Point-free Simplification

José M. P. Proença
jproenca@di.uminho.pt

Techn. Report DI-PURe-05.08.01
2005, August

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

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

DI-PURe-05.08.01
Point-free *Simplification* by José M. P. Proença

Abstract

A collection of libraries and tools that support the automatic conversion of programs to *point-free* and some manipulation, were recently developed as part of the *UMinho Haskell Libraries*. The process used in the calculation of *point-free* terms produce complex terms that, in most cases, can be simplified.

In this work rule-driven simplification and transformation of *point-free* terms are studied in more detail. For this purpose a tool called SimpliFree was developed. The main idea is to automate the construction of a *Haskell* file that applies some given strategies to *point-free* terms, by the use of generic traversals.

Some default strategies were included in the tool, that can simplify most terms and can be easily updated by the user, to minimise the needed intervention.

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).

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.

To join the advantages of both styles, the translation of *pointwise* to pure *point-free* was previously studied in [Cun05,Pro05]. For this matter, a tool called DRHYLO [Cun05] was created, that can not only translate *pointwise Haskell* definitions to *point-free* expressions, but it can also remove explicit recursion and convert it to the hylomorphism recursion pattern. One of the main problems with this conversion to *point-free* code 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 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, where the code is annotated with special commented blocks containing information on how to apply transformations to the code.

A more complete introduction to the SIMPLIFREE tool is made after some background introduction, in section 4.

Report structure

In the first sections some theoretical background is presented. In section 2 the *point-free* language is introduced and defined, and some work around this style is also referred. In the next section two generic approaches to traverse terms are presented: using the *Scrap your boilerplate* approach (subsection 3.1) and using the generic libraries in *Strafunski* software bundle (subsection 3.2).

In the rest of the report the SimpliFree tool is explored. A more descriptive introduction is made in section 4. The explanation of the tool is made in two main sections: in section 5 the problem of defining generic traversals for this particular case is explored; and in section 6 the construction of monadic functions capable of applying rules, based on the rule description, is explained in detail. Finally, in section 7, several strategies are tested and explained, starting by very simple simplification strategies and finishing by strategies that applied the cata-fusion law.

In the last section a global vision of the SimpliFree tool is performed, together with some ideas for future work.

2 Programming in *point-free*

The *point-free* style of programming was first introduced in 1977, by John Backus, in an ACM Turing Award talk [Bac78]. He introduced several combinators used in the *point-free Haskell* library.

In the *UMinho Haskell Libraries* several libraries related to the *point-free* programming are defined:

- `Language/Pointfree` – to allow for the manipulation of typed and untyped *point-free* terms;
- `Language/Pointwise` – to allow for the manipulation of *pointwise* terms, to be later converted to *point-free*;
- `Pointless` – to allow for the type-checking and execution of *point-free* code with recursion patterns, parameterised by data types, using a similar syntax to the theoretical notation, to be introduced below.

A tool called drhylo, that is able to perform the *pointwise*/*point-free* translation (in most cases), is also available from the same repository.

Point-free. In the *point-free* language, types are defined according to the syntax in figure 1.

For example, booleans can be expressed as $\mathsf{Bool} = 1+1$, and lists with elements of type A as $\mathsf{List}A = \mu(\underline{1} \oplus \underline{A} \otimes \mathsf{Id})$.

A *point-free* term can be defined by using a set of combinators enumerated in figure 2. These combinators come from universal constructors in categories with products, non-empty sums, exponentials, and terminal object.

The subscripts of in and out are omitted when clear from context. The swap function, for example, can be defined in *point-free* as $\mathsf{snd} \vartriangle \mathsf{fst}$.

$$
\begin{array}{rll}
A, B ::= & 1 & - \text{ single element type} \\
| & A \to B & - \text{ continuous functions from } A \text{ to } B \\
| & A \times B & - \text{ cartesian product} \\
| & A + B & - \text{ separated sum} \\
| & \mu F & - \text{ recusrive (regular) type, defined as the fixed point of a functor} \\
F, G ::= & \mathsf{Id} & - \text{ identity functor} \\
| & \underline{A} & - \text{ constant functor that always returns } A \\
| & F \otimes G & - \text{ lifted product bifunctor} \\
| & F \oplus G & - \text{ lifted sum bifunctor} \\
| & F \circledcirc G & - \text{ composition of functors}
\end{array}
$$

Fig. 1. *Point-free* type syntax (A and B)

$$
\begin{array}{rll}
\mathsf{fst} & : A \times B \to A & - \text{ projection of the first element of a pair} \\
\mathsf{snd} & : A \times B \to B & - \text{ projection of the second element of a pair} \\
\mathsf{inl} & : A \to A + B & - \text{ injection on the left} \\
\mathsf{inr} & : B \to A + B & - \text{ injection on the right} \\
\mathsf{in}_{\mu F} & : F(\mu F) \to \mu F & - \text{ construct values of a given type} \\
\mathsf{out}_{\mu F} & : \mu F \to F(\mu F) & - \text{ inspect values of a given type} \\
\mathsf{id} & : A \to A & - \text{ identity function} \\
\mathsf{bang} & : A \to 1 & - \text{ constant function that returns the least element} \\
\mathsf{ap} & : (A \to B) \times A \to B & - \text{ application of a function to an element} \\
(\cdot \circ \cdot) & : (B \to C) \to (A \to C) \to A \to C & - \text{ composition of functions} \\
(\cdot \vartriangle \cdot) & : (A \to B) \to (A \to C) \to A \to (B \times C) & - \text{ split combinator} \\
(\cdot \triangledown \cdot) & : (A \to C) \to (B \to C) \to (A + B) \to C & - \text{ either combinator} \\
\bar{\cdot} & : (A \times B \to C) \to A \to B \to C) & - \text{ curry combinator}
\end{array}
$$

Fig. 2. Combinators that define the *point-free* language

Recursion is expressed by a special combinator: the *hylomorphism*. This recursion pattern was proved to be powerful enough for the definition of any fixed point [MH95], and it can be defined as:

$$\begin{aligned}&\mathsf{hylo}_{\mu F} : (F\ B \to B) \to (A \to F\ A) \to A \to B\\&\mathsf{hylo}_{\mu F}\ g\ h = g \circ F(\mathsf{hylo}_{\mu F}\ g\ h) \circ h\end{aligned}$$

where h computes the values passed to the recursive calls and g computes the results from the recursive call, and returns the final result. μF models the recursion tree of a function defined as a hylomorphism. For example, the disjunction of a list of booleans can be defined as the following hylomorphism:

$$\begin{aligned}&or : \mathsf{List\ Bool} \to \mathsf{Bool}\\&or = \mathsf{hylo}_{\mathsf{List\ Bool}}\ (\mathsf{inr} \triangledown \overline{or\_bin})\ \mathsf{out}_{\mathsf{List\ Bool}}\end{aligned}$$

The *or_bin* function is the binary *or* for the boolean type mentioned above, equivalent to the *Haskell* operator `&&`.

3 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.

3.1 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. Lets 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 lets suppose that we want to do two different operations:

1. change the names (of functions, variables, ...) that began with *normalize* to *normalise*;
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.

3.2 The Strafunski approach

Strafunski is a software bundle for implementing language processing documents [LV03]. In this work we will focus 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 definition of 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 3.

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]
```

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

Fig. 3. *Some basic strategy combinators*

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.

4 The SimpliFree tool

One of the main problems with the conversion to *point-free* code described in section 2 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 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 [dMS99]: 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 4 illustrates the way files are organised when using the tool.

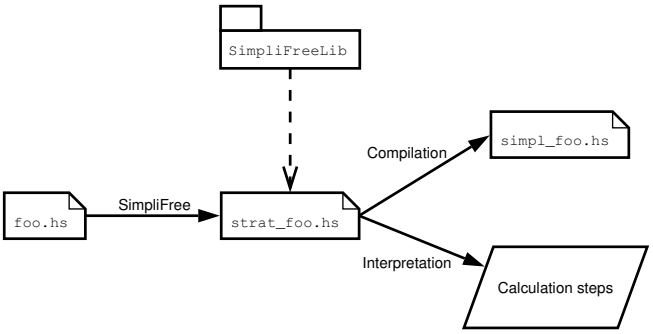

Fig. 4. Diagram with SimpliFree architecture

5 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 was used the Strafunski library, 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.

5.1 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, 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.

5.2 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.

5.3 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 5.2, 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)
```

5.4 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.

5.5 Builtin strategies

To facilitate the user task, a few default strategies were included in the SimpliFree tool. 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 A. There is a base strategy (appendix A.1) that is used in the creation of other strategies, like the advanced strategy (appendix A.2). The base strategy not only simplification rules but also rules to fold and unfold several known macros, like swap ($= \mathsf{snd} \triangledown \mathsf{fst}$) 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 Rule construction

6.1 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. $\mathsf{natId1} : id \circ f \to f$

```
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.

6.2 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 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 6.2.

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
```

[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.

ii) Only in some cases – The pattern matching succeeds, but the *ending* case needs to be considered (as described in section 6.2).

    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)
    ```

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 6.2, 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 (6.4). 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 7.

6.4 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 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 conditions. 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 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 were now changed, as described in table 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)) = Just ((t:.:x):xs, e)`	`addComp t lst = [((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 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. 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.

6.5 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 6.4);
– 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 7.3). This is internally represented as a macro with a special name that is ignored when printing.

7 Testing Strategies

7.1 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 5.3):

```
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.

7.2 *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 patter. 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 5.5, 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 A.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'
```

7.3 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. $\mathsf{cataList\_fusion} : f.(\!|g|\!)_{\mathsf{List}} = (\!|h|\!)_{\mathsf{List}} \quad \Leftarrow \quad f \circ g = h \circ (\mathsf{List}\ f)$

```
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.

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

So, if the difficult step $\dagger$ 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 6.5, 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 A.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 6.5) 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.

7.4 Cata-Fusion for Rose Trees

In a very similar way to the strategy defined in the previous section (7.3), 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.

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

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.

8 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 testing will be done with a sample file resulting from drhylo, that includes the functions tested in section 7.2, together with the advanced strategy pack described in appendix A.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 testing file, 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.

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

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 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).

The approximate user CPU time of each approach can be seen in figure 5. 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:

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

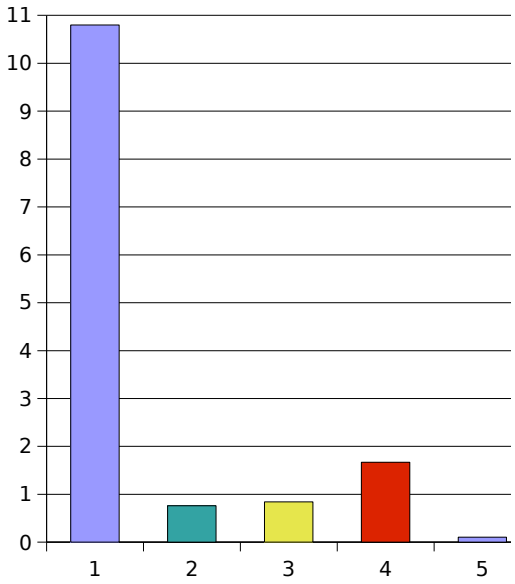

Fig. 5. User CPU time in second of the traversal of terms for each approach

9 Conclusion and future work

The proposed goals were to simplify complex *point-free* terms in an automated way and with the minimum intervention possible. For that 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 for more efficient pattern matching with the terms, which was considered more relevant in this project.

Future Work

An important issue that was not approached here was the **formal validation** of the implemented algorithm for pattern matching. When a *left variable* is found in a rule definition, auxiliary functions that take advantage of *Haskell* pattern matching are used, to check for conditions and to produce the resulting expression. It would be important to prove the the soundness of the expressions produced, as well as the way the expressions are produced. This is specially important in this project since every other step in the conversion from *pointwise* is based on a strong theoretical basis.

Another important issue is the **type** information. It is possible to infer types using the *Haskell* pattern matching mechanism in a similar way to what was done in SimpliFree to apply rules, 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 (that is not very hard to do), but also on all tools previously defined for untyped *point-free* expressions. The untyped approach in this tool was necessary before moving to the typed expressions, and the results obtained without types were already 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 7.3, and a more general solution for *cata-fusion* that looks at 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 enter in infinite loops. A loop detection mechanism to detect circularities would be a way of preventing these cases, that was not implemented;
- 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.

References

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.

Cun05. Alcino Cunha. *Point-free Program Calculation*. PhD thesis, University of Minho, 2005.

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.

LJ03. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37. ACM Press, 2003.

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.

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.

Pro05. José Proença. Transformações pointfree - pointwise. Technical report, University of Minho, Department of Informatics, 2005.

A Rules repository

To facilitate the simplification of terms when using the SimpliFree tool, several rules and strategies were defined. This 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.

A.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
-}
```

A.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
-}
```

A.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)
-}
```