Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 229 (5) (2011) 57–74
www.elsevier.com/locate/entcs

Algebraic Specialization of Generic Functions for Recursive Types

Alcino Cunha[1] and Hugo Pacheco[2]

Dep. de Informática (Centro de Ciências e Tecnologias da Computação), Universidade do Minho, Campus de Gualtar, P-4710-057, Braga, Portugal

Abstract

Defining functions over large, possibly recursive, data structures usually involves a lot of boilerplate. This code simply traverses non-interesting parts of the data, and rapidly becomes a maintainability problem. Many generic programming libraries have been proposed to address this issue. Most of them allow the user to specify the behavior just for the interesting bits of the structure, and provide traversal combinators to "scrap the boilerplate". The expressive power of these libraries usually comes at the cost of efficiency, since runtime checks are used to detect where to apply the type-specific behavior.
In previous work we have developed an effective rewrite system for specialization and optimization of generic programs. In this paper we extend it to also cover recursive data types. The key idea is to specialize traversal combinators using well-known recursion patterns, such as folds or paramorphisms. These are ruled by a rich set of algebraic laws that enable aggressive optimizations. We present a type-safe encoding of this rewrite system in Haskell, based on recent language extensions such as type-indexed type families.

Keywords: Generic programming, Recursion patterns, Rewrite systems, Specialization, Type families

1 Introduction

Modeling real-world problems in a functional language typically leads to a large set of recursive data types, each with a lot of different constructors. That is the case, for example, when developing language processing tools, where grammars are represented using a different data type for each non-terminal and a different constructor for each production rule. Similarly, schema-aware XML processing usually involves mapping a huge schema to an equivalent data type, with each of the many elements mapped to a different type. Such proliferation of data types makes it hard to implement even conceptually simple functions, that manipulate a very small subset of the data constructors.

[1] Email: alcino@di.uminho.pt
[2] Email: hpacheco@di.uminho.pt

doi:10.1016/j.entcs.2011.02.016

A classic (but still benign) example used to illustrate this problem is the so called "paradise benchmark" [13]. Suppose one has a XML schema to model a company with several departments, each having a name, a manager and a collection of employees or sub-departments. This schema could be represented by the following Haskell data type.

```
data Company = C [Dept]
data Dept = D Name Manager [Either Employee Dept]
data Employee = E Person Salary
data Person = P Name Address
data Salary = S Int
type Manager = Employee
type Name = String
type Address = String
```

Suppose one also wants to define a function to increase all salaries by a fixed amount k. A possible definition of this function could be

```
increase :: Int → Company → Company
increase k (C ds) = C (map (incD k) ds)
  where incD k (D nm mgr us) = D nm (incE k mgr) (map (incU k) us)
        incU k (Left e)  = Left (incE k e)
        incU k (Right d) = Right (incD k d)
        incE k (E p s) = E p (incS k s)
        incS k (S s) = S (s + k)
```

Even this rather simple definition is filled with boilerplate code, whose only purpose is to perform a standard traversal of the *Company* data type to find salaries to increase. Apart from aesthetical reasons, this kind of boilerplate has some major drawbacks: (1) it makes code understanding rather difficult, since the only interesting functions (in this case *incS*) are lost amid bucketloads of uninteresting code; and (2) it rapidly becomes a maintainability problem, since each model evolution implies changes to many functions that are only concerned with parts of the model not affected by the evolution.

Many generic programming libraries have been proposed to address this issue. Most of them allow the user to specify the behavior just for the interesting bits of the structure, and provide traversal combinators to encode the remaining boilerplate. One of the most successful libraries is the conveniently named "Scrap you Boilerplate" (SYB), first introduced in [13] and subsequently extended with additional functionalities [14]. Using this library, the *increase* function could be redefined as follows.

```
increase :: Int → Company → Company
increase k = everywhere (mkT (incS k))
  where incS k (S s) = S (s + k)
```

The *everywhere* combinator traverses a data structure in bottom-up fashion, applying the given generic transformation to all its nodes. The mkT combinator builds a generic transformation from a type specific one: given a function $f :: a \to a$, $mkT\ f$ behaves like f for all values of type a and like the identity function otherwise. Besides being much easier to understand what *increase* does, its definition will stay the same even if the *Company* data type changes.

Unfortunately, the obvious advantages provided by this style of generic programming come at a price: the performance of generic functions is much worse than analogous non-generic ones. In [18], the SYB implementation of a standard set of benchmark functions was reported to run 7 times slower in average than the non-generic implementation. Part of this performance loss is due to the run-time checks needed to determine at each node whether to apply specific or generic behavior. The remaining is due to structural reasons inherent to this style of generic programming: the traversal combinators must blindly traverse the whole data structure, even if a certain branch does not mention types where the specific behavior applies.

Some new SYB-like generic programming libraries have been proposed to address this efficiency problem. According to a recent survey [20], two of the most efficient are Uniplate [18] and Smash [11]. The former outperforms SYB by restricting the power of the traversal combinators. The latter offsets some of the run-time checks to compile-time, but needs extra work from the programmer in order to support new data types.

In previous work [7], we have taken a different approach to tackle this problem: we developed a rewrite system that specializes generic functions for specific types. This specialization proceeds in two phases: (1) the generic functions are specialized to non-optimized point-free definitions; (2) these definitions are then optimized using standard algebraic laws for point-free combinators. The major drawback of that approach was the lack of support for user defined recursive types, such as the *Company* type above.

The major contribution of this paper is to extend that specialization mechanism to also cover recursive data types. More specifically, we will focus on inductive data types that can be defined as fixpoints of functors. The key idea is to specialize traversal combinators using well-known recursion patterns for inductive types, such as folds or paramorphisms. Likewise to standard point-free combinators, these recursion patterns are also characterized by a rich set of algebraic laws that enable further optimizations after specialization. Most of the definitions that result from the new rewrite system have runtimes close to the hand-written non-generic ones. Another contribution is the Haskell encoding of these new laws: thanks to recent language extensions such as type-indexed type families, we managed to get an implementation that closely mimics the theory.

Section 2 briefly surveys the SYB approach to generic programming and recaps our previous work on the specialization of generic functions for non-recursive types. Section 3 extends this work with new algebraic rules for the specialization for recursive types. Section 4 discusses how these new rules can be accommodated in a

type-safe rewrite system implemented in Haskell. Section 5 presents specialization examples and compares the respective speedups. Section 6 presents related work, and Section 7 makes some concluding remarks, including some future work ideas.

2 Specialization for Non-recursive Types

In this paper we will focus on a limited set of combinators that capture the essence of strategic generic programming libraries like SYB. Generic functions come in two flavors: type-preserving (transformations) and type-unifying (queries). For defining type-preserving functions we have the following combinators:

$$\begin{array}{ll} nop & :: T \\ (\triangleright) & :: T \to T \to T \\ gmapT & :: T \to T \\ everywhere & :: T \to T \\ mkT_A & :: (A \to A) \to T \\ apT_A & :: T \to (A \to A) \end{array}$$

In the SYB library, the type T of type-preserving generic functions is defined as $\forall a.\ Data\ a \Rightarrow a \to a$. Type classes like *Data* are extensively used in SYB to infer type representations for data types. Among others, these are necessary in the definition of mkT to determine where the type-specific transformation should be applied. To simplify the presentation, instead of using type classes, we will parameterize mkT with an explicit type representation. Besides *everywhere*, we have combinators to map a transformation over all direct children of node ($gmapT$), to sequence transformations ($\triangleright$), and to denote the identity transformation (nop). We also have an explicit combinator to apply a generic transformation to a particular type (apT). In SYB this is done implicitly via type-classes. The transformation to increase all salaries would now be written as follows.

$$\begin{array}{l} increase :: Int \to Company \to Company \\ increase\ k = apT_{Company}\ everywhere\ (mkT_{Salary}\ (incS\ k)) \\ \quad \textbf{where}\ incS\ k\ (S\ s) = S\ (s + k) \end{array}$$

For defining type-unifying functions we have the following combinators:

$$\begin{array}{ll} \varnothing & :: Q\ R \\ (\cup) & :: Q\ R \to Q\ R \to Q\ R \\ gmapQ & :: Q\ R \to Q\ R \\ everything & :: Q\ R \to Q\ R \\ mkQ_A & :: (A \to R) \to Q\ R \\ apQ_A & :: Q\ R \to (A \to R) \end{array}$$

In this case, $Q\ R$ represents the type of generic queries with result type R. Once more, to simplify the presentation of the specialization laws we will assume that R is a monoid, with a *zero* element and an associative *plus* operator. In practice,

$$
\begin{array}{ll}
id & :: A \to A \\
(\circ) & :: (B \to C) \to (A \to B) \to (A \to C) \\
\pi_1 & :: A \times B \to A \\
\pi_2 & :: A \times B \to B \\
(\triangle) & :: (A \to B) \to (A \to C) \to (A \to B \times C) \\
\times & :: (A \to B) \to (C \to D) \to (A \times C \to B \times D) \\
i_1 & :: A \to A + B \\
i_2 & :: B \to A + B \\
(\nabla) & :: (A \to C) \to (B \to C) \to (A + B \to C) \\
(+) & :: (A \to B) \to (C \to D) \to (A + C \to B + D)
\end{array}
$$

Fig. 1. Point-free combinators.

this makes little difference since most typical results, namely lists and integers, are indeed monoids. mkQ_A creates a generic query out of a type-specific one, returning *zero* for types other than A. *everything* collects all results in a bottom-up traversal using the *plus* operator. *gmapQ* collects the results of applying a query to all direct children, $\cup$ sums the results of two queries, and $\varnothing$ denotes the query that always returns *zero*. To apply a generic query to a particular type we have *apQ*. For example, to compute the total salary bill of a company we could define the following generic query:

$$
\begin{array}{l}
salaries :: Company \to Int \\
salaries = apQ_{Company}\ everything\ (mkQ_{Salary}\ billS) \\
\quad \textbf{where}\ billS\ (S\ s) = s
\end{array}
$$

In previous work [7] we have presented a rewrite system to specialize generic functions to type-specific point-free definitions. In the point-free style of programming, functions are composed using a standard set of higher-order combinators, avoiding the need to explicitly mention the domain points as variables. This variable-free style (popularized by John Backus in his 1977 Turing award lecture [2]) is particularly amenable for program calculation since its combinators are characterized by a rich set of algebraic laws. We use a rather standard set of point-free combinators for handling products and sums (see Figure 1). Their behavior should be clear from the type signatures. For more information about the laws ruling this combinators and point-free program calculation in general see [9,5].

Most user defined data types can be defined as the fixpoint of a regular functor. The *base functor* that captures the signature of a data type A will be denoted F_A (when the type A is clear from the context we will often omit it from the subscript). A regular functor is either the identity functor Id, the constant functor $\underline{A}$ (that always returns A), the lifting of the sum $\oplus$ and product bifunctors $\otimes$, or the composition $\odot$ of functors. For example, for lists we have $F_{[A]} = \underline{1} \oplus \underline{A} \otimes Id$. If the type is not recursive, its base functor will not have any identity. For example, $F_{(Maybe\ A)} = \underline{1} \oplus \underline{A}$. Associated with each data type A we also have two unique functions $in_A : F_A\ A \to A$ and $out_A :: A \to F_A\ A$, that are each other's inverse. They allow us to encode and inspect values of the given type, respectively.

$$apT_A\ nop = id \qquad nop\text{-Apply}$$
$$apT_A\ (f \rhd g) = apT_A\ f \circ apT_A\ g \qquad \rhd\text{-Apply}$$
$$\left.\begin{array}{l} apT_A\ (gmapT\ f) = id, \textbf{if } A \textit{ base type} \\ apT_A\ (gmapT\ f) = in_A \circ apT_{F\ A}\ f \circ out_A, \textbf{if } A \textit{ datatype} \end{array}\right\} \qquad gmapT\text{-Apply}$$
$$apT_A\ (everywhere\ f) = apT_A\ (gmapT\ (everywhere\ f) \rhd f) \qquad everywhere\text{-Apply}$$
$$\left.\begin{array}{l} apT_A\ (mkT_A\ f) = f \\ apT_A\ (mkT_B\ f) = id, \textbf{if } A \not\equiv B \end{array}\right\} \qquad mkT\text{-Apply}$$
$$apT_{A\times B}\ f = apT_A\ f \times apT_B\ f \qquad \times\text{-Apply}$$
$$apT_{A+B}\ f = apT_A\ f + apT_B\ f \qquad +\text{-Apply}$$
$$apT_1\ f = id \qquad 1\text{-Apply}$$

Fig. 2. Laws for specializing generic transformations.

Figure 2 presents the laws used to specialize type-preserving combinators into point-free. Specialization proceeds by pushing down the *apT* combinator until it gets consumed by the *mkT*-Apply law. Similar laws exist for the type-unifying combinators. Although not generic, the definitions produced by the specialization phase are very inefficient because they still traverse the whole data structure. However, using point-free program calculation laws they can be optimized in order to eliminate redundant traversals. Notice that the *everywhere*-Apply law uses the recursive definition of this traversal combinator using *gmapT* and $\rhd$. Since the previous rewrite system only handled non-recursive user defined data types, this law did not pose any termination problems. However, for recursive types it cannot be used since it would lead to an infinite expansion of the definition (due to successive expansions of *everywhere* in recursive occurrences of the type).

3 Specialization for Recursive Types

The key to avoid infinite expansions is to specialize traversal combinators using an alternative definition based on standard recursion patterns such as folds. Likewise to point-free combinators, these recursion patterns are characterized by powerful algebraic laws, that will enable us to optimize the specialized definitions. For a comprehensive presentation of most standard recursion patterns and the respective laws see [17].

The standard recursion pattern of iteration, usually known as fold or *catamorphism*, consumes an inductive type by replacing its constructors with a given argument function. For an inductive type A, given a function $g{::}F\ B \to B$, $(\!|g|\!)_A{::}A \to B$ denotes a fold over that type that produces values of type B. Its recursive definition can be clearly depicted in the following diagram.

$$\begin{array}{ccc} A & \xrightarrow{out_A} & F\ A \\ {\scriptstyle (\!|g|\!)_A}\downarrow & & \downarrow{\scriptstyle F\ (\!|g|\!)_A} \\ B & \xleftarrow[g]{} & F\ B \end{array}$$

While folds can express functions defined by iteration, *paramorphisms* can ex-

$(\!\|in_A\|\!)_A = id$	*reflex*-CATA
$(\!\|g\|\!)_A \circ in_A = g \circ F\ (\!\|g\|\!)_A$	*cancel*-CATA
$f \circ (\!\|g\|\!)_A = (\!\|h\|\!)_A \Leftarrow f \circ g = h \circ F\ f$	*fusion*-CATA
$(\!\|in_{[A]} \circ (id + f \times id)\|\!)_{[A]} = map\ f$	*map*-CATA
$(\!\langle in_A \circ F\ \pi_1 \rangle\!)_A = id$	*reflex*-PARA
$(\!\langle g \rangle\!)_A \circ in_A = g \circ F\ ((\!\langle g \rangle\!)_A \triangle id)$	*cancel*-PARA
$f \circ (\!\langle g \rangle\!)_A = (\!\langle h \rangle\!)_A \Leftarrow f \circ g = h \circ F\ (f \times id)$	*fusion*-PARA
$(\!\langle f \circ F\ \pi_1 \rangle\!)_A = (\!\|f\|\!)_A$	*cata*-PARA

Fig. 3. Some laws for folds and paramorphisms.

press all functions that can be defined by primitive recursion [16]. In practice, this means that the result can depend not only on the recursive result, but also on the recursive occurrence of the type. For an inductive type A, given a function $g :: F\ (B \times A) \to B$, $(\!\langle g \rangle\!)_A :: A \to B$ denotes a paramorphism over that type that produces values of type B. Again, its recursive definition can be expressed by a diagram.

$$\begin{array}{ccccc} A & \xrightarrow{out_A} & F\ A & \xrightarrow{F\ (id \triangle id)} & F\ (A \times A) \\ {\scriptstyle (\!\langle g \rangle\!)_A}\downarrow & & & & \downarrow{\scriptstyle F\ ((\!\langle g \rangle\!)_A \times id)} \\ B & & \xleftarrow[g]{} & & F\ (B \times A) \end{array}$$

Notice how a copy of the recursive occurrence is made before the recursive invocation. For optimization of functions defined as folds and paramorphisms we will use the laws presented in Figure 3.

When applied to an inductive type, the bottom-up traversal *everywhere* will be specialized into a fold over that type. The *everywhere*-APPLY law will now be defined as follows.

$$apT_A\ (everywhere\ f) = (\!\|apT_A\ f \circ in_A \circ apT_{F\ \overline{A}}\ (everywhere\ f)\|\!)_A$$

The behavior of this fold is better understood with the help of the following diagram:

$$\begin{array}{ccccccc} A & & \xrightarrow{\qquad\qquad out_A \qquad\qquad} & & & & F\ A \\ {\scriptstyle (\!|\cdot|\!)_A}\downarrow & & & & & & \downarrow{\scriptstyle F\ (\!|\cdot|\!)_A} \\ A & \xleftarrow[apT_A\ f]{} & A & \xleftarrow[in_A]{} & F\ A & \xleftarrow[apT_{F\ \overline{A}}\ (everywhere\ f)]{} & F\ A \end{array}$$

The intent of the function $apT_{F\ \overline{A}}\ (everywhere\ f)$ is to apply the transformation to all content of the type, apart from its recursive occurrences (which were already processed recursively by the fold itself). This behavior is achieved by adding the following law to the set presented in Figure 2:

$$apT_{\overline{A}}\ f = id \qquad \textit{rec}\text{-APPLY}$$

This law guarantees that a type marked with an overline is ignored by the apT combinator. For example, for lists $apT_{F\ \overline{A}}\ (everywhere\ f)$ would be instantiated as $apT_{1+A\times\overline{[A]}}\ (everywhere\ f)$, which is equivalent to $id + apT_A\ (everywhere\ f) \times id$.

Since *everywhere* f is a bottom-up traversal, after transforming both the recursive occurrences and the remaining content, f still needs to be applied to the resulting value. To do so, in_A is first used to reconstruct a value of type A, followed by an application of $apT_A\ f$.

To exemplify the specialization of a generic transformation to a recursive type, consider the following example, where $f = mkT_{Int}\ succ$:

$$
\begin{aligned}
& apT_{[Int]}\ (everywhere\ f) \\
= & \{\ everywhere\text{-}\textsc{Apply}\ \} \\
& (\!|apT_{[Int]}\ (mkT_{Int}\ succ) \circ in_{[Int]} \circ apT_{F\ \overline{[Int]}}\ (everywhere\ f)|\!)_{[Int]} \\
= & \{\ mkT\text{-}\textsc{Apply}; +\text{-}\textsc{Apply}; \times\text{-}\textsc{Apply}; rec\text{-}\textsc{Apply}\ \} \\
& (\!|id \circ in_{[Int]} \circ (id + apT_{Int}\ (everywhere\ f)\ \times\ id)|\!)_{[Int]} \\
= & \{\ everywhere\text{-}\textsc{Apply}\ \} \\
& (\!|in_{[Int]} \circ (id + apT_{Int}\ (gmapT\ (everywhere\ f) \triangleright f)\ \times\ id)|\!)_{[Int]} \\
= & \{\ \triangleright\text{-}\textsc{Apply}; gmapT\text{-}\textsc{Apply}\ \} \\
& (\!|in_{[Int]} \circ (id + (id \circ apT_{Int}\ (mkT_{Int}\ succ))\ \times\ id)|\!)_{[Int]} \\
= & \{\ mkT\text{-}\textsc{Apply}\ \} \\
& (\!|in_{[Int]} \circ (id + succ\ \times\ id)|\!)_{[Int]} \\
= & \{\ map\text{-}\textsc{Cata}\ \} \\
& map\ succ
\end{aligned}
$$

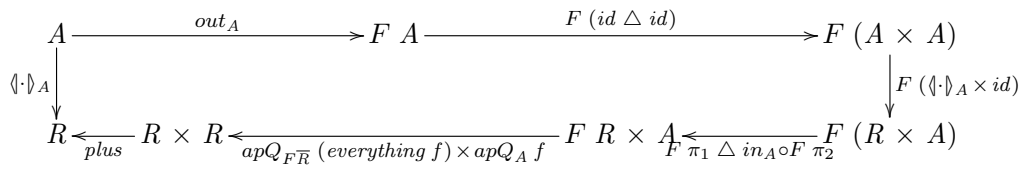
As expected, if f is applied to a type that does not contain integers the result is the identity function:

$$
\begin{aligned}
& apT_{[Char]}\ (everywhere\ f) \\
= & \{\ldots\} \\
& (\!|in_{[Char]} \circ (id + (id \circ apT_{Char}\ (mkT_{Int}\ succ))\ \times\ id)|\!)_{[Char]} \\
= & \{\ mkT\text{-}\textsc{Apply}\ \} \\
& (\!|in_{[Char]} \circ (id + id\ \times\ id)|\!)_{[Char]} \\
= & \{\ id\ \times\ id = id; id + id = id\ \} \\
& (\!|in_{[Char]}|\!)_{[Char]} \\
= & \{\ reflex\text{-}\textsc{Cata}\ \} \\
& id
\end{aligned}
$$

The bottom-up *everything* combinator will be specialized into a paramorphism:

$$
\begin{gathered}
apQ_A\ (everything\ f) = \\
(\!|plus \circ (apQ_{F\overline{R}}\ (everything\ f)\ \times\ apQ_A\ f) \circ (F\ \pi_1\ \triangle\ in_A \circ F\ \pi_2)|\!)_A
\end{gathered}
$$

Again, this paramorphism is easier to understand with a diagram:



After recursion, the input value is reconstructed using in_A in order to feed it to the generic query. Simultaneously, the query is applied to the non-recursive type contents, and finally both results are put together with the monoid *plus* operator.

4 Encoding in Haskell

In order to harness the above algebraic laws into a type-safe rewrite system for the specialization of generic functions, we must provide type-safe representations for both functions and types. For functions we will use the same representation presented in [7], based on a *generalized algebraic data type* (GADT) [19]:

$$
\begin{array}{l}
\textbf{data}\ PF\ a\ \textbf{where} \\
\quad Id \quad :: PF\ (a \to a) \\
\quad (\triangle) \ :: PF\ (a \to b) \to PF\ (a \to c) \to PF\ (a \to (b, c)) \\
\quad (\times) \ :: PF\ (a \to b) \to PF\ (c \to d) \to PF\ ((a, c) \to (b, d)) \\
\quad (\nabla) \ :: PF\ (a \to c) \to PF\ (b \to c) \to PF\ (Either\ a\ b \to c) \\
\quad (+) \ :: PF\ (a \to b) \to PF\ (c \to d) \to PF\ (Either\ a\ c \to Either\ b\ d) \\
\quad mkT :: Type\ a \to PF\ (a \to a) \to PF\ T \\
\quad apT \ :: Type\ a \to PF\ T \to PF\ (a \to a) \\
\quad ...
\end{array}
$$

This type contains both point-free combinators and SYB combinators. Generic transformations and queries have the following types:

$$
\begin{array}{l}
\textbf{type}\ T = \forall a.\ Type\ a \to a \to a \\
\textbf{type}\ Q\ r = \forall a.\ Type\ a \to a \to r
\end{array}
$$

Instead of using type classes to infer type representations, these are explicitly passed to generic functions. Type representations are also defined using a GADT. For base types, sums and products, *Type a* can be defined as follows.

$$
\begin{array}{l}
\textbf{data}\ Type\ a\ \textbf{where} \\
\quad Int \quad :: Type\ Int \\
\quad Char :: Type\ Char \\
\quad ... \\
\quad Sum \ :: Type\ a \to Type\ b \to Type\ (Either\ a\ b) \\
\quad Prod :: Type\ a \to Type\ b \to Type\ (a, b) \\
\quad Func :: Type\ a \to Type\ b \to Type\ (a \to b)
\end{array}
$$

One consequence of using a GADT to encode the combinators is the ability to define an evaluation function $eval :: PF\ a \to a$. For example, the evaluation of mkT follows closely the SYB semantics:

$$
\begin{array}{l}
eval\ (mkT\ a\ f) = \lambda b\ x \to \textbf{case}\ teq\ a\ b\ \textbf{of}\ Just\ Eq \to eval\ f\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Nothing \to x
\end{array}
$$

The resulting generic function behaves as f if applied to a value of type a, or as the identity function otherwise. Function *teq* tests equality of type representations, and is nowadays a classical example of the usefulness of GADTs [19]:

```
data Equal a b where Eq :: Equal a a
teq :: Type a → Type b → Maybe (Equal a b)
teq Int Int = return Eq
teq (Sum a b) (Sum c d) = do Eq ← teq a c
                             Eq ← teq b d
                             return Eq
...
teq _ _ = Nothing
```

The constructor *Eq* of the *Equal* GADT can be seen as a proof that types a and b are indeed equal.

The representation of user defined recursive types follows directly from the theoretical definition given in Section 2. For each type A we need to represent its base functor $F\ A$. When applying a functor to another type we want to get a sum of products as result, capable of being processed with point-free combinators. If functors are defined with normal Haskell polymorphic data types, it is impossible to obtain this behavior, since type-equivalence in Haskell is not structural but name-based.

In order to overcome this problem, we decided to represent functors using type-indexed type families [21,3], a new extension to the Haskell type system already supported in GHC. Developed with type-level programming in mind, type families are type constructors that represent sets of types. Set members are aggregated according to the type parameters passed to the type family constructor, called type indices: family constructors can have different representation types for different type indices. A type family to represent functors can defined as follows.

```
type family F a x :: *
```

In this definition a is the type index that stands for the type whose functor is being defined, and x is the type argument of the functor itself. For example, for lists we have the following instance:

```
type instance F [a] x = Either One (a, x)
```

The GADT that represents functions can now be extended with constructors for the recursion patterns, together with the *in* and the *out* functions.

```
data PF a where
  ...
  In  :: PF (F a a → a)
  Out :: PF (a → F a a)
```

$$Cata :: PF\ (F\ a\ c \to c) \to PF\ (a \to c)$$
$$Para :: PF\ (F\ a\ (c, a) \to c) \to PF\ (a \to c)$$

In our rewrite system we will need to apply a functor both to type and function representations. Using the above type family, we can capture this behavior in the following data type, that represents the functor of an inductive type a.

$$\textbf{data}\ Functor\ a = Functor\{\ mapT :: \forall b.\ Type\ b \to Type\ (F\ a\ b),$$
$$mapF :: \forall x\ y.\ PF\ (x \to y) \to PF\ (F\ a\ x \to F\ a\ y)\}$$

Our type representation can now be extended with a new constructor to represent user defined recursive types.

$$\textbf{data}\ Type\ a\ \textbf{where}$$
$$\ldots$$
$$Data :: String \to Functor\ a \to Type\ a$$

Given a ground type a, it is possible to use the Haskell type system to infer its representation. We can define a class with all representable types:

$$\textbf{class}\ Typeable\ a\ \textbf{where}$$
$$typeof :: Type\ a$$

For example, for lists the *Typeable* instance can be defined as follows:

$$\textbf{instance}\ Typeable\ a \Rightarrow Typeable\ [a]\ \textbf{where}$$
$$typeof = Data\ name\ functor$$
$$\textbf{where}\ name = \texttt{"["} +\!\!+ show\ (typeof :: Type\ a) +\!\!+ \texttt{"]"}$$
$$functor = Functor\{\ mapT = \lambda b \to Sum\ One\ (Prod\ typeof\ b),$$
$$mapF = \lambda f \to Id + (Id \times f)\}$$

In order to guarantee that the rewrite system is type-safe, rewrite rules are represented by a monadic function that takes a function representation and returns a representation of the same type.

$$\textbf{type}\ Rule = \forall a.\ Type\ a \to PF\ a \to RewriteM\ (PF\ a)$$

RewriteM is a stateful monad that keeps a trace of the applied rules and is an instance of *MonadPlus*, thus modeling partiality in rule application. The extra type representation passed as argument allows the rule to make type-based rewriting decisions.

Both the specialization laws of Figure 2, and the point-free optimization laws (such as the ones presented in Figure 3 for folds and paramorphisms) are encoded as rewrite rules. For example, the reflexivity rule for folds can be defined as follows.

$$reflex\_Cata :: Rule$$
$$reflex\_Cata\ (Func\ a\ b)\ (Cata\ In) = \textbf{do}\ Eq \leftarrow teq\ a\ b$$

```
                                        success "reflex-Cata" Id
reflex_Cata _ _ = mzero
```

This rule uses *teq* to guarantee that is only applied to functions of type $a \to a$. The monadic function *success* updates the *RewriteM* monad to keep trace of the successful reduction.

Rewrite systems are built from basic rules using a standard set of strategic combinators. There are two main top-level strategies: *optimize_syb* for specialization of type-preserving and type-unifying generic programs into point-free expressions; and *optimize_pf* for simplification and optimization of point-free definitions. The latter also applies some "beautifying" rules to produce more concise results.

5 Examples

We will now present some specialization examples, and compare the performance between the resulting definitions and the original SYB functions.

The first example is the generic transformation to increase all salaries. In order to increase the readability of the specialized point-free definitions, we will consider that the type-specific behavior is for the *Employee* type instead of *Salary*. In SYB we have the following definition.

```
increase :: Int → Company → Company
increase k = everywhere (mkT (incE k))
  where incE k (E p (S s)) = E p (S (s + k))
```

After encoding this definition using type $PF\ a$, and applying the specialization and optimization strategies, we get the following definition, where C stands for *Company* and D for *Dept*.

$$in_C \circ map\ (\!|in_D \circ (id \times (incE\ k \times map\ (incE\ k + id)))|\!)_D \circ out_C$$

This definition is a pretty-print of the respective representation in $PF\ a$. It approximates the hand-written presented in Section 1: the fold will be recursively applied to each department of a company; at each department both the manager and all direct employees will have their salaries increased by the function $incE\ k$.

The second example, presented before in Section 2, addresses the specialization of a generic query to compute the total salary bill of a company.

```
salaries :: Company → Int
salaries k = everything (mkQ billE)
  where billE (E p (S s)) = s
```

In this case we get the following definition.

$$sum \circ map\ (\!|plus \circ (billE \times (sum \circ map\ (billE \triangledown id)) \circ \pi_2)|\!)_D \circ out_C$$

Note how the paramorphism was simplified as a fold, since the query does not mention the recursive type *Dept*. The expression $sum \circ map\ (billE \triangledown id)$ collects

all salaries from the direct employees of a department, and all recursively computed salaries from sub-departments. This result is then summed with the salary of the manager to compute the total salary bill. The expression $sum \circ map\ f$ was used in this example just to make the result more readable. In fact, they are fused together as a single fold by the rewrite system.

The last example combines the two previous examples into a single function:

$$\begin{aligned} &higher\_salaries :: Int \to Company \to Int \\ &higher\_salaries\ k = salaries \circ increase\ k \end{aligned}$$

Although the two operations are performed in sequence in the original query, after specialization we get a result very similar to the previous one, with a single traversal over the type.

$$\begin{aligned} &\textbf{let}\ aux = billE \circ incE\ k \\ &\textbf{in}\ \ sum \circ map\ (\!|plus \circ (aux \times (sum \circ map\ (aux \nabla id)) \circ \pi_2)|\!)_D \circ out_C \end{aligned}$$

Unlike systems specially designed to implement fusion (such as [23]), our rewrite system cannot implement the full power of the fusion laws. However it covers most of the particular instances that occur during the specialization of generic functions. For example, the above optimization was possible due to the following instance of *fusion*-Cata.

$$(\!|f|\!)_A \circ (\!|in_A \circ g|\!)_A = (\!|f \circ g|\!)_A \ \Leftarrow\ F\ (\!|f|\!)_A \circ g = g \circ F\ (\!|f|\!)_A$$

To verify the side-condition of this law, we first apply the rewrite system *optimize_pf* to $F\ (\!|f|\!)_A \circ g$ and $g \circ F\ (\!|f|\!)_A$, and then check for syntactic equality.

Performance analysis

We have compared the runtimes of the first two examples for hand-written, specialized, and generic definitions written in SYB and Uniplate. The results are presented in Figure 4. A large part of SYB's inefficiency is due to the heavy use of type-classes to infer type representations. To factor out this penalty, and better quantify the speedup achieved by our specialization mechanism, we also include the runtimes of both generic functions obtained by evaluating their representation using the *eval* function presented in section 4 (denoted in the graphic as SYB GADT). We compiled each function using GHC 6.8.2 with optimization flag `O2`. Each example was tested with *Company* values of increasing size (measured in kBytes needed to store the Haskell definition of each value).

As expected, for both examples, the SYB generic definitions perform much worse than the hand-written, and the loss factor grows with the database size. The SYB GADT variant is at least twice as fast, but still much slower than the hand-written. The specialized point-free definitions perform closer to the hand-written, with loss factors of 1.11 (*increase* 100) and 2.85 (*salaries*) for the biggest sample. This performance loss is mainly due to the use of *in* and *out* to convert between user defined types and they structural representation as a sum of products. For these particular

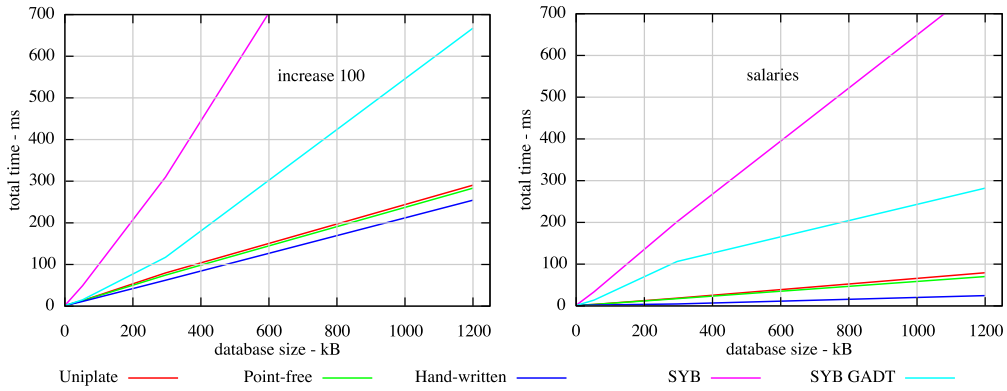


Fig. 4. Timing results.

examples, the performance of the specialized point-free code is tangentially better than Uniplate. As discussed in the next section, Uniplate also has some mechanisms to avoid traversing unnecessary branches, which justify the proximity in the results.

Although quite standard when comparing generic programming libraries, these example are not particularly flattering to our optimization mechanism: in fact, there are no large branches of data that can be avoided in the traversals. For example, if the *Company* data type had any other information besides departments (not containing the type *Salary*), the runtime would remain the same, further widening the gap to SYB. We also achieve a significant advantage when optimizing compositions of generic functions: for example, in the *higher_salaries* example our specialized point-free definition was already 1.35 times faster than Uniplate for the biggest sample.

6 Related Work

Uniplate

Unlike SYB, some generic programming libraries have been designed with performance issues in mind, usually at the cost of expressiveness. One such library is Uniplate [18], that is among the fastest libraries currently available for generic programming in Haskell [20]. That fact, together with the SYB-like flavor of its combinators, motivated an obvious inclusion in the comparative performance analysis of the previous section. The key idea behind Uniplate is that most generic traversals have value-specific behavior for just one type. Building on this insight, this library provides two key combinators to specify bottom-up generic transformations:

$$transform :: Uniplate\ a \Rightarrow (a \to a) \to a \to a$$
$$transformBi :: Biplate\ b\ a \Rightarrow (a \to a) \to b \to b$$

The *transform* combinator applies its argument to every a occurring inside a value of type a, while *transformBi* applies its argument to every a occurring inside a value of a different type b. Recalling our examples, the *increase* transformation can be defined using *transformBi*, since it looks for all salaries inside a company. The

Uniplate and *Biplate* classes contain primitive methods to find the substructures of type a inside values of type a and b, respectively. Instances of these classes can be defined using a variety of methods, ranging from more generic and less efficient to more verbose and more efficient. The most efficient method (used in the comparison of the previous section) is to define the instances by hand, which for *Biplate* requires defining n^2 instances to support n types. When defining instances for *Biplate* b a it is possible to avoid traversing down branches of a that do not contain the target type b, thus optimizing generic traversals.

The main advantage of *Uniplate* is that generic functions execute fast out of the box, without the need of an explicit optimization phase. On the other hand, likewise to SYB, our optimization technique can handle more powerful combinators, that target different types in a single traversal. Using fusion techniques, our approach can also further optimize combinations of traversals, while Uniplate speedups are constrained to individual traversals.

Other techniques to optimize generic programs

Another very efficient SYB-like generic programming library is Smash [11]. Instead of using run-time checks to find the target types, it offsets them to compile-time by using heterogeneous collections [12] to encode the type-specific cases of generic functions. Unfortunately, the speedup obtained with this technique comes at the cost of extra work from the programmer: in order to support a new data type, all different traversal combinators must be defined from scratch, while in SYB they can all be generically implemented using just two primitive methods.

A different approach has been followed in [1], where a technique named symbolic evaluation was developed to optimize Generic Haskell programs [15]. It focus on the specialization of fully applied functions and tries to eliminate conversions between types and their structural representations. Symbolic evaluation guarantees that the intermediate structures are completely removed from the optimized code. A similar technique could be used in our framework to further optimize the point-free definitions, via an additional translation step to explicitly recursive point-wise code.

Application scenarios

As previously mentioned, our main goal was to extend the specialization mechanism presented in [7] to also cover inductive types. In [7] we already described how it could be used to optimize the structure-shy XPath query language. This technique was harnessed into the prototype schema-aware XPath compiler XPTO [8]. Query compilation in XPTO proceeds as follows: the XML Schema is parsed into a sum of products representation using *Type* a; the XPath query is parsed into a type-safe representation of type *PF* a; the rewrite system is used to specialize the query to the given schema; the specialized point-free definition is output into a new Haskell program to be compiled and linked with an XML parser and point-free execution library; the resulting program can then be used to execute the original query against XML files conforming to the given schema. We are currently deploying the new technique presented here into the XPTO compiler in order to handle

some recursive XML Schemas.

A similar type-safe rewriting system was also used in [6] to optimize two-level data transformations [4]. A two-level data transformation consists of a type-level transformation coupled with value-level transformations of the respective inhabitants. More specifically, we developed a framework that allows us to specify data type refinements $A \leqslant B$ using strategic combinators, and get for free the migration functions between values of type A and B, and vice-versa. Both the types and the migration functions are again encoded using *Type a* and *PF a*, allowing us to use the rewrite system to optimize them, and migrate queries/producers from the abstract type A to the concrete type B. The inclusion of inductive types in the rewrite system will allow us to extend the applicability of this framework.

Template meta-programming

We believe that our algebraic approach could be instructed at a lower level to provide compile-time specialization of generic functions, through template meta-programming [22]. This rewriting process would encompass transformation of Haskell generic programs through direct manipulation of their abstract syntax trees. However, since the current implementation of template meta-programming in Haskell is completely untyped, we would loose the guarantee that the rewrite system is type-safe. Template meta-programming could also be used to infer automatically the recursive types' sum of products representation.

7 Concluding Remarks

We have extended an existent mechanism to specialize SYB-like generic functions to also cover user defined recursive data types. By focusing on inductive types (fixpoints of functors) we were able to use recursion patterns such as folds and paramorphisms to encode generic traversals. These recursion patterns are characterized by nice algebraic laws, that were incorporated in a type-safe rewrite system to further optimize the specialized code. The definitions produced by our specialization mechanism perform close to hand-written non-generic ones. Thanks to recent extensions of the Haskell type-system, such as type-indexed type families or generalized algebraic data types, our implementation of the rewrite system closely mimics the theoretical presentation.

The major limitation of the current approach is that it only supports single-recursive inductive types. We are currently investigating how to extend it to cover more general forms of recursion, such as mutually-inductive data types or nested data types. Particularly relevant to this endeavor is the work described in [10], showing that higher-order functors can be used to give an initial algebra semantics to nested data types (likewise to standard inductive types).

References

[1] Alimarine, A. and S. Smetsers, *Optimizing generic functions*, in: D. Kozen and C. Shankland, eds., "Proc. of 7th Int. Conf. on Mathematics of Program Construction, MPC 2004 (Stirling, July 2004)," Lecture Notes in Computer Science **3125**, Springer, 2004, pp. 16–31.

[2] Backus, J., *Can programming be liberated from the von Neumann style? a functional style and its algebra of programs*, Commun. of ACM **21**(8) (1978), pp. 613–641.

[3] Chakravarty, M. M. T., G. Keller and S. Peyton Jones, *Associated type synonyms*, in: "Proc. of 10th ACM SIGPLAN Int. Conf. on Functional programming, ICFP '05 (Tallinn, Sept. 2005)," ACM Press, 2005, pp. 241–253.

[4] Cunha, A., J. N. Oliveira and J. Visser, *Type-safe two-level data transformation*, in: J. Misra, T. Nipkow and E. Sekerinski, eds., "Proc. of 14th International Symposium on Formal Methods, FM 2006 (Hamilton, Ont., Aug. 2006)," Lecture Notes in Computer Science **4085**, Springer, 2006, pp. 284–299.

[5] Cunha, A., J. S. Pinto and J. Proença, *A framework for point-free program transformation*, in: A. Butterfield, C. Grelck and F. Huch, eds., "Revised Selected Papers from 17th Int. Wksh. on Implementation and Application of Functional Languages, IFL 2005 (Dublin, Sept. 2005)," Lecture Notes in Computer Science **4015**, Springer, 2006, pp. 1–18.

[6] Cunha, A. and J. Visser, *Strongly typed rewriting for coupled software transformation*, Electron. Notes in Theor. Comput. Sci. **174**(1) (2007), pp. 17–34.

[7] Cunha, A. and J. Visser, *Transformation of structure-shy programs - applied to XPath queries and strategic functions*, in: "Proc. of 2007 ACM SIGPLAN 2007 Wksh. on Partial Evaluation and Program Manipulation, PEPM '07 (Nice, Jan. 2007)," ACM Press, 2007, pp. 11–20.

[8] Ferreira, F. and H. Pacheco, *XPTO - an Xpath preprocessor with type-aware optimization*, in: V. Santos, P. R. Henriques and S. M. de Sousa, eds., "Proc. of 1st Conf. on Compilers, Related Technologies and Applications, CoRTA '07 (Covilhã, July 2007)," Universidade da Beira Interior, 2007.

[9] Gibbons, J., *Calculating functional programs*, in: R. Backhouse, R. Crole and J. Gibbons, eds., "Revised Lectures from Int. Summer School and Wksh. on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, ACMMPC 2000 (Oxford, Apr. 2000)," Lecture Notes in Computer Science **2297**, Springer, 2002, pp. 148–203.

[10] Johann, P. and N. Ghani, *Initial algebra semantics is enough!*, in: S. Ronchi Della Rocca, ed., "Proc. of 8th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2007 (Paris, June 2007)," Lecture Notes in Computer Science **4583**, Springer, 2007, pp. 207–222.

[11] Kiselyov, O., *Smash your boilerplate without class and typeable*, message to the Haskell mailing list, 2006. Available at http://article.gmane.org/gmane.comp.lang.haskell.general/14086.

[12] Kiselyov, O., R. Lämmel and K. Schupke, *Strongly typed heterogeneous collections*, in: "Proc. of 2004 ACM SIGPLAN Haskell Workshop, Haskell '04 (Snowbird, UT, Sept. 2004)," ACM Press, 2004, pp. 96–107.

[13] Lämmel, R. and S. Peyton Jones, *Scrap your boilerplate: a practical design pattern for generic programming*, in: "Proc. of 2003 ACM SIGPLAN Int. Wksh. on Types in Language Design and Implementation, TLDI '03 (New Orleans, LA, Jan. 2003)," ACM Press, 2003, pp. 26–37.

[14] Lämmel, R. and S. Peyton Jones, *Scrap your boilerplate with class: extensible generic functions*, in: "Proc. of 10th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '05 (Tallinn, Estonia, Sept. 2005)," ACM Press, 2005, pp. 204–215.

[15] Löh, A., J. Jeuring, D. Clarke, R. Hinze, A. Rodriguez and J. de Wit, *The generic Haskell user's guide – version 1.42 (Corla)*, Technical Report UU-CS-2005-004, Universiteit Utrecht, 2005. Available at http://www.cs.uu.nl/research/techreps/UU-CS-2005-004.html.

[16] Meertens, L., *Paramorphisms*, Formal Asp. of Comput. **4**(5) (1992), pp. 413–424.

[17] Meijer, E., M. Fokkinga and R. Paterson, *Functional programming with bananas, lenses, envelopes and barbed wire*, in: J. Hughes, ed., "Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA '91 (Cambridge, MA, Aug. 1991)," Lecture Notes in Computer Science **523**, Springer, 1991, pp. 124–144.

[18] Mitchell, N. and C. Runciman, *Uniform boilerplate and list processing*, in: "Proc. of 2007 ACM SIGPLAN Haskell Workshop, Haskell' 07 (Freiburg, Sept. 2007)," ACM Press, 2007, pp. 49–60.

[19] Peyton Jones, S., D. Vytiniotis, S. Weirich and G. Washburn, *Simple unification-based type inference for GADTs*, in: "Proc. of 11th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '06 (Portland, OR, Sept. 2006)," ACM Press, 2006, pp. 50–61.

[20] Rodriguez, A., J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov and B. C. d. S. Oliveira, *Comparing libraries for generic programming in Haskell*, Technical Report UU-CS-2008-010, Universiteit Utrecht, 2008. Available at http://www.cs.uu.nl/research/techreps/UU-CS-2008-010.html

[21] Schrijvers, T., S. Peyton Jones, M. Chakravarty, M. Sulzmann, *Type checking with open type functions*, in: "Proc. of 13th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '08 (Victoria, BC, Sept. 2008)," ACM Press, 2008, pp. 51–62.

[22] Sheard, T. and S. Peyton Jones, *Template meta-programming for Haskell*, ACM SIGPLAN Notices **37**(12) (2002), pp. 60–75.

[23] Sittampalam, G. and O. de Moor, *Mechanising fusion*, in: J. Gibbons and O. de Moor, eds., "The Fun of Programming," Cornerstones of Computing, Palgrave, 2003, pp. 79–104.