Afstudeerverslag

# Specializing Type-Indexed Values by Partial Evaluation

**Martijn de Vries**

**begeleider Dr. J. Terlouw**



Rijksuniversiteit Groningen
Informatica
Postbus 800
9700 AV Groningen

03 mei 2004

# Abstract

The Generic Haskell programming language allows functions to be defined by induction on the structure of data types. This gives rise to generic functions which can be applied to values of any conceivable data type.

Compiling a Generic Haskell program amounts to generating a Haskell program in which all generic functions have been translated to ordinary Haskell functions. Since the Haskell language only allows functions to be defined on the values of a data type, translating generic functions defined on the structure of data types is not straightforward.

The application of a generic function to a value involves specializing the function to the type of its parameter. For every distinct specialization of a generic function in a Generic Haskell program, an ordinary Haskell function is generated in the compilation process. Hence, the compilation of a generic function will typically yield several ordinary functions.

The current method that is used to translate specializations is rather unsophisticated. At run-time, values are frequently converted back and forth to a structural representation which simplifies the code generation process considerably. This approach is often extremely detrimental to the space and time efficiency of the generated functions.

In this thesis an optimization method is described which attempts to eliminate all structural conversions from the generated functions by applying partial evaluation in combination with a number of program transformations. This approach essentially evaluates all conversions in the representation of values at compile-time. The functions of the resulting optimized program approach the efficiency of hand-written Haskell functions in terms of space and time usage.

An implementation of the described method is included as an optimizer in the final phase of the Generic Haskell compiler.

# Contents


**1 Introduction** **1**
- 1.1 Generic Functional Programming . . . . . 1
- 1.2 Data Types . . . . . 2
- 1.3 Type-indexed Values . . . . . 4
- 1.4 Kind-indexed Types . . . . . 6
- 1.5 Specialization . . . . . 7
- 1.6 Optimizing Specializations . . . . . 7

**2 Principles of Specialization** **9**
- 2.1 Structural Conversions . . . . . 9
- 2.2 Specialization of Type-Indexed Values . . . . . 11

**3 Compiling Generic Haskell** **15**
- 3.1 Compiling Generic Haskell . . . . . 15
- 3.2 Data Types . . . . . 16
- 3.3 Translating Kind-Indexed Types . . . . . 18
- 3.4 Translating Type-Indexed Values . . . . . 19

**4 Partial Evaluation of Specializations** **21**
- 4.1 Partial Evaluation . . . . . 21
- 4.2 Symbolic Evaluation . . . . . 24
- 4.3 Pattern Matching with Symbolic Values . . . . . 26
- 4.4 Specializations . . . . . 29

**5 Case Transformations** **33**
- 5.1 Case Terminology . . . . . 33
- 5.2 Case Elimination . . . . . 34
- 5.3 Case Cleanup . . . . . 35
- 5.4 Case Floating . . . . . 35
- 5.5 Case of Case Transformation . . . . . 36
- 5.6 Generalizing Case of Case . . . . . 37

**6 Optimizing Specializations** **39**
- 6.1 Eliminating Structural Conversions . . . . . 39
- 6.2 Optimization Algorithm . . . . . 42

**7 Optimization in Practice** **43**
7.1 Optimizer Implementation . . . . . 43
7.2 Examples . . . . . 43
7.3 Performance . . . . . 46

**8 Conclusion** **49**
8.1 Results . . . . . 49
8.2 Implementation . . . . . 50
8.3 Future Work . . . . . 50
8.4 Related Work . . . . . 51

**A Partial Evaluation of Specializations** **53**
A.1 Partial Evaluation of $poly_T$ . . . . . 53
A.2 Partial Evaluation of $poly_{K_i}$ . . . . . 56

**B Definitions Used For Benchmarking** **57**

**Bibliography** **59**

# Chapter 1

# Introduction

## 1.1 Generic Functional Programming

Generic Haskell [HJ03] allows functions to be defined by induction on the structure of data types. This gives programmers the ability to write *generic functions* which can operate on any conceivable data type.

Defining a function generically is not just beneficial when this function will be used for many different data types. Haskell data types may have many constructors which means that non-generic definitions of functions for such data types are often lengthy and tedious. It is often possible to define functions on large data types more concisely by using generic definitions, which will lead to fewer errors caused by carelessness.

When a data type is altered while a program is still in development, it is usually necessary that functions defined for this particular data type are altered as well. For instance, a *complete function* defined for a data type $D$ may become a *partial function* when a constructor is added in the definition of $D$. Unexpected run-time errors may result. Since generic function definitions can be applied to any data type, it is often not necessary to modify function definitions after data type definitions have been altered. Still, one must of course verify that the generic function produces the desired result for any value of a modified data type.

Occasionally one may want to define functions for values of a data type not yet known. When new data types are introduced, old generic functions can be applied to values of these data types without changes to the function definitions. This also facilitates writing functions which are truly generic in nature. Examples of this sort of functions are: *compress*, *hash*, *encrypt*, *save* and *search*.

As well as being useful for general purpose programming, Generic Haskell is particularly well suited for:

- XML processing and XML tools
- Applications with large data types, such as compilers and interpreters

## 1.2 Data Types

New algebraic data types can be introduced in Haskell [PJ03] by using the **data** construct:

$$\textbf{data}\ T\ u_1 \ldots u_k = K_1\ t_{11} \ldots t_{1k_1} \mid \cdots \mid K_n\ t_{n1} \ldots t_{nk_n}$$

This declaration introduces a new type constructor $T$ with $n$ data constructors $K_1, \ldots, K_n$. Type constructor $T$ has $k$ type parameters $u_1, \ldots, u_k$ and data constructor $K_i$ takes $k_i$ value arguments corresponding to the types $t_{i1}, \ldots, t_{ik_i}$.

In addition to algebraic data type definitions, new types can also be introduced using **newtype** and **type** declarations. With the **type** keyword, *type synonyms* can be defined:

$$\textbf{type}\ T\ u_1 \ldots u_k = t$$

This declaration introduces a new type constructor $T$ with $k$ type parameters $u_1, \ldots, u_k$. The type constructor $T$ applied to $k$ type arguments is declared to be equivalent to an existing type $t$ in which type variables $u_1, \ldots, u_k$ may appear.

$$\textbf{newtype}\ T\ u_1 \ldots u_k = N\ t$$

A **newtype** declaration also introduces a new type but with the representation of an existing type. The main difference with a type synonym is that type-coercions must be carried out explicitly. Types defined by **newtype** are more efficient because an extra level of indirection is omitted.

### 1.2.1 Representing Data Types

In order to let functions be defined over the structure of data types, one needs to be able to represent this structure in the programming language. In Haskell, a data type declaration can be regarded as a sum of products where the type parameters of each constructor form a product. The different constructors of a data type represent the components of the sum. The sum of two types can be represented in Generic Haskell using the *Sum* data type:

$$\textbf{data}\ Sum\ a\ b = Inl\ a \mid Inr\ b$$

The type *Sum* $a$ $b$ may also be written as $a$ :+: $b$. To represent the product of two types, the *Prod* data type is used:

$$\textbf{data}\ Prod\ a\ b = a :*: b$$

As with *Sum*, there is a shorthand notation for the *Prod* type constructor: :*:. Observe that there exist both a data constructor and a type constructor with the name :*:.

Since a constructor in Haskell need not have parameters, it is also necessary to be able to represent products of zero types. This can be done using the *Unit* type:

$$\textbf{data}\ Unit = Unit$$

Using *Unit*, :+: and :*:, it is possible to describe the structure of any Haskell data type. Haskell has a number of built-in primitive data types that can not easily or efficiently be represented in the sum of products form. These types (*Int*, *Char*, (→) and *IO*) are referred to as *primitive types* and must be used directly in Generic Haskell.

**Example**

The standard Haskell data type *Maybe* is introduced by the following declaration:

**data** *Maybe a* = *Nothing* | *Just a*

The structure of the *Maybe Int* type can be described as:

*Unit* :+: *Int*

### 1.2.2 Structure Types

The concept of the structure of a data type that we have now informally described, gives rise to the definition of *structure types*. A *structure type* of a type $t$ is a type which is isomorphic to the top-level structure of $t$ and is constructed using the binary sum type constructor :+:, the binary product type constructor :*: and the *Unit* type constant defined above.

Given a data type $T$, the structure type of $T$ (usually written as $T^\circ$) can be constructed as follows:

1. Replace every constructor alternative $K_i\ t_{i1} \ldots t_{ik_i}$ of $T$ by the product of the types of the arguments of $K_i$: $t_{i1} :*: \ldots :*: t_{ik_i}$. If a constructor of $T$ does not have arguments, replace it by the *Unit* type constant.

2. Construct the sum of the types obtained for each constructor in step 1: $(t_{11} :*: \ldots :*: t_{1k_1}) :+: \ldots :+: (t_{n1} :*: \ldots :*: t_{nk_n})$.

Furthermore, all type parameters of $T$ are inherited by $T^\circ$.

As will become clear later on, structure types play an important role in the process of translating Generic Haskell code to traditional Haskell code.

**Example**

The *Tree* data type is commonly defined as:

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

The structure type of *Tree* is given by the following type synonym:

**type** *Tree*° *a* = *Unit* :+: (*a* :*: (*Tree a*) :*: (*Tree a*))

Observe that the *Tree* data type is a recursive type whereas *Tree*° is not a recursive type.

### 1.2.3 Kinds

Type constructors that take one or more types as arguments can be regarded as functions on types which also produce a type as a result. Analogous to determining types for values and for functions on values, we can determine *kinds* for types and for functions on types (i.e. type constructors). A kind can thus informally be thought of as a type of a type.

Type constants (i.e. type constructors taking zero arguments) such as *Int* and *Bool* have kind $*$. The type constructor *Maybe*, which takes one type argument, has kind $* \rightarrow *$ and type constructor *Either* (which takes two arguments) has kind $* \rightarrow * \rightarrow *$.

Similar to functions taking functions as arguments, type constructors may take type constructors as arguments. A type constructor which takes a type constructor as its argument has a *higher-order kind.*

**Example**

An example of a type constructor with a higher-order kind is the generalized rose tree [HJ03]

$$\textbf{data}\ \mathit{GRose}\ f\ a = \mathit{GBranch}\ a\ (f\ (\mathit{GRose}\ f\ a))$$

which is a generalization of the rose tree using lists:

$$\textbf{data}\ \mathit{Rose}\ a = \mathit{Branch}\ a\ [\mathit{Rose}\ a]$$

The GRose type constructor has kind $(* \rightarrow *) \rightarrow * \rightarrow *$.

As will become clear later on (see section 1.4), kinds play an important role in determining types for generic functions.

## 1.3 Type-indexed Values

A type-indexed value is a function that takes a type as its argument and produces a regular value (i.e. a value that does not depend on a type). In Generic Haskell, the type argument to type-indexed values is placed between {| and |} brackets (i.e. {| |}).

```
struct{|Char|} = "Char"
struct{|Int|} = "Int"
struct{|Unit|} = "Unit"
struct{|Maybe|} a = "Maybe" ++ a
struct{|:+:|} a b = "(" ++ a ++ ":+:" ++ b ++ ")"
struct{|:*:|} a b = "(" ++ a ++ ":*:" ++ b ++ ")"
```

The type-indexed value *struct* defined above is a string whose content depends on the type argument given. Given the definition above, applying *struct* to a type yields the following results:

```
struct{|Int|} = "Int"
struct{|Maybe (Either (Int, Char) Bool)|} =
        "Maybe((Int:*:Char):+:(Unit:+:Unit))"
```

Except for the sum and product cases, the definition of *struct* in the example above is straightforward. The $:+:$ and $:*:$ type constructors each take two type arguments, say $\alpha$ and $\beta$. For this reason, the cases of *struct* for the $:+:$ and $:*:$ type constructors also each each take two arguments. These arguments correspond with the recursive calls for the arguments of the sum or product, in this case $struct\{\!|\alpha|\!\}$ and $struct\{\!|\beta|\!\}$.

The definition of *struct* above, introduced a type-indexed string value. In most practical cases however, type-indexed values are used to define generic functions which are also occasionally referred to as *polytypic functions* [JJ97].

### Example

A standard example of a generic function is the *gmap* function which is a generic version of the well-known *map* function in Haskell. Although the original *map* only operates on list-types, its generic counterpart can handle arguments of any type.

$$
\begin{aligned}
&gmap\{\!|Unit|\!\} = id \\
&gmap\{\!|Int|\!\} = id \\
&gmap\{\!|:+:|\!\}\ gmapA\ gmapB\ (Inl\ a) = Inl\ (gmapA\ a) \\
&gmap\{\!|:+:|\!\}\ gmapA\ gmapB\ (Inr\ b) = Inr\ (gmapB\ b) \\
&gmap\{\!|:*:|\!\}\ gmapA\ gmapB\ (a:*:b) = (gmapA\ a):*:(gmapB\ b)
\end{aligned}
$$

It now becomes possible to define the original *map* function on lists as:

$$map = gmap\{\!|[\,]|\!\}$$

In Haskell, the built-in type constructor for lists is written as $[\,]$, which has kind $* \to *$. The type $[\alpha]$ for a list containing values of type $\alpha$ is shorthand for $[\,]\ \alpha$. Similarly the tuple-type $(\alpha, \beta)$ is shorthand for $(,)\ \alpha\ \beta$ where $(,)$ is a built-in type constructor with kind $* \to * \to *$.

The built-in Haskell data type for lists, $[\,]$ is defined as follows:

$$\textbf{data}\ [\,]\ a = [\,] \mid (:)\ a\ ([\,]\ a)$$

However, for readability we shall occasionally make use of the following isomorphic definition:

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

Based on this definition we can derive the structure type of *List*:

$$\textbf{type}\ List^{\circ}\ a = Unit:+:(a:*:List\ a)$$

As was mentioned before, the type-operators $:+:$ and $:*:$ are syntactic sugar for the type constructors *Sum* and *Prod* respectively.

The declaration given above for *gmap* only defines the *gmap* function for structure types and for one primitive type (*Int*). When we supplied the *List* type constructor as the type argument to *gmap* in the declaration of *map*, a new definition of *gmap* was created for the *List* type constructor. Since *gmap* was not originally defined for *List*, the big question is: how is this new definition for list-types created? While the intricate details are postponed until the next chapter, we will say for now that the structure type of *List* is used to construct a version of *gmap* tailored for type *List*.

The parameters *gmapA* and *gmapB* of the *gmap* cases for types `:+:` and `:*:` represent the recursive invocations of *gmap* for the type parameters of `:+:` and `:*:`. Since the *List* type constructor only takes a single type argument (i.e. it has kind $* \to *$), the specialization $gmap\{\!|List|\!\}$ takes one function as its argument. In the case of $gmap\{\!|List|\!\}$, this function shall be applied to all elements of the supplied list.

## 1.4 Kind-indexed Types

In the previous section type-indexed values were introduced which allow generic functions to be defined. Just as with regular functions, generic functions also have a type. However, as we have seen in the previous section, the arity of a generic function can differ per case since it depends on the arity of particular type constructors. This suggests that the type of a generic function depends on the kind of the type it is applied to. Since the Haskell type system is not expressive enough for such types, a more sophisticated mechanism is required. *Kind-indexed types* provide the flexibility that is needed. The type of a generic function can be defined by induction on the structure of kinds [Hin00].

A kind-indexed type is a function that takes a kind as an argument and produces a regular type (i.e. a type that does not depend on a kind). The kind argument to kind-indexed type is placed between $\{\!|$ and $|\!\}$ brackets (i.e. `{[ ]}`).

Kind-indexed types are defined in Generic Haskell by defining a case for the $*$ kind as well as a case for the kind $k \to l$ where $k$ and $l$ are also kinds.

The following kind-indexed type definition can be used to assign a type to the *struct* type-indexed value defined in the previous section.

$$\begin{array}{l}\textbf{type}\ Struct\{\!|*|\!\} = String \\ \textbf{type}\ Struct\{\!|k \to l|\!\} = Struct\{\!|k|\!\} \to Struct\{\!|l|\!\}\end{array}$$

The following declaration expresses that the type-indexed value *struct* has a kind-indexed type *Struct*:

$$struct\{\!|t :: k|\!\} :: Struct\{\!|k|\!\}$$

It is easy to verify that the definition of *Struct* captures the notion that the arity of *struct* depends on the arity of the provided type argument:

$$\begin{array}{l}struct\{\!|t :: *|\!\} :: String \\ struct\{\!|Maybe|\!\} :: String \to String \\ struct\{\!|Either|\!\} :: String \to String \to String\end{array}$$

The kind-indexed type that is necessary to assign a type to the *gmap* function is somewhat more involved.

$$\begin{array}{l}\textbf{type}\ Map\{\!|*|\!\}\ t_1\ t_2 = t_1 \to t_2 \\ \textbf{type}\ Map\{\!|k \to l|\!\}\ t_1\ t_2 = \textbf{forall}\ u_1\ u_2\,.\,Map\{\!|k|\!\}\ u_1\ u_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad \to Map\{\!|l|\!\}\ (t_1\ u_1)\ (t_2\ u_2) \\ gmap\{\!|t :: k|\!\} :: Map\{\!|k|\!\}\ t\ t\end{array}$$

The following type declarations show the type of *gmap* for several type constructors:

$$gmap\{\!|Int|\!\} :: Int \to Int$$

$$
\begin{aligned}
&gmap\{\![\,[\,]\,]\!\} :: \textbf{forall}\ u_1\ u_2\,.\,(u_1 \to u_2) \to [\,]\ u_1 \to [\,]\ u_2 \\
&gmap\{\![\,:+:\,]\!\} :: \textbf{forall}\ u_1\ u_2\,.\,(u_1 \to u_2) \\
&\qquad\qquad \to \textbf{forall}\ u_3\ u_4\,.\,(u_3 \to u_4) \to (u_1 :+: u_3) \to (u_2 :+: u_4)
\end{aligned}
$$

The **forall** quantifications may of course be omitted so that $u_1 \ldots u_4$ become free type variables.

## 1.5 Specialization

Defining generic functions is only useful if there exists the possibility of using them on values of concrete Haskell data types. As was demonstrated in the previous sections, a generic function must first be applied to a type argument before it can be called from Haskell code.

The application of a type-indexed value to a type yields an ordinary value which does not depend on a type. The process of generating an ordinary value from a type-indexed value is called *specialization*. When a generic function is specialized for a concrete data type, the Generic Haskell compiler generates a definition of a concrete function for the supplied data type.

A generic function $g$ is defined by specifying a value for a number of primitive types and for each constructor of structure types. The challenging part of specializing $g$ is generating an implementation for a Haskell data type $T$ given just the structure type of $T$ and the definition of $g$ for structure types. Although it is technically possible to generate a definition of $g$ for type $T$ directly, this is not how the problem is handled in the Generic Haskell compiler, at least for the time being.

When a specialization $g\{\!|T|\!\}$ is processed by the compiler, functions are generated that can be used to convert back and forth between values of $T$ and the structure type $T^\circ$. With these conversion functions at hand, the task of specialization has become much easier. The current implementation of the compiler will generate a specialization of $g$ for values of the structure type and will wrap calls to this function in calls to the conversion functions. Generating a specialization of $g$ for a structure type is straightforward as $g$ has been directly defined for structure types.

Further details on how the process of function specialization is currently carried out in practice will be given in chapter 3. First the principles of specialization will be discussed in chapter 2.

## 1.6 Optimizing Specializations

Converting values back and forth between their original representation and structure type representation simplifies the process of specialization considerably. However, this approach also leads to superfluous computation and thus

bad run-time performance of generic functions.

### Example

If we let *from* and *to* represent conversion functions generated to convert a list value to structural representation and back, then a sketch of the evaluation process of the following expression illustrates the inefficiency of the compiled specializations:

$$
\begin{aligned}
& \mathit{gmap}\{\!|\mathit{List}|\!\}\ f\ [1,2,3] \\
= {} & \mathit{to}\ (\mathit{gmap}\{\!|\mathit{List}^\circ|\!\}\ (\mathit{from}\ [1,2,3])) \\
= {} & \mathit{to}\ (\ldots(\mathit{to}\ (\mathit{gmap}\{\!|\mathit{List}^\circ|\!\}\ (\mathit{from}\ [2,3])))\ldots) \\
= {} & \mathit{to}\ (\ldots(\mathit{to}\ (\ldots(\mathit{to}\ (\mathit{gmap}\{\!|\mathit{List}^\circ|\!\}\ (\mathit{from}\ [3])))\ldots))\ldots) \\
= {} & \mathit{to}\ (\ldots(\mathit{to}\ (\ldots(\mathit{to}\ (\ldots(\mathit{to}\ (\mathit{gmap}\{\!|\mathit{List}^\circ|\!\}\ (\mathit{from}\ [\,])))\ldots))\ldots))\ldots)
\end{aligned}
$$

The repeated conversions stem from the fact that the *List* type is among the components of $\mathit{List}^\circ$. This means that the specialization $\mathit{gmap}\{\!|\mathit{List}^\circ|\!\}$ will include a call to $\mathit{gmap}\{\!|\mathit{List}|\!\}$. The latter function will subsequently wrap a call to $\mathit{gmap}\{\!|\mathit{List}^\circ|\!\}$ in calls to the conversion functions.

A relatively easy method of overcoming the overhead caused by conversion between representations is by performing all conversion-related computations at compile-time using partial evaluation. This idea is described in more detail in chapters 4, 5 and chapter 6. Finally chapter 7 demonstrates how our solution performs in practice.

# Chapter 2

# Principles of Specialization

In this chapter we will examine the fundamentals upon which specialization in Generic Haskell is based. Specializing a generic function *poly* for a concrete data type $t$, amounts to using a specialization of *poly* for $t^\circ$ in combination with conversion functions to convert values of type $t$ to values of type $t^\circ$ and vice versa. The characteristics of these conversion functions will be discussed in section 2.1. Section 2.2 explains how type-indexed values are specialized for structure types.

## 2.1 Structural Conversions

### 2.1.1 Embedding-Projection Pairs

In section 1.2.2 we discussed how a structure type $t^\circ$ can be constructed from a data type $t$. It is important to observe that the types $t$ and $t^\circ$ are isomorphic, which means that it is possible to define functions $from :: t \rightarrow t^\circ$ and $to :: t^\circ \rightarrow t$ such that $to \cdot from = id$ and $from \cdot to = id$. Together these two functions form an *embedding-projection pair* with *from* being the *embedding* and *to* being the *projection* [Pie91].

Generating embedding *from* and projection *to* for a type $t$ is not difficult if we recall the definitions of type constructors *Sum* and *Prod* on page 2. The *from* function must combine all arguments of each constructor of $t$ into a value of type *Prod*. These products must then be stored in a value of type *Sum*. Constructors that do not have arguments can not be stored as a *Prod* value and must be stored as *Unit*. The *to* function is also straightforward since it is simply the inverse of *from*.

#### Example

Below an embedding-projection pair is given for the *Tree* data type as defined on page 3:

$$
\begin{aligned}
&from_{Tree} :: Tree\ a \rightarrow Tree^\circ\ a \\
&from_{Tree}\ Leaf = Inl\ Unit \\
&from_{Tree}\ (Node\ x\ l\ r) = Inr\ (x :*: (l :*: r))
\end{aligned}
$$

$$
\begin{aligned}
&to_{Tree} :: Tree^{\circ}\ a \rightarrow Tree\ a \\
&to_{Tree}\ (Inl\ Unit) = Leaf \\
&to_{Tree}\ (Inr\ (x :*: (l :*: r))) = Node\ x\ l\ r
\end{aligned}
$$

As we shall see in the next chapter, embedding-projection pairs are conveniently stored using the *EP* data type:

$$\mathbf{data}\ EP\ a\ b = EP\ (a \rightarrow b)\ (b \rightarrow a)$$

Thus:

$$
\begin{aligned}
&ep_{Tree} :: EP\ Tree\ \ Tree^{\circ} \\
&ep_{Tree} = EP\ from_{Tree}\ to_{Tree}
\end{aligned}
$$

The following two functions on embedding-projection pairs come in handy:

$$
\begin{aligned}
&from :: EP\ a\ b \rightarrow (a \rightarrow b) \\
&from\ (EP\ a2b\ b2a) = a2b \\
\\
&to :: EP\ a\ b \rightarrow (b \rightarrow a) \\
&to\ (EP\ a2b\ b2a) = b2a
\end{aligned}
$$

Another useful operation on embedding-projection pairs is the composition operation $\circ$, which can be defined as follows [HJ03]:

$$
\begin{aligned}
&(\circ) :: EP\ b\ c \rightarrow EP\ a\ b \rightarrow EP\ a\ c \\
&(EP\ b2c\ c2b)\ \circ\ (EP\ a2b\ b2a) = EP\ (b2c\ \cdot\ a2b)\ (c2b\ \cdot\ b2a)
\end{aligned}
$$

### 2.1.2 Maps

The Haskell *map* function has type $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ and is intuitively usually interpreted as a function that takes a function as its first argument, which is then applied to all elements of the list in its second argument. However, by currying and the fact that the type-operator $(\rightarrow)$ associates to the right, the *map* function can also be thought of as a function that *lifts* a function with type $a \rightarrow b$ to a function with type $[a] \rightarrow [b]$, which operates on lists. Or more generally, a map for a type constructor $F$ of kind $* \rightarrow *$ lifts a function of type $a \rightarrow b$ to a function of type $F\ a \rightarrow F\ b$ [HJ03].

So far we have seen that converting a value of type $t$ to a value of type $t^{\circ}$ and vice versa can be done by using the functions $from_t$ and $to_t$. However, given a type constructor $F$ of kind $* \rightarrow *$ we are not able to easily convert a value of type $F\ t$ to a value of type $F\ t^{\circ}$. For this purpose the generic *bimap* function will now be introduced [dW02]. For a type constructor $F$, the *bimap* function can be used to lift an embedding-projection pair with type $EP\ a\ b$ to an embedding-projection pair with type $EP\ (F\ a)\ (F\ b)$. The *bimap* function may thus be regarded as a variant of *map* that lifts two functions packed in an embedding-projection pair at once.

$$
\begin{aligned}
&\mathbf{type}\ Bimap\{\!|*|\!\}\ t_1\ t_2 = EP\ t_1\ t_2 \\
&\mathbf{type}\ Bimap\{\!|k \rightarrow l|\!\}\ t_1\ t_2 = \\
&\quad \mathbf{forall}\ a_1\ a_2\ .\ Bimap\{\!|k|\!\}\ a_1\ a_2 \rightarrow Bimap\{\!|l|\!\}\ (t_1\ a_1)\ (t_2\ a_2)
\end{aligned}
$$

$$
\begin{aligned}
&bimap\{\!|t :: k|\!\} :: Bimap\{\!|k|\!\}\ t\ t \\
&bimap\{\!|Unit|\!\} = EP\ id\ id \\
&bimap\{\!|Int|\!\} = EP\ id\ id \\
&bimap\{\!|:+:|\!\}\ (EP\ a2b\ b2a)\ (EP\ c2d\ d2c) \\
&\qquad = EP\ (\lambda x \rightarrow \text{case } x \text{ of } Inl\ a \rightarrow Inl\ (a2b\ a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad Inr\ c \rightarrow Inr\ (c2d\ c)) \\
&\qquad\qquad\qquad\quad (\lambda x \rightarrow \text{case } x \text{ of } Inl\ b \rightarrow Inl\ (b2a\ b) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad Inr\ d \rightarrow Inr\ (d2c\ d)) \\
&bimap\{\!|:*:|\!\}\ (EP\ a2b\ b2a)\ (EP\ c2d\ d2c) \\
&\qquad = EP\ (\lambda(a:*:c) \rightarrow (a2b\ a):*:(c2d\ c)) \\
&\qquad\qquad\quad (\lambda(b:*:d) \rightarrow (b2a\ b):*:(d2c\ d)) \\
&bimap\{\!|(\rightarrow)|\!\}\ (EP\ a2b\ b2a)\ (EP\ c2d\ d2c) \\
&\qquad = EP\ (\lambda a2c \rightarrow c2d\ \cdot\ a2c\ \cdot\ b2a) \\
&\qquad\qquad\quad (\lambda b2d \rightarrow d2c\ \cdot\ b2d\ \cdot\ a2b)
\end{aligned}
$$

The most interesting case is for *bimap* specialized for the function type constructor $(\rightarrow)$ [1] which has kind $* \rightarrow * \rightarrow *$. Instantiating the kind-indexed type *Bimap* with kind $* \rightarrow * \rightarrow *$ gives $bimap\{\!|(\rightarrow)|\!\}$ the following type:

$$bimap\{\!|(\rightarrow)|\!\} :: EP\ a\ b \rightarrow EP\ c\ d \rightarrow EP\ (a \rightarrow c)\ (b \rightarrow d)$$

The specialization $bimap\{\!|(\rightarrow)|\!\}$ results in a function that takes an embedding-projection pair between types $a$ and $b$ and an embedding-projection pair between types $c$ and $d$, and produces an embedding-projection pair that can be used to convert between functions of types $(a \rightarrow c)$ and $(b \rightarrow d)$. This will turn out to be a very useful operation because it can be used to convert a specialization of a generic function for structure type $t^\circ$ into a specialization of a generic function for the original type $t$.

**Example**

Suppose that a specialization of *gmap* for the structure type $Tree^\circ$ is known. This function will have the following type:

$$gmap\{\!|Tree^\circ|\!\} :: (a \rightarrow b) \rightarrow (Tree^\circ\ a) \rightarrow (Tree^\circ\ b)$$

Using *bimap* we can derive a specialization of *gmap* for the *Tree* data type:

$$
\begin{aligned}
&gmap\{\!|Tree|\!\} :: (a \rightarrow b) \rightarrow (Tree\ a) \rightarrow (Tree\ b) \\
&gmap\{\!|Tree|\!\} = to\ (bimap\{\!|(\rightarrow)|\!\}\ ep_{Tree}\ ep_{Tree})\ gmap\{\!|Tree^\circ|\!\}
\end{aligned}
$$

## 2.2 Specialization of Type-Indexed Values

Now that we are able to convert a generic function specialized for a structure type $t^\circ$ to a function specialized for type $t$, we can concentrate on specializing generic functions for structure types. The general form of a type-indexed value *poly* is:

[1] Since $(\rightarrow)$ is a Haskell built-in type constructor that can not be defined using the **data** construct, it does not have a structure type. Consequently if a generic function can sensibly be defined for the function space, a definition for the $(\rightarrow)$ type constructor must be given explicitly (i.e. $(\rightarrow)$ is a primitive type)

$$
\begin{aligned}
&poly\{\!|t :: k|\!\} :: PolyT\{\!|k|\!\}\ t \\
&poly\{\!|Int|\!\} = \ldots \\
&poly\{\!|Unit|\!\} = \ldots \\
&poly\{\!|:+:|\!\}\ f\ g = \ldots \\
&poly\{\!|:*:|\!\}\ f\ g = \ldots
\end{aligned}
$$

Type-indexed values are typically defined for the primitive types and for structure type constructors *Unit*, *Sum* (i.e. $:+:$) and *Prod* (i.e. $:*:$). This makes specialization for structure type constructors easy.

### 2.2.1 Rewriting Type Applications

One of the fundamental ideas behind generic definitions in Generic Haskell is that type application must be interpreted as value application [Hin00]. This means that if $F :: \kappa \to \nu$ and $A :: \kappa$ are types with $\kappa$ and $\nu$ being kinds, then $poly\{\!|F\ A|\!\}$ is to be interpreted as $poly\{\!|F|\!\}\ poly\{\!|A|\!\}$.

**Example**

The specialization of $poly\{\!|Maybe\ (Tree\ Int)|\!\}$ is rewritten to

$$poly\{\!|Maybe|\!\}\ (poly\{\!|Tree|\!\}\ poly\{\!|Int|\!\})$$

Obviously for this translation to work, a specialization must accept the extra arguments that it is passed. A specialization $poly\{\!|F|\!\}$ for a type constructor $F$ with arity $n$ will have to receive $n$ arguments not including the value that it should operate on. For example, the component $poly\{\!|:+:|\!\}$ of the declaration of *poly* takes two arguments $f$ and $g$ because type constructor $:+:$ has kind $* \to * \to *$ (i.e. its arity is 2).

The fact that type-indexed values have kind-indexed types [Hin00] as we mentioned in the previous chapter, is a consequence of this principle of specialization.

### 2.2.2 Specialization for Structure Types

In summary, the method that is used to specialize a generic function *poly* for a type $t$ that does not contain type applications, can be described by the following two rules:

a. If $poly\{\!|t|\!\}$ is a component of the declaration of *poly* use the body of the component.

b. If $poly\{\!|t|\!\}$ is not a component of the declaration of *poly*, specialize *poly* for the structure type of $t$ and, using *bimap* and an embedding-projection pair between $t$ and $t^\circ$, transform this specialization to a function for the original type.

A structure type $t^\circ$ generally exhibits the following form:

$$\textbf{type}\ t^\circ\ u_1 \ldots u_k = (t_{11} :*: \ldots :*: t_{1k_1}) :+: \ldots :+: (t_{n1} :*: \ldots :*: t_{nk_n})$$

where $t_{11} \ldots t_{nk_n}$ are types such as the structure type constructor *Unit*, primitive type *Int*, type-variable $u_i$ or any other type such as $[u_i]$. If type $t$ is a recursive type, the structure type $t^\circ$ will contain $t_{ij} = t$.

Specializing type-indexed value *poly* for a structure type $t^\circ$ means we must first interpret all type applications in $t$ using the principle discussed in the previous subsection. Doing so for specialization $poly\{\!|t^\circ|\!\}$ yields:

$$
\begin{array}{l}
poly\{\!|t^\circ|\!\} = \\
\quad poly\{\!|:+:|\!\}\ (poly\{\!|:*:|\!\}\ poly\{\!|t_{11}|\!\} \ldots (poly\{\!|:*:|\!\} \ldots poly\{\!|t_{1k_1}|\!\})) \\
\qquad\qquad \ldots \\
\qquad\qquad (poly\{\!|:+:|\!\} \ldots (poly\{\!|:*:|\!\}\ poly\{\!|t_{n1}|\!\} \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad (poly\{\!|:*:|\!\} \ldots poly\{\!|t_{nk_n}|\!\})))
\end{array}
$$

This preliminary definition of $poly\{\!|t^\circ|\!\}$ can be further refined by replacing specializations for structure type constructors by the appropriate right hand side from the declaration of *poly*. Other specializations $poly\{\!|t_{ij}|\!\}$ are to be treated as any other specialization contained in a Generic Haskell expression.

### Example

We will show how the *gmap* function defined on page 5 is specialized for the structure type $Tree^\circ$, which is defined as:

$$\textbf{type}\ Tree^\circ\ a = Unit :+: (a :*: (Tree\ a) :*: (Tree\ a))$$

Processing all type-applications we obtain:

$$
\begin{array}{l}
gmap\{\!|Tree^\circ|\!\}\ f = gmap\{\!|:+:|\!\}\ gmap\{\!|Unit|\!\} \\
\qquad\qquad\qquad\qquad (gmap\{\!|:*:|\!\}\ f \\
\qquad\qquad\qquad\qquad\qquad\qquad (gmap\{\!|:*:|\!\}\ (gmap\{\!|Tree|\!\}\ f) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (gmap\{\!|Tree|\!\}\ f)))
\end{array}
$$

Then by consulting the definition of *gmap* to replace the specializations $gmap\{\!|Unit|\!\}$, $gmap\{\!|:+:|\!\}$ and $gmap\{\!|:*:|\!\}$ we get:

$$
\begin{array}{l}
gmap\{\!|Tree^\circ|\!\}\ f\ (Inl\ a) = Inl\ (id\ a) \\
gmap\{\!|Tree^\circ|\!\}\ f\ (Inr\ (a :*: (ba :*: bb))) = \\
\qquad Inr\ (f\ a :*: (gmap\{\!|Tree|\!\}\ f\ ba :*: gmap\{\!|Tree|\!\}\ f\ bb))
\end{array}
$$

This is the final result of specializing *gmap* for structure type $Tree^\circ$.

# Chapter 3

# Compiling Generic Haskell

In the previous chapter we examined the principles upon which specialization in Generic Haskell depends. Now it is time to take a closer look at how the Generic Haskell compiler does its work in practice. Knowing what the exact output of the compiler looks like is essential for being able to carry out optimizations. This chapter describes how the generic programming constructs are translated into plain Haskell by the current version [1] of the Generic Haskell compiler.

An overview of the tasks performed by the Generic Haskell compiler is provided in section 3.1. Section 3.2 shows what Haskell code is generated for data type declarations in a Generic Haskell program. In section 3.3 the translation of kind-indexed types will be examined and finally section 3.4 has the details on the method by which type-indexed values are specialized in practice.

For reasons of presentation some code presented in examples in this chapter has been slightly simplified to increase its comprehensibility.

## 3.1 Compiling Generic Haskell

A Generic Haskell program may contain any [2] Haskell declaration. As discussed in chapter 1, a Generic Haskell program may also contain type-indexed value and kind-indexed type declarations. Furthermore, in expressions of a Generic Haskell program, kind-indexed types may be applied to kinds and type-indexed values may be applied to types.

The main task of compiling Generic Haskell programs to Haskell is generating definitions for specializations of generic functions and replacing the uses of specializations with calls to the generated functions. Furthermore to assign types to these generated functions, kind-indexed types must be instantiated with suitable kinds.

The compiler is also responsible for type-checking Generic Haskell programs to catch programming errors at an early stage. However, this aspect of the compilation process is currently still under development.

[1] At the time of writing the latest version of the Generic Haskell compiler is 1.40.1

[2] There are a few small exceptions; consult [CJL01] for details.

## 3.2 Data Types

### 3.2.1 Structure Types

For every data type used in a Generic Haskell program, a structure type definition is emitted by the compiler. As was mentioned before, any Haskell data type can be represented as a sum of products with the type parameters of each constructor forming a product.

Conceptually a structure type is a type synonym of which the right hand side is built using the type constructors introduced in section 1.2.1. In practice however, the Generic Haskell compiler declares a number of type-synonyms to achieve the same effect.

**Example**

The *Tree* data type declared by:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

can be thought of as having the following structure type:

$$\textbf{type}\ \mathit{Tree}^{\circ}\ a = \mathit{Unit} :+: (a :*: (\mathit{Tree}\ a) :*: (\mathit{Tree}\ a))$$

However, the declarations generated by the Generic Haskell compiler for the structure type of *Tree* are as follows:

$$\begin{array}{l}
\textbf{type}\ \mathit{GHC_Leaf}\ a = \mathit{Tree}\ a \\
\textbf{type}\ \mathit{GHC_Node}\ a = \mathit{Tree}\ a \\
\\
\textbf{type}\ \mathit{GHt_Tree}\ a = \mathit{Sum}\ (\mathit{Con}\ (\mathit{GHC_Leaf}\ a))\ (\mathit{Con}\ (\mathit{GHC_Node}\ a)) \\
\textbf{type}\ \mathit{GHt_GHC_Leaf}\ a = \mathit{Unit} \\
\textbf{type}\ \mathit{GHt_GHC_Node}\ a = \mathit{Prod}\ a\ (\mathit{Prod}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a))
\end{array}$$

The prefix *GHC_* of the name of a type is used to indicate that it represents the type of a constructor-case. Names of structure types are prefixed by *GHt_*. The *Con* [3] constructor in the definition of *GHt_Tree* is used to wrap a value so that information about constructors can be used in generic functions. This is a feature of Generic Haskell that we have not mentioned since to us it is not of particular importance. The uses of the *Con* constructor can therefore safely be ignored in the rest of this chapter.

This shows that for every constructor two type synonyms are generated. The purpose of these types will become clear in the next subsection on the generation of embedding-projection pairs.

For a data type that is defined by:

$$\textbf{data}\ T\ u_1 \ldots u_k = K_1\ t_{11} \ldots t_{1k_1} \mid \cdots \mid K_n\ t_{n1} \ldots t_{nk_n}$$

the following structure type declarations are generated:

$$\begin{array}{l}
\textbf{type}\ \mathit{GHC_K_1}\ u_1 \ldots u_k = T\ u_1 \ldots u_k \\
\quad \ldots \\
\textbf{type}\ \mathit{GHC_K_n}\ u_1 \ldots u_k = T\ u_1 \ldots u_k
\end{array}$$

[3] The *Con* data type is defined by: **newtype** *Con c* = *Con c*

$$\begin{array}{l}
\textbf{type}\ GHt_T\ u_1 \ldots u_k = Sum\ (Con\ (GHC_K_1\ u_1 \ldots u_k)) \\
\qquad\qquad\qquad\qquad\qquad\qquad (Sum \ldots (Con\ (GHC_K_n\ u_1 \ldots u_k))) \\
\textbf{type}\ GHt_GHC_K_1\ u_1 \ldots u_k = Prod\ t_{11}\ (Prod \ldots t_{1k_1}) \\
\ldots \\
\textbf{type}\ GHt_GHC_K_n\ u_1 \ldots u_k = Prod\ t_{n1}\ (Prod \ldots t_{nk_n})
\end{array}$$

### 3.2.2 Conversion from/to Structural Representation

Since the translated definition for a generic function will operate on values of a structure type, the compiler has to generate conversion functions that can be used to convert back and forth between a value and its structural equivalent. In the previous chapter we have seen that the functions we need together form an embedding-projection pair.

The following data type is used to represent embedding-projection pairs:

$$\textbf{data}\ EP\ a_1\ a_2 = EP\ (a_1 \rightarrow a_2)\ (a_2 \rightarrow a_1)$$

The following two functions are used to extract respectively the embedding and the projection function from an embedding-projection pair.

$$\begin{array}{l}
from\ (EP\ a\ b) = a \\
to\ (EP\ a\ b) = b
\end{array}$$

As we have seen in the previous section, a product structure type declaration is generated for each constructor of a data type. This means the compiler must also generate an embedding-projection pair for each constructor of a data type to be able to convert an instance of the constructor back and forth to structural representation. Besides that, an embedding-projection pair must of course also be generated for the type the constructors belong to. Generating these functions is not difficult as can be seen in the example below.

**Example**

For the *Tree* data type defined in the previous section, the following embedding-projection pairs are generated.

$$\begin{array}{l}
ghEP_Tree :: EP\ (Tree\ a)\ (GHt_Tree\ a) \\
ghEP_Tree = \textbf{let}\ \{ gh_From\ gh_A@Leaf = Inl\ (Con\ gh_A) \\
\qquad\qquad\qquad ; gh_From\ gh_A@(Node\ gh_a\ gh_b\ gh_c) = Inr\ (Con\ gh_A) \\
\qquad\qquad\qquad ; gh_To\ (Inl\ (Con\ gh_A)) = gh_A \\
\qquad\qquad\qquad ; gh_To\ (Inr\ (Con\ gh_A)) = gh_A \\
\qquad\qquad\qquad \}\ \textbf{in}\ EP\ gh_From\ gh_To
\end{array}$$

$$\begin{array}{l}
ghEP_GHC_Leaf :: EP\ (GHC_Leaf\ a)\ (GHt_GHC_Leaf\ a) \\
ghEP_GHC_Leaf = \textbf{let}\ \{ gh_From\ Leaf = Unit \\
\qquad\qquad\qquad\qquad ; gh_To\ Unit = Leaf \\
\qquad\qquad\qquad\qquad \}\ \textbf{in}\ EP\ gh_From\ gh_To
\end{array}$$

$$\begin{array}{l}
ghEP_GHC_Node :: EP\ (GHC_Node\ a)\ (GHt_GHC_Node\ a) \\
ghEP_GHC_Node = \\
\qquad\qquad \textbf{let}\ \{ gh_From\ (Node\ gh_a\ gh_b\ gh_c) = (gh_a : * : (gh_b : * : gh_c)) \\
\qquad\qquad\quad ; gh_To\ (gh_a : * : (gh_b : * : gh_c)) = Node\ gh_a\ gh_b\ gh_c \\
\qquad\qquad\quad \}\ \textbf{in}\ EP\ gh_From\ gh_To
\end{array}$$

## 3.3 Translating Kind-Indexed Types

The type of a type-indexed value depends on the kind of the type constructor that it is applied to. Kind-indexed types provide the flexibility that is required to assign types to type-indexed values.

In the next section we will show that when a type-indexed value is specialized for a particular data type, the Generic Haskell compiler will generate a Haskell function which is an implementation of the type-indexed value for the particular data type. Along with the declaration of this function the compiler emits a type declaration that assigns a type to the function that was generated. In order to determine this type, the kind-indexed type that was assigned to the type-indexed value as part of its its declaration, is instantiated with the kind of the type that the type-indexed value was specialized for. This means the kind-indexed type can be specialized for a particular kind. The type that is obtained from carrying out this specialization is assigned to the Haskell function that was generated for the specialization of the type-indexed value.

The general form of a kind-indexed type is as follows:

$$\begin{array}{l}\textbf{type}\ PolyT\{\!\![*]\!\!\}\ t_1 \ldots t_n = T\ t_1 \ldots t_n \\ \textbf{type}\ PolyT\{\!\![k \to l]\!\!\}\ t_1 \ldots t_n = \\ \quad \textbf{forall}\ u_1 \ldots u_n\ .\ PolyT\{\!\![k]\!\!\}\ u_1 \ldots u_n \to PolyT\{\!\![l]\!\!\}\ (t_1\ u_1) \ldots (t_n\ u_n)\end{array}$$

To specialize a kind-indexed type for a kind, the compiler will solely expand the definition above to construct a type.

### Example

In section 1.4 the *gmap* function was given the following kind-indexed type:

$$\begin{array}{l}\textbf{type}\ Map\{\!\![*]\!\!\}\ t_1\ t_2 = t_1 \to t_2 \\ \textbf{type}\ Map\{\!\![k \to l]\!\!\}\ t_1\ t_2 = \textbf{forall}\ u_1\ u_2\ .\ Map\{\!\![k]\!\!\}\ u_1\ u_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad \to Map\{\!\![l]\!\!\}\ (t_1\ u_1)\ (t_2\ u_2) \\ gmap\{\!\![t :: k]\!\!\} :: Map\{\!\![k]\!\!\}\ t\ t\end{array}$$

The *Tree* type constructor has kind $* \to *$.

When the *gmap* function is specialized for the *Tree* type, the type of the resulting function is determined by specializing the *Map* kind-indexed type to the kind of the *Tree* type constructor. *Tree* has kind $* \to *$, which gives us:

$$gmap\{\!\![Tree]\!\!\} :: Map\{\!\![* \to *]\!\!\}\ Tree\ Tree$$

expanding the definition of *Map* subsequently gives:

$$gmap\{\!\![Tree]\!\!\} :: \textbf{forall}\ u_1\ u_2\ .\ (u_1 \to u_2) \to Tree\ u_1 \to Tree\ u_2$$

Since the explicit quantification of $u_1$ and $u_2$ is not valid in Haskell, the type declaration generated by the compiler is:

$$ghs\_gmap\_\_Tree :: (u_1 \to u_2) \to Tree\ u_1 \to Tree\ u_2$$

## 3.4 Translating Type-Indexed Values

The most important translation that is carried out by the compiler is the generation of a Haskell function as a result of the specialization of a type-indexed value for a particular data type. A type-indexed value has the following general form:

$$
\begin{array}{l}
poly\{\!|t :: k|\!\} :: PolyT\{\!|k|\!\}\ t \\
poly\{\!|Int|\!\} = \ldots \\
poly\{\!|Unit|\!\} = \ldots \\
poly\{\!|:+:|\!\}\ f\ g = \ldots \\
poly\{\!|:*:|\!\}\ f\ g = \ldots
\end{array}
$$

Applying the type-indexed value *poly* to a type $t$ yields a specialization $poly\{\!|t|\!\}$ for which a Haskell definition with the name *ghs_poly_t* is generated. The prefix *ghs_* indicates that the function under consideration is the result of a specialization. The suffix *poly_t* attests that this function is a specialization of function *poly* for the type $t$.

### Example

Below is the definition of the *gmap* function which was discussed earlier in section 1.3.

$$
\begin{array}{l}
gmap\{\!|Unit|\!\} = id \\
gmap\{\!|Int|\!\} = id \\
gmap\{\!|:+:|\!\}\ gmapA\ gmapB\ (Inl\ a) = Inl\ (gmapA\ a) \\
gmap\{\!|:+:|\!\}\ gmapA\ gmapB\ (Inr\ b) = Inr\ (gmapB\ b) \\
gmap\{\!|:*:|\!\}\ gmapA\ gmapB\ (a :*: b) = (gmapA\ a) :*: (gmapB\ b)
\end{array}
$$

For the specialization $gmap\{\!|Tree|\!\}$ the compiler produces the following Haskell definition:

$$
\begin{array}{l}
ghs\_gmap\_Tree\ a = to\ (ghs\_bimap\_\ (\rightarrow)\ ghEP\_Tree\ ghEP\_Tree) \\
\qquad\qquad\qquad\qquad (ghs\_gmap\_GHt\_Tree\ a)
\end{array}
$$

The *bimap* specialization for the function type constructor $(\rightarrow)$, which was discussed in section 2.1.2, is used to construct an embedding-projection pair which can be used to transform a function which operates on values of type *GHt_Tree* into a function which operates on values of type *Tree*. We now turn our attention to the *ghs_gmap_GHt_Tree* function which is a specialization of *gmap* for the *GHt_Tree* type:

$$
\begin{array}{l}
ghs\_gmap\_GHt\_Tree\ a = \\
\qquad ghs\_gmap\_Sum\ ((ghs\_gmap\_Con\ ghC\_Leaf)\ (ghs\_gmap\_GHC\_Leaf\ a)) \\
\qquad\qquad\qquad ((ghs\_gmap\_Con\ ghC\_Node)\ (ghs\_gmap\_GHC\_Node\ a))
\end{array}
$$

The *ghC_Leaf* and *ghC_Node* values contain information about the constructors *Leaf* and *Node* respectively. The *ghs_gmap_Con* function is defined as follows:

$$
\begin{array}{l}
ghs\_gmap\_Con\ \_\ c = to\ (ghs\_bimap\_\ (\rightarrow)\ ghEP\_Con\ ghEP\_Con) \\
\qquad\qquad\qquad\qquad (ghs\_gmap\_GHt\_Con\ c)
\end{array}
$$

The *ghs_gmap_Con* function uses *bimap* to transform a function for values of type *GHt_Con* into a function for values of type *Con*. Since the definition of *gmap* does not use any constructor information, the implementation of *gmap_gmap_GHt_Con* is equivalent to the identity function:

```
ghs_gmap_GHt_Con c = c
```

This justifies rewriting the definition of *ghs_gmap_GHt_Tree* to the following for simplicity:

```
ghs_gmap_GHt_Tree a =
    ghs_gmap_Sum (ghs_gmap_GHC_Leaf a) (ghs_gmap_GHC_Node a)
```

The definition of *ghs_gmap_GHC_Leaf* shows that once again *bimap* is used to transform a function operating on values of type *GHt_GHC_Leaf* into a function operating on values of type *GHC_Leaf*:

```
ghs_gmap_GHC_Leaf a = to (ghs_bimap_ (→) ghEP_GHC_Leaf ghEP_GHC_Leaf)
                         (ghs_gmap_GHt_GHC_Leaf a)
ghs_gmap_GHt_GHC_Leaf a = ghs_gmap_Unit
```

This same technique is repeated for the specialization of *gmap* for the other constructor of *Tree*:

```
ghs_gmap_GHC_Node a = to (ghs_bimap_ (→) ghEP_GHC_Node ghEP_GHC_Node)
                         (ghs_gmap_GHt_GHC_Node a)
ghs_Main_gmap_GHt_GHC_Node a =
  ghs_gmap_Prod a (ghs_gmap_Prod (ghs_gmap_Tree a) (ghs_gmapTree a))
```

The only remaining definitions are the ones for *ghs_gmap_Sum* and *ghs_gmap_Prod*, which are both included explicitly as components in the definition of the *gmap* function:

```
ghs_gmap_Prod gh_a gh_b gh_c = ghc_gmap_Prod gh_a gh_b gh_c
ghs_gmap_Sum gh_a gh_b gh_c = ghc_gmap_Sum gh_a gh_b gh_c
```

Notice that the prefix of the functions that are called is *ghc_*. This indicates that the original definition of *gmap* includes a component for the type constructor under consideration (in this case *Prod* and *Sum*):

```
ghc_gmap_Sum gmapA gmapB (Inl a) = Inl (gmapA a)
ghc_gmap_Sum gmapA gmapB (Inr b) = Inr (gmapB b)
ghc_gmap_Prod gmapA gmapB (a:*:b) = (gmapA a):*:(gmapB b)
```

This completes the generation of Haskell code for the specialization *gmap*{|*Tree*|}.

# Chapter 4

# Partial Evaluation of Specializations

In this chapter we will investigate how partial evaluation can be used to alleviate the issues associated with structural conversions in compiled specializations. In section 4.1 the basics of partial evaluation are introduced. One of the techniques that plays a prominent rôle in partial evaluation is symbolic evaluation. The operational semantics of symbolic evaluation for a limited subset of Haskell are given in section 4.2. Applying symbolic evaluation successfully requires an extended form of pattern matching that is capable of matching patterns against symbolic values. Our extended pattern matching semantics are presented in section 4.3. Finally in section 4.4 the compiled specializations that were described in chapter 3 are modelled in the Haskell subset used in section 4.2 and the effects of applying partial evaluation are demonstrated.

## 4.1 Partial Evaluation

Partial evaluation [JGS93] is an optimization technique typically used at compile-time in which one or more inputs of a function are set to fixed values. During partial evaluation, all computations in the definition of a function that do not depend on missing data, are performed at compile-time in order to save computational resources at run-time. Once the remaining input data becomes available (e.g. at run-time), the partially evaluated function definitions can be used to produce a final result (ideally in less time than if the original function definitions had been used).
A partial evaluator takes the following two inputs:

1. A function (or more generally a program) that one would like to optimize.
2. Static input data for the supplied function.

When provided with these two arguments, a partial evaluator will produce a new function that only depends on the input data which was not supplied. The parameters of the original function for which values are not available are called *dynamic parameters*, and the parameters for which a value was supplied are called *static parameters*. The new function that was generated by the partial

evaluator is usually referred to as the *residual function* (or more generally the *residual program*). It may come as no surprise that partial evaluation can only succesfully be used to optimize functions with at least one static parameter.

**Example**

In the following declaration a function *power* is defined which computes $x^n$:

$$\begin{array}{l} power :: Int \rightarrow Int \rightarrow Int \\ power\ x\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * power\ x\ (n-1) \end{array}$$

The *power* function takes two arguments in order to produce a final result. However, if the value of one of the two arguments is already known, it is possible to partially evaluate the definition with respect to this input to produce an optimized version of the definition. In the following definition of *power4* which computes $x^4$, the *power* function is used with the fixed value of 4 for parameter $n$:

$$power4\ x = power\ x\ 4$$

Since the value of one of the two arguments to *power* is known prior to execution, it is possible to apply partial evaluation to reach a more efficient definition of *power4*. If in the original definition of *power* the parameter $n$ is fixed to 4, the following reduction-steps illustrate how a more efficient version of *power4* can be constructed.

$$\begin{array}{l} \quad power4\ x = power\ x\ 4 \\ \Rightarrow power4\ x = \textbf{if}\ 4 \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * power\ x\ 3 \\ \Rightarrow power4\ x = x * \textbf{if}\ 3 \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * power\ x\ 2 \\ \Rightarrow power4\ x = x * x * power\ x\ 2 \\ \Rightarrow power4\ x = x * x * \textbf{if}\ 2 \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * power\ x\ 1 \\ \Rightarrow power4\ x = x * x * x * \textbf{if}\ 1 \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * power\ x\ 0 \\ \Rightarrow power4\ x = x * x * x * x * power\ x\ 0 \\ \Rightarrow power4\ x = x * x * x * x * \textbf{if}\ 0 \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * power\ x\ (-1) \\ \Rightarrow power4\ x = x * x * x * x * 1 \end{array}$$

It is clear that the final definition of *power4* obtained by partial evaluation is more efficient than its original since several recursive calls and tests for equality have been avoided.

Besides being useful for constructing optimized versions of programs, partial evaluation can be used as well for many other purposes. For example, an interpreter for a programming language can be partially evaluated for a certain program source to mimic the process of compilation. This idea can be taken a step further by partially evaluating the partial evaluator itself with an interpreter of a programming language as its first input argument. This leaves us with a residual program which takes a program source as its argument and returns an interpreter partially evaluated for the given program source, also known as a compiler. Partial evaluation thus allows for a compiler to be generated automatically for a language from an implementation of the operational semantics of this language [JGS93]. The ideas of self-application of a partial evaluator and using partial evaluation to construct compiler generators were first explored in [Fut71].

The example above demonstrates the use of two important partial evaluation techniques: *unfolding* and *symbolic evaluation*. These and the most important other partial evaluation techniques are described below.

### 4.1.1 Unfolding

In the *unfolding* technique (also called *inlining*), an applicative expression is replaced by the body of the function that is being called. Parameters of the function in the replacement body are substituted for the arguments of the applicative expression.

**Example**

$$power4\ x = power\ x\ 4$$

Unfolding the definition of the *power* function yields:

$$power4\ x = \textbf{if}\ 4 \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * power\ x\ (4-1)$$

### 4.1.2 Definition Creation

When an applicative expression $f\ x_1 \ldots x_n$ has static arguments, it is usually worthwhile to create a new partially evaluated version of the function $f$. Introducing a new definition based on an existing function definition is called *definition creation*.

**Example**

$$f\ x = power\ x\ 4$$

Since this expression calls *power* with a static second argument (the value 4), a new and possibly more efficient partially evaluated definition of *power* can be introduced:

$$power4\ x = x * x * x * x * 1$$

The *folding* technique described in the next section is used to replace the original applicative expression with a call to the new partially evaluated definition.

### 4.1.3 Folding

The inverse of the unfolding technique is called *folding*. Folding replaces an expression with a call to a function which usually has been introduced by the definition creation technique.

**Example**

$$f\ x = power\ x\ 4$$

This call to the *power* function can be replaced with a call to the more efficient *power4* function that was generated by declaration creation.

$$f\ x = power4\ x$$

### 4.1.4 Symbolic Evaluation

Typically not all inputs to a function are available when performing partial evaluation. This means that when the definition of a function is unfolded, not all occurrences of parameters in the body of the function can be replaced by values. Instead, the parameters of a function may been instantiated by variables, which we shall refer to as *symbolic values*. Unfortunately operational semantics for most programming languages are not sufficient for performing reductions in the presence of symbolic values.

*Symbolic evaluation* provides alternative operational semantics suited to properly deal with free variables. Obviously these alternative semantics resemble those of the programming language under consideration as closely as possible.

**Example**

The expression

$$(\lambda x\ y \rightarrow \textbf{if}\ x \equiv 0\ \textbf{then}\ y\ \textbf{else}\ y * y)\ 1\ z$$

can be symbolically evaluated to:

$$\textbf{if}\ 1 \equiv 0\ \textbf{then}\ z\ \textbf{else}\ z * z$$

which can subsequently be symbolically evaluated further to:

$$z * z$$

Symbolic evaluation halts here since the value of $z$ is necessary to perform further reductions.

Symbolic evaluation for a subset of Haskell is defined in section 4.2.

## 4.2 Symbolic Evaluation

Reduction strategies used to perform evaluation in traditional functional languages typically are not well-suited for reducing expressions containing free variables. Conventional reduction strategies define a *redex* to be an expression matching the left hand side of a reduction rule or definition. However, in the presence of free variables, a redex by this definition is not necessarily reducible. This is because a redex containing a free variable can not be reduced if the value if the variable is required. Symbolic evaluation employs a reduction strategy which will not attempt to reduce expressions when the value of free variables are needed.

To be more precise about the operational semantics of symbolic evaluation, we shall define the evaluation relation $\Rightarrow$. Unfortunately it is not feasible to define $\Rightarrow$ for the Haskell language as Haskell is too large to make a formal treatment useful. For the sake of clarity and brevity we shall constrain ourselves to a limited subset of the Haskell language in which a program consists of a number of declarations of the following form:

$$\begin{aligned} D ::= & \textbf{data}\ C\ V_1 \ldots V_n = C_1\ V_{11} \ldots V_{1k_1} \mid \cdots \mid C_n\ V_{n1} \ldots V_{nk_n} \\ & \mid V = E \end{aligned}$$

$$
\begin{aligned}
E ::= {} & V \mid (C\ E_1 \ldots E_n) \mid (E_1, \ldots, E_n) \\
& \mid (\lambda V \rightarrow E) \mid (E_1\ E_2) \\
& \mid \textbf{case}\ E\ \textbf{of}\ P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n
\end{aligned}
$$

$$P ::= \_ \mid V \mid C\ P_1 \ldots P_n \mid (P_1, \ldots, P_n)$$

Note that $C$ and $V$ resemble constants and variables respectively and that a sequence such as $E_1 \ldots E_n$ may also denote an empty sequence (i.e. $n \geqslant 0$). Furthermore, $(\lambda V_1 \ldots V_n \rightarrow E)$ may be used as an abbreviation for $(\lambda V_1\ (\ldots (\lambda V_n \rightarrow E)))$.

The $\Rightarrow$ relation is of the following form:

$$e_1 \Rightarrow e_2$$

where $e_1$ and $e_2$ are expressions. The $\Rightarrow$ relation is defined by the rules below:

$$\frac{}{V \Rightarrow V}\ \text{SE-Var1} \qquad \text{(if there is no declaration } V = E\text{)}$$

$$\frac{E_1 \Rightarrow E_2}{V \Rightarrow E_2}\ \text{SE-Var2} \qquad \text{(if there is a declaration } V = E_1\text{)}$$

$$\frac{E_1 \Rightarrow E_2}{(\lambda V \rightarrow E_1) \Rightarrow (\lambda V \rightarrow E_2)}\ \text{SE-Abs}$$

$$\frac{E_{11} \Rightarrow E_{12} \ldots E_{n1} \Rightarrow E_{n2}}{C\ E_{11} \ldots E_{n1} \Rightarrow C\ E_{12} \ldots E_{n2}}\ \text{SE-Con}$$

$$\frac{E_{11} \Rightarrow E_{12} \ldots E_{n1} \Rightarrow E_{n2}}{(E_{11}, \ldots, E_{n1}) \Rightarrow (E_{12}, \ldots, E_{n2})}\ \text{SE-Prod}$$

$$\frac{E_1 \Rightarrow (\lambda V \rightarrow E'_1) \qquad E_2 \Rightarrow E'_2 \qquad E'_1\,[E'_2\,/\,V] \Rightarrow E_3}{(E_1\ E_2) \Rightarrow E_3}\ \text{SE-App}$$

$$\frac{E \Rightarrow E' \qquad \langle P_1, \ldots, P_n \rangle \triangleq E'}{\textbf{case}\ E\ \textbf{of}\ P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n \Rightarrow E_i\ [\langle P_1, \ldots, P_n \rangle \triangle E']}\ \text{SE-Case1}$$

$$\frac{E \Rightarrow E' \qquad \langle P_1, \ldots, P_n \rangle \not\triangleq E' \qquad E_i \Rightarrow E'_i}{\textbf{case}\ E\ \textbf{of}\ P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n \Rightarrow \textbf{case}\ E'\ \textbf{of}\ P_1 \rightarrow E'_1; \ldots; P_n \rightarrow E'_n}\ \text{SE-Case2}$$

In these rules the simultaneous substitution of $e_1$ for $x_1$, ..., $e_k$ for $x_k$ in $e$ is written as $e\,[e_1\,/\,x_1, \ldots, e_k\,/\,x_k]$. Note that $x_1, \ldots, x_k$ are assumed to be distinct.

From the SE-App rule we can see that the argument in a function application is evaluated first before it is substituted in the body of the $\lambda$-abstraction. This means that these rules implement a strict evaluation order. Since Haskell is non-strict, it is of particular importance that the type of symbolic evaluation defined here is only applied to expressions of which is known that they can safely be evaluated strictly.

The SE-Case rule uses two operators which have not yet been defined: $\triangleq$ and $\triangle$. The relation $\triangleq$ expresses that one of the patterns in the sequence of patterns on the left hand side matches the expression on the right hand side. If a pattern that matches contains variables, as a result of the match the variables

will be bound to expressions. If $\langle P_1, \ldots, P_n \rangle \triangleq E$ then $\langle P_1, \ldots, P_n \rangle \triangle E$ is a simultaneous substitution of these pattern-variable bindings. The definitions of $\triangleq$ and $\triangle$ are given in section 4.3.

**Example**

$$\textbf{case } (1,2) \textbf{ of } (x, y) \rightarrow f\ x\ y$$

Intuitively it is clear that $\langle (x, y) \rangle \triangleq (1, 2)$. Evaluating this expression will cause the variables $x$ and $y$ in the expression $f\ x\ y$ to be substituted with expressions 1 and 2 respectively.

$$(f\ x\ y)\ [\langle (x, y) \rangle \triangle (1, 2)] = (f\ x\ y)\ [1\ /\ x, 2\ /\ y] = f\ 1\ 2$$

## 4.3 Pattern Matching with Symbolic Values

Pattern matching is used in case expressions for both binding variables to expressions and selecting an expression from a number of alternatives. As was mentioned in the previous section, symbolic evaluation cannot continue when the value of a free variable is required in order to make further reductions. This usually occurs during the evaluation of case expressions because an alternative expression is typically selected based on the value of some variable. For this reason, a clever pattern matching mechanism can greatly improve the power and potential of symbolic evaluation.

### 4.3.1 Traditional Pattern Matching

Traditional Haskell pattern matching of course does not allow patterns to be matched against variables. We must therefore extend the pattern matching semantics to handle matching against free variables optimally. The following example illustrates the problem:

**Example**

A variant of the *power* function that was introduced in section 4.1 can be defined in our limited Haskell subset as follows:

$$power = (\lambda z \rightarrow \textbf{case } z \textbf{ of } (x, 0) \rightarrow 1; (x, n) \rightarrow x * power\ (x, n - 1))$$

Given this definition of *power*, the evaluation of the expression $power\ (a, 0)$ where $a$ is a free variable, should cause the first alternative of the case expression to be selected upon symbolic evaluation. This means that the pattern matching semantics should allow pattern-variable $x$ to match against free variable $a$. Although this case is rather uncomplicated, as illustrated by the following examples, the problem in general is not trivial.

**Example**

The *Either* data type and *either* function are introduced by the following declarations:

$$\begin{aligned}&\textbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b\\ &\mathit{either} = (\lambda lf \rightarrow (\lambda rf \rightarrow (\lambda e \rightarrow \textbf{case}\ e\ \textbf{of}\ (\mathit{Left}\ x) \rightarrow lf\ x\\ &\qquad\qquad\qquad\qquad\qquad\qquad ;(\mathit{Right}\ y) \rightarrow rf\ y)))\end{aligned}$$

Symbolic evaluation of the expression *either* (*power* 3) ($\lambda x \rightarrow$ *power* $x$ 3) $e$, where $e$ is an unknown variable, is more difficult since it is unknown whether $e$ is a *Left* or a *Right* value. It is therefore not possible to select an alternative of the case expression.

**Example**

In the following declarations a function is defined which determines equivalence of *Bool* values:

$$\begin{aligned}&\textbf{data}\ \mathit{Bool} = \mathit{True} \mid \mathit{False}\\ &\mathit{equivBool} = (\lambda x \rightarrow (\lambda y \rightarrow \textbf{case}\ (x, y)\ \textbf{of}\ (\mathit{False}, \mathit{False}) \rightarrow \mathit{True}\\ &\qquad\qquad\qquad\qquad\qquad\qquad ;(\mathit{True}, \mathit{True}) \rightarrow \mathit{True}\\ &\qquad\qquad\qquad\qquad\qquad\qquad ;(\_, \_) \rightarrow \mathit{False}))\end{aligned}$$

As will be defined more precisely in the next section, the wildcard pattern, _, matches any expression without introducing any bindings. When the expression *equivBool b True* is partially evaluated, where $b$ is an unknown *Bool* value, the variable $b$ can *not* be matched against the wildcard pattern for if the value of $b$ had been known, a different match might have resulted.

It is clear that pattern matching with unknown variables is more involved than traditional pattern matching because it is necessary to consider matching-results to previous patterns as well. In the next section the semantics of pattern matching with symbolic values will be given.

### 4.3.2 Extended Pattern Matching Semantics

Pattern matching is about matching patterns against expressions. When these expressions do not contain free variables, the pattern matching semantics presented here are not different from traditional pattern matching. However, when free variables do enter the picture we must be particularly wary when matching values against variable patterns and wildcard patterns.

Unlike traditional pattern matching, when matching a pattern against an expression, it is necessary to consider what other patterns an expression has already been matched with and how many patterns will be matched with later on. We therefore define pattern matching for *pattern sequences* from which at any time exactly one pattern is called the *active pattern*. The active pattern is the pattern against which a value is being matched.

Extended pattern matching for a pattern sequence $\langle P_1, \ldots, P_n \rangle$ can either succeed or fail. Failure will result if no pattern in the sequence matches with the given expression. When pattern matching on the sequence succeeds, the first pattern in the sequence that matches the given value is called the *matching pattern*.

Pattern matching with free variables for a single pattern $p$ in a pattern sequence may either *succeed*, *fail possibly* or *fail necessarily*. The distinction in failure stems from the fact that when a pattern is matched against a value $e$ containing free variables, it may be possible to conclude from the information available, that the pattern can never match $e$ or that the pattern could possibly not match $e$. This difference is illustrated by the following example.

**Example**

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$
$$\mathit{fromMaybe} = (\lambda d \rightarrow (\lambda m \rightarrow \textbf{case}\ m\ \textbf{of}\ \mathit{Nothing} \rightarrow d; (\mathit{Just}\ a) \rightarrow a))$$

In the expression *fromMaybe* 0 $x$ where $x$ is a free variable, $x$ will be matched against pattern *Nothing* as well as pattern (*Just* $a$) in order to determine which alternative of the case expression should be selected. Because nothing is known about the value of $x$, the matching of $x$ against both patterns will result in a possible failure.

Consider on the other hand the expression *fromMaybe* 0 (*Just* $x$) which also contains a free variable $x$. In this case the expression (*Just* $x$) shall be matched to both *Nothing* and (*Just* $a$). This time the first match will fail necessarily because no matter the value of $x$, the pattern *Nothing* will never match (*Just* $x$). As will become clear later on, when expression (*Just* $x$) is matched against pattern (*Just* $a$) the match will succeed in this case.

Upon a succesful match, the result of pattern matching for both single patterns and pattern sequences is a set of bindings in which every pattern-variable occurring in the matching pattern is bound to an expression.

Matching a pattern sequence $\langle P_1, \ldots, P_n \rangle$ with active pattern $P_i$ to an expression $e$ possibly containing free variables follows the following set of rules. It is important to note that applicative order reduction is assumed, which means that expressions that are matched against are assumed to be in normal form.

1. When $P_i$ is a pattern-variable, matching succeeds only if no match has *failed possibly* against this pattern before. If there *has* been at least one previous possibly failed match against this pattern, matching *fails possibly* as well. Upon a succesful match, $P_i$ is bound to $e$.

2. When $P_i$ is a wildcard pattern, matching succeeds only if no match has *failed possibly* against this pattern before. If there *has* been at least one previous possibly failed match against this pattern, matching *fails possibly* as well. No bindings will result from a succesful match.

3. When the expression, $e$, that is being matched is a variable, matching succeeds only if the pattern sequence consists of a single pattern. If matching succeeds, $P_i$ is bound to $e$, otherwise matching fails possibly.

4. When $P_i$ is of the form $C\ P'_1 \ldots P'_m$ where $C$ is a constructor, matching succeeds only if $e$ is of the form $C\ E_1 \ldots E_m$ and all of $P'_1 \ldots P'_m$ sucessfully match $E_1 \ldots E_m$. If any of $P'_1 \ldots P'_m$ results in an unsuccesful match, matching will fail necessarily.

5. When $P_i$ is another kind of pattern, the traditional pattern matching rules apply where failure should be interpreted as necessarily failure. Traditional Haskell pattern matching rules can be found in [PJ03].

We can now give definitions of the pattern matching operators that were introduced informally in the previous section:

$$\langle P_1, \ldots, P_n \rangle \triangleq E \quad = \quad (\exists i : 1 \leqslant i \leqslant n \wedge P_i \text{ matches } E)$$

Given a sequence of patterns $P_1, \ldots, P_n$ and an expression $E$. Let $P_i$ be the *first* pattern in the sequence that matches $E$ and let the resulting set of bindings be $[e_1 / x_1, \ldots, e_k / x_k]$. Then:
$\langle P_1, \ldots, P_n \rangle \triangle E \quad = \quad E\,[e_1 / x_1, \ldots, e_k / x_k]$

## 4.4 Specializations

Now that we have constrained ourselves to a limited subset of Haskell, it has become feasible to examine the effects of applying partial evaluation to the definitions generated by the compiler for specializations of generic functions.

### 4.4.1 Modelling Specializations

The specialization $poly\{\!|T|\!\}$ of a generic function $poly$ for a type $T$ given by:

$$\textbf{data}\ T\ u_1 \ldots u_k = K_1\ t_{11} \ldots t_{1k_1} \mid \cdots \mid K_n\ t_{n1} \ldots t_{nk_n}$$

will cause the Generic Haskell compiler to produce a number of Haskell definitions [1]. In order to show the results of partial evaluation on compiled specializations, it is convenient to model the definitions produced in the limited Haskell subset that was given in section 4.2.

The main definition resulting from a specialization $poly\{\!|T|\!\}$ is named $poly_T$ and has the following general form:

$$poly_T = (\lambda u_1 \ldots u_k \rightarrow to\ (bimap_{(\rightarrow)}\ ep_T\ ep_T)\ (poly_{T^\circ}\ u_1 \ldots u_k))$$

The definition for $poly_T$ makes use of the specialization of $poly$ for the structure type of $T$, $poly_{T^\circ}$. The structure type of $T$ is introduced by the following type synonyms:

$$\textbf{type}\ T^\circ\ u_1 \ldots u_k = Sum\ (K_1\ u_1 \ldots u_k) \ldots (Sum \ldots (K_n\ u_1 \ldots u_k))$$

And for every constructor $K_i$ of $T$:

$$\textbf{type}\ K_i\ u_1 \ldots u_k = T\ u_1 \ldots u_k$$

The definition of $poly_{T^\circ}$ which operates on values of $T^\circ$ is as follows:

$$poly_{T^\circ} = (\lambda u_1 \ldots u_k \rightarrow poly_{Sum}\ (poly_{K_1}\ u_1 \ldots u_k)\ (poly_{Sum} \ldots (poly_{K_n}\ u_1 \ldots u_k)))$$

Note that when $T$ has just a single constructor, the $Sum$ type constructor can not be used in the structure type $T^\circ$ and the $Sum$ data constructor can not be used in the definition of $poly_{T^\circ}$. In such cases :

$$\textbf{type}\ T^\circ = K_1$$
$$poly_{T^\circ} = (\lambda u_1 \ldots u_k \rightarrow poly_{K_1}\ u_1 \ldots u_k)$$

For every constructor $K_i$ of $T$ a function $poly_{K_i}$ and a type synonym $K_i^\circ$ exist:

$$poly_{K_i} = (\lambda u_1 \ldots u_k \rightarrow to\ (bimap_{(\rightarrow)}\ ep_{K_i}\ ep_{K_i})\ (poly_{K_i^\circ}\ u_1 \ldots u_k))$$

For the form of the type synonym $K_i^\circ$ we can distinguish three cases:

[1] Details of the translation process were discussed in chapter 3

a. If $k_i = 0$:

$$\textbf{type}\ K_i^\circ\ u_1 \ldots u_k = Unit$$

b. If $k_i = 1$:

$$\textbf{type}\ K_i^\circ\ u_1 \ldots u_k = t_{i1}$$

c. If $k_i > 1$:

$$\textbf{type}\ K_i^\circ\ u_1 \ldots u_k = Prod\ t_{i1} \ldots (Prod \ldots t_{ik_i})$$

Furthermore, for every constructor $K_i$ of $T$ there is a definition for $poly_{K_i^\circ}$ which is based on the form of type synonym $K_i^\circ$.

$$poly_{K_i^\circ} = (\lambda u_1 \ldots u_k \rightarrow poly_{Prod}\ \sigma(t_{i1}) \ldots (poly_{Prod} \ldots \sigma(t_{ik_i})))$$

where $\sigma(\tau)$ is defined by:

$$\sigma(\tau) = \begin{cases} \tau, & \text{if for a certain } j,\ \tau = u_j\ (\text{for } 1 \leqslant j \leqslant k) \\ poly_C\ \sigma(\tau_1) \ldots \sigma(\tau_j), & \text{if } \tau = C\ \tau_1 \ldots \tau_j \end{cases}$$

Note that when a constructor $K_i$ has no parameters (i.e. $k_i = 0$), the definition of the corresponding $poly_{K_i^\circ}$ equals $poly_{Unit}$. When a constructor takes just a single parameter (i.e. $k_i = 1$) it is not necessary (or possible) to use $poly_{Prod}$. In this case $poly_{K_i^\circ}$ equals $(\lambda u_1 \ldots u_k \rightarrow \sigma(t_{i1}))$.

**Example**

Specializing *gmap* for the *Either* data type will produce the following function declarations and type synonyms:

$$\begin{aligned}
&\textbf{data}\ Either\ a\ b = Left\ a \mid Right\ b \\
&gmap_{Either} = (\lambda a\ b \rightarrow to\ (bimap_{(\rightarrow)}\ ep_{Either}\ ep_{Either})\ gmap_{Either^\circ}) \\
&gmap_{Either^\circ} = (\lambda a\ b \rightarrow gmap_{Sum}\ (gmap_{Left}\ a\ b)\ (gmap_{Right}\ a\ b)) \\
&gmap_{Left} = (\lambda a\ b \rightarrow to\ (bimap_{(\rightarrow)}\ ep_{Left}\ ep_{Left})\ gmap_{Left^\circ}) \\
&gmap_{Left^\circ} = (\lambda a\ b \rightarrow a) \\
&gmap_{Right} = (\lambda a\ b \rightarrow to\ (bimap_{(\rightarrow)}\ ep_{Right}\ ep_{Right})\ gmap_{Right^\circ}) \\
&gmap_{Right^\circ} = (\lambda a\ b \rightarrow b)
\end{aligned}$$

The following definition for $gmap_{Sum}$ can be translated directly from the original definition of the *gmap* function:

$$\begin{aligned}
gmap_{Sum} = (\lambda gmapA\ gmapB\ x \rightarrow \textbf{case}\ x\ \textbf{of}\ &(Inl\ a) \rightarrow Inl\ (gmapA\ a) \\
&(Inr\ b) \rightarrow Inr\ (gmapB\ b))
\end{aligned}$$

What is left now is the definition of $bimap_{(\rightarrow)}$ and the embedding-projection pairs that are used to convert values to structural representation and back. For the conversion of values from type $T$ and structure type $T^\circ$ the embedding-projection pair $ep_T$ is used. This conversion involves wrapping a value into a *Sum*-value. The conversion of values of a type $K_i$ to values of type $K_i^\circ$ on the other hand, involves wrapping a given value into a *Prod*-value. This is done using an $ep_{K_i}$ embedding-projection pair. The $ep_T$ embedding-projection pair is declared by:

$$
\begin{aligned}
ep_T = EP\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ & (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
& \ldots \\
& (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
& \ldots \\
& (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x))) \\
(\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ & (Inl\ v) \rightarrow v \\
& \ldots \\
& (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
& \ldots \\
& (Inr\ (\ldots (Inr\ v))) \rightarrow v)
\end{aligned}
$$

and $ep_{K_i}$ for a constructor $K_i$ of $T$:

$$
\begin{aligned}
ep_{K_i} = EP\ & (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (K_i\ v_1 \ldots v_{k_i}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_i})) \\
& (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_i})) \rightarrow K_i\ v_1 \ldots v_{k_i})
\end{aligned}
$$

The definition of $bimap_{(\rightarrow)}$ is given by:

$$
\begin{aligned}
bimap_{(\rightarrow)} = (\lambda x\ y \rightarrow \textbf{case}\ x\ \textbf{of}\ & (EP\ a2b\ b2a) \rightarrow \\
& \textbf{case}\ y\ \textbf{of}\ (EP\ c2d\ d2c) \rightarrow \\
& \quad EP\ (\lambda a2c \rightarrow c2d \cdot a2c \cdot b2a) \\
& \qquad (\lambda b2d \rightarrow d2c \cdot b2d \cdot a2b))
\end{aligned}
$$

The definitions of *from* and *to* are:

$$
\begin{aligned}
from &= (\lambda ep \rightarrow \textbf{case}\ ep\ \textbf{of}\ (EP\ a\ b) \rightarrow a) \\
to &= (\lambda ep \rightarrow \textbf{case}\ ep\ \textbf{of}\ (EP\ a\ b) \rightarrow b)
\end{aligned}
$$

For completeness we also give a definition of the $(\cdot)$ function composition operator:

$$f \cdot g = (\lambda x \rightarrow f\ (g\ x))$$

### 4.4.2 Partially Evaluating Specializations

In order to optimize the code resulting from the specialization $poly\{\!|T|\!\}$ of a generic function *poly* for a data type $T$, the generated definition $poly_T$ can be partially evaluated using the symbolic evaluation semantics that we have introduced in section 4.2. Ideally this would produce a semantically equivalent definition in which no superfluous conversions are made. However, as will become clear in this section, more work will be necessary after partial evaluation has completed.

Partially evaluating the definition of $poly_T$ given in the previous subsection will first inline the definitions of *to*, $bimap_{(\rightarrow)}$, $ep_T$ and $poly_{T^\circ}$ and will subsequently perform symbolic evaluation. In our presentation we shall follow the partial evaluation process as closely as possible. However, to conserve space and to improve legibility of expressions, we shall occasionally use an alternative evaluation order or not inline certain definitions right away. These measures will not affect the result of partial evaluation.

The results of performing partial evaluation are presented in appendix A. Section A.1 shows how partial evaluation is performed on the definition of $poly_T$. The definition of $poly_{K_i}$ is partially evaluated in section A.2. Since $poly_T$ and $poly_{K_i}$ have a similar structure, the first number of evaluation steps are the

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k\ x \rightarrow \\
\quad \textbf{case}\ (poly_{Sum} \\
\quad\quad (\lambda x \rightarrow \textbf{case}\ ((poly_{Prod}\ \sigma(t_{11}) \ldots (poly_{Prod} \ldots \sigma(t_{1k_1}))) \\
\quad\quad\quad\quad\quad\quad\quad\quad (\textbf{case}\ x\ \textbf{of}\ (K_1\ v_1 \ldots v_{k_1}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_1}))) \\
\quad\quad\quad\quad\quad \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_1})) \rightarrow K_1\ v_1 \ldots v_{k_1}) \\
\quad\quad (poly_{Sum} \ldots (\lambda x \rightarrow \textbf{case}\ ((poly_{Prod}\ \sigma(t_{n1}) \ldots (poly_{Prod} \ldots \sigma(t_{nk_n}))) \\
\quad\quad\quad\quad\quad\quad\quad\quad (\textbf{case}\ x\ \textbf{of}\ (K_n\ v_1 \ldots v_{k_n}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_n}))) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_n})) \rightarrow K_n\ v_1 \ldots v_{k_n})) \\
\quad\quad (\textbf{case}\ x\ \textbf{of}\ (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x)))) \\
\quad\quad \textbf{of}\ (Inl\ v) \rightarrow v \\
\quad\quad\quad \ldots \\
\quad\quad\quad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\quad\quad\quad \ldots \\
\quad\quad\quad (Inr\ (\ldots (Inr\ v))) \rightarrow v)
\end{array}
$$

Figure 4.1: Final result of partially evaluating $poly_T$.

same for both definitions. For this reason some intermediate results have been omitted in section A.2. Figure 4.1 contains the final result of section A.1 that is obtained by partially evaluating $poly_T$.

The partially evaluated definition of $poly_T$ presented here still contains references to specializations $poly_{Sum}$ and $poly_{Prod}$. We will assume that the generic definition of *poly* contains components for structure type constructors *Sum* and *Prod*. Consequently the specializations $poly_{Sum}$ and $poly_{Prod}$ will not require much effort as the bodies of components in the generic definition can be copied directly. To avoid cluttering the new definition of $poly_T$ these definitions have not been inlined.

It is obvious from looking at figure 4.1 that the structural conversions have not yet been eliminated. As was already announced, the partial evaluation techniques described in this chapter are not sufficient for solving the problem. In chapter 6 we will see how the transformations that are defined in the next chapter can be used to make further optimizations.

# Chapter 5

# Case Transformations

The partial evaluation techniques discussed in the previous chapter by themselves are inadequate to achieve the desired results. Often, the symbolic evaluator is unable to perform reductions to case expressions that could still be optimized further if one or more transformations would be carried out first.

The results of partial evaluation will be even worse if too many definitions are unfolded into expressions that can not be fully optimized directly by symbolic evaluation. Uncontrolled unfolding in such case can cause expressions to grow to undesirable lengths.

This chapter discusses transformations on case expressions that are used to create more opportunities for symbolic evaluation. As we shall see later on in chapter 6, symbolic evaluation combined with a number of these transformations is a good recipe for optimizing the specializations described in section 4.4.

Most transformations in this chapter or variations of these transformations also appear in [San95].

## 5.1 Case Terminology

The general form of a case expression is as follows:

$$\text{case } E \text{ of } \{ P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n \}$$

where $E$, $E_1$ ... $E_n$ are expressions and $P_1$ ... $P_n$ are patterns. In style with [San95], we say that the case expression above *scrutinizes* expression $E$. Therefore $E$ shall be referred to as the *scrutinee* of the case expression. Each part $P_i \rightarrow E_i$ will be called an *alternative* of the case expression.

The case expressions that we will mostly be dealing with will scrutinize tuples. These expressions are thus of the following form:

$$\text{case } (E_{s_1}, \ldots, E_{s_k}) \text{ of } \{ P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n \}$$

The expressions $E_{s_1} \ldots E_{s_k}$ shall be called *expression components*, or *components* for short, of the case expression. For convenience if a case expression scrutinizes an expression that is not a tuple, this expression is considered to be the only component of the case expression. A case expression which scrutinizes the nullary tuple () is considered to have no components. In an analogous manner, the pattern of an alternative consists of *pattern components*.

We say a pattern component $P$ *corresponds* to an expression component $E$ when the pattern matching process that is used to select an alternative would attempt to match $E$ against $P$. This is illustrated by the following example:

**Example**

$$\textbf{case } (1,(2,3),x) \textbf{ of } \{(5,a,(b,c)) \rightarrow 1; (d,(e,f),g) \rightarrow 2; h \rightarrow 3\}$$

The following table shows what expression each pattern corresponds to:

| Pattern component | Corresponds to expression component |
|---|---|
| $5$ | $1$ |
| $a$ | $(2,3)$ |
| $(b,c)$ | $x$ |
| $d$ | $1$ |
| $(e,f)$ | $(2,3)$ |
| $g$ | $x$ |
| $h$ | $-$ |

Notice that a pattern component corresponds to at most one expression component and that multiple pattern components may correspond to the same expression component.

## 5.2 Case Elimination

Case expressions with just a single alternative $P \rightarrow E$ can be replaced by $E$ in which the variables bound by $P$ are substituted. An expression

$$\textbf{case } E_s \textbf{ of } \{P \rightarrow E\}$$

can be transformed into the following equivalent expression:

$$E\,[\langle P\rangle \vartriangle E_s]$$

This technique is called *case elimination* [San95].

**Example**

$$\textbf{case } (1,2) \textbf{ of } \{(x,y) \rightarrow f\ x\ y\}$$

can be reduced to:

$$f\ 1\ 2$$

Note that this transformation is only valid under the assumption that the case expression is *total*. Analogous to total functions, a *total case expression* provides a matching case for all constructors of a data type. This assumption is reasonable for our purpose since matching an unsupported constructor against a non-total case expression would normally produce a run-time error.

## 5.3 Case Cleanup

When the pattern components corresponding to an expression component do not bind variables and the expression component is not relevant in determining which alternative is to be selected, the case component and all corresponding pattern components may be removed.

**Example**

$$\textbf{case}\ (a, b, c)\ \textbf{of}\ \{(0, y, z) \rightarrow z; (x, y, z) \rightarrow x\}$$

The middle pattern components do not bind values that are used in any of the alternatives and the value of the second case component has no influence on the alternative selection process. The middle component can therefore safely be removed:

$$\textbf{case}\ (a, c)\ \textbf{of}\ \{(0, z) \rightarrow z; (x, z) \rightarrow x\}$$

To aid the detection of these "don't care components" and to increase the readability of expressions, pattern-variables are replaced by pattern-wildcards if the bound variables do not occur in the alternative expression body:

$$\textbf{case}\ (a, c)\ \textbf{of}\ \{(0, z) \rightarrow z; (x, \_) \rightarrow x\}$$

## 5.4 Case Floating

The symbolic evaluator will occassionally encounter case expressions that scrutinize expressions with free variables. The presence of free variables in the scrutinee of a case expression usually prevents the symbolic evaluator from choosing a matching alternative. This gives rise to applicative expressions of the following form:

$$(\textbf{case}\ E_f\ \textbf{of}\ \{P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n\})\ E_x$$

where $E_x$ is an expression and $E_f$ is an expression containing at least one free variable that is keeping the symbolic evaluator from choosing an alternative. To create more possibilities for further partial evaluation, which is often desirable, *case floating* [San95] can be used. Case floating transforms the expression above into:

$$\textbf{case}\ E_f\ \textbf{of}\ \{P_1 \rightarrow E_1\ E_x; \ldots; P_n \rightarrow E_n\ E_x\}$$

The downside of case floating is that the size of an expression may increase dramatically if $E_x$ is large. However, in our application subsequent partial evaluation usually reduces the alternatives of the resulting case expression to satisfactory lenghts.

**Example**

$$(\textbf{case}\ x\ \textbf{of}\ \{0 \rightarrow \mathit{const}\ 0; x \rightarrow \mathit{power}\ x\})\ 4$$

Since the value of $x$ is unknown an alternative of the case expression can not be selected. By applying the case floating transformation the case expression can be written as follows:

$$\textbf{case}\ x\ \textbf{of}\ \{0 \rightarrow \mathit{const}\ 0\ 4; x \rightarrow \mathit{power}\ n\ 4\}$$

The transformed case expression introduces opportunities for further partial evaluation.

## 5.5 Case of Case Transformation

In the process of partially evaluating generated specializations the symbolic evaluator frequently encounters applicative expressions which apply the composition of two functions to a free variable. Typically the bodies of these two functions will contain case expressions which scrutinize the function's argument. In general this type of expression has the following structure:

$$(\lambda x \rightarrow \textbf{case } x \textbf{ of } \{P_{11} \rightarrow E_{11}; \ldots; P_{1m} \rightarrow E_{1m}\})$$
$$((\lambda y \rightarrow \textbf{case } y \textbf{ of } \{P_{21} \rightarrow E_{21}; \ldots; P_{2n} \rightarrow E_{2n}\})\ z)$$

Since the value of $z$ is unknown, it is usually not possible to select a matching alternative of the inner case expression. Hence it is also not usually possible to select an alternative in the outer case expression. After symbolic evaluation of an expression displaying the structure given above, we are generally left with an expression of the following form:

$$\textbf{case } (\textbf{case } z \textbf{ of } \{P_{21} \rightarrow E_{21}; \ldots; P_{2n} \rightarrow E_{2n}\}) \textbf{ of } \{P_{11} \rightarrow E_{11}; \ldots; P_{1m} \rightarrow E_{1m}\}$$

This result is unsatisfactory because it does not provide new possibilities for performing further symbolic evaluation. It is possible to transform the above expression into:

$$\begin{aligned}\textbf{case } z \textbf{ of } \{ & P_{21} \rightarrow \textbf{case } E_{21} \textbf{ of } \{P_{11} \rightarrow E_{11}; \ldots; P_{1m} \rightarrow E_{1m}\}; \\ & \ldots \\ & P_{2n} \rightarrow \textbf{case } E_{2n} \textbf{ of } \{P_{11} \rightarrow E_{11}; \ldots; P_{1m} \rightarrow E_{1m}\} \\ \}\end{aligned}$$

This is the Case of Case transformation [San95]. The Case of Case transformation can thus be used to transform a case expression that scrutinizes another case expression into a semantically equivalent case expression that does not scrutinize case expressions. This creates new possibilities for further symbolic evaluation of the alternatives of the resulting case expression at the expense of possibly duplicating expressions. As shall become clear in chapter 6, this is less of an issue for our particular application.

**Example**

Let functions $f$ and $g$ be defined by:

$$f = (\lambda x \rightarrow \textbf{case } x \textbf{ of } \{\mathit{False} \rightarrow 0; \mathit{True} \rightarrow 1\})$$
$$g = (\lambda x \rightarrow \textbf{case } x \textbf{ of } \{A \rightarrow \mathit{True}; B \rightarrow \mathit{False}\})$$

The expression $f\ (g\ x)$ where $x$ is a free variable can be partially evaluated using the symbolic evaluation rules in section 4.2. This would result in the following case expression:

$$\textbf{case } (\textbf{case } x \textbf{ of } \{A \rightarrow \mathit{True}; B \rightarrow \mathit{False}\}) \textbf{ of } \{\mathit{False} \rightarrow 0; \mathit{True} \rightarrow 1\}$$

Applying the Case of Case transformation yields:

$$\begin{aligned}\textbf{case } x \textbf{ of } \{ & A \rightarrow \textbf{case } A \textbf{ of } \{A \rightarrow \mathit{True}; B \rightarrow \mathit{False}\}; \\ & B \rightarrow \textbf{case } B \textbf{ of } \{A \rightarrow \mathit{True}; B \rightarrow \mathit{False}\} \\ \}\end{aligned}$$

By applying symbolic evaluation this expression can be further optimized to an expression containing no nested case expressions:

$$\text{case } x \text{ of } \{A \rightarrow \mathit{True}; B \rightarrow \mathit{False}\}$$

## 5.6 Generalizing Case of Case

The Case of Case transformation as described in the previous section can be applied if the scrutinee of a case expression is also a case expression. However, if the scrutinee of a case expression consists of multiple components (i.e. the scrutinee is a tuple expression), the Case of Case transformation cannot be used to lift case expression components. A case expression that is a component of another case expression shall be referred to as a *case component*.

The form of a case expression with at least one case component is:

$$\begin{aligned}&\text{case } (Ex_1, \ldots, \text{case } (Ez_1, \ldots, Ez_k) \text{ of } \{P_{21} \rightarrow E_{21}; \ldots; P_{2n} \rightarrow E_{2n}\},\\ &\qquad \ldots, Ex_l) \text{ of } \{P_{11} \rightarrow E_{11}; \ldots; P_{1m} \rightarrow E_{1m}\}\end{aligned}$$

Note that this expression can be regarded as a generalization of the kind of expression that the original Case of Case transformation operates on. This suggests that the Case of Case transformation can also be generalized and that the above can be transformed to:

$$\begin{aligned}&\text{case } (Ez_1, \ldots, Ez_k) \text{ of}\\ &\quad \{P_{21} \rightarrow \text{case } (Ex_1, \ldots, E_{21}, \ldots, Ex_l) \text{ of } \{P_{11} \rightarrow E_{11}; \ldots; P_{1m} \rightarrow E_{1m}\}\\ &\quad \ldots\\ &\quad P_{2n} \rightarrow \text{case } (Ex_1, \ldots, E_{2n}, \ldots, Ex_l) \text{ of } \{P_{11} \rightarrow E_{11}; \ldots; P_{1m} \rightarrow E_{1m}\}\\ &\quad \}\end{aligned}$$

**Example**

Let us again consider the definition of *power* for our subset of Haskell:

$$\mathit{power} = (\lambda z \rightarrow \text{case } z \text{ of } (x, 0) \rightarrow 1; (x, n) \rightarrow x * \mathit{power}\ (x, n-1))$$

The expression $\mathit{power}\ (2, \mathit{power}\ (x, 4))$ with $x$ being a free variable may be partially evaluated to the following expression:

$$\begin{aligned}&\text{case } (2, \text{case } (x, 4) \text{ of } \{(x, 0) \rightarrow 1; (x, n) \rightarrow x * \mathit{power}\ (x, n-1)\})\\ &\qquad \text{of } \{(x, 0) \rightarrow 1; (x, n) \rightarrow x * \mathit{power}\ (x, n-1)\}\end{aligned}$$

Using the generalized Case of Case transformation this expression can be rewritten as:

$$\begin{aligned}&\text{case } (x, 4) \text{ of}\\ &\quad \{(x, 0) \rightarrow \text{case } (2, 1) \text{ of } \{(x, 0) \rightarrow 1; (x, n) \rightarrow x * \mathit{power}\ (x, n-1)\};\\ &\quad (x, n) \rightarrow \text{case } (2, x * \mathit{power}\ (x, n-1)) \text{ of}\\ &\qquad\qquad \{(x, 0) \rightarrow 1; (x, n) \rightarrow x * \mathit{power}\ (x, n-1)\};\\ &\quad \}\end{aligned}$$

Once again, the transformed expression can be further reduced by subsequent symbolic evaluation to:

$$\text{case } (2, x * x * x * x) \text{ of } \{(x, 0) \rightarrow 1; (x, n) \rightarrow x * \mathit{power}\ (x, n-1)\}$$

# Chapter 6

# Optimizing Specializations

In this chapter we return to the setting of section 4.4 where a definition $poly_T$ representing the specialization $poly\{\!|T|\!\}$ was partially evaluated. With the transformations that were introduced in the previous chapter at hand, we will attempt to optimize the partially evaluated definition of $poly\{\!|T|\!\}$ until it is no longer composed of structural conversions.

Section 6.1 examines the general format of partially evaluated specializations and shows how structural conversions can be eliminated. In section 6.2 an algorithm for optimizing compiled specializations is described.

This chapter inherits all definitions that were introduced in chapter 4.

## 6.1 Eliminating Structural Conversions

The last step in the partial evaluation process of $poly_T$ as presented in section A.1 shows a lengthy definition of $poly_T$. For clarity we back up one step and start off our endeavour with the second to last definition of $poly_T$ in which the definitions of $poly_{K_i}$ have not yet been inlined. This definition is also shown in figure 6.1. Figure 6.2 shows the partially evaluated definition of $poly_{K_i}$.

The format of both the definition of $poly_T$ and $poly_{K_i}$ can be characterized as follows:

$$poly_\tau = (\lambda x \rightarrow \textbf{case}\ (f\ (\textbf{case}\ x\ \textbf{of}\ \{\, alts_1 \,\}))\ \textbf{of}\ \{\, alts_2 \,\})$$

with $f$ given by:

$$f = (\lambda z \rightarrow body_f)$$

The list of alternatives $alts_1$ is used to convert value $x$ to an alternative representation. The alternatives of $alts_2$ are used to convert a value in the alternative representation back to the original representation. Function $f$ operates on values in the alternative representation and produces a result that is also in the alternative representation. The goal of optimization is to create an equivalent definition in which representational conversions have been removed. In order to accomplish this, expression $body_f$ will be rewritten.

It is important to realize that the constructors of a variable $z$ with a non-primitive type can only be accessed when $z$ is scrutinized inside a case expression. In the process of rewriting expression $body_f$ to eliminate representational

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k\ x \rightarrow \\
\quad \textbf{case}\ (poly_{Sum}\ (poly_{K_1}\ u_1 \ldots u_k)\ (poly_{Sum} \ldots (poly_{K_n}\ u_1 \ldots u_k))) \\
\qquad\qquad (\textbf{case}\ x\ \textbf{of}\ (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x)))) \\
\quad \textbf{of}\ (Inl\ v) \rightarrow v \\
\qquad \ldots \\
\qquad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\qquad \ldots \\
\qquad (Inr\ (\ldots (Inr\ v))) \rightarrow v
\end{array}
$$

Figure 6.1: Intermediate result of partially evaluating $poly_T$ (step (8) section A.1).

$$
\begin{array}{l}
poly_{K_i} = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\lambda x \rightarrow \textbf{case}\ ((poly_{Prod}\ \sigma(t_{i1}) \ldots (poly_{Prod} \ldots \sigma(t_{ik_i}))) \\
\qquad\qquad\qquad\qquad (\textbf{case}\ x\ \textbf{of}\ (K_i\ v_1 \ldots v_{k_i}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_i}))) \\
\qquad\qquad\qquad \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_i})) \rightarrow K_i\ v_1 \ldots v_{k_i}))
\end{array}
$$

Figure 6.2: Final result of partially evaluating $poly_{Ki}$ (section A.2).

conversions, we can therefore concentrate on rewriting case expressions in which formal parameter $z$ of $f$ appears as a scrutinee. However, $z$ may also occur in $body_f$ as an argument in an applicative expression. Any applicative expressions which involves $z$ will have to be evaluated symbolically and any resulting case expressions scrutinizing $z$ will have to be transformed. When the optimization process has completed, all scrutinizations of $z$ will have been altered to reflect that $z$ will have a different representation.

A case expression scrutinizing $z$ that is encountered in the process of symbolically evaluating $body_f$ will have the following form:

$$\textbf{case}\ (z)\ \textbf{of}\ \{P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n\}$$

Finding expressions of this kind is not difficult. Function $f$ is applied to argument (case $x$ of $\{alts_1\}$), which means the symbolic evaluation rule SE-App will replace any occurrence of $z$ in $body_f$ by (case $x$ of $\{alts_1\}$). This will result in the following type of expression which can not be evaluated any further:

$$\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{alts_1\})\ \textbf{of}\ \{P_1 \rightarrow E_1; \ldots; P_n \rightarrow E_n\}$$

After symbolic evaluation has halted, we can scan the resulting expression for case expressions scrutinizing a case expression and apply the Case of Case transformation which was described in section 5.6. The Case of Case transformation will merge alternatives $P_1 \ldots P_n$ with alternatives $alts_1$ which will create new opportunities for further partial evaluation. The combination of case transformation and partial evaluation will take care of the elimination of all represen-

tation conversions in $body_f$.

**Example**

Let the functions $f$, $g$ and $h$ be defined as:

$$\begin{aligned}
&\mathit{from} = (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ \{1 \rightarrow \mathit{One}; 2 \rightarrow \mathit{Two}\}) \\
&\mathit{to} = (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\}) \\
&f = (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ \{\mathit{One} \rightarrow \mathit{Two}; \mathit{Two} \rightarrow \mathit{One}\})
\end{aligned}$$

In the following expression $x$ is a free variable:

$$\mathit{to}\ (f\ (\mathit{from}\ x))$$

This expression can be partially evaluated to the following case expression:

$$\begin{aligned}
&\textbf{case}\ (f\ (\textbf{case}\ x\ \textbf{of}\ \{1 \rightarrow \mathit{One}; 2 \rightarrow \mathit{Two}\})) \\
&\quad \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\}
\end{aligned}$$

Note that we have purposely not inlined the definition of $f$ to show that its format matches the format discussed above. Inlining the definition of $f$ gives:

$$\begin{aligned}
&\textbf{case}\ (\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{1 \rightarrow \mathit{One}; 2 \rightarrow \mathit{Two}\}) \\
&\qquad\qquad \textbf{of}\ \{\mathit{One} \rightarrow \mathit{Two}; \mathit{Two} \rightarrow \mathit{One}\}) \\
&\quad \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\}
\end{aligned}$$

By applying the Case of Case transformation we obtain:

$$\begin{aligned}
&\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{1 \rightarrow \mathit{One}; 2 \rightarrow \mathit{Two}\}) \\
&\quad \textbf{of}\ \{\mathit{One} \rightarrow \textbf{case}\ \mathit{Two}\ \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\}; \\
&\qquad \mathit{Two} \rightarrow \textbf{case}\ \mathit{One}\ \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\} \\
&\qquad \}
\end{aligned}$$

Then by applying the Case of Case transformation a second time:

$$\begin{aligned}
&\textbf{case}\ x\ \textbf{of}\ \{1 \rightarrow \textbf{case}\ \mathit{One}\ \textbf{of}\ \{\mathit{One} \rightarrow \textbf{case}\ \mathit{Two}\ \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\}; \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Two} \rightarrow \textbf{case}\ \mathit{One}\ \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\} \\
&\qquad\qquad\qquad\qquad\qquad \}; \\
&\qquad\qquad 2 \rightarrow \textbf{case}\ \mathit{Two}\ \textbf{of}\ \{\mathit{One} \rightarrow \textbf{case}\ \mathit{Two}\ \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\}; \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Two} \rightarrow \textbf{case}\ \mathit{One}\ \textbf{of}\ \{\mathit{One} \rightarrow 1; \mathit{Two} \rightarrow 2\} \\
&\qquad\qquad\qquad\qquad\qquad \} \\
&\qquad\qquad \}
\end{aligned}$$

Using further symbolic evaluation (SE-Case1 and SE-App) we finally obtain an expression from which the representational conversions have been removed:

$$\begin{aligned}
&\textbf{case}\ x\ \textbf{of}\ \{1 \rightarrow 2; \\
&\qquad\qquad 2 \rightarrow 1 \\
&\qquad\qquad \}
\end{aligned}$$

The partially evaluated optimizations of $poly_T$ and $poly_{K_i}$ in figures 6.1 and 6.2 respectively are both instances of the general definition that we gave earlier:

$$poly_\tau = (\lambda x \rightarrow \textbf{case}\ (f\ (\textbf{case}\ x\ \textbf{of}\ \{\mathit{alts}_1\}))\ \textbf{of}\ \{\mathit{alts}_2\})$$

More accurately, for $poly_T$ function $f$ is given by:

$$f = poly_{Sum}\ (poly_{K_1}\ u_1 \ldots u_k)\ (poly_{Sum} \ldots (poly_{K_n}\ u_1 \ldots u_k))$$

and for $poly_{K_i}$ function $f$ is given by:

$$f = poly_{Prod}\ \sigma(t_{i1}) \ldots (poly_{Prod} \ldots \sigma(t_{ik_i}))$$

The definitions of $poly_{Sum}$, $poly_{K_i}$ and $poly_{Prod}$ and $\sigma(t_{ij})$ depend on the concrete definitions of $poly$ and $T$ and can therefore not be expanded.

## 6.2 Optimization Algorithm

From the previous section we may conclude that all tools necessary for optimizing compiled specializations of type-indexed values are available. However, in order to allow for automated optimization, an algorithm must be defined which describes when to apply which tool.

The optimization algorithm has the following inputs:

a. A definition $poly_T$ which is the compiled specialization that is to be optimized.

b. All other definitions that were generated by the Generic Haskell compiler along with $poly_T$.

A compiled specialization $poly_T$ can be optimized by performing the following sequence of steps:

1. Inline all available definitions in $poly_T$. Do not inline recursive calls.
2. Perform symbolic evaluation until no more reductions can be made (i.e. until a normal form has been reached).
3. Scan the partially evaluated definition of $poly_T$ for case expressions scrutinizing a case expression and apply the Case of Case transformation.
4. Repeat steps 2 and 3 until no more case expressions scrutinizing case expressions are found.

The optimization algorithm will produce an optimized version of $poly_T$ from which all structural conversions have been eliminated.

# Chapter 7

# Optimization in Practice

So far we have seen how compiled specializations can be modelled in a simplified version of Haskell and how partial evaluation defined for this simplified language combined with a number of transformations can be used to optimize these modelled specializations. In practice however, the optimization will have to be carried out on specializations which have been compiled to normal Haskell code. Although in essence the process of optimization is the same, the expressive richness of the Haskell language does make partial evaluation in particular more elaborate. More specifically, a partial evaluator for Haskell is a fully fledged Haskell interpreter which is also capable of performing symbolic evaluation.

The implementation of the optimizer for specializations in Haskell shall not be discussed here as its workings largely resemble the process that has been described in the previous chapters. This chapter however shall be mostly concerned with the *results* of the optimizer in practice. A brief description of the optimizer implementation shall be given in section 7.1. Section 7.2 gives several examples of generic functions applied to a number of data types. In section 7.3 the performance of optimized specializations is compared against their unoptimized equivalents.

## 7.1 Optimizer Implementation

The optimizer is implemented as the last phase of the Generic Haskell compiler. Since development is still in an experimental stage, the optimizer is not enabled by default but must be activated using the `--poptimize` command line flag. When the optimizer is enabled, a file with the extension `.optimized.hs` is written in addition to the regular `.hs` file.

## 7.2 Examples

Figure 7.1 shows the context in which the optimizer operates. Internally the optimizer consists of a Haskell partial evaluator and a number of transformations that will help the partial evaluator make as many reductions as possible. Since the partial evaluator is also a regular Haskell interpreter, some of the standard Haskell Prelude definitions must be loaded to be able to make reductions. The file `OptimizerPrelude.hs` is located in the library directory of the Generic

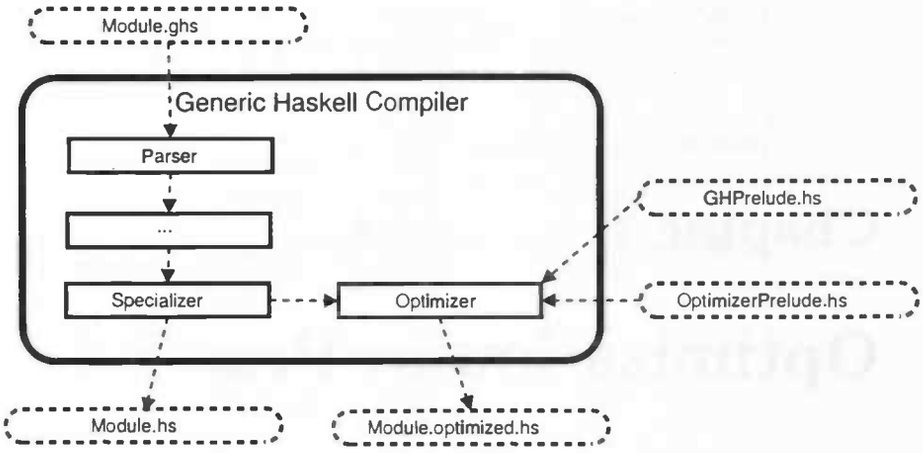


Figure 7.1: Optimizer displayed in its context.

Haskell compiler and is loaded by the optimizer before any reductions are performed. Unlike the regular Haskell Prelude, the Optimizer Prelude contains just a limited number of functions which are important for the partial evaluation of specializations.

As we have seen in chapter 3 compiled specializations make heavy use of structure types and the generic *bimap* function. Specialized versions of *bimap* as well as declarations for the type constructors used to build structure types are available in the Generic Haskell Prelude, `GHPrelude.hs`. Normally the Generic Haskell Prelude is loaded by the Haskell compiler or interpreter that is used to compile or execute a compiled Generic Haskell program. This means our optimizer must also load the `GHPrelude.hs` file before starting reductions.

Since any Haskell program is also a Generic Haskell program, with the optimizer enabled the Generic Haskell compiler can also be used as a general partial evaluator for Haskell. However, one must keep in mind that the partial evaluator has been tailored for evaluating specializations and that the Optimizer Prelude does not contain many definitions which are normally found in the Haskell Prelude. Also, by default only Generic Haskell specializations (i.e. declarations with names prefixed by *ghs_*) are optimized. This means all other declarations are normally left in their original form.

The *gmap* function specialized for the list type constructor [] should clearly be equivalent to the well-known *map* function in Haskell. Although this fact is not immediately apparent from definitions that are originally generated by the compiler, it is not difficult to recognize the structure of Haskell's *map* in the optimized version of $gmap\{\![[]]\!\}$ shown below [1].

$$
\begin{array}{l}
ghs\_gmap\_\ []\ ghs\_gmap\_a = (\lambda x \rightarrow \textbf{case}\ (x)\ \textbf{of} \\
\quad \{([]) \rightarrow []; \\
\quad\ (gh\_A@(\_ : \_)) \rightarrow \\
\qquad\qquad ((:)\ (ghs\_gmap\_a\ ((\lambda((a : b)) \rightarrow a)\ gh\_A)) \\
\qquad\qquad\quad ((ghs\_gmap\_\ []\ ghs\_gmap\_a)\ ((\lambda((a : b)) \rightarrow b)\ gh\_A)))
\end{array}
$$

[1] For clarity some of the variables used in generated definitions in this chapter have been renamed

```
  })
```

Expressions generated by the optimizer can typically easily be optimized further because the optimizer will only attempt to reduce the overhead that stems from the process of specialization. This keeps the process of optimization as simple as possible and is not detrimental to the performance of optimized code because most Haskell compilers are equipped with excellent optimizers capable of removing the overhead introduced by the Generic Haskell optimizer.

For a more interesting example we can specialize *gmap* for a *nested data type* [BM98]: a data type whose declaration contains different instances of its type parameters. For example, the *Bush* data type:

```
data Bush a = Nil | Cons a (Bush (Bush a))
```

Optimizing the Haskell code generated for the specialization *gmap*{|*Bush*|} yields the following result:

```
ghs_gmap_Bush ghs_gmap_a = (λx → case (x) of
  {(Nil) → Nil;
   (gh_A@(Cons _ _)) →
       (Cons (ghs_gmap_a ((λ((Cons a b)) → a) gh_A))
             ((ghs_gmap_Bush (ghs_gmap_Bush ghs_gmap_a))
                             (λ((Cons a b)) → b) gh_A))
  })
```

It is not difficult to verify that this function indeed meets the requirements of a specialization of *gmap* for data type *Bush*.

Specializing the generic equality function, *eq*, for the *Bush* data type after optimization yields:

```
ghs_eq_Bush ghs_eq_a = (λx → (λx' → (case (x, x') of
  {(Nil, Nil) → True;
   (Nil, (Cons _ _)) → False;
   ((Cons _ _), Nil) → False;
   (gh_A'@(Cons _ _), gh_A@(Cons _ _)) →
    (∧) (ghs_eq_a ((λ((Cons a b)) → a) gh_A') ((λ((Cons a b)) → a) gh_A))
        ((ghs_eq_Bush (ghs_eq_Bush ghs_eq_a))
                      ((λ((Cons gh_a gh_b)) → gh_b) gh_A')
                      ((λ((Cons gh_a gh_b)) → gh_b) gh_A))
  })))
```

The generic equivalent of the *foldr* Haskell function is the *rreduce* function defined below:

```
type RReduce{[*]}t b = t → b → b
type RReduce{[k → l]}t b = forall u .
                           RReduce{[k]}u b → RReduce{[l]}(t u) b
rreduce{| t :: k |} :: RReduce{[k]}t b
rreduce{| Unit |}x e = e
rreduce{| Int |}n e = e
rreduce{| :+: |}fA fB (Inl a) e = fA a e
rreduce{| :+: |}fA fB (Inr b) e = fB b e
rreduce{| :*: |}fA fB (a:*:b) e = fA a (fB b e)
```

| Function Used | Reductions | Memory Cells |
|---|---|---|
| Unoptimized *gmap*⦃[ ]⦄ | 50057 | 118017 |
| Optimized *gmap*⦃[ ]⦄ | 15020 | 24927 |
| Haskell *map* | 11020 | 20927 |
| Unoptimized *gmap*⦃*Bush*⦄ | 61897 | 110236 |
| Optimized *gmap*⦃*Bush*⦄ | 39288 | 91834 |
| Haskell *mapBush* | 33931 | 86988 |
| Unoptimized *eq*⦃*Bush*⦄ | 39174 | 53102 |
| Optimized *eq*⦃*Bush*⦄ | 11241 | 16359 |
| Haskell *eqBush* | 2058 | 4115 |
| Unoptimized *rreduce*⦃*Tree*⦄ | 34244 | 94032 |
| Optimized *rreduce*⦃*Tree*⦄ | 4863 | 7422 |
| Haskell *foldrTree* | 2565 | 4614 |

Figure 7.2: Run-time statistics showing the results of optimization.

Specializing *rreduce* for the *Tree* data type (as defined on page 3) yields the following optimized result:

$$
\begin{array}{l}
ghs\_rreduce\_Tree\ ghs\_rreduce\_a = (\lambda x \rightarrow (\lambda x' \rightarrow (\mathbf{case}\ (x)\ \mathbf{of} \\
\ \{(Leaf) \rightarrow x'; \\
\ \ (gh\_A@(Node\ \_\ \_\ \_)) \rightarrow (ghs\_rreduce\_a\ ((\lambda((Node\ a\ b\ c)) \rightarrow a)\ gh\_A) \\
\ \ \ \ ((ghs\_rreduce\_Tree\ ghs\_rreduce\_a)\ ((\lambda((Node\ a\ b\ c)) \rightarrow b)\ gh\_A) \\
\ \ \ \ \ \ ((ghs\_rreduce\_Tree\ ghs\_rreduce\_a)\ ((\lambda((Node\ a\ b\ c)) \rightarrow c)\ gh\_A)\ x'))) \\
\ \}))
\end{array}
$$

As we shall see in the next section, the time and space usage of the optimized *rreduce*⦃*Tree*⦄ are much lower than of its unoptimized version.

## 7.3 Performance

After all this effort of optimizing code generated for specializations of generic functions, obviously we are interested in how our improvements matter in terms of space- and time usage of a given function. Unfortunately it is nearly impossible to predict exactly what the speed-up will be for a given function and input. Even when using a profiler it is difficult to obtain reliable measurements that show the exact speed-up. We have found that the information on the number of reductions and the memory usage given by the Haskell interpreter Hugs [JP] is the most useful indicator of the efficiency of a function. Timing a function by measuring the number of reductions performed eliminates the external influence that other processes running on the same computer may have. Moreover, it is also more precise for small inputs.

The overhead that is introduced by the process of specialization for a given generic function depends mainly on the complexity of the data type to which the function is specialized. If the data type in question has many constructors each with many fields, the generated code will require more structural conversions and hence more overhead will be introduced.

Figure 7.2 shows the space- and time usage of the functions that were used as examples in the previous section. For each function the table lists the run-time

performance of the optimized version, the unoptimized version and a hand-written version which was created in Haskell directly. All functions have been benchmarked with appropriate inputs. Details can be found in appendix B.

From the table it is clear that optimization brings significant improvements in the number of reductions necessary as well as in the use of memory cells. Speed-up factors roughly range from 2 to 7 on the number of reductions and from 1.2 to 12 on memory usage.

The difference in performance between the Haskell functions and the optimized specializations can be explained by the fact that the code generated by the optimizer uses a slightly less efficient way of selecting fields from a constructor value. However, an optimizing Haskell compiler will most likely not have any problems removing this small overhead [2].

Since practical applications (e.g. compilers) typically have substantial data types, we may conclude that the benefits of optimization are greater for generic functions in practical applications than for the examples demonstrated here.

[2] It is actually quite difficult to establish this as a fact because the run-time behavior of compiled Haskell programs cannot be measured in terms of reductions. Even without optimization in the Haskell compiler, the difference between an optimized specialization and a native Haskell equivalent would not be reliably measurable.

# Chapter 8

# Conclusion

## 8.1 Results

In the earlier chapters the inefficiency problem associated with compiled specializations of type-indexed values has been discussed. Our goal was to devise a method that would eliminate all overhead from compiled specializations of type-indexed values. This goal has been accomplished for a large class of generic functions by applying techniques of program transformation and partial evaluation.

For the purpose of performing partial evaluation, we have introduced a symbolic evaluation scheme along with an algorithm for performing symbolic pattern matching. In addition, a number of transformations were formulated to further aid the optimization process. We have shown that partial evaluation in combination with the described transformations can be used to eliminate all redundancies introduced by the process of specialization. In practice optimization turns out to be quite rewarding as time and space usage for evaluation of compiled specializations are typically reduced by 50% or more.

Unfortunately we discovered in a late stage that not all compiled specializations can easily be partially evaluated. It turns out that specializations of generic functions with types containing recursive types cannot be optimized using the approach we presented. However, since generic functions of this type are rarely encountered, in practice the shortcoming of the current method is tolerable. The problems associated with generic functions with types containing recursive types is that the embedding-projection pairs generated for performing structural conversions are indirectly recursive. Partially evaluating expressions involving calls to these functions will currently cause the partial evaluator to run indefinitely. Solving this issue is certainly not trivial and requires techniques for detecting non-termination during partial evaluation. Resolving this problem shall therefore remain future work for now.

In summary, the main contributions of this thesis are:

- A thorough description of the specialization process as carried out by the Generic Haskell compiler.
- An extension of traditional pattern matching to allow symbolic values to be matched against (nested) patterns.

- A number of program transformations and a generalization of an existing transformation.
- The application of partial evaluation techniques in generic functional programming.
- The implementation of the optimizer in the Generic Haskell compiler.

## 8.2 Implementation

The implementation of the optimizer is included as the last stage in the Generic Haskell compiler. For the most part the optimizer implementation consists of a partial evaluator for Haskell. Because Haskell is a large language, supporting all of its features in the partial evaluator would be a substantial undertaking. The following Haskell features are currently not supported:

- Type classes
- **do**-notation
- Record data types
- List comprehensions
- Function guards

Generic functions making use of one or more of these features will not be optimized fully or at all.

Since the use of monads in Haskell relies heavily on type classes, monadic-style programming is consequently also currently not supported.

## 8.3 Future Work

The shortcomings in the implementation of the optimizer mentioned in the previous section are certainly not all trivial to overcome. Most notably, adding support for type classes would be a rather cumbersome task which would involve adding a type-inferencing mechanism to the partial evaluator. Because implementing a partial evaluator that supports all features of Haskell would require a large effort, integrating the partial evaluator with an existing Haskell interpreter may be more feasible.

Judging from the results of chapters 4 and 6, it would be interesting to investigate how to generate more efficient code for specializations of type-indexed values directly. Since we have been able to show the results of partially evaluating the compiled specialization of a generic function for *any* given data type, it seems likely that the efficiency of generated code could be improved with some effort. Effectively this would integrate the process of optimization into the other phases of Generic Haskell.

Once an implementation is available of Dependency-style Generic Haskell [LCJ03] it will be necessary to investigate if the current optimizer can be used to remove overhead from compiled dependency-style specializations. Dependency-style Generic Haskell is a successor to "Classic" Generic Haskell and addresses

the issue that it is often difficult to define generic functions for more involved applications. It is particularly cumbersome to define generic functions that depend upon other generic functions because this typically requires access to the type arguments. Using Dependency-style Generic Haskell generic functions can be defined in a more intuitive way. However, the extra flexibility that Dependency-style Generic Haskell introduces does come at the price of some added complexity in general. It is likely that the optimizer will have to be modified in order to support Dependency-style Generic Haskell. Of course this will depend mostly on the implementation of the specialization process.

## 8.4 Related Work

The implementation of the extension [AP02] to the Clean language that allows generic functions to be defined, suffers from the same efficiency problems as the implementation of Generic Haskell. In [AS03b] a remedy is proposed similar to our approach which attempts to remove structural translations introduced by the process of specialization through partial evaluation. Symbolic evaluation is accomplished by using typing information of expressions to determine what constructors may appear after reduction. Another approach using an extension to the fusion algorithm [Chi94] is described in [AS03a]. Both methods can be applied succesfully only to generic functions with types containing only non-recursive types.

## Acknowledgements

I would like to thank Johan Jeuring and Andres Löh at Utrecht University for suggesting the topic of this thesis and for commenting on my drafts. Finally, I would also like to thank Jan Terlouw at the University of Groningen for being my thesis advisor.

# Appendix A

# Partial Evaluation of Specializations

This appendix is associated with section 4.4. In the sections below a specialization of a type-indexed value modelled in a subset of Haskell, is partially evaluated step-by-step. All relevant definitions and reduction rules can be found in sections 4.2 and 4.4.

Note that the application of the SE-Abs reduction rule is not made explicit in the comments below since SE-Abs is needed in almost every step.

## A.1 Partial Evaluation of $poly_T$

$$poly_T = (\lambda u_1 \ldots u_k \rightarrow to\ (bimap_{(\rightarrow)}\ ep_T\ ep_T)\ (poly_{T^\circ}\ u_1 \ldots u_k))$$

$\Rightarrow$ (1) { inlining definition of $bimap_{(\rightarrow)}$ }

$$\begin{aligned} poly_T = (\lambda u_1 \ldots u_k \rightarrow to\ ((\lambda x\ y \rightarrow & \textbf{case}\ x\ \textbf{of}\ (EP\ a2b\ b2a) \rightarrow \\ & \textbf{case}\ y\ \textbf{of}\ (EP\ c2d\ d2c) \rightarrow \\ & EP\ (\lambda a2c \rightarrow c2d \cdot a2c \cdot b2a) \\ & \quad (\lambda b2d \rightarrow d2c \cdot b2d \cdot a2b)) \\ & ep_T\ ep_T)\ (poly_{T^\circ}\ u_1 \ldots u_k)) \end{aligned}$$

$\Rightarrow$ (2) { inlining definition of $to$; SE-App }

$$\begin{aligned} poly_T = (\lambda u_1 \ldots u_k \rightarrow & (\lambda ep \rightarrow \textbf{case}\ ep\ \textbf{of}\ (EP\ a\ b) \rightarrow b) \\ & (\textbf{case}\ ep_T\ \textbf{of}\ (EP\ a2b\ b2a) \rightarrow \\ & \quad \textbf{case}\ ep_T\ \textbf{of}\ (EP\ c2d\ d2c) \rightarrow \\ & \quad EP\ (\lambda a2c \rightarrow c2d \cdot a2c \cdot b2a) \\ & \quad\quad (\lambda b2d \rightarrow d2c \cdot b2d \cdot a2b)) \\ & poly_{T^\circ}\ u_1 \ldots u_k) \end{aligned}$$

$\Rightarrow$ (3) { SE-App }

$$\begin{aligned} & poly_T = (\lambda u_1 \ldots u_k \rightarrow \\ & \quad (\textbf{case} \\ & \quad\quad (\textbf{case}\ ep_T\ \textbf{of}\ (EP\ a2b\ b2a) \rightarrow \\ & \quad\quad\quad \textbf{case}\ ep_T\ \textbf{of}\ (EP\ c2d\ d2c) \rightarrow \end{aligned}$$

$$
\begin{array}{l}
\quad EP\ (\lambda a2c \rightarrow c2d \cdot a2c \cdot b2a) \\
\qquad (\lambda b2d \rightarrow d2c \cdot b2d \cdot a2b)) \\
\quad \textbf{of}\ (EP\ a\ b) \rightarrow b)\ poly_{T^\circ}\ u_1 \ldots u_k)
\end{array}
$$

$\Rightarrow$ (4) { inlining definition of $ep_T$; }

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\textbf{case} \\
\quad\quad (\textbf{case}\ (EP\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x))) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (Inl\ v) \rightarrow v \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (Inr\ (\ldots (Inr\ v))) \rightarrow v)) \\
\quad\quad\quad\quad\quad\quad \textbf{of}\ (EP\ a2b\ b2a) \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad \textbf{case}\ (EP\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x))) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (Inl\ v) \rightarrow v \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (Inr\ (\ldots (Inr\ v))) \rightarrow v)) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{of}\ (EP\ c2d\ d2c) \rightarrow EP\ (\lambda a2c \rightarrow c2d \cdot a2c \cdot b2a) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\lambda b2d \rightarrow d2c \cdot b2d \cdot a2b)) \\
\quad\quad\quad \textbf{of}\ (EP\ a\ b) \rightarrow b)\ (poly_{T^\circ}\ u_1 \ldots u_k))
\end{array}
$$

$\Rightarrow$ (5) { SE-App; SE-Case1 }

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\lambda b2d \rightarrow \\
\quad\quad\quad (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (Inl\ v) \rightarrow v \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (Inr\ (\ldots (Inr\ v))) \rightarrow v) \\
\quad\quad\quad\quad \cdot\ b2d \\
\quad\quad\quad\quad \cdot\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x))) \\
\quad\quad ) \ (poly_{T^\circ}\ u_1 \ldots u_k))
\end{array}
$$

$\Rightarrow$ (6) { SE-App }

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\lambda x \rightarrow \textbf{case } x \textbf{ of } (Inl\ v) \rightarrow v \\
\qquad\qquad \ldots \\
\qquad\qquad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\qquad\qquad \ldots \\
\qquad\qquad (Inr\ (\ldots (Inr\ v))) \rightarrow v) \\
\quad \cdot (poly_{T^\circ}\ u_1 \ldots u_k) \\
\quad \cdot (\lambda x \rightarrow \textbf{case } x \textbf{ of } (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\qquad\qquad \ldots \\
\qquad\qquad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\qquad\qquad \ldots \\
\qquad\qquad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x))))
\end{array}
$$

$\Rightarrow$ (7) { inlining definition of $(\cdot)$; SE-App }

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\lambda x \rightarrow \textbf{case } ((poly_{T^\circ}\ u_1 \ldots u_k) \\
\qquad\qquad (\textbf{case } x \textbf{ of } (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x)))) \\
\qquad \textbf{of } (Inl\ v) \rightarrow v \\
\qquad\quad \ldots \\
\qquad\quad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\qquad\quad \ldots \\
\qquad\quad (Inr\ (\ldots (Inr\ v))) \rightarrow v))
\end{array}
$$

$\Rightarrow$ (8) { $\lambda$-abbreviation; inlining definition of $poly_{T^\circ}$; SE-App }

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k\ x \rightarrow \\
\quad \textbf{case } (poly_{Sum}\ (poly_{K_1}\ u_1 \ldots u_k)\ (poly_{Sum} \ldots (poly_{K_n}\ u_1 \ldots u_k))) \\
\qquad (\textbf{case } x \textbf{ of } (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\qquad\qquad \ldots \\
\qquad\qquad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\qquad\qquad \ldots \\
\qquad\qquad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x)))) \\
\quad \textbf{of } (Inl\ v) \rightarrow v \\
\qquad \ldots \\
\qquad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\qquad \ldots \\
\qquad (Inr\ (\ldots (Inr\ v))) \rightarrow v
\end{array}
$$

$\Rightarrow$ (9) { inlining definitions of $poly_{K_i}$ from section A.2; SE-App }

$$
\begin{array}{l}
poly_T = (\lambda u_1 \ldots u_k\ x \rightarrow \\
\quad \textbf{case } (poly_{Sum} \\
\qquad (\lambda x \rightarrow \textbf{case } ((poly_{Prod}\ \sigma(t_{11}) \ldots (poly_{Prod} \ldots \sigma(t_{1k_1}))) \\
\qquad\qquad\qquad (\textbf{case } x \textbf{ of } (K_1\ v_1 \ldots v_{k_1}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_1}))) \\
\qquad\qquad \textbf{of } (v_1 : * : \ldots (\ldots : * : v_{k_1})) \rightarrow K_1\ v_1 \ldots v_{k_1})
\end{array}
$$

$$
\begin{array}{l}
(poly_{Sum} \ldots (\lambda x \rightarrow \textbf{case}\ ((poly_{Prod}\ \sigma(t_{n1}) \ldots (poly_{Prod} \ldots \sigma(t_{nk_n}))) \\
\qquad\qquad (\textbf{case}\ x\ \textbf{of}\ (K_n\ v_1 \ldots v_{k_n}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_n}))) \\
\qquad\qquad\quad \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_n})) \rightarrow K_n\ v_1 \ldots v_{k_n})) \\
(\textbf{case}\ x\ \textbf{of}\ (K_1\ v_{11} \ldots v_{1k_1}) \rightarrow Inl\ x \\
\qquad\qquad \ldots \\
\qquad\qquad (K_i\ v_{i1} \ldots v_{ik_i}) \rightarrow Inr\ (\ldots (Inl\ x)) \\
\qquad\qquad \ldots \\
\qquad\qquad (K_n\ v_{n1} \ldots v_{nk_n}) \rightarrow Inr\ (\ldots (Inr\ x)))) \\
\quad \textbf{of}\ (Inl\ v) \rightarrow v \\
\qquad \ldots \\
\qquad (Inr\ (\ldots (Inl\ v))) \rightarrow v \\
\qquad \ldots \\
\qquad (Inr\ (\ldots (Inr\ v))) \rightarrow v)
\end{array}
$$

## A.2 Partial Evaluation of $poly_{K_i}$

$$poly_{K_i} = (\lambda u_1 \ldots u_k \rightarrow to\ (bimap_{(\rightarrow)}\ ep_{K_i}\ ep_{K_i})\ (poly_{K_i^\circ}\ u_1 \ldots u_k))$$

$\Rightarrow$ (1) { analogous to steps (1) through (6) of section A.1}

$$
\begin{array}{l}
poly_{K_i} = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_i})) \rightarrow K_i\ v_1 \ldots v_{k_i}) \\
\quad \cdot (poly_{K_i^\circ}\ u_1 \ldots u_k) \\
\quad \cdot (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ (K_i\ v_1 \ldots v_{k_i}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_i})))
\end{array}
$$

$\Rightarrow$ (2) { inlining definition of $(\cdot)$; SE-App }

$$
\begin{array}{l}
poly_{K_i} = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\lambda x \rightarrow \textbf{case}\ ((poly_{K_i^\circ}\ u_1 \ldots u_k) \\
\qquad\qquad\qquad (\textbf{case}\ x\ \textbf{of}\ (K_i\ v_1 \ldots v_{k_i}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_i}))) \\
\qquad\qquad \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_i})) \rightarrow K_i\ v_1 \ldots v_{k_i}))
\end{array}
$$

$\Rightarrow$ (3) { inlining definition of $poly_{K_i^\circ}$; SE-App }

$$
\begin{array}{l}
poly_{K_i} = (\lambda u_1 \ldots u_k \rightarrow \\
\quad (\lambda x \rightarrow \textbf{case}\ ((poly_{Prod}\ \sigma(t_{i1}) \ldots (poly_{Prod} \ldots \sigma(t_{ik_i}))) \\
\qquad\qquad\qquad (\textbf{case}\ x\ \textbf{of}\ (K_i\ v_1 \ldots v_{k_i}) \rightarrow v_1 : * : \ldots (\ldots : * : v_{k_i}))) \\
\qquad\qquad \textbf{of}\ (v_1 : * : \ldots (\ldots : * : v_{k_i})) \rightarrow K_i\ v_1 \ldots v_{k_i}))
\end{array}
$$

# Appendix B

# Definitions Used For Benchmarking

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
data Bush a = Nil | Cons a (Bush (Bush a))

    -- Test1
testList = [1 .. 1000]
mapListTest f = f (+1) testList
gmapList = gmap{|[]|}
mapList = map

    -- Test 2
growBush :: Int → Bush Int
growBush 0 = Nil
growBush n = Cons n (Cons (growBush (n − 1))
                          (Cons (Cons (growBush (n − 1)) Nil) Nil))
testBush = growBush 8

mapBush :: (a → b) → (Bush a) → (Bush b)
mapBush f Nil = Nil
mapBush f (Cons x b) = Cons (f x) (mapBush (mapBush f) b)

gmapBush = gmap{|Bush|}
mapBushTest f = f (+1) testBush

    -- Test 3
geqBush = eq{|Bush|}

eqBush :: (a → b → Bool) → Bush a → Bush b → Bool
eqBush f Nil Nil = True
eqBush f (Cons x1 b1) (Cons x2 b2) = (f x1 x2) ∧ (eqBush (eqBush f) b1 b2)
eqbush f _ _ = False

eqBushTest f = f (≡) testBush testBush
```

```
    -- Test 4
growTree :: Int → Tree Int
growTree 0 = Leaf
growTree n = Node n (growTree (n − 1)) (growTree (n − 1))

testTree = growTree 8

foldrTree :: (a → b → b) → (Tree a) → b → b
foldrTree op Leaf e = e
foldrTree op (Node x l r) e = op x (foldrTree op l (foldrTree op r e))

rreduceTree = rreduce{|Tree|}
rreduceTreeTest f = f (:) testTree []
```

# Bibliography


[AP02] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for clean. In Thomas Arts and Markus Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001*, volume 2312, pages 168–185. Springer-Verlag, Berlin, 2002.

[AS03a] Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. 2003.

[AS03b] Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. 2003.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422, pages 52–67. Springer-Verlag, Berlin, 1998.

[Chi94] Wei-Ngan Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.

[CJL01] Dave Clarke, Johan Jeuring, and Andres Löh. The Generic Haskell User's Guide. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2001.

[CL03] Dave Clarke and Andres Löh. Generic Haskell, Specifically. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47. Kluwer, B.V., 2003.

[dW02] Jan de Wit. A technical overview of Generic Haskell. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2002.

[Fut71] Yoshihiko Futamura. Partial evaluation of Computation process - an approach to a compiler-compiler. In *Systems, Computers, Controls*, pages 45–50, 1971.

[Hin00] Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, pages 2–27, 2000.

[HJ03] Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. 2003.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[JJ97] Patrik Jansson and Johan Jeuring. Polyp - a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programing Languages*, pages 470–482, 1997.

[JP] Mark P. Jones and John C. Peterson. Hugs 98: A functional programming system based on haskell 98.

[LCJ03] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the International Conference on Functional Programming, ICFP'03*, pages 141–152, 2003.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, Cambridge, MA, 1991.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, Cambridge, UK, 2003.

[San95] André Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, 1995.