Improving Generic Views
M. Sc. Thesis Proposal

Thomas van Noort
trnoort@cs.uu.nl

Center for Software Technology
Universiteit Utrecht, The Netherlands

February 28, 2007

Abstract

Generic views resolve the tension between data abstraction and induction in generic programming. This thesis proposal discusses two improvements of generic views: generic views for generic types and programmable views. Both improvements increase the capabilities and usability of generic views for generic programming.

1 Introduction

Parametric polymorphism is an important concept in functional programming: polymorphic functions are defined independently of the type of values. However, parametric polymorphism restricts us as well because we can not exploit the structure of the type of the values. Using generic programming we can define *generic functions*; functions defined by induction on the structure of types. Several approaches to generic programming exist, which differ in the representation used for the structure of a type. This thesis proposal uses an extension of Haskell called Generic Haskell [5].

Generic Views

Data abstraction and induction are important concepts in the area of functional programming. Unfortunately, these concepts do not go well together. The concept of data abstraction decouples the usage and definition of data, which improves modularity and maintainability. However, the concept of induction is based on a certain representation of data. It is evident that there is tension between hiding the representation and exploiting the representation in order to define functions using induction. This tension is resolved by the concept of *views* [7].

Generic Haskell uses views to define generic functions. Since it is unknown beforehand by which type a generic function is indexed, there should be a uniform representation of types. In Generic Haskell, a type is viewed using the standard view as a sum of products. This view allows us to define generic functions by induction on the structure of types. Besides the standard view, other views of types exist, e.g., the balanced sums view, the list-like view, and the recursive view. Together with the standard view, these views are called *generic views* [3]. In this thesis proposal we discuss two improvements of generic views: generic views for generic types (Section 2) and programmable views (Section 3). Furthermore, a road map (Section 4) is discussed briefly, followed by some concluding remarks (Section 5).

2 Generic Views for Generic Types

We use the domain of (generic) term rewriting to illustrate the need for generic views for generic types. We start by defining a small library for rewriting mathematical expressions, using plain Haskell, in Section 2.1. Then, we abstract over rewriting the language of simple mathematical expressions and define functions for generic rewriting using Generic Haskell. Finally, we propose an improvement of generic views in order to solve the problems encountered when defining a generic rewriting library. In Section 2.2 we discuss what approach will be taken in solving the problems.

2.1 Motivation

Rewriting Expressions

Several arithmetic laws (e.g., distributivity of $*$ over $+$) hold on mathematical expressions. For example, an expression of the form, $x * (y + z)$, is equivalent to, $(x * y) + (x * z)$. Using plain Haskell, we define a small library for rewriting expressions. Consider the following data type for simple expressions:

```
data Expr = Const Int
          | Add   Expr Expr
          | Mul   Expr Expr.
```

Applying arithmetic laws to expressions amounts to rewriting values of type Expr. Applying the distributivity rule to, for example,

```
Mul (Const 2)
    (Add (Const 3) (Const 4)),
```

yields

```
Add (Mul (Const 2) (Const 3))
    (Mul (Const 2) (Const 4)).
```

In order to define such a rule on Expr values, a constructor for representing metavariables must be added to the Expr type. Otherwise, a specific rewrite rule must be defined for each possible Expr value, which is undesirable for evident reasons. The extended Expr type is defined as follows:

```
data MVExpr = MetaVar  String
            | MVConst  Int
            | MVAdd    MVExpr MVExpr
            | MVMul    MVExpr MVExpr.
```

Using a type synonym for rewrite rules,

```
type Rule (t :: ⋆) = (t, t),
```

we define the distributivity rule on the extended Expr type as

```
distr :: Rule MVExpr
distr = (lhs, rhs)
  where lhs = MVMul (MetaVar "x")
                    (MVAdd (MetaVar "y") (MetaVar "z"))
        rhs = MVAdd (MVMul (MetaVar "x") (MetaVar "y"))
                    (MVMul (MetaVar "x") (MetaVar "z")).
```

Using this rule, we can rewrite expressions by composing functions for matching and building:

```
applyExpr                   :: Rule MVExpr → Expr → Maybe Expr
applyExpr (lhs, rhs) expr = do subst ← matchExpr lhs expr
                               return (buildExpr rhs subst).
```

Matching the left-hand side of a rewrite rule to an expression might fail; therefore, this functions returns a substitution in the Maybe monad. The substitution is built using explicit pattern matching in a top-down fashion:

```
type Subst (t :: ⋆) = Map String t
matchExpr                                    :: MVExpr → Expr → Maybe (Subst Expr)
matchExpr (MetaVar s)      expr          = Just (singleton s expr)
matchExpr (MVConst i)      (Const i′)    | i ≡ i′      = Just empty
                                         | otherwise = Nothing
matchExpr (MVAdd e1 e2)  (Add e′1 e′2) = mergeSubst (matchExpr e1 e′1) (matchExpr e2 e′2)
matchExpr (MVMul e1 e2) (Mul e′1 e′2) = mergeSubst (matchExpr e1 e′1) (matchExpr e2 e′2)
matchExpr _                _              = Nothing.
```

Matching against a metavariable always succeeds and results in a singleton substitution. If a constant is encountered, it suffices to verify that the values are equal in order to succeed matching. The remaining constructors are matched pairwise. The resulting substitutions of the recursive calls are combined by the function *mergeSubst*; its type is

```
mergeSubst :: (Eq t) ⇒ Maybe (Subst t) → Maybe (Subst t) → Maybe (Subst t).
```

This function is responsible for verifying that equivalent metavariables occurring in different parts of the expression, map to the same expression. If *matchExpr* succeeds in producing a substitution, this can be used by the function *buildExpr* to construct a rewritten expression:

```
buildExpr                          :: MVExpr → Subst Expr → Expr
buildExpr (MetaVar s)      subst = subst ! s
buildExpr (MVConst i)      _     = Const i
buildExpr (MVAdd e1 e2)  subst = Add (buildExpr e1 subst) (buildExpr e2 subst)
buildExpr (MVMul e1 e2) subst = Mul (buildExpr e1 subst) (buildExpr e2 subst).
```

All metavariables used at the right-hand side of a rewrite rule must be bound at the left-hand side. Therefore, the function *buildExpr* fails in case a metavariable is not found in the substitution. Building a constant is relatively easy: it amounts to changing the constructor. Again, the remaining constructors pairwise call the function recursively. The resulting expressions are combined into a single expression using the corresponding constructor of the original Expr type.

Generic Rewriting

Previously, the functions used for rewriting expressions were specific to the type of expressions. If we would like to rewrite values of other types (e.g., linear equations), we would have to implement these functions all over again. Fortunately, using Generic Haskell, it is possible to define *generic* rewriting functions which allow us to rewrite values of any type.

Three functions are specific to rewriting mathematical expressions: *applyExpr*, *matchExpr*, and *buildExpr*. Furthermore, the original Expr type is extended to MVExpr to facilitate matching arbitrary expressions against metavariables. We define a generic rewriting library by defining the three functions as generic functions, which are all indexed by the type of expressions. First, we define *apply* which is a generic abstraction and is indexed by the type to which rewrite rules are applied:

```
apply ⟨t :: ⋆⟩                :: (Eq t) ⇒ Rule t → t → Maybe t
apply ⟨τ⟩ (lhs, rhs) term = do subst ← match ⟨τ⟩ lhs term
                                return (build ⟨τ⟩ rhs subst).
```

The definition of *apply* is similar to *applyExpr*. They only differ in the type and the functions used for matching and building values. The type of *apply* tells us that rules are defined on values on type t and that rules are applied to values of type t as well. This in contrast to the definition of *applyExpr*: rules are defined on MVExpr values and Expr values are rewritten. This is a consequence of using Generic Haskell where functions with multiple type indexes can not be defined. Since the generic function *match* must index on two arguments, as we will see next, the types of these arguments must be the same. This has quite some consequences regarding the usability of the generic rewriting library, which will be discussed later on. Given two values, the generic abstraction *match* returns a possible substitution since matching might fail:

```
match ⟨t :: ⋆⟩        :: (Eq t) ⇒ t → t → Maybe (Subst t)
match ⟨τ⟩ lhs term =
  case getMetaVar ⟨τ⟩ lhs of
    Just s     → Just (singleton s term)
    Nothing → case topleveleq ⟨τ⟩ lhs term of
                  True  → let lhsC    = children ⟨τ⟩ lhs
                              termC   = children ⟨τ⟩ term
                              matches = zipWith (match ⟨τ⟩) lhsC termC
                          in  foldr mergeSubst (Just empty) matches
                  False → Nothing.
```

The generic definition of *match* has the same pattern as the original definition, specific to expressions. First, encountering a metavariable results in a single substitution. The generic function *getMetaVar* is defined in the generic rewriting library and returns the field of the *MetaVar* constructor if the argument is a metavariable. If the argument is not a metavariable, the two values are verified to be equal at top level using *topleveleq*, which is defined using the fixed-point view on the value level. Using this generic view, recursive calls in the type definition are made explicit. Since the function only has to verify equality on top level, we stop the recursion by returning *True* for recursive calls on the value level using a local redefinition:

```
topleveleq ⟨t :: ⋆ viewed Fix⟩   :: t → t → Bool
topleveleq ⟨Fix φ⟩ (In x) (In y) = let eq ⟨α⟩ _ _ = True
                                   in  eq ⟨φ α⟩ x y.
```

If the values are not equal at top level, matching stops immediately. Otherwise, construction of the substitution continues. A helper generic function *children* is used to collect the direct recursive children in a list [4]. These children are matched pairwise and the resulting substitutions are merged as before. The generic function *children* uses the fixed-point view on the value level as well:

```
children ⟨t :: ⋆ viewed Fix⟩  :: t → [t]
children ⟨Fix φ⟩ (In r)       = let collect ⟨α⟩ x = [x]
                                in  collect ⟨φ α⟩ r.
```

Now that we have computed the substitution, we can build the right-hand side of the rewrite rule using this substitution. The definition of the generic abstraction *build* is not that similar to the original definition, specific to expressions:

```
build ⟨t :: ⋆⟩         :: t → Subst t → t
build ⟨τ⟩ rhs subst = mapRec ⟨τ⟩ substitute rhs
  where substitute term = case getMetaVar ⟨τ⟩ term of
```

```
                    Just s     → subst ! s
                    Nothing → term.
```

This definition uses a recursive map, which maps the given function in a top-down fashion. The function that is mapped has to verify whether the current value is a metavariable. If this is the case, it looks up the term in the substitution and returns it as a result. Otherwise, the original value being considered is returned. The recursive map uses the fixed-point view to recursively traverse the data structure:

```
mapRec ⟨t :: ⋆ viewed Fix⟩  :: (gmap ⟨t⟩) ⇒ (t → t) → t → t
mapRec ⟨Fix φ⟩ g (In r)     = let (In r′)    = g (In r)
                                  gmap ⟨α⟩  = mapRec ⟨Fix φ⟩ g
                              in  In (gmap ⟨φ α⟩ r′).
```

The definition of *mapRec* first applies the given function to the top level value. Then, it applies itself recursively to the children of the modified top level value. This causes the given function to be applied in a top-down fashion.

All the necessary functions are lifted to generic functions, and the generic rewriting library is complete. Using this library, the distributivity rule is applied as follows,

```
apply ⟨MVExpr⟩ distr ((MVMul (MVConst 2)
                              (MVAdd (MVConst 3)
                                     (MVConst 4)))),
```

which still yields,

```
Just (MVAdd (MVMul (MVConst 2)
                   (MVConst 3))
            (MVMul (MVConst 2)
                   (MVConst 4))).
```

It is important to realize that rewrite rules are applied to the extended type (e.g., MVExpr) and *apply* returns a value of this type as well.

Extending Data Types

In addition to the the three functions necessary for generic rewriting, we also have to extend the original type with a constructor representing a metavariable. This is a process that currently has to be done manually, which is undesirable. Therefore, we would like to have *generic views for generic types*, such that an annotation like,

```
MV ⟨t :: ⋆ viewed Fix⟩ :: ⋆,
```

would instruct the Generic Haskell compiler to convert the given type to the right type from the fixed-point view. Furthermore, the implementation of the generic rewriting library is based on a number of assumptions regarding the constructor representing a metavariable: its name is *MetaVar* and the constructor has a single field of type String. Forcing the user to be aware of such assumptions is undesirable. These assumptions are reflected in the definition the generic function *getMetaVar*:

```
getMetaVar ⟨t :: ⋆⟩               :: MV ⟨t⟩ → Maybe String
getMetaVar ⟨Sum α β⟩ (Inr y) = Just y
getMetaVar ⟨_ :: ⋆⟩       _       = Nothing.
```

We would like the generic rewriting library to be an exact generalization of the expression rewriting library as discussed before. Therefore, we would like to change the type of the generic abstraction *apply* to

```
apply ⟨t :: ⋆⟩ :: (Eq t) ⇒ Rule (MV ⟨t⟩) → t → Maybe t,
```

such that the application of a rewrite rule does not result in a value of the extended type.

We begin by defining a type which automatically extends a type with a constructor. Our first attempt to add a constructor for metavariables is a type synonym,

```
type MV (t :: ⋆) = Sum t String,
```

which adds a sum to an existing type. Unfortunately, it is not that easy to extend a data type with another constructor. This approach only adds a constructor at top level, which does not fit our needs. The power of a generic rewriting library depends on the power of the rewrite rules that can be defined. If metavariables can occur at top level only, the power of rewrite rules is severely limited. In fact, we are not able to define the distributivity rule with metavariables occurring at top level only. Therefore, we would like to add a constructor at all levels of a type. This implies that we have to make the recursive calls in a type definition explicit. As we have seen before on the value level, the fixed-point view allows us to do so. In order to make the points of recursion explicit on the type level, we have to lift the fixed-point view to the type level.

If we define a generic type using the fixed-point view for the purpose of adding a constructor, we have to define a result using the Fix f type. This is because we make the recursive calls in the type definition explicit, then we add a constructor to these recursive calls, and finally we make the new recursive calls explicit again. Since the argument f of Fix f has kind $\star \to \star$, we have to use the lifted versions of sum and product:

```
data K (a :: ⋆) (b :: ⋆)                     = K a
data Id (a :: ⋆)                             = Id a
data LSum (f :: ⋆ → ⋆) (g :: ⋆ → ⋆) (a :: ⋆) = LInl (f a) | LInr (g a)
data LProd (f :: ⋆ → ⋆) (g :: ⋆ → ⋆) (a :: ⋆) = f a ⊗ g a.
```

Using these types, we define a generic type which, using the fixed-point view, extends the given type with a constructor representing a metavariable:

```
MV ⟨t :: ⋆ viewed Fix⟩ :: ⋆
type MV ⟨Fix φ⟩        = Fix (LSum φ (K String)).
```

We assume for now that the name generated for the extra constructor is `"MetaVar"`.

Now that we have defined the generic type MV, we adjust the generic rewriting library according to the new type of *apply* as introduced earlier. The definition of *apply* does not change, however, the types and definitions of the functions called by *apply* do change. We start by changing the type and definition of *match*:

```
match ⟨t :: ⋆⟩        :: (Eq t) ⇒ MV ⟨t⟩ → t → Maybe (Subst t)
match ⟨τ⟩ lhs term =
  case getMetaVar ⟨τ⟩ lhs of
    Just s    → Just (singleton s term)
    Nothing → case topleveleq ⟨τ⟩ lhs term of
                      True  → let lhsC     = children ⟨MV ⟨τ⟩⟩ lhs
                                  termC    = children ⟨τ⟩ term
                                  matches = zipWith (match ⟨τ⟩) lhsC termC
                              in  foldr mergeSubst (Just empty) matches
                      False → Nothing.
```

Since the first argument (i.e., *lhs*, the left-hand side of the rewrite rule) now has the extended type, applying *children* to *lhs* changes as well. Another consequence of changing the type is that the definition of *topleveleq* has to be changed accordingly. Previously, this generic function was defined using the fixed-point view and a local redefinition on the generic function *eq*. Unfortunately, the type of the arguments of *topleveleq* are no longer the same. The new definition of *topleveleq* removes the additional sum using a case analysis on the first argument. This definition uses knowledge from the context in which it is used, we already know that the first argument is not a metavariable, thus, it always contains an *Inl* constructor:

$$\begin{array}{ll} \mathit{topleveleq}\ \langle \mathrm{t} :: \star\ \textbf{viewed}\ \mathrm{Fix}\rangle & ::\ \mathrm{MV}\ \langle \mathrm{t}\rangle \to \mathrm{t} \to \mathit{Bool} \\ \mathit{topleveleq}\ \langle \mathrm{Fix}\ \varphi\rangle\ (\mathit{In}\ (\mathit{Inl}\ x))\ (\mathit{In}\ y) & = \textbf{let}\ \mathit{eq}\ \langle \alpha\rangle\ \_\ \_ = \mathit{True} \\ & \quad \textbf{in}\ \ \mathit{eq}\ \langle \varphi\ \alpha\rangle\ x\ y. \end{array}$$

The definition of *build* does not change; only its type changes to

$$\mathit{build}\ \langle \mathrm{t} :: \star\rangle :: \mathrm{MV}\ \langle \mathrm{t}\rangle \to \mathrm{Subst}\ \mathrm{t} \to \mathrm{t}.$$

This generic abstraction is defined using a generic map called *mapRec*. Again, the definition of *mapRec* does not change, but its type is adapted to

$$\mathit{mapRec}\ \langle \mathrm{t} :: \star\ \textbf{viewed}\ \mathrm{Fix}\rangle :: (\mathit{gmap}\ \langle \mathrm{t}\rangle) \Rightarrow (\mathrm{MV}\ \langle \mathrm{t}\rangle \to \mathrm{t}) \to \mathrm{MV}\ \langle \mathrm{t}\rangle \to \mathrm{t}.$$

Finally, we adjust the type and definition of the distributivity rule to

```
distr :: Rule (MV ⟨Expr⟩)
distr = (lhs, rhs)
  where lhs = Mul_MV (MetaVar "x")
                     (Add_MV (MetaVar "y") (MetaVar "z"))
        rhs = Add_MV (Mul_MV (MetaVar "x") (MetaVar "y"))
                     (Mul_MV (MetaVar "x") (MetaVar "z")).
```

The syntactic sugar used in this definition (e.g., $\mathit{Add}_{\mathrm{MV}}$) is explained in Section 2.2.

Other Examples

The domain of generic rewriting is not the only domain which benefits from generic views for generic types. In the domain of structure editors, incomplete values exist because not all the required information is available at once. Therefore, data types are extended with holes to make the current incomplete values, type correct [1]. These holes are filled (i.e., replaced) later on when more information is available. Data types are extended with holes by taking an approach similar to the first approach taken by this proposal for extending data types. As argued before, this approach is undesirable and a better approach is to use the fixed-point view on a generic type. Similar problems concerning holes have been encountered in the generic structure editor Proxima [6].

Another application which can benefit from generic views for generic types is the Zipper [2]. The Zipper is a data structure representing a tree together with a subtree, which is the current focus of attention. Navigation functions allow the user to move the point of focus, but only to recursive components. The Zipper is implemented using generic types which expects a pattern functor as type argument, to use it as if the generic type is viewed as a fixed-point. In the current implementation, the user has to define the pattern functor manually. Using generic views for generic types, this would not be necessary any more because pattern functors will be generated automatically for generic types.

Other Views

Up until now, we considered lifting only the fixed-point view to the type level. However, there are other generic views which might be useful on the type level as well. A view that closely resembles the fixed-point view is the recursive view. Since the fixed-point view is useful on the type level, the recursive view is expected to be useful on the type level as well. It is yet unknown whether other views, such as the balanced sums view and the list-like view, are useful on the type level.

2.2 Approach

In Section 2.1 we have motivated our desire to lift generic views to the type level. We distinguish four parts of generic views for generic types: language design, type translation, definition of constructors, and referring to constructors. Note that the definition of a view [3, 4] does not need to change, no additional definitions are required to lift the generic views to the type level.

Language Design

A generic function which is defined using a generic view other than the standard view (e.g., the generic function *children*), uses the following syntax in its type:

$$\mathit{children}\ \langle \mathrm{t} :: \star\ \textbf{viewed}\ \mathrm{Fix}\rangle :: \mathrm{t} \to [\mathrm{t}].$$

In order to define a generic type using a generic view, an obvious choice would be to use similar syntax. For example, the generic type MV using the fixed-point view will be defined as:

$$\mathrm{MV}\ \langle \mathrm{t} :: \star\ \textbf{viewed}\ \mathrm{Fix}\rangle :: \star.$$

Formally, we will extend the language introduced for generic views [3] to include generic types and especially generic views for generic types. We will define appropriate typing and kinding rules for the extensions to the language. These rules will be used in order to perform static analyses on the generic types: verifying the kind of the indexed type and verifying the kind of the generic type.

Type Translation

In order to include generic views for generic types, only the process of translating generic functions on generic types [2] has to be modified. Currently, the standard view is used by default in the translation process. Ideally, the process would parameterize over the view to be used in the translation. Therefore, we will generalize the process by parameterizing over the view which is used in the translation. This will allow us to plug in any embedding-projection pair corresponding to the view defined on the generic type.

Formalizing this approach should allow us to prove correctness of the translation process of generic views for generic types.

Definition of Constructors

As can be seen from the definition of the generic type MV,

$$\begin{array}{ll} \mathrm{MV}\ \langle \mathrm{t} :: \star\ \textbf{viewed}\ \mathrm{Fix}\rangle & ::\ \star \\ \textbf{type}\ \mathrm{MV}\ \langle \mathrm{Fix}\ \varphi\rangle & = \mathrm{Fix}\ (\mathrm{LSum}\ \varphi\ (\mathrm{K\ String})), \end{array}$$

the name associated with the added constructor is not mentioned explicitly. It is desirable to have the possibility to be able to give the constructor a name. Using the constructor descriptor type *Con c* a, the constructor name can be defined explicitly. Assuming that the first field of a constructor descriptor *Con* contains the name of the constructor, the new generic type would be defined as

$$\begin{array}{ll} \mathrm{MV}\ \langle \mathrm{t} :: \star\ \textbf{viewed}\ \mathrm{Fix}\rangle & ::\ \star \\ \textbf{type}\ \mathrm{MV}\ \langle \mathrm{Fix}\ \varphi\rangle & = \mathrm{Fix}\ (\mathrm{LSum}\ \varphi\ (\mathrm{Con}\ \texttt{"MetaVar"}\ (\mathrm{K\ String}))). \end{array}$$

This definition requires special attention. Since we define a generic type, it seems invalid to specify a name (i.e., a value) for the extra constructor. However, a similar approach is used for generic functions. Consider the following definition of the generic function *show*, where the other arms are omitted for brevity,

```
show ⟨a :: ⋆⟩                   :: (show ⟨a⟩) ⇒ (String → String) → a → String
show ⟨Con c α⟩ p (Con x) = let parens x = "(" ++ x ++ ")"
                               body     = show ⟨α⟩ parens x
                           in  if   null body
                               then c
                               else p (c ++ " " ++ body).
```

The identifiers in the type are considered to be type variables. However, the identifier *c* is used in the body of *show* as a value. Actually, the identifier *c* is an abstract value generated by the compiler. With generic views for generic types, we would like to be able to define this abstract value, instead of letting the compiler generate it for us. Evidently, if there is no constructor name specified, the compiler still must generate a name.

Referring to Constructors

The example of generic rewriting shows us that it is desirable to be able to refer to the constructors of the indexed type of a generic type. For example, if we define a rewrite rule on expressions, we have to know how to refer to the constructors of the Expr type. Unfortunately, we can not use the constructor names of the original type. This is because the compiler generates a new type for each arm of a generic type, introducing new constructors since duplicate constructor names are not allowed in Haskell. We do not want to confront the user with compiler generated constructor names, therefore, we propose syntactic sugar in order to refer to the right constructor. For example, the type Expr has a constructor *Add*. If we instantiate the generic type MV to Expr, this introduces new constructors for the generic type. If we still want to refer to the constructor *Add*, corresponding to the instantiated generic type MV, we use syntactic sugar like Add_{MV}. The compiler will map this name to the associated generated constructor name.

As mentioned earlier, a generic type can also contain new constructors in its definition. Referring to these constructors does not involve syntactic sugar. If the extra constructor is assigned a name, this name can be used to refer to the constructor. Otherwise, the name generated by the compiler has to be used. However, this is a situation that should be avoided because forcing the user to depend on compiler generated names is worse than depending on assumptions made by the library.

3 Programmable Views

In this section we discuss the second improvement of generic views: programmable views. We motivate the need for programmable views (Section 3.1) and describe the approach which will be taken to solve the problems discussed (Section 3.2).

3.1 Motivation

Currently, four views are implemented in the Generic Haskell compiler: the standard view, the identity view, the list-like view, and the fixed-point view. These views are defined by adding and modifying several files of the compiler. If a user wants to add a view to the compiler, the user has to perform the changes and recompile the compiler. Furthermore, if a new version of the compiler is released, the user has to perform the modifications again to include their own views. Clearly, this is not an elegant and modular approach. We would like to be able to define views in an elegant and modular fashion using *programmable views*.

Having the opportunity to define your own views in an elegant and convenient fashion will stimulate users to experiment with views. Furthermore, it is expected that experimenting with views will quickly lead to new useful views.

The original concept of generic views will not change; generic views will become more useful because the users can define their own views. Additionally, programmable views do not apply to generic functions only, but to generic types as well. As explained in Section 2.2, we do not need any additional specifications to lift the generic views to the type level.

3.2 Approach

We split up the concept of programmable views in three parts: language design, code generation and integration, and verification of view specifications. We discuss the approach separately for each part.

Language Design

The design of the language will depend on the information required to define a view. Currently, the user is forced to define several Attribute Grammar (AG) files and to adjust existing AG files to include the new view in the compiler. We will start with a minimal abstraction of this approach: extracting *all* what is defined in AG files for a new view to an external file defined by the user. Then, we will search for common patterns in the view definitions in order to provide useful abstractions. These abstractions will be defined in a Domain-Specific Embedded Language (DSEL) using Generic Haskell as the host language. It is yet unknown whether the power of Generic Haskell is needed to specify the library. It could be that Haskell would suffice as well as the host language. We can abstract even more by looking for information that can be inferred from a view definition, and what information has to be defined explicitly. It is expected that this phase in the abstraction process is most likely to introduce syntax specific to our domain; transforming the DSEL into a DSL.

A problem expected to occur in the first steps of the abstraction process involves extracting the AG code. Since the AG code defines a traversal over the abstract-syntax tree and defines attributes for each syntactic construct, the DSEL must do this as well. Unfortunately, this will quickly clutter the definition of a view because of the large amounts of syntactic constructs. Since we want to minimize the effort required to implement a new view, we will have to abstract over the syntactic constructs in some way to provide an elegant approach to programmable views.

Code Generation and Integration

The GH compiler uses AG to generate code by traversing over the abstract syntax tree. The views defined externally by the user will be compiled to an AG definition as well. We would like to incorporate these generated view definitions in the GH compiler, to let the compiler generate code according to these view definitions. However, these views are defined by the user and the GH compiler itself is already compiled. There are two solutions to this problem, we can either dynamically load the AG definitions or compile the AG definitions separately. The former would be the best solution since this would optimally incorporate the external view definitions in the current AG definitions of the GH compiler. Unfortunately, the dynamic loading of AG definitions is not possible yet. Therefore, we will use the latter approach which compiles the AG definitions separately. The GH compiler will have to assume that there is an interface available to the compiled AG definition with which it can generate the right code.

Verification of View Specifications

Since the user can define views which are used in the Generic Haskell compiler, we have to be aware of the fact that invalid views can be defined. A view is valid if it is kind preserving and generates well-typed, well-behaved conversions [3, 4]. A Haskell compiler can verify the validity of a view specification to a certain extent. The kind preservation and well-typedness of conversions can be verified since kind and type checking is performed on the generated structural representation and embedding-projection pairs. Since it is undecidable to verify that the composition of two functions is the identity, it is impossible to verify that a view generates well-behaved conversions. Satisfying this property is the responsibility of the user of the programmable views.

The design of the specification language determines what kind of analyses can be performed on a view specification. It is important that all the required information is available in a view specification. Therefore, using a DSL instead of an DSEL might be relevant as well to the kind of analyses performed on a view specification.

4 Road Map

This section describes roughly how the time available is divided over each of the parts of the thesis. We can distinguish four parts: starting up the thesis, generic views for generic types, programmable views, and finishing the thesis. The whole month of August is reserved for a holiday in South-Africa.

Part 1: March 2007

The first part of the thesis consists of writing the outline of the thesis. Together with the outline, the introductory parts of the thesis will be written which is the formal basis of the thesis. This formal basis will be used in the theoretical approach to the problems discussed.

Part 2: April, May 2007

Generic views for generic types is the first improvement of generic views. First, the required features will be implemented in the current Generic Haskell compiler. The results of this implementation, and especially any unexpected problems encountered, will be documented afterwards. Additionally, a formal approach to generic views for generic types will be documented. This formal approach will consist of the formal basis as discussed before and the existing formal approach to generic views [3].

Part 3: June, July 2007

The third part will use the same planning as for generic views for generic types. We will start by developing a view definition language and integrating this with the existing Generic Haskell compiler. This process will mainly focus on defining the existing views using the new view definition language. Following that, the design of the view definition language will be documented. Additionally, we will try to define new views using the view definition language to reveal any of its shortcomings and discover its capabilities.

Part 4: September 2007

The last part consists of finishing the thesis by removing any rough edges left. Furthermore, the thesis will be defended at the Software Technology Colloquium.

5 Conclusion

Previous work on generic views [3] leaves room for improvement. In this thesis we will try to improve generic views in two ways: lifting the generic views to the type level (generic views for generic types) and defining a view definition language (programmable views). The former improves the capabilities and usability of generic types by automating the conversions of types. The latter allows us to define views in an elegant and modular fashion, independent from the Generic Haskell compiler. It is expected that programmable views leave room for improvement as well. Since this improvement involves the complete design of a language, it is expected that we are not able to solve all problems encountered in this thesis, and further research in this area is inevitable.

References

[1] Paul Hagg. A framework for developing generic XML tools. Master's thesis, Universiteit Utrecht, 2002.

[2] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.

[3] Stefan Holdermans. Generic views. Master's thesis, Universiteit Utrecht, 2005.

[4] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriquez. Generic views on data types. In *MPC '06: Proceedings of the 8th International Conference on Mathemathics of Program Construction*, pages 209–234. Springer-Verlag, 2006.

[5] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Universiteit Utrecht, 2004.

[6] Martijn Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Universiteit Utrecht, 2004.

[7] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313. ACM Press, 1987.