF` (iteratePF strat)) `orPF` return

rulePF :: MonadPlus m => String -> Strat m -> Strat m
rulePF _ r = once (mkMp r)

andPF :: MonadPlus m => Strat m -> Strat m -> Strat m
andPF r g e = r e >>= g
```

The function that applies a strategy to a term only needs to use an *Haskell* application, without the use of other combinators. No further work was performed using the SyB approach, since the resulting code was not so efficient as the code produced with *Strafunski*'s libraries.

6.2.2 Simple traversal with Strafunski

The main advantage of the *Strafunski* approach is that the resulting code is much more efficient (apparently twice as fast). And since this tool was specially developed to make traversals on any data type (while the *Generics* library deals with other concerns involving generic programming as well as traversals), it already has more specialised combinators to facilitate the traversal definitions.

Strafunski has type preserving and type unifying strategy combinators. In the present case only the simplified term is calculated, so only type preserving strategy combinators need to be used. The same example functions as before are now:

```
normaliseStrat :: MonadPlus m => TP m
normaliseStrat = iterateTP strat
  where strat = once_tdTP (adhocTP failTP flat)
```

```
        flat ((x :.: y) :.: z) = return (x :.: (y :.: z))
        flat _ = fail "no need to flat"
        iterateTP :: MonadPlus m => TP m -> TP m
        iterateTP strat = (strat `seqTP` (iterateTP strat)) `choiceTP` idTP


rulePF :: (MonadPlus m) => String -> (Term -> m Term) -> TP m
rulePF _ r = (once_tdTP (adhocTP failTP r)) `seqTP` normaliseStrat

andPF :: MonadPlus m => TP m -> TP m -> TP m
andPF s1 s2 = s1 `seqTP` s2
```

Note that a rule is still a monadic function that changes a *point-free* term, but a strategy is now of type $TP\ m$ (type preserving), and to apply a strategy to a term the combinator `applyTP` has to be used.

6.2.3 Advanced traversal with Strafunski

In this approach the intermediate calculation steps are calculated. To accomplish this it is necessary to combine both strategies of *Strafunski* mentioned in section 6.2.2, in order to change a term (type preservation) and to collect all changed terms and rules applied (type unification).

The equivalent functions to the previous examples are more complicated in this approach, except for the normalise strategy that remains unaltered. This is due to the need of combining type preserving and type unifying strategy combinators.

```
rulePF :: (MonadPlus m) => String ->
          (Pointfree.Term -> m Pointfree.Term) -> TU Computation m
rulePF name r =
    idComp `passTU` \(_,t1) ->
           (((once_tdTP (adhocTP failTP r)) `seqTP` normaliseStrat) `seqTU` idComp)
           `passTU` \(_,t2) ->
           constTU ([(t1,name)],t2)

idComp :: MonadPlus m => TU Computation m
idComp = adhocTU (constTU (mempty::Computation)) (\x->return ([],x))

andPF :: MonadPlus m => TU Computation m -> TU Computation m -> TU Computation m
andPF s1 s2 = s1 `passTU` \(lst1,t1) ->
                (constTP t1) `seqTU` s2 `passTU` \(lst2,t2) ->
                   (constTU (lst1++lst2,t2))
  where constTP t = adhocTP idTP (\_->return t)
```

6.2.4 More complex justifications

In the definition of rules it is possible to apply strategies to the resulting expression. This means that the rule is only succesfully applied if the strategies do not fail. It also means that,

in case of success, the intermediate steps associated to the strategies applied inside the rule cannot be visualised.

In order to access the intermediate steps, the rule functions are changed so that they return not only the resulting expression, but also a list of the computations associated to the strategies applied inside the rule. So a rule will have the following signature:

$$rule :: MonadPlus\ m \Rightarrow Term \rightarrow m\ (Term, [Computation])$$

This implies changing the definition of a *Computation*, to add the possibility of having other computations inside a main computation:

```
type CalcTerm = (Pf.Term,[Computation])
data Computation = Comp { csteps :: [(CalcTerm,String)],
                          cresult :: Pf.Term}
```

It is also necessary to change the strategy combinator that applies a rule, which becomes much more complicated, since the intermediate computations are extracted using a state monad:

```
rulePF :: (MonadPlus m) => String ->
          (Pf.Term -> m CalcTerm) -> TU Computation m
rulePF name r =
 let tu_state =
           idComp `passTU` \(Comp _ t1) ->
           (((once_tdTP (adhocTP failTP (aux r))) `seqTP` normaliseStrat)
           `seqTU` (insSteps t1))
     tu_nostate =
           -- conversion of "StateT [Computation] m (a,b,c)" to "m (a,b,c)"
           msubstTU (flip evalStateT []) tu_state
 in tu_nostate `passTU` \(t1,t2,comps) -> constTU (Comp [((t1,comps),name)] t2)
  where aux :: MonadPlus m => (Pf.Term -> m CalcTerm) -> Pf.Term
                           -> StateT [Computation] m Pf.Term
        aux r term = do (t,comp) <- lift (r term)
                        put comp
                        return t
        -- convertion of "m a" to "StateT s m a"
        lift m = StateT (\s -> m >>= \a -> return (a,s))
        insSteps :: MonadPlus m => Pf.Term
                 -> TU (Pf.Term,Pf.Term,[Computation]) (StateT [Computation] m)
        insSteps t1 = adhocTU (constTU (ID,ID,[]))
                              (\t2 -> get >>= \comps -> return (t1,t2,comps))
```

After defining a *Show* instance for *Computations*, it is possible to visualise the calculations associated to strategies called from rule definitions.

The main problem with this approach is that the application of any rule involves more calculations, so the produced code is slower than the one obtained with the previous approaches.

6.2.5 Builtin strategies

To facilitate the user task, a few default strategies were included in the SimpliFreetool. The possibility of redefining strategies made the process of adding new rules to existing strategies easier.

The strategies present in the tool can be found in appendix B. There is a base strategy (appendix B.1) that is used in the creation of other strategies, like the advanced strategy (appendix B.2). The base strategy not only simplification rules but also rules to fold and unfold several known macros, like swap ($=$ snd $\triangledown$ fst) and exp ($= \overline{f \circ \mathsf{ap}}$). The main idea of this strategy is:

1. Apply the simplification rules as many times as possible;
2. Unfold known macros if possible, and return to previous step until no more known macros exist.
3. Fold known macros, to make the result easier to read.

During the simplification, the rules follow the idea that the composition should be the inner operator. For example, $(f \vartriangle g) \circ h$ is converted to $(f \circ h) \vartriangle (g \circ h)$, and not the other way (unless followed by certain strategies), since it can *trigger* more cancelation rules.

The advanced strategy uses the base strategy, but some more simplification rules are added. The reason why a new strategy pack was created is because there are several rules that can give type error, like the convertion of $\mathsf{id} \times \mathsf{id}$ to id.

6.3 Rule construction

6.3.1 Basic principles

A rule is converted to a *Haskell* function of type `Term -> m Term`, where `m` is a *MonadPlus*, such that:

- it applies a transformation to a term or to the prefix of a composition, not to other subterms;
- it fails (`fail "..."`) when it is not possible to apply the transformation;
- it assumes that the composition is normalised (associated to the right).

Example 1 *natId1* : $id \circ f \rightarrow f$

```
natId1 (ID :.: f) = return f
natId1 _          = fail "..."
```

In example 1 a very simple example is shown, but the definition of these functions can get very complicated, as described in the next subsections.

6.3.2 Adition of the *ending*

The first problem comes from the fact that the rules can be applied not only to terms, but also to **prefix** of terms, relatively to compositions. Remember that the composition is assumed to be always associated to the right (this way the composition operator behaves similarly to the constructor of lists in *Haskell*).

The best way to solve this problem is to add a new match to each rule definition when necessary, to pattern match the *ending* (when the last element is also a composition), and to place it in the resulting expression, as can be seen in example 2.

Example 2 *sumCancel1* : $(f \triangledown g) \circ inl \rightarrow f$

```
sumCancel1 ((f :\/: g) :.: INL)         = return (f)
sumCancel1 ((f :\/: g) :.: (INL :.: x)) = return (f :.: x)
sumCancel1 _                            = fail "..."
```

The *ending* case needs to be added when the outermost operator is the composition, and the last element is **not** a variable.

6.3.3 Left variables (l.v.)

Variables on the right of compositions match with the biggest composition possible, as expected when looking at example 1, since the composition is associated to the right. The main problem is when variables are found on the **left** of a composition (including the case of variables in the middle of compositions, since it is on the left of a subterm).

Each composition with a left variable (l.v.) is substituted by another variable that is analysed by a new auxiliary function (top down substitution).

Example 3 *exp_fold* : $\overline{f \circ ap} \rightarrow f^{\circ}$

```
exp_fold (Curry f') | ... = ... aux_f  f' ...
  where aux_f (f :.: AP) = ...
        ...
...
```

Inside the guards it is verified if the variable(s) representing the composition with a l.v. match with the term being evaluated, also by the use of the auxiliary function.

The empty spaces in example 3 will be explored in full detail in the next subsections.

Auxiliary functions

Each new auxiliary function is responsible by the association of elements of a composition to match with a l.v., through recursion and pattern matching. It calculates:

a) if the pattern matching succeeded;

b) a list of terms corresponding to each of the variables inside the expression;

c) if the the auxiliary function is associated with a l.v. that precedes the last element of a composition, where this composition is the outermost operator, and if the last element of this composition is not a variable, then it also calculates the possible *ending*, similarly to what has been done in subsection 6.3.2.

In the auxiliary functions an intermediate structure was used – `Maybe ([Term], Maybe Term)`, and several new functions were added to the library *TravGenLib.hs*:

success - Verifies if the pattern matching was successful.
```
success = isJust
```

noVar - When the pattern matching fails
```
noVar = Nothing
```

emptyVar - When the pattern matching succeeds, but no variable was found yet.
```
emptyVar = Just ([],Nothing)
```

addTerm - Adds a new term to the list of expressions associated to variables.
```
addTerm t (Just (l,e)) = Just (t:l,e)
```

addComp - Composes a term (through the constructor `:.:`) with the last term added to the list with `addTerm`.
```
addComp t (Just (x:xs, e)) = Just ((t:.:x):xs, e)
```

joinVars - Joins the results of more than one auxiliary functions.
```
joinVars = fmap mconcat . sequence
```
[1]

getTerm - Return the n^{th} term.
```
getTerm n (Just (l,e)) = l !! n
```

getEnd - Add the end to a given term.
```
getEnd (Just (_,Just t1)) t2 = t2 :.: t1
getEnd _ t = t
```

Each auxiliary function consists of 3 or 4 cases:

i) The pattern matching succeeds.

In example 3:
```
aux_f (f:.:AP) = addTerm f emptyVar
```

ii) Only in some cases – The pattern matching succeeds, but the *ending* case needs to be considered (as described in section 6.3.2).

In example 3 this case is not needed since the composition where `f` appears is not the outtermost operation. If it were, then it would be written as:
```
aux_f (f:.:(AP:.:x)) = addEnd x (addTerm f emptyVar)
```

[1] where `Maybe` belongs to the classes `Functor` and `Monad`, and an instance to the class `Monoid` where added; in GHC $\leq$ 6.4 an instance of `Monoid` for pair were added.

iii) The pattern matching is not well succeeded, but it is a composition case, so the auxiliary function is recursively called to the *tail* of the composition.

In example 3:
```
aux_f (f0:.:f1) | success (aux_f f1)
                = addComp f0 (aux_f f1)
```

iv) The pattern matching fails.

In example 3:
```
aux_f _ = noVar
```

If a composition with a l.v. is found inside the expression that is matched (*e.g.*, `g:.:ID`), then it is substituted by another variable (`g'`), and the condition `success (aux_g g')` is added to the guards of the cases *i*) and *ii*), where `aux_g` is the auxiliary function associated to `g` (built in a similar way to `aux_f`). The body of the *i*) and *ii*) cases are also changed, by substituting `emptyVar` by `aux_g g'`, and if more l.v. are found it is replaced by `joinVars [aux_g g',...]`.

It is important to note that the order in which the terms are added to the intermediate structure is always the same, regardless of the terms that are instantiated with the auxiliary functions, making the controlled extraction of elements from the list possible.

Main function

For each l.v. `f`, the condition `success (aux_f f')` is added to the guard. In the main body two transformations are applied to the returned term:

- The variables inside the compositions with a l.v., calculated by the auxiliary functions, are collected by the function `getTerm`, that is applied to the union of the result of the auxiliary functions. They are then "put inside" the resulting term through a lambda abstraction. The order is very important here.

  In example 3:
  ```
  exp_fold (Curry f') | success (aux_f f')
                      = (\f -> Macro "exp" [f])  (getTerm 0 (aux_f f'))
  ```

- The possible *ending* is added to the term, by the function `getEnd`.

  In example 3:
  ```
  ... (\f -> getEnd (aux_f f') (Macro "exp" [f]))  (getTerm 0 (aux_f f'))
  ```

The addition of the *ending* in normal expressions, as described in subsection 6.3.2, may also be needed when no l.v. are found in the end of the outer composition. In this case a duplication of the function and all of the auxiliary functions is made, with the small difference that the `ending` is contemplated (in a similar way to example 2).

Extras

An important feature that was added was the use of **conditions**. But the correct verification of conditions required important changes to the code, as will be seen in the next section (6.3.4). The main problem with the way conditions are handled is when l.v.'s occur. So far the auxiliary functions only return a single possible pattern match, and do not backtrack if the condition fails.

With the verification of conditions working correctly, new features were added to the tool. One example is when **equal variables** are found in the left hand side of a rule. In this case they are substituted by fresh variables, and the tests for their equality are added to the list of conditions.

Another improvement is the possibility of **applying strategies** to the result of a rule. The test to check if the strategies are successfully applied is added to the list of conditions. This allows the definition of much more complex rules, as the fusion for catamorphism on lists, that will be presented in section 6.4.

6.3.4 Conditions

So far a condition is interpreted as a string containing a *Haskell* expression that returns a boolean. When equal names for variables are used in the left-hand side of an expression, then one of it is replaced by a fresh variable, and the condition that verifies their equality is automatically added.

A way of verifying conditions correctly was introduced by applying some changes to the produced code. The changes described in subsection 6.3.3 were not sufficient for the verification of conditions. In this subsection the changes to the previous algorithm are explained in detail.

Simplest case

In the cases of a rule where the left hand side is a composition, where the last element is a variable (not a l.v.), and conditions are supplied by the user, two new matches are added before the usual matches. For example, consider the case of a rule called *rule*, that matches with a composition ending with variable *f*, returning the expression *exp* with conditions *cond*. In this case the main match will be replaced by the 3 following matches:

```
rule (... :.:f:.:x) | cond = return (exp:.:x)
rule (... :.:f:.:(x1:.:x2)) = rule (... :.:(f:.:x1):.:x2)
rule (... :.:f) | cond = return exp
```

where the third one remains equal (appart from the presence of the condition inside the guard).

Even if there are left variables (as long as f is not a l.v.), the validation of the auxiliary functions will still be evaluated in the first and third match without undesirable side condi-

tions. But still some is attention needed in the presence of left variables, as it will be shown in the next subsection. Then, so far, the conditions are correctly evaluated when:

- The outermost operator is not the composition;
- The outermost operator is the composition and the last element is not a variable;
- The outermost operator is the composition and the last element is a variable – not a left variable (only here the proposed changes are needed).

In the presence of left variables

In subsection 6.3.3 the auxiliary functions return an intermediate structure with a possible definition for each of the variables in a sub-expression and a possible *ending*, in the data type – `Maybe ([Term], Maybe term)`. Instead of just returning the first possible solution for each variable, the intermediate structure was changed to `[([Term], Maybe Term)]`, and the auxiliary functions now return the list of all possible solutions for each variable. The new functions added to *TravGenLib.hs* in subsection 6.3.3 were now changed, as described in table 6.1.

Maybe ([Term], Maybe term)	**[([Term], Maybe Term)]**
`success = isJust`	`success = not.null`
`noVar = Nothing`	`noVar = []`
`emptyVar = Just ([],Nothing)`	`emptyVar = [([],Nothing)]`
`addTerm t (Just (l,e)) = Just (t:l,e)`	`addTerm t lst = [(t:l,nt) | (l,nt) <- lst]`
`addComp t (Just (x:xs,e))`	`addComp t lst`
`  = Just ((t:.:x):xs, e)`	`  = [((t:.:x):xs,nt) | (x:xs,nt) <- lst]`
`addEnd t (Just (x,_)) = Just (x,Just t)`	`addEnd t lst = [(l,Just t) | (l,_) <- lst]`
`joinVars = fmap mconcat . sequence`	`joinVars = fmap mconcat . sequence`
`getTerm n (Just (l,_)) = l !! n`	`getTerm n ((l,_):_) = l !! n`
`getEnd (Just (_,Just t1)) t2 = t2:.:t1`	`getEnd ((_,Just t1):_) t2 = t2:.:t1`
`getEnd _ t = t`	`getEnd _ t = t`
–	`verifyCond = isJust`
–	`testCond f = findIndex (f.fst)`
–	`getIndex i = drop (fromJust i)`

Table 6.1: Changes to the functions that operate on the intermediate structure

Specific changes to the auxiliary functions and to the main functions will be explored in more detail in the next subsections.

Auxiliary functions

Recall the 4 possible matches added for each auxiliary function described in subsection 6.3.3. The cases *i*) and *ii*) need to be changed, in order to gather all the possible

results (without regarding the condition), instead of just returning the first possible pattern match. The new two cases are:

i) If the left hand side is a composition that ends in a variable, then the result of a recursive call to the *tail* of the composition is added to the possible results.

> In example 3 the result does not end in a variable, so the match remains unchanged. If it did, the new match would be:
>
> ```
> aux_f (f:.:AP:.:g) = addTerm f (addTerm g emptyVar)
>                    ++ addComp f (aux_f (AP:.:g)
> ```

ii) Only in some cases, as before – The pattern matching succeeds, but the *ending* case needs to be considered. In this case the recursive call to the *tail* of the composition is always added to the possible results.

> In example 3 this case is not needed since the composition where `f` appears is not the outermost operation. If it were, then it would be written as:
>
> ```
> aux_f (f:.:(AP:.:x)) = addEnd x (addTerm f emptyVar)
>                      ++ addComp f (aux_f (AP:.:x)
> ```

Note that the guards, when present, do not suffer any change.

Main function

To facilitate the reading and the writing of the rule function, three new pattern binds (*aliases*) were added to declarations of the main function, together with the auxiliary functions:

all_pattern = `joinVars [`*auxiliary_functions_calls*`]` – gathers all the possible results for each variable inside compositions in the *scope* of l.v.. Will be used only in the evaluation of the conditions;

index = `testCond (\[`*name_of_variables*`] -> `*conditions*`)` `all_pattern` – looks for the first variable attribution that satisfies the conditions (`index :: Maybe Int`). The abstraction only captures the names of the variables defined inside the compositions with l.v. because the others are already captured by the pattern matching in the main function;

all_expression = = `getIndex index all_pattern` – puts a solution that satisfies the conditions at the head of the list with all the terms that pattern matched, by *dropping* possibilities. It will be used inside the main expression, when evaluating the final result.

When there are no conditions to be evaluated, then the *alias* `all_expression` is enough, defined in the same way as `all_pattern`, since all possible results returned by the auxiliary functions are correct.

In the end it is only necessary to add the predicate `verifyIndex index` to the guard of the main function, when there are user defined conditions.

6.3.5 Syntactic sugar

The use of syntactic sugar enables the possibility of using simpler instructions that will be converted to more complex set of instructions.

Some important additions to the language are:

- The possibility of using **equal variables**, that are replaced by fresh variables and compared inside the conditions (already mentioned before in subsection 6.3.4);
- A special rule that constructs a new rule expressing the **associativity** property, as shown in the following example:

```
Assoc assoc_cat : 'cat'        ⟹   catAssoc : (curry 'cat') . 'cat' ->
                                        'comp' . ((curry 'cat') >< (curry 'cat'))
```

- Another special rule, given a set of macros definitions, expands each macro to the corresponding fold and unfold rule, and also creates strategies that gather the folding rules (named `macros_fold`) and the unfolding rules (named `macros_unfold`), as shown in the following example:

```
Macro swap :                    macros_fold : swap_fold orOF coswap_fold
      snd /\ fst                macros_unfold : swap_unfold orOF coswap_unfold
Macro coswap :          ⟹
      Right \/ Left             swap_unfold  : 'swap' -> snd /\ fst
                                coswap_unfold: 'coswap' -> Right \/ Left
                                swap_fold  : snd /\ fst -> 'swap'
                                coswap_fold: Right \/ Left -> 'coswap'
```

- The introduction of **lists of terms**, that allows the use of rules with more than one argument (as will be seen when the strategy for the fusion of catamorphisms is introduced in section 6.4.3). This is internally represented as a macro with a special name that is ignored when printing.

6.4 Testing Strategies

6.4.1 Simple example

In this section a very small example will be shown, without importing any set of rules from the rules repository. It will simply be shown how to iterate the product cancellation rule:

$$\mathsf{Prod\text{-}Cancel}_1 : \mathsf{fst} \circ (f \vartriangle g) = f$$
$$\mathsf{Prod\text{-}Cancel}_2 : \mathsf{snd} \circ (f \vartriangle g) = g$$

The original file is:

```
f = curry ((snd.(snd /\ fst)).(fst /\ fst))

{- Rules:
simplify: many (prodCancel1 or prodCancel2)
```

```
prodCancel1: fst.(f/\g) -> f
prodCancel2: snd.(f/\g) -> g
-}

{- Optimizations: f -> simplify -}
```

And the resulting file after the application of the SimpliFree tool is (using *SimpliFreeLib* described in section 6.2.3):

```
module Main where

import SimpliFreeLib
import Language.Haskell.Syntax
import Language.Haskell.Pretty
import Language.Pointfree.Pretty

prodCancel1 (FST :.: (f :/\: g)) = return (f)
prodCancel1 (FST :.: ((f :/\: g) :.: x)) = return (f :.: x)
prodCancel1 _ = fail "rule prodCancel1 not applied"
prodCancel2 (SND :.: (f :/\: g)) = return (g)
prodCancel2 (SND :.: ((f :/\: g) :.: x)) = return (g :.: x)
prodCancel2 _ = fail "rule prodCancel2 not applied"

simplify
  = manyPF
      ((rulePF "prodCancel1" prodCancel1) `orPF`
         (rulePF "prodCancel2" prodCancel2))

f = Curry (SND :.: ((SND :/\: FST) :.: (FST :/\: FST)))

f_simplify = unOk (applyPF simplify f)

what = putStrLn "Avaiable results:\n  - f_simplify\n"
main
  = putStrLn
      (prettyPrint
         (HsModule .... f_simplify .... )
  where f_simplify_ = pf2hs (snd f_simplify)
```

When interpreting the resulting file it is possible to list all existing optimizations by the function `what` (in this case there is only one). The function `f_simplify` returns the computation with the intermediate steps, which in this case yields:

```
*Main> f_simplify
curry (snd.(snd /\ fst).(fst /\ fst))
  = { prodCancel2 }
curry (fst.(fst /\ fst))
  = { prodCancel1 }
curry fst
```

The `main` function return the original *Haskell* file, but with the transformed terms instead of the original expressions. The comments and the original indentation are lost during this

process. To make life easier for those who just want the simplified terms, not the intermediate calculations, a shell script that receives the original file and produces the simplified one was created. In this script SimpliFree program is applied, and then the resulting code is compiled and executed.

6.4.2 *DrHylo* results

The DrHylo tool is part of the *Uminho Haskell Libraries*, developed in the University of Minho. As described in the introduction, it can translate normal recursive *Haskell* functions to *point-free* terms, where recursion is only expressed by the hylomorphism recursion pattern. The main problem with this translation is that the resulting *point-free* expressions are usually more complex than the expected ones. In this section some functions obtained by the DrHylo tool are analysed, and simplified automatically by the SimpliFree tool. The advanced strategy, introduced in section 6.2.5, will be used to simplify the expressions.

The code bellow was produced by DrHylo.

```
module Sample where
import Pointless.Functors
import Pointless.Combinators
import Pointless.Combinators.Uncurried
import Pointless.RecursionPatterns

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

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

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

coswap :: Either a b -> Either b a
coswap
  = app .
```

```
      (((curry
           (app .
              ((eithr . ((curry (inr . snd)) /\ (curry (inl . snd)))) /\ snd)))
            . bang)
         /\ id)

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

data Nat = Zero
         | Succ Nat
         deriving Show

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

instance FunctorOf ((:+:) (Const One) Id) Nat where
        inn' (Inl (Const _)) = Zero
        inn' (Inr (Id v1)) = Succ v1
        out' (Zero) = Inl (Const _L)
        out' (Succ v1) = Inr (Id v1)
```

To use the advanced strategy described in appendix B.2 it is enough to use an extra argument when calling the SimpliFree program:

```
SimpliFree -i adv_strat < Samples_drHylo.hs > out_drHylo.hs
```

The simplified code is:

```
module Sample where
import Pointless.Functors
import Pointless.Combinators
import Pointless.Combinators.Uncurried
import Pointless.RecursionPatterns

comp :: (b -> c, a -> b) -> a -> c
comp = curry (app . ((fst . fst) /\ (app . (snd >< id))))

swap :: (a, b) -> (b, a)
swap = swap

assocr :: ((a, b), c) -> (a, (b, c))
assocr = (fst . fst) /\ (snd >< id)

coswap :: Either a b -> Either b a
coswap = coswap

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

data Nat = Zero
         | Succ Nat
         deriving Show

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

instance FunctorOf ((:+:) (Const One) Id) Nat where
        inn' (Inl (Const _)) = Zero
        inn' (Inr (Id v1)) = Succ v1
        out' (Zero) = Inl (Const _L)
        out' (Succ v1) = Inr (Id v1)
```

Note that in the case of the `swap` and `coswap` functions the simplification derived the corresponding name, because both are known macros in the used strategy. When looking at the derived computation of `coswap`, for example, it is possible to follow all the steps taken in the process.

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

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

6.4.3 Cata-Fusion for Lists

The rule cata-fusion for lists is a good example to illustrate the advantages of allowing the application of strategies in the result of a rule.

Example 4 *cataList_fusion* : $f.(\!|g|\!)_{List} = (\!|h|\!)_{List} \quad \Leftarrow \quad f \circ g = h \circ (List\ f)$

```
cataList_fusion (f :.: (Macro "cata" [g]))
     = Macro "cata" [`apply deriveGene (f:.:g)`]
```

The cata-fusion rule is not so straightforward as the previous rules, and there are many ways of calculating the h value using strategies.

Since $\mathsf{F}_{\mathsf{List}\ A} = \underline{1} \oplus \underline{A} \otimes \mathsf{Id}$, it is reasonable to assume that the gene of the catamorphism is an *either* ($g = g_1 \triangledown g_2$). So the difficult task is to find h such that

$$f \circ (g_1 \triangledown g_2) = h \circ (id + id \times f)$$

It is still possible to do some calculations to facilitate the definition of a strategy for calculating h.

$$\left[\begin{array}{ll} & f \circ (g_1 \triangledown g_2) \\ = & \quad \{\ \text{Sum-Fusion}\ \} \\ & f \circ g_1 \triangledown f \circ g_2 \\ = & \quad \{\ \dagger\ \} \\ & f \circ g_1 \triangledown i \circ (j \times k \circ f) \\ = & \quad \{\ \text{Natural Id, Prod-Functor}\ \} \\ & f \circ g_1 \triangledown i \circ (j \times k) \circ (id \times f) \\ = & \quad \{\ \text{Sum-Absortion}\ \} \\ & (f \circ g_1 \triangledown i \circ (j \times k)) \circ (id + id \times f) \end{array}\right.$$

So, if the difficult step † is possible, then it is possible to match h with $f \circ g_1 \triangledown i \circ (j \times k)$. In other words, $h = h_1 \triangledown h_2$ where:

- $h_1 = f \circ g_1$
- $h_2 = i \circ (j \times k)$, if $f \circ g_2 = i \circ (j \times k \circ f)$

At this stage it is possible to note that:

- h_1 can be easily calculated, since f and g_1 are already known;
- h_2 is not so easy to calculate, since the values if i, j and k are not known yet. They can be calculated by equational reasoning on the equality $f \circ g_2 = i \circ (j \times k \circ f)$. This can be achieved by the definition of a strategy that begins by transforming $f \circ g_2$ into $i \circ (j \times k \circ f)$, and then extracts the values of i, j and k to produce the $h_2 = i \circ (j \times k)$;
- After having the h_1 and h_2 value, the new catamorphism can be easily defined as $(\!|h_1 \triangledown h_2|\!)$

The strategy can be written in the SimpliFree *language* as:

```
cataList : cataList_rule

cataList_rule : f . ('cataList' [g1\/g2]) ->
                    'cataList' [(f.g1) \/ (apply getH2 [f,f.g2])]


getH2 : extractH2 or (cataList_step and getH2)

extractH2 : extractH2A or extractH2B or extractH2C or extractH2D

extractH2A : [g,a.(b >< (c.g))] -> a.(b><c)
extractH2B : [g,a.(b >< g)] -> a.(b><id)
extractH2C : [g,b >< (c.g)] -> b><c
extractH2D : [g,b >< g] -> b><id

cataList_step : user_cataL_rules or swapLeft or base_rule or base_unfMacros

swapLeft : (f >< g) . 'swap' -> 'swap' . (g >< f)

cataList_rules : catAssoc

catAssoc : (curry 'cat') . 'cat' -> 'comp' . ((curry 'cat') >< (curry 'cat'))
```

where the only user defined rule is

$$\mathsf{cata\text{-}assoc} : \overline{cat} \circ cat \rightarrow comp \circ (\overline{cat} \times \overline{cat})$$

that describes the *cat* associativity property. Note that, with the syntatic sugar added in section 6.3.5, the `catAssoc` rule could be defined as:

```
Assoc catAssoc: 'cat'
```

The strategy called `base_rules` is imported from a rules repository, and its main task is to simplify terms and to put composition inside splits and eithers.

For example, this strategy, using the associative property of concatenation described above, allows the simplification of

$$\overline{cat} \circ (\!|nil \triangledown (cat \circ swap \circ (wrap \times \mathsf{id}))|\!)$$

to

$$(\!|(\overline{cat} \circ nil) \triangledown (comp \circ swap \circ ((\overline{cat} \circ wrap) \times \mathsf{id}))|\!)$$

which computes the reverse of a list using an accumulator.

The corresponding *pointwise* functions to these two *point-free* definitions can be defined in *Haskell*, respectively, as:

```
reverse [] = []
reverse (x:xs) = cat (reverse xs, wrap x)
```

```
reverse_t [] y = y
reverse_t (x:xs) y = reverse_t xs (x:y)
```

The computation steps produced by the strategy defined before are:

```
*Main> test_cataList
curry 'cat'.('cataList' ['nil' \/ ('cat'.'swap'.('wrap' >< id))])
  = { cataList_rule
      --- and ---
      [curry 'cat',curry 'cat'.'cat'.'swap'.('wrap' >< id)]
        = { catAssoc }
      [curry 'cat','comp'.(curry 'cat' >< curry 'cat').'swap'.('wrap' >< id)]
        = { swapLeft }
      [curry 'cat','comp'.'swap'.(curry 'cat' >< curry 'cat').('wrap' >< id)]
        = { prodFun }
      [curry 'cat','comp'.'swap'.((curry 'cat'.'wrap') >< (curry 'cat'.id))]
        = { natId2 }
      [curry 'cat','comp'.'swap'.((curry 'cat'.'wrap') >< curry 'cat')]
        = { extractI2B }
      'comp'.'swap'.((curry 'cat'.'wrap') >< id)
    }
'cataList' [(curry 'cat'.'nil') \/ ('comp'.'swap'.((curry 'cat'.'wrap') >< id))]
```

But there are still some important issues that can be relevant, like the fact that the cata-fusion would not be possible after an application of the exp-fusion rule ($\overline{g} \circ f \rightarrow \overline{g \circ (f \times \mathsf{id})}$). This is because the strategy would try to fuse the function *cat* (instead of $\overline{cat}$) with the catamorphism, which fails. This indicates that some manipulation and backtracking may be necessary before the strategy can be applied.

This strategy for fusing catamorphisms for lists was added to the rules repository (as can be seen in appendix B.3), so it can be reused for other cases as well. This and the fact that the associativity property can be automatically written as a special rule (as described in section 6.3.5) makes this strategy very easy to use. In the case presented in this section it would be enough to import the strategy pack with the cata-fusion law for lists, and to define the associativity property using the special rule.

6.4.4 Cata-Fusion for Rose Trees

In a very similar way to the strategy defined in the previous section (6.4.3), a more complex strategy can be defined for Rose Trees.

A rose tree can be defined in *Haskell* as:

```
data Rose a = Node a [Rose a]
```

The fixed point associated to this type can be easily defined as $\mathsf{Rose}\ A = \mu(\underline{A} \times \mathsf{List})$. So in this case, the functor on functions can be defined as $F\ f = \mathsf{id} \times \mathsf{map}_{\mathsf{List}}\ f$.

In the definition of this strategy, the *extract* function needs to recognise a more concrete expression. Besides that, the strategy is very similar to the previous case:

```
cataRoseTree_strat : opt cataRoseTree

cataRoseTree : f . ('cataRoseTree' [g]) -> 'cataRoseTree' [apply
               getRoseTreeH2 [f,f.g]]

getRoseTreeH2 : extractRoseTreeH2 or (cataRoseTree_step and
                getRoseTreeH2)

extractRoseTreeH2 : extractRoseTreeH2A or extractRoseTreeH2B or
                    extractRoseTreeH2C or extractRoseTreeH2D
extractRoseTreeH2A : [f, a . (b >< (c . ('mapList' [f])))] -> a . (b >< c)
extractRoseTreeH2B : [f, b >< (c . ('mapList' [f]))] -> b >< c
extractRoseTreeH2C : [f, a . (b >< ('mapList' [f]))] -> a . (b >< id)
extractRoseTreeH2D : [f, b >< ('mapList' [f])] -> b >< id

cataRoseTree_step : cataRoseTree_rules or swapLeft or adv_rules or
base_unfMacro
```

The testing function will be a function that performs a postorder traversal in a rose tree, and returns the corresponding list. In the *point-free* style, this function can be defined as follows.

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

The specification for the optimization can be written as:

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

It is now possible to apply the cata-fusion law for rose trees, using the associativity property of the cat operator (mentioned in the previous section), the cata-fusion for lists, and the following property:

$$(\!|f \triangledown g \circ (h \times id)|\!)_{\mathsf{List}} = (\!|f \triangledown g|\!)_{\mathsf{List}} \circ \mathsf{map}_{\mathsf{List}}\ h)$$

The output produced by the SimpliFree tool is:

```
*Main> post_t_cataRoseTree_strat
curry 'cat'.('cataRoseTree' ['cat'.'swap'.('wrap' ><
            ('cataList' [('pnt' ['nil']) \/ 'cat']))])
  = { cataRoseTree
      --- and ---
```

```
      [curry 'cat',curry 'cat'.'cat'.'swap'.('wrap' ><
      ('cataList' [('pnt' ['nil']) \/ 'cat']))]
        = { catAssoc }
      [curry 'cat','comp'.(curry 'cat' >< curry 'cat').'swap'.
      ('wrap' >< ('cataList' [('pnt' ['nil']) \/ 'cat']))]
        = { swapLeft }
      [curry 'cat','comp'.'swap'.(curry 'cat' >< curry 'cat').
      ('wrap' >< ('cataList' [('pnt' ['nil']) \/ 'cat']))]
        = { prodFun }
      [curry 'cat','comp'.'swap'.((curry 'cat'.'wrap') ><
      (curry 'cat'.('cataList' [('pnt' ['nil']) \/ 'cat'])))]
        = { cataList
            --- and ---
            [curry 'cat',curry 'cat'.'cat']
              = { catAssoc }
            [curry 'cat','comp'.(curry 'cat' >< curry 'cat')]
              = { extractH2B }
            'comp'.(curry 'cat' >< id)
    }
      [curry 'cat','comp'.'swap'.((curry 'cat'.'wrap') ><
      ('cataList' [(curry 'cat'.('pnt' ['nil'])) \/
      ('comp'.(curry 'cat' >< id))]))]
        = { foldMapFusAdv }
      [curry 'cat','comp'.'swap'.((curry 'cat'.'wrap') ><
      (('cataList' [(curry 'cat'.('pnt' ['nil'])) \/ 'comp']).
      ('mapList' [curry 'cat'])))]
        = { extractRoseTreeH2A }
      'comp'.'swap'.((curry 'cat'.'wrap') ><
      ('cataList' [(curry 'cat'.('pnt' ['nil'])) \/ 'comp']))
    }
'cataRoseTree' ['comp'.'swap'.((curry 'cat'.'wrap') ><
               ('cataList' [(curry 'cat'.('pnt' ['nil'])) \/ 'comp']))]
```

As it can be seen, the fusion is successfully applied when using this strategy. The final catamorphism represents the more efficient version of the postorder traversal, that uses an accumulator.

6.5 Implementation details and Efficiency

There are three main steps involved in the simplification of an *Haskell* file:

1) Application of the SimpliFree tool to obtain the intermediate *Haskell* file, where the traversals and rules are encoded in *Haskell*;

2) Compilation (or interpretation) of the intermediate *Haskell* file;

3) Application of the traversals in the process of transforming the *point-free* terms.

The efficiency of each step will be analysed separately in this section. For that, the example in appendix C will be used, together with the advanced strategy pack described

in appendix B.2, which has 68 rules (including the folding and unfolding of macros). The computer involved in the process is a Pentium M at 1.3 GHz, with GHC 6.2.2.

Application of SimpliFree

In the first step – the application of the SimpliFree tool – most of the time is consumed during the parsing. So in this subsection the only concern will be the parser, since we are looking at the efficiency problems.

In the first versions of this tool the parsing was done in two ways:

- the *point-free* functions were parsed using the parser included in GHC (*Language.Haskell.Parser*), that are later traversed to collect the existing *point-free* expressions.
- the special code inside the blocks, containing rules, strategies, optimisations and program options, is parsed with a library using parsing combinators.

Later a tool called Happy[2] was used to build the parser for the syntax inside the special blocks. Happy is a parser generator for *Haskell*, similar to the *yacc* tool for the *C* language, that takes an annotated BNF specification of a grammar and produces a *Haskell* module with a parser for the grammar. The main advantages of using this tool are the fact that the grammar is now much easier to understand and change, and the produced code is much more efficient than when using the parsing combinators. Using the example in appendix C the user CPU time is around **0.33 seconds**, versus the **2.1 seconds** that the parsing combinators took in several tests. This means that, in this case, the code produced by Happy is around 6 times faster than with the parsing combinators.

Compilation and interpretation

Recall the several different approaches made in this tool:

1) Calculation of the final result with ***Data.Generics*** libraries (SyB approach);
2) Calculation of the final result with *Strafunski*;
3) Calculation of intermediate steps with *Strafunski*;
4) Calculation of **all** intermediate steps also with *Strafunski*.

Later, a 5th version with no generic traversals was developed. In this version only a single function to traverse the *point-free* abstract syntax tree was defined, producing identical results to the last approach, where all the intermediate results were computed.

This boilerplate code involved in this approach is not too extensive, since the definition of the *point-free* language is very succinct. The main disadvantage is the fact that the solution is

[2] http://www.haskell.org/happy

not so general, since instead of using a generic monad, it uses a particular monad (all results have a specific type, and are not parameterised), and the fact that all the combinators to traverse terms needed to be defined (but so far only a few were used).

The 5th version is, as expected, much more efficient, not only in the traversal of terms, but also at the compilation or interpretation of the intermediate result. This is just because the generic libraries are no longer compiled, making the compilation process faster. The user CPU time obtained with the `time` command is, in the 4th approach, around **10.5 seconds**, while in the 5th approach is around **6.0 seconds**. The difference is not so big in the second compilation, because the generic libraries are already compiled. In this case the 4th approach took about 5.2 seconds, while the 5th one took 4.5 seconds.

Application of transformations

The last step consists on the application of the rules and strategies to the expressions. It can be considered to be the most important stage in this process. The five different approaches were measured using the unix `time` command (including the 5th version, that is expected to be much more efficient, since it is not generic).

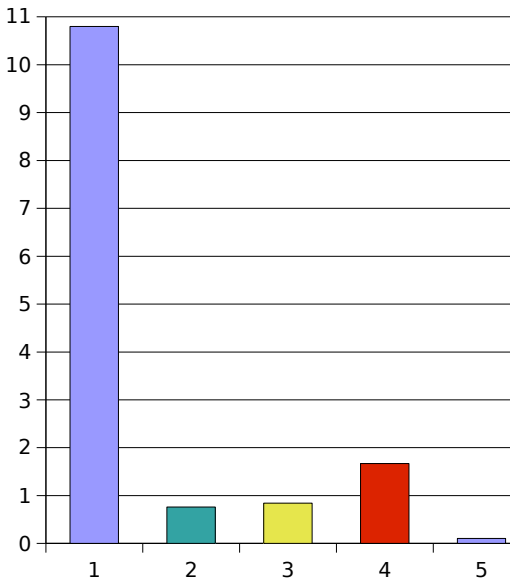


Figure 6.2: User CPU time (in seconds) of the traversal of terms for each approach

The approximate user CPU time of each approach can be seen in figure 6.2. The worse result is definitely the traversal using the SyB approach, which took more than 10 seconds to apply the traversals, and only computes the final result (as the 2nd approach that took around 0.76 seconds). This was the main reason why only the libraries using *Strafunski* were developed. In the 4th approach, the application of each rule involves collecting the computations that the rule may return, and for this a state monad was introduced in the application of each rule. This lifting of monads was the main responsable for the duplication of the time involved. In the last approach, the removal of generic traversals generated a much

more efficient code, that took around 0.11 seconds to execute.

So, using the 5th approach, the total time to obtain a simplified *Haskell* file is:

$$0.33(\mathsf{SimpliFree}) + 6.0(\text{compilation}) + 0.11(\text{transformations}) = 6.44s$$

Chapter 7

Conclusions and Future Work

The main goal of this project was the automatic simplification of expressions written in a *point-free* style. Using the SimpliFree tool this is possible with none or very little human intervention. Other issues like the definition of refactorings to convert *Haskell* to *point-free* and to remove some syntactic sugar (to be applied before the conversion to *point-free*), and program transformation of *point-free* expressions based on rules and strategies, were also covered.

Refactorings. Two refactorings were developed. In the first a translation from *pointwise* to *point-free* code is performed. This translation is applied to expressions and not to function definitions. There are some minor differences with respect to the syntax of the recognised *pointwise* code. The second refactoring tries to address one of DrHylo's problems: the small subset of *Haskell* code covered. For that the removal of guards was studied, which proved to be a more complicated task than expected.

The SimpliFree tool. The proposed goals were to simplify complex *point-free* terms in an automated way and with the minimum intervention possible. For that purpose a special syntax was created to be used within the code (this concept is known as *active source*). With this tool it is possible to give some hints (not always needed), to guide the simplification process of the terms. It is also possible to define strategies powerful enough to apply program transformations like the cata-fusion (in some cases).

The visualisation of the intermediate steps was considered important, so the user can have some feedback when choosing the right hints to give. A number of powerful strategy combinators are also available, so the user can describe transformations using a much richer syntax than would be possible using only rules.

The fact that an intermediate file had to be produced may be seen as a disadvantage, but it allows to implement pattern-matching more efficiently, which was considered prioritary in this project.

Future Work

The first of the two implemented refactorings adresses the conversion from *pointwise* to *point-free*. The final solution resembles the DrHylo results, and there is a clear reason for this. When this library was created, the development of DrHylo was still in progress and at that stage of DrHylo it was not clear yet whether the translation of *pointwise* to *point-free* should be a part of the tool or not, since the main goal was the derivation of hylomorphisms from explicit recursion. Most formal and implementation issues concerning the implemented refactoring were already presented in [Cun05, Pro05b].

In the second refactoring the main concern was the removal of guards. It manages to remove guards in most cases, but there are still some cases that involve a more complex semantic analysis to make this removal possible. The main motivation of this refactoring was to enrich the language covered by tools like DrHylo, that do not recognise guards. Other refactorings to remove syntactic sugar could have been developed, but the automatic simplification of the resulting *point-free* expressions from DrHylo was considered to be a priority.

In the SimpliFree tool, in spite of the very satisfactory results, there are several issues that can still be improved. An important issue that was not approached here was the **formal validation** of the implemented algorithm for pattern matching. This means that the soundness of the produced functions in the conversion of rules, as well as the way they are produced, is not proved yet. This would be specially important in this project, since every other step involved in the conversion from *pointwise* already have a strong theoretical basis.

A second important issue is the **type** information. It is possible to infer types using the *Haskell* pattern matching mechanism (as the construction of rules in SimpliFree), as shown in [Cun05]. Using types, some rules that could not be applied in this system may be possible to apply. But this would require big changes not only in the syntax tree, but also in all tools previously defined for untyped *point-free* expressions. The untyped approach taken in the development of this tool already proved to be very satisfactory.

There are also other issues that can be improved, namely:

- The *cata-fusion* for lists may still require some previous manipulation, as mentioned in section 6.4.3, and a more generic solution for *cata-fusion* parameterised by the associated base functor may be possible;
- A more direct and hidden interaction with the *DrHylo* tool may be a good idea, so that in a final stage the user only asks to convert to *point-free* and obtains the simplified expression (although in some proofs human interaction may still be necessary);
- The strategies defined in the rules repository created for this tool could still be improved, and new strategies could also be defined. A friendly way of inserting new strategies to the repository may also be a good idea, but at the moment the rules are all compiled with the program, not imported at run time;
- It is very easy to define strategies that create infinite loops. A loop detection mechanism to detect circularities would be a way of preventing these cases;

- When simplifying an expression it is possible to obtain a *Computation* with the intermediate steps. A textual view was defined for these computations, but there could be a way to convert the result to other formats, like LaTeX expressions.

Bibliography

[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.

[CPP05] Alcino Cunha, Jorge Sousa Pinto, and José Proença. Down with variables. Technical Report DI-PURe-05.06.01, Departamento de Informática, Universidade do Minho, June 2005.

[Cun05] Alcino Cunha. *Point-free Program Calculation*. PhD thesis, Departamento de Informática, Universidade do Minho, 2005.

[Cur93] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhuser, 2nd edition, 1993.

[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.

[FM91] Maarten Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

[HHJK04] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the programatica toolset. 2004. Presented at the High Confidence Software and Systems Conference, HCSS04.

[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.

[HL78] Grard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[Jon00] Mark Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[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 Lmmel 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] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Haskell '03: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2003. ACM Press.

[LV02a] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[LV02b] Ralf Lämmel and Joost Visser. Design Patterns for Functional Strategic Programming. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.

[LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.

[Mec05] Serge Mechveliani. *Term rewriting, Equational reasoning, Automatic proofs.* Program Systems Institute, Pereslavl-Zalessky, Russia, 2005. Manual on the the Dumatel - 1.02 system.

[Mee92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[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.

[Pro05a] José Proença. Point-free simplification. Technical Report DI-PURe-05.06.02, Universidade do Minho, 2005.

[Pro05b] José Proença. Tranformações pointwise - point-free. Technical Report DI-PURe-05.02.01, Departamento de Informática, Universidade do Minho, February 2005.

[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.

Appendix A

Examples: Removal of Guards

In section 5.2 a refactoring to remove guards was described. In this section some examples of the application of this refactoring are shown. In some cases the guards cannot be removed.

Crossed merge
Original code:

```
-- Since the four matches are disjoint, it is possible to swap them
--  to merge matches before removing guards.
crossed (h:t1) x | h == 0 = x
crossed []     y | y > 0  = 1
crossed (h:t2) z | otherwise = -z
crossed []     v | v < 0  = -1
```

Resulting code:

```
module Crossed where

-- since the four matches are disjoint, it is possible to swap them
--  to merge matches before removing guards.
crossed ((h : xx_0)) xx_1
    = if h == 0 then xx_1 else -xx_1
crossed [] xx_2
    =   if xx_2 > 0
        then 1
        else if xx_2 < 0
             then -1
             else error "UnMatched Pattern"
```

With declarations
Original code:

```
-- The declaration will be converted into a let inside the expressions.

f x y | x < 0 = x where x = y
```

```
f u v | u > 0 = -v
f a b | otherwise = 0
```

Resulting code:

```
-- The declaration will be converted into a let inside the expressions.

f xx_2 xx_3
    =   if let x = xx_3 in x < 0
        then let x = xx_3 in x
        else if xx_2 > 0 then -xx_3 else 0
```

Irrefutable patterns
Original code:

```
-- A let will be added to the expressions.

func :: Maybe Int -> Int
func ~(v@(Just x)) | isJust v = x
func v             | otherwise = 0
```

Resulsting code:

```
-- A let will be added to the expressions.

func :: Maybe Int -> Int
func xx_1
    =   if let (v@(Just x)) = xx_1 in isJust v
        then let (v@(Just x)) = xx_1 in x
        else 0
```

User data type - Success
Original code:

```
data X a = A | B a | C {x,y::a}

-- only the second match will not be merged, and the otherwise case is not
--  present in none of the expressions.

f (C {x = myx}) ([],0)    x | x < 0  = myx
f A             (lst,0)   x | x == 1 = head lst  -- disjoint from others
f (C a b)       r@([] ,0) x | x > 0  = snd r
f (B 0)         (a,b)     x | x == 2 = b
f ((B 0))       (a,c)     x | x == 3 = -c
```

Resulting code:

```
data X a = A | B a | C {x,y::a}

-- only the second match will not be merged, and the otherwise case is not
--  present in none of the expressions.
```

```
f A (lst, 0) x
    =   if x == 1
        then head lst
        else error "UnMatched Pattern"
f (C xx_0 xx_1) xx_2@([], 0) x
    =   if x < 0
        then xx_0
        else if x > 0
             then snd xx_2
             else error "UnMatched Pattern"
f (B 0) (a, xx_3) x
    =   if x == 2
        then xx_3
        else if x == 3
             then -xx_3
             else error "UnMatched Pattern"
```

User data type - Fail
Original code:

```
data X a = A | B a | C {x,y::a}

-- Same as in previous example, but the last match makes every other match
--  inconsitent, because after converting to an if then else the last match
--  may become impossible to reach.
-- So no alterations will be made, except for some indentation differences.

f (C {x = myx}) ([],0)    x | x < 0  = myx
f A             (lst,0)   x | x == 1 = head lst  -- disjoint from others
f (C a b)       r@([] ,0) x | x > 0  = snd r
f (B 0)         (a,b)     x | x == 2 = b
f ((B 0))       (a,c)     x | x == 3 = -c
f _             _         _ = 0
```

Resulting code:

```
data X a = A | B a | C {x,y::a}

-- Same as in previous example, but the last match makes every other match
--  inconsitent, because after converting to an if then else the last match
--  may become impossible to reach.
-- So no alterations will be made, except for some indentation differences.

f (C{x = myx}) ([], 0) x | x < 0 = myx
f A (lst, 0) x | x == 1 = head lst
f (C a b) r@([], 0) x | x > 0 = snd r
f (B 0) (a, b) x | x == 2 = b
f ((B 0)) (a, c) x | x == 3 = -c
f _ _ _ = 0
```

Appendix B

Rules repository

To facilitate the simplification of terms when using the SimpliFree tool, several rules and strategies were defined. These rules can be imported by activating the corresponding option, as an argument on the command line, or as a special annotated block. In most cases these rules and strategies are enough to simplify the *point-free* terms.

B.1 Base strategy

Several strategies were here created with the purpose of being redefined later, if the user wants to reuse the strategy and add something more:

- `base_pre`
- `base_pos`
- `macros_fold`
- `macros_unfold`

The strategy consists on applying a set of rules as much as possible, then unfold the known macros, and loop this process until no rules can be applied. In the end the known macros are folded again to make the reading easier.

The main strategy is called `base_strat`, and its definition is as follows.

```
{- Strategies:
base_strat : base_tmp1 and (many base_fMacro)

base_tmp1  : base_tmp2 and (opt ((oneOrMore base_unfMacro) and base_tmp1))
base_tmp2  : many base_rules
base_pre      : fail
base_pos      : fail
macros_fold   : fail
macros_unfold : fail
base_rules : base_pre or base_simpl or base_comp_ins or base_pos
```

```
base_simpl : ((many prodFusInv) and eitherConst)
             or natId1 or natId2 or prodCancel1 or prodCancel1'
             or prodCancel2 or prodCancel2' or prodRefl or sumCancel1
             or sumCancel1' or sumCancel2 or sumCancel2' or sumRefl
             or expCancel or constFus
base_comp_ins : prodAbs or prodFun or prodFus or sumAbs or sumFun or sumFus
                or expCancAdv1 or expCancAdv2 or expCancAdv3 or expCancAdv4
                or expFus
base_unfMacro : exp_unfold or pxe_unfold or unpnt_unfold or pnt_unfold
                or swap_unfold or coswap_unfold or distl_unfold
                or distr_unfold or split_unfold or macros_unfold
base_fMacro   : exp_fold or pxe_fold or unpnt_fold or swap_fold or coswap_fold
                or distl_fold or distr_fold or split_fold or either_fold
                or expFus_fold or macros_fold
-}

-- simplification
{- Rules:
natId1 : id . f -> f
natId2 : f . id -> f
prodCancel1 : fst . (f /\ g) -> f
prodCancel1': fst . (f >< g) -> f.fst
prodCancel2 : snd . (f /\ g) -> g
prodCancel2': snd . (f >< g) -> g.snd
prodRefl    : fst /\ snd -> id >< id
sumCancel1  : (f \/ g)  . Left  -> f
sumCancel1' : (f -|- g) . Left  -> Left . f
sumCancel2  : (f \/ g)  . Right -> g
sumCancel2' : (f -|- g) . Right -> Right . g
sumRefl     : Left \/ Right -> id -|- id
expCancel   : app . ((curry f) >< id) -> f
constFus    : ('pnt' [f]) . g -> 'pnt' [f]
eitherConst : 'eithr'.((curry (f.snd)) /\ (curry (g.snd))) -> curry ((f \/ g).snd)
prodFusInv  : (f.h) /\ (g.h) -> (f/\g) . h
-}
-- composition to sub terms
{- Rules:
prodAbs  : (i><j) . (g/\h) -> (i.g) /\ (j.h)
prodFun  : (f><g) . (h><i) -> (f.h) >< (g.i)
prodFus  : (f/\g) . h -> (f.h) /\ (g.h)
sumAbs   : (g\/h) . (i-|-j) -> (g.i) \/ (h.j)
sumFun   : (f-|-g) . (h-|-i) -> (f.h) -|- (g.i)
sumFus   : f . (g\/h) -> (f.g) \/ (f.h)
expFus   : (curry g) . f -> curry (g . (f >< id))
expCancAdv1: app . (((curry f) . g)><h) -> f . (g><h)
expCancAdv2: app . ((curry f)><h) -> f . (id><h)
expCancAdv3: app . (((curry f) . g)/\h) -> f . (g/\h)
expCancAdv4: app . ((curry f)/\h) -> f . (id/\h)
-}
-- fold and unfold of macros
{- Rules:
exp_unfold   : 'exp' [f] -> curry (f . app)
pxe_unfold   : 'pxe' [f] -> curry (app . (id >< f))
```

```
unpnt_unfold : 'unpnt' [f] -> app . ((f.bang) /\ id)
pnt_unfold   : 'pnt' [f]-> curry (f.snd)
swap_unfold  : 'swap' -> snd /\ fst
coswap_unfold: 'coswap' -> Right \/ Left
distl_unfold : 'distl' -> app . (((curry Left)\/(curry Right)) >< id)
distr_unfold : 'distr' -> ('swap'-|-'swap') . ('distl' . 'swap')
split_unfold : 'split' -> curry ((app.(fst >< id)) /\ (app.(snd >< id)))
either_unfold: 'eithr' -> curry ((app \/ app) . (((fst >< id) -|- (snd >< id)) . 'distr'))
exp_fold   : curry (f . app) -> 'exp' [f]
pxe_fold   : curry (app . (id >< f)) -> 'pxe' [f]
unpnt_fold : app . ((f.bang) /\ id) -> 'unpnt' [f]
pnt_fold   : curry (f.snd) -> 'pnt' [f]
swap_fold  : snd /\ fst -> 'swap'
coswap_fold: Right \/ Left -> 'coswap'
distl_fold : app . (((curry Left)\/(curry Right)) >< id) -> 'distl'
distr_fold : ('swap'-|-'swap') . ('distl' . 'swap') -> 'distr'
split_fold : curry ((app.(fst >< id)) /\ (app.(snd >< id))) -> 'split'
either_fold: curry ((app \/ app) . (((fst >< id) -|- (snd >< id)) . 'distr')) -> 'eithr'
expFus_fold: curry (g . (f >< id)) -> (curry g) . f
-}
```

B.2 Advanced Strategy

Reuses the base strategy and adds some more rules. The reason this is a separated strategy is that some of the rules might not type check.

As in the definition of the base strategy, there are some rules that are meant to be redefined by the user, if needed:

- `adv_pre`
- `adv_pos`
- `macros_fold`
- `macros_unfold`

The main strategy is called `adv_strat`, and its definition is as follows.

```
{- import base_strat }

{- Strategies:
adv_strat : base_strat

adv_rules : adv_pre or base_rules or adv_pos
adv_pre : fail
adv_pos : fail
base_pre : adv_pre or toProd1 or toProd2 or toProd3
base_pos : idTUn1 or idTUn2 or apTUn or bangTUn or prodReflAdv or
           sumReflAdv or unfix1 or unfix2 or expCancAdv5 or
           toSum or toSum2 or
```

```
           toSum3 or fixProdCancel1 or fixProdCancel2 or
           fixProdCancel3 or fixProdCancel4 or fixProdCancel5
           or adv_pos
-}
{- Rules:
idTUn1: id >< id  -> id
idTUn2: id -|- id -> id
apTUn: curry app -> id
bangTUn: bang.f -> bang
prodReflAdv: (fst.f) /\ (snd.f) -> f
sumReflAdv: (f.Left) \/ (f.Right) -> f
expCancAdv5: curry (app . (f><id)) -> f
toProd1: (f.fst) /\ (g.snd) -> f >< g
toProd2: (f.fst) /\ snd -> f >< id
toProd3: fst /\ (g.snd) -> id >< g
toSum:  (Left . f) \/ (Right . g) -> f -|- g
toSum2: (Left . f) \/ Right -> f -|- id
toSum3: Left \/ (Right . g) -> id -|- g
unfix1: 'fix' . (curry fst)         -> id
unfix2: 'fix' . (curry (curry snd)) -> (curry snd) . bang
fixProdCancel1: 'fix' . (curry (curry snd)) -> curry snd
fixProdCancel2: 'fix' . (curry (curry (f.snd))) -> curry (f.snd)
fixProdCancel3: 'fix' . (curry (curry ((f.snd) /\ (g.snd)))) -> curry ((f/\g).snd)
fixProdCancel4: 'fix' . (curry (curry ((f.snd) /\ snd))) -> curry ((f/\id).snd)
fixProdCancel5: 'fix' . (curry (curry (snd /\ (g.snd)))) -> curry ((id/\g).snd)
-}
{-
expCancAdv5:
curry (app . (f><id)) = {exp fusion}
curry app . f = { exp Refl (not in base rules) }
id . f = f
Without types I think it can be dangerous to use
 - id >< id  -> id
 - id -|- id -> id
 - curry app  -> id
So they are not in base rules
-}
```

B.3 Fusion of catamorphisms for lists

This strategy was created to show that the main idea underneath the fusion of catamorphisms for lists could also be generalised in a strategy. But in some cases the user still have to add several *hints* before the fusion is possible.

The advanced strategy is used, so some of the strategies to be redefined are still exported:

- `adv_pos`
- `macros_fold`
- `macros_unfold`

The main strategy is called `cataList_strat`, and its definition is as follows.

```
{- import adv_strat -}

{- Strategies:
cataList_strat : adv_strat

adv_pre : cataList
cataList_rules : fail

cataList : f . ('cataList' [g1\/g2]) ->
                'cataList' [(f.g1) \/ (apply getH2 [f,f.g2])]

getH2 : extractH2 or (cataList_step and getH2)

extractH2 : extractH2A or extractH2B or extractH2C or extractH2D
extractH2A : [f,a.(b >< (c.f))] -> a.(b><c)
extractH2B : [f,a.(b >< f)] -> a.(b><id)
extractH2C : [f,b >< (c.f)] -> b><c
extractH2D : [f,b >< f] -> b><id

cataList_step : cataList_rules or swapLeft or adv_rules or base_unfMacro

swapLeft : (f >< g) . 'swap' -> 'swap' . (g >< f)
-}
```

Appendix C

Examples: DrHylo and SimpliFree

In this chapter a *Haskell* file with different known functions will be presented, that will be translated first into *point-free* with the DrHylo tool, and later simplified with the SimpliFree tool

C.1 Original *Haskell* file

The original file to be tested is presented bellow.

```
module Sample where

import Pointless.Functors

comp :: (b->c, a->b) -> (a->c)
comp (f,g) y = f (g y)

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

assocr :: ((a,b),c) -> (a,(b,c))
assocr ((x,y),z) = (x,(y,z))

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

coswap :: Either a b -> Either b a
coswap (Left  y) = Right y
coswap (Right y) = Left  y

undistr :: Either (a,b) (a,c) -> (a, Either b c)
undistr (Left  (y,z)) = (y, Left  z)
undistr (Right (x,z)) = (x, Right z)

data Nat = Zero | Succ Nat deriving Show

plus :: (Nat, Nat) -> Nat
```

```
plus (Zero,   z) = z
plus (Succ n, z) = Succ (plus (n,z))

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

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

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

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

cat :: [a] -> [a] -> [a]
cat []    l = l
cat (h:t) l = h:(cat t l)

data Tree a = Leaf | Node a (Tree a) (Tree a)

inorder :: Tree a -> [a]
inorder Leaf = []
inorder (Node x l r) = cat (inorder l) (x:(inorder r))
```

C.2 DrHylo Results

In this section the modified *Haskell* file with *point-free* expressions obtained by the DrHylo tool are presented. The DrHylo tool was called by the following line:

```
$ drhylo < foo.hs > foo_drhylo.hs
```

```
module Sample where
import Pointless.Functors
import Pointless.Combinators
import Pointless.Combinators.Uncurried
import Pointless.RecursionPatterns

comp :: (b -> c, a -> b) -> a -> c
comp
  = app .
      (((curry
           (curry
              (app .
                 ((fst . (fst . ((snd . fst) /\ snd))) /\
```

```
                    (app .
                       ((snd . (fst . ((snd . fst) /\ snd))) /\
                          (snd . ((snd . fst) /\ snd))))))))
          . bang)
         /\ id)

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

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

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


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

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

data Nat = Zero
         | Succ Nat
```

```
         deriving Show

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

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

fact :: Nat -> Nat
fact
  = hylo (_L :: Mu ((:+:) (Const One) ((:*:) (Const a0) Id)))
      (app .
         (((curry
              (app .
                 ((eithr .
```

```
                     ((curry (inn . (inr . (inn . (inl . bang))))) /\
                        (curry
                           (app .
                              (((pnt mult) . bang) /\
                                 ((inn . (inr . (fst . snd))) /\ (snd . snd)))))))
                  /\ snd)))
             . bang)
            /\ id))
      (app .
         (((curry
              (app .
                 ((eithr . ((curry (inl . bang)) /\ (curry (inr . (snd /\ snd)))))
                    /\ (out . snd))))
             . bang)
            /\ id))

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

len :: [a] -> Nat
```

```
len
  = hylo (_L :: Mu ((:+:) (Const One) Id))
      (app .
         (((curry
              (app .
                 ((eithr .
                     ((curry (inn . (inl . bang))) /\ (curry (inn . (inr . snd)))))
                    /\ snd)))
             . bang)
            /\ id))
      (app .
         (((curry
              (app .
                 ((eithr . ((curry (inl . bang)) /\ (curry (inr . (snd . snd))))) /\
                    (out . snd))))
             . bang)
            /\ id))

cat :: [a] -> [a] -> [a]
cat
  = app .
      (((fix .
           (curry
              (curry
                 (curry
                    (app .
                       ((eithr .
                           ((curry (snd . (((snd . fst) . fst) /\ (snd . fst)))) /\
                              (curry
                                 (inn .
                                    (inr .
                                       ((fst . snd) /\
                                          (app .
                                             ((app . ((((snd . fst) . fst) . fst) /\ (snd . snd)))
                                                /\
                                                (snd . (((snd . fst) . fst) /\ (snd . fst))))))))))
                          /\ (out . (fst . ((snd . fst) /\ snd))))))))))
          . bang)
         /\ id)

data Tree a = Leaf
            | Node a (Tree a) (Tree a)

inorder :: Tree a -> [a]
inorder
  = hylo
      (_L :: Mu ((:+:) (Const One) ((:*:) Id ((:*:) (Const a0) Id))))
      (app .
         (((curry
              (app .
                 ((eithr .
                     ((curry (inn . (inl . bang))) /\
                        (curry
```

```
                          (app .
                             ((app . (((pnt cat) . bang) /\ (fst . snd))) /\
                                (inn .
                                   (inr .
                                      ((fst . (fst . (snd . snd))) /\ (snd . (snd . snd))))))))))
                    /\ snd)))
              . bang)
             /\ id))
      (app .
         (((curry
              (app .
                 ((eithr .
                     ((curry (inl . bang)) /\
                        (curry
                           (inr . ((fst . (snd . snd)) /\ (snd /\ (snd . (snd . snd))))))))))
                    /\ (out . snd))))
              . bang)
             /\ id))

instance FunctorOf ((:+:) (Const One) Id) Nat where
        inn' (Inl (Const _)) = Zero
        inn' (Inr (Id v1)) = Succ v1
        out' (Zero) = Inl (Const _L)
        out' (Succ v1) = Inr (Id v1)

instance FunctorOf
         ((:+:) (Const One) ((:*:) (Const a) ((:*:) Id Id))) (Tree a) where
        inn' (Inl (Const _)) = Leaf
        inn' (Inr ((Const v1 :*: (Id v2 :*: Id v3)))) = Node v1 v2 v3
        out' (Leaf) = Inl (Const _L)
        out' (Node v1 v2 v3) = Inr ((Const v1 :*: (Id v2 :*: Id v3)))
```

C.3 SimpliFree Results

In this section the modified *Haskell* file with simplified *point-free* expressions obtained by the SimpliFree tool are presented. The SimpliFree tool was called by the following line:

```
$ SimpliScript.sh -i adv_strat foo_drhylo.hs foo_simplifree.hs
```

```
module Sample where
import Pointless.Functors
import Pointless.Combinators
import Pointless.Combinators.Uncurried
import Pointless.RecursionPatterns

comp :: (b -> c, a -> b) -> a -> c
comp = curry (app . ((fst . fst) /\ (app . (snd >< id))))

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

```
assocr :: ((a, b), c) -> (a, (b, c))
assocr = (fst . fst) /\ (snd >< id)

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

coswap :: Either a b -> Either b a
coswap = coswap

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

data Nat = Zero
     | Succ Nat
     deriving Show

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

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

fact :: Nat -> Nat
fact
  = hylo (_L :: Mu ((:+:) (Const One) ((:*:) (Const a0) Id)))
      (app .
     ((eithr .
         ((curry (inn . (inr . (inn . (inl . bang))))) /\
        (curry (mult . ((inn . (inr . (fst . snd))) /\ (snd .
snd))))))
        /\ id))
      (app .
     ((eithr . ((curry (inl . bang)) /\ (curry (inr . (snd /\ snd)))))
```

```
        /\ out))

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

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

cat :: [a] -> [a] -> [a]
cat
  = app .
      ((fix .
      (curry
         (curry
        (curry
           (app .
              ((eithr .
              ((curry (snd . fst)) /\
                 (curry
                (inn .
```

```
                    (inr .
                       ((fst . snd) /\
                      (app .
                         ((app . ((snd . (fst . fst)) >< snd)) /\
                            (snd . fst)))))))))
              /\ (out . (snd . fst))))))))
     /\ id)

data Tree a = Leaf
        | Node a (Tree a) (Tree a)

inorder :: Tree a -> [a]
inorder
  = hylo
      (_L :: Mu ((:+:) (Const One) ((:*:) Id ((:*:) (Const a0) Id))))
      (app .
     ((eithr .
         ((curry (inn . (inl . bang))) /\
        (curry
           (app .
              ((cat . (fst . snd)) /\
             (inn .
                (inr . ((fst . (fst . (snd . snd))) /\ (snd . (snd .
snd)))))))))
        /\ id))
      (app .
     ((eithr .
         ((curry (inl . bang)) /\
        (curry
           (inr . ((fst . (snd . snd)) /\ (snd /\ (snd . (snd .
snd))))))))
        /\ out))

instance FunctorOf ((:+:) (Const One) Id) Nat where
    inn' (Inl (Const _)) = Zero
    inn' (Inr (Id v1)) = Succ v1
    out' (Zero) = Inl (Const _L)
    out' (Succ v1) = Inr (Id v1)

instance FunctorOf
     ((:+:) (Const One) ((:*:) (Const a) ((:*:) Id Id))) (Tree a)
where
    inn' (Inl (Const _)) = Leaf
    inn' (Inr ((Const v1 :*: (Id v2 :*: Id v3)))) = Node v1 v2 v3
    out' (Leaf) = Inl (Const _L)
    out' (Node v1 v2 v3) = Inr ((Const v1 :*: (Id v2 :*: Id v3)))
```