Dynamic Construction of Generic Functions

Ronny Wichers Schreur and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{ronny,rinus}@cs.ru.nl
http://www.cs.ru.nl/{~ronny,~rinus}

Abstract.. This paper presents a library for the run-time construction and specialisation of generic or polytypic functions. This library utilises the type information that is available in dynamics to implement generic functions on their values. The library closely follows the static generic framework, both in its use and in its implementation. It can dynamically construct generic operations ranging from equality, *map* and parsers to pretty printers and generic graphical editors. A special feature of the library is that it can also be used to derive meaningful specialisations of generic functions that operate on the type representation of the dynamic.

1 Introduction

This paper is about constructing generic functions for dynamically typed values (or shortly, dynamics). Let us first explain what we mean by generic functions and dynamics.

In Generic Haskell [13] as well as in Clean [15] it is possible to define *generic functions* [4, 8]. A generic function is an ultimate reusable function that allows reflection on the structure of data in a type-safe way.

Once defined, a generic function can be applied on any value of any given concrete static type. Generic functions can be used to define work that is of a general nature. The technique has successfully been applied to define functions like equality, *map*, *fold*, to construct parsers and pretty printers, to create GUI applications [3] and to generate test data [10].

A generic function is actually not a single function, but rather a special kind of overloaded function. To define a generic function, instances for the generic function are defined for a finite number of type constructors. Given these base instances, the compiler can fully automatically derive an instance for the generic function for any given concrete *static* type.

Both in Haskell as well as in Clean one can use *dynamics*. Dynamics allow the programmer to associate a run-time value with its type. The are some differences between dynamics in Haskell and in Clean. In Clean dynamics are incorporated in the language while in Haskell dynamics are made available via a library facility. Dynamics in Clean can be of polymorphic type, and one can do run-time type unification using type pattern variables [14]. Furthermore, dynamics (even

C. Grelck et al. (Eds.): IFL 2004, LNCS 3474, pp. 160–176, 2005.

functions) can be serialised , stored to disk and read it in by some other running application. In this way one can easily create persistency, type-safe plug-ins, and mobile code [17]. The facility has been used to create a type safe functional operating system [16] that uses a typed file system in which all files are dynamics stored on disk.

Dynamics enable the type safe communication of data and code between independently programmed distributed applications. It would therefore be very nice if we would also be able to apply generic functions to a dynamic, in particular to a "foreign" dynamic. In theory it should be possible to construct such a generic function, since a dynamic contains information about its type.

The ability to construct such a generic function that can be applied on any value of any type stored in a dynamic would give us new possibilities. For instance, in our functional operating system we will be able to test the equality of two (unknown) dynamics. It also means that if we receive a dynamic from somewhere, we can automatically create a parser or pretty printer for it. From that moment on, the operating system shell is able to recognise expressions of the types involved.

Figure 1 gives an impression of what we want to achieve . The program at the top writes a tree value in a dynamic to disk. This dynamic value is read by the bottom application. Note that the Tree type is not available at compile time in the bottom application. By using the library it is still possible to create a graphical editor for the tree in the dynamic value.

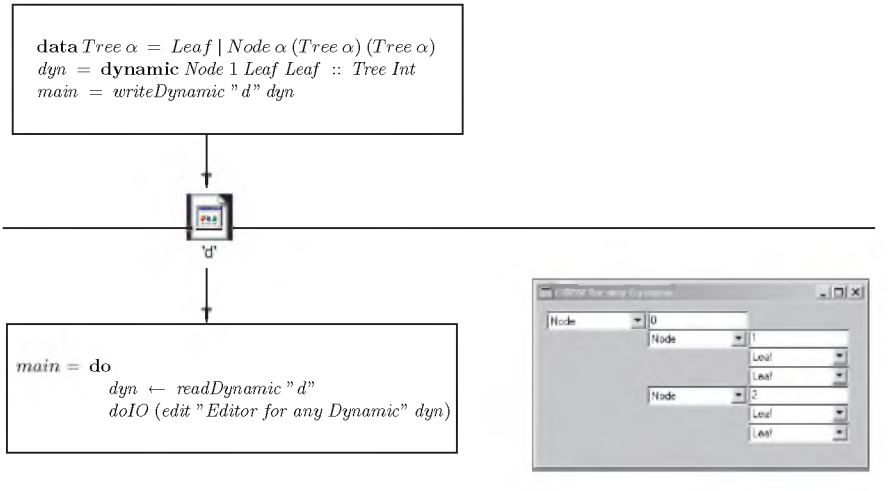

Fig. 1. A dynamically constructed generic editor

In practice this means that all the conversions and constructions that are currently done by the compiler at compile-time now somehow have to be accomplished at run-time. This is not so easy. A compiler can do full reflection on the representation of types and terms, but a running application (Clean uses compiled code) can only do some limited reflection on the representation of the

types. Furthermore one has to be able to construct new functions at run-time. The research question is: is it nevertheless possible to create generic functions for dynamics? In this paper we explain how one can do it, and explain what language facilities are needed to realise it.

The main contributions of this paper are:

- We show that that our library enables the construction of generic functions at run-time in the same spirit as the well-known static generic translation scheme (section 6 and 4);
- We show that our generic functions cannot only be used on the value part but also on the type part of a dynamic (section 4).

The code and examples in this paper are presented in Haskell, because it is more widely known. In any case, the differences are insignificant. The library is implemented in Clean and available from the web-page that accompanies this paper (`http://www.cs.ru.nl/~ronny/DynGen/`).

The remainder of the paper is organised as follows. In section 2 we briefly recap the dynamic machinery. In section 3 we describe how a generic function is statically defined in the language. Then we explain in section 4 how, with help of our library, a generic function for dynamics can be constructed in a very similar way as in the static case. The translation scheme for generics as implemented in the compiler is illustrated in section 5. In section 6 we explain how we manage to realise this translation scheme at run-time. In section 7 we show some extensions, present example applications, and discuss the efficiency of the library. After discussing related work in section 8, we end in section 9 with conclusions and future work.

2 Dynamics

Dynamically typed values, or dynamics for short, combine a value with a representation of its type [1, 15]. Here are some examples of dynamics.

$$
\begin{aligned}
&\mathit{twoDynamics} :: (\mathit{Dynamic}, \mathit{Dynamic}) \\
&\mathit{twoDynamics} = (\mathbf{dynamic}\ 3 :: \mathit{Int},\ \mathbf{dynamic}\ \mathit{id} :: \forall \alpha.\alpha \rightarrow \alpha)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dynApply} :: \mathit{Dynamic} \rightarrow \mathit{Dynamic} \rightarrow \mathit{Dynamic} \\
&\mathit{dynApply}\ (\mathbf{dynamic}\ f :: a \rightarrow b)\ (\mathbf{dynamic}\ x :: a) = \mathit{dynamic}\ f\ x :: b \\
&\mathit{dynApply}\ \_\ \_ = \mathit{error}\ \text{“}\mathit{dynApply} : \mathit{type\ error}\text{”}
\end{aligned}
$$

The first alternative of *dynApply* only matches if the first dynamic argument contains a function value and the second dynamic argument a value of a type that matches the argument type of the function. This example shows how matching on dynamic values involves dynamic unification of types. This guarantees that the application $f\ x$ is safe.

The type pattern variable in a dynamic can also arise from a type variable in the signature of the function. Such a type variable is postfixed with an upward arrow, as in the following two functions.

```
toDyn :: ∀α.Typeable α ⇒ α → Dynamic
toDyn x = dynamic x :: α↑

fromDyn :: ∀α.Typeable α ⇒ Dynamic → α
fromDyn (dynamic x :: α↑) = x
fromDyn _ = error "type mismatch"
```

These so called *type dependent functions* [14] are overloaded in the type representation of that type variable, indicated by the *Typeable* class. For example in *fromDyn* the type of α is determined by the context in which the function is used.

The dynamic system in Clean has more features that are not used in this paper, but that do greatly enhance the applicability of dynamics. One can serialise any dynamic (even functions) and store its value to disk or send it over to another running application. Any other Clean application can read in or receive such a dynamic. Clean uses compiled code which means that a dynamic linker is required that is able to link in code to a running application [17].

2.1 Obtaining Additional Information About Dynamic Types

In Haskell access to the representations of types and data type definitions is available in the Data.Generics library that was developed to support the techniques in the "Scrap your boilerplate articles" [11, 12].

In Clean, dynamics, patterns match on dynamics, as well as dynamic unification are part of the language. Access to the representation of types and the type definitions is therefore less important to the average Clean user. To realise our library, we do need access to this type of information. The representation contains all the information needed to construct the generic representation for dynamic types at run-time. The actual representations of types and data types in both Haskell and Clean differ from the one presented in this paper. We have simplified it a bit to increase readability.

The following library functions are used to obtain additional information about types. The *typeOf* function returns the representation of a type.

```
typeOf :: ∀α. Typeable α ⇒ α → TypeRep
data TypeRep
   =TyCon TyCon | TyApp TypeRep TypeRep
   | TyForAll VarId TypeRep | TyVar VarId
```

The function *typeDefOf* returns a representation of the data type.

```
typeDefOf :: TyCon → TyDef
data TyDef = AlgType {arity :: Int, conses :: [(Constr, [Type]]} | NoType
```

The *Constr* data type represents a data constructor from an algebraic type. It supports the following operations.

```
data Constr =     — abstract type
instance show Constr
build :: Constr → Dynamic
match :: Constr → Dynamic
```

The function *build* returns a dynamic that contains the constructor. The function *match* returns a dynamic with a function that matches on the constructor. For example for the *Cons* constructor in the *List* type these dynamics have the following values.

```
build_Cons = dynamic Cons
match_Cons = dynamic λ l f x → case l of Cons h t → f h t ; _ → x
```

3 Generic Programming

This section describes the basics of generic or *polytypic* programming à la Hinze [9]. Generic functions are defined on the sum-of-products structure of algebraic data types. The following code shows the generic constructors from which the generic structure is build and presents the generic structure for a user defined list type.

```
data 1       = 1                    — unit
data α × β   = α × β                — product
data α + β   = InL α | InR β        — sum

data List α  = Nil | Cons α (List α)   — user defined algebraic type
type List° α = 1 + (α × (List α))      —     and its generic structure
```

In the full blown generic framework the generic structure is much richer with information about data constructors and record fields (their name, arity, and so on). This information is necessary for generic parsers and pretty-printers, but we do not consider it further for clarity's sake.

The remainder of this section illustrates how a programmer defines and uses a generic function in the static generic framework. The running example is a generic equality function that is used to compare two integer lists.

3.1 Define the Type Signature of the Generic Function

The generic equality function is defined as follows.

```
type Eq α    = α → α → Bool
generic eq a :: Eq a
```

In this example there is only one generic variable before the double colon (a), but in general there can be several. The type after the double colon can also be polymorphic in other type variables. We do not consider higher-ranked types in this paper, so all polymorphic variables must be quantified at the top level.

3.2 Provide the Base Instances

The programmer provides each base instance by defining a function with the name of the generic function subscripted with the name of the type constructor.

$$
\begin{array}{lcl}
eq_{Int}\ a\ b & = & a == b \\
eq_1\ 1\ 1 & = & True \\
eq_\times\ eq_1\ eq_2\ (a_1 \times a_2)\ (b_1 \times b_2) & = & eq_1\ a_1\ b_1\ \&\&\ eq_2\ a_2\ b_2 \\
eq_+\ eq_l\ eq_r\ (InL\ a)\ (InL\ b) & = & eq_l\ a\ b \\
eq_+\ eq_l\ eq_r\ (InR\ a)\ (InR\ a) & = & eq_r\ a\ b \\
eq_+\ eq_l\ eq_r\ \_\ \_ & = & False
\end{array}
$$

The number of arguments of a base instance depends on the arity of the type constructor. For example, $eq_\times$ receives equality functions for the first and second elements of the pairs.

3.3 Specialise the Generic Function for a Particular Type

A specialisation is denoted by putting the type between braces after the name of the generic function.

$$main\ =\ print\ (eq\{List\ Int\}\ (Cons\ 1\ Nil)\ (Cons\ 2\ Nil))$$

Here $eq\{List\ Int\}$ is the specialisation of the generic equality function for lists of integers. It is also possible to specialise for types of higher kind such as *List* (kind $* \rightarrow *$). In this paper the type for which a generic function is specialised is assumed to be monomorphic.

4 Dynamic Generic Library

In the previous section 3 we showed how to statically define and use a generic equality function. Here we show how to do the same dynamically. For this purpose the library offers a number of functions to construct a generic function at run-time. Basically, we do the same steps as before. For each step a library function is offered (*defineGeneric*, *baseInstance*, *specialise*). All definitions of the dynamic generic function given so far are collected in an abstract type (*GenFun*).

```
data GenFun    — abstract data type
defineGeneric :: Int → Type → GenFun
baseInstance  :: TyCon → Dynamic → GenFun → GenFun
specialise    :: GenFun → Type → Dynamic
```

We will demonstrate the use of each library function for the equality example from section 2. Because several base instances have to be provided for any generic function, we make the notation a little lighter with an infix variant of the *baseInstance* function. It is defined as follows:

$$
\begin{aligned}
&(:+:)\ \mathit{infixl}\ 4\\
&(:+:)\ ::\ \mathit{GenFun} \rightarrow (\mathit{TyCon},\ \mathit{Dynamic}) \rightarrow \mathit{GenFun}\\
&\mathit{genFun} :+: (\mathit{tyCon},\ \mathit{dyn}) = \mathit{baseInstance}\ \mathit{tyCon}\ \mathit{dyn}\ \mathit{genFun}
\end{aligned}
$$

Below we use the notation $\lceil a \rceil$ as a short-cut for the representation of the type a. For example $\lceil \mathit{List}\ \mathit{Int} \rceil$ denotes $\mathit{typeOf}\ (\bot :: \mathit{List}\ \mathit{Int})$. The same notation is also overloaded to denote the representation of a type constructor. For example $\lceil \mathit{List} \rceil$ denotes the representation of the *List* type constructor. The context always indicates which of the two variants is meant.

4.1 Define the Type Signature of the Generic Function

The first step is to provide the signature of the generic function. For the generic equality it is:

$$
\begin{aligned}
&\mathit{defEq}\ ::\ \mathit{GenFun}\\
&\mathit{defEq} = \mathit{defineGeneric}\ 1\ \lceil \forall a.\ \mathit{Eq}\ a \rceil
\end{aligned}
$$

The generic type variables and any other type variables are all bound by one quantifier in the second argument of *defineGereric*. By convention, the generic type variables are given first, and the integer argument indicates how many generic type variables the function takes. In the example the first variable (a) is the generic type variable.

4.2 Provide the Base Instances

After defining the type of the dynamic generic equality function, we extend it by providing the base instances.

$$
\begin{aligned}
&\mathit{baseEq}\ ::\ \mathit{GenFun}\\
&\mathit{baseEq} = \mathit{defEq} \quad :+: (\lceil \mathit{Int} \rceil, \mathbf{dynamic}\ \mathit{eq}_{\mathit{Int}})\\
&\qquad\qquad\quad :+: (\lceil\ 1\ \rceil, \mathbf{dynamic}\ \mathit{eq}_{1}\ )\\
&\qquad\qquad\quad :+: (\lceil\ \times\ \rceil, \mathbf{dynamic}\ \mathit{eq}_{\times}\ )\\
&\qquad\qquad\quad :+: (\lceil\ +\ \rceil, \mathbf{dynamic}\ \mathit{eq}_{+}\ )
\end{aligned}
$$

Assuming that we already have a static generic function for equality defined, the definition is rather straightforward. The instances of the static generic function *eq* can directly serve as the base instances for the dynamic generic equality.

This code shows that it can be tiresome to populate the generic function with the base instances for all base and primitive types (we should also have provided base instances for *Float*, *Char*, *Bool*). It may be useful to have some language support to make it easier to add all available static base instances.

4.3 Specialise the Generic Function for a Particular Type

Finally we can apply our dynamic generic function to check if two dynamics are equal.

$$
\begin{aligned}
&genEq :: TypeRep \rightarrow Dynamic \\
&genEq = specialise\ baseEq \\
&main\ \ = print\ (genEq\lceil List\ \ Int\rceil \\
&\qquad\qquad\qquad\qquad `dynApply`\ (\textbf{dynamic}\ Cons\ 1\ Nil) \\
&\qquad\qquad\qquad\qquad `dynApply`\ (\textbf{dynamic}\ Cons\ 2\ Nil))
\end{aligned}
$$

The example shows that using the dynamic generic library is very similar to using the static generic framework. In the example above we made good use of the static instances of the generic equality function to serve as the base instances of the dynamic generic equality. However, it is also possible to use the dynamic generic library without using the static generic framework.

5 Generic Translation

Before we explain how generic functions are constructed dynamically we first review the static translation scheme as originated from Hinze [8].

We present the translation scheme by studying the code that the compiler generates for our running example. The purpose of this exposition is to point out the information that is needed to perform the translation and to get an idea of the language features that are used in the generated code. In the next section we will then see how this corresponds to the dynamic setting.

5.1 Overview

The compiler uses the following information for the translation scheme (readily available from the compiler's syntax tree):

- the signature of the generic function;
- the base instances for this generic function;
- the type for which the generic function has to be specialised;
- the type definitions of all types that appear in this type.

The remainder of this section describes the different parts of the translation: the specialisation of the generic function for a type expression, the conversion between values and their generic representation, and the derivation of the generic function for an algebraic type.

5.2 Specialisation

The specialisation of a generic function for a specific type is an easy transformation. It is nothing more than replacing type constructors with the instance of the generic function for that type, and replacing type application by term application. For the specialisation of the generic equality function for list of integers the compiler performs the following transformation.

$$eq\{List\ Int\} \Longrightarrow eq_{List}\ eq_{Int}$$

The eq_{Int} function was provided by the programmer (*Int* is a primitive type), but the compiler must derive the eq_{List} function. The remainder of the section describes how this is accomplished.

5.3 Equality on the Generic Representation

The first step is to specialise the generic equality function for the generic representation type $List^{\circ}$, again by replacing each type constructor with the generic instance for that type.

$$eq_{List^{\circ}} \quad :: \forall\alpha.\ Eq\ \alpha \rightarrow\ Eq\ (List^{\circ}\ \alpha)$$
$$eq_{List^{\circ}}\ a = eq_1\ `eq_+`\ (a\ `eq_{\times}`\ eq_{List}\ a)$$

5.4 Embedding Projection

We now have an equality function on the generic representation of lists, but we need an equality function on lists. We can adapt one to the other by using a so called *embedding projection*. Conveniently enough this embedding projection itself can be implemented as a generic function. It has the following definition.

$$\textbf{data}\ \alpha\ \rightleftarrows\ \beta\ \ =\ EP\ \{from :: \alpha \rightarrow \beta,\ to :: \beta \rightarrow \alpha\}$$
$$\textbf{generic}\ ep\ a\ b =\ a\ \rightleftarrows\ b$$

For the generic equality function only the conversion in one direction is needed because the generic type variable occurs on negative positions (to the left of an arrow), but to cover the general case we combine the conversions both ways.

The embedding projection for the equality function is the specialisation of the generic function *ep* on the structure of signature of the generic function, in our example the equality type $\alpha \rightarrow \alpha \rightarrow Bool$.

$$ep_{eq} \quad :: \forall\alpha\beta.\ (\alpha\ \rightleftarrows\ \beta)\ \rightarrow\ (Eq\ \alpha\ \rightleftarrows\ Eq\ \beta)$$
$$ep_{eq}\ a = a\ `ep_{\rightarrow}`\ (a\ `ep_{\rightarrow}`\ ep_{id})$$

This specialisation deviates from the standard scheme in one place. The type constructor *Bool* is replaced by ep_{id} (defined as $\{from = id,\ to = id\}$) instead of ep_{Bool}. In fact, the embedding projection for any type that does not involve a generic type variable is the identity projection. With this observation the number of embedding projections can be reduced.

The function $ep_{\rightarrow}$ composes the embedding projections for the argument type and the result type.

$$ep_{\rightarrow}\ arg\ res = EP\ (from\ arg\ o\ from\ result)\ (to\ result\ o\ to\ arg))$$

5.5 Conversion Functions

The implementation of the conversion functions from a list to its generic representation and the other way around is a simple exercise in case distinction, based on the algebraic structure of the type definition.

$$from_{List}\ :: \forall\alpha.List\ \alpha \rightarrow List^{\circ}\ \alpha$$
$$from_{List}\ Nil = InL\ 1$$
$$from_{List}\ (Cons\ a\ b) = InR\ (a{\times}b)$$

$$to_{List}\ :: \forall\alpha.List^{\circ}\ \alpha \rightarrow List\ \alpha$$
$$to_{List}\ (InL\ 1) = Nil$$
$$to_{List}\ (InR\ (a{\times}b)) = Cons\ a\ b$$

The two conversion functions are grouped by $convert_{List}$.

$$convert_{List} :: \forall\alpha.\ List\ \alpha \rightleftarrows List^{\circ}_{o}\ \alpha$$
$$convert_{List} = EP\ from_{List}\ to_{List}$$

5.6 Derived Function

The last step in the derivation is to combine the specialisation on the generic representation, the conversion function and the embedded projection for the generic function.

$$adapt_{List} :: \forall\alpha.\ Eq\ (List\ \alpha) \rightarrow Eq\ (List^{\circ}\ \alpha)$$
$$adapt_{List} = epfrom\ (ep_{eq}\ convert_{List})$$

$$eq_{List} :: \forall\alpha.\ Eq\ \alpha \rightarrow Eq\ (List\ \alpha)$$
$$eq_{List}\ a = adapt_{List}\ (eq_{List^{\circ}}\ a)$$

Note that eq_{List} is a recursive function (indirectly through $eq_{List^{\circ}}$).

6 Dynamic Generic Translation

In this section we implement the dynamic generic library functions from section 4 by adapting the static generic transformations from section 5.

6.1 Basic Implementation

As can be seen from the type signatures in section 4, a *GenFun* value is passed between the library functions. It contains information about the generic function that was stored in the compiler's syntax tree in the static translation scheme. The abstract type is defined as a record with the following fields.

```
data GenFun = GenFun { arity     :: Int
                     , signature :: TypeRep
                     , instances :: FiniteMap TyCon Dynamic
                     , ep        :: Dynamic                 }
```

This record is created by the *defineGeneric* function that stores the arity and the type signature, creates an empty map of instances and constructs the embedding projection for the type signature. The *specialiseEP* function performs the specialisation for the embedding projection of the generic type signature as described in section 5.4.

```
defineGeneric :: Int → Type → GenFun
defineGeneric a s = GenFun { arity     = a
                           , signature = s
                           , instances = emptyFM
                           , ep        = specialiseEP a s}
```

The *baseInstance* function adds an instance to the map of instances.

```
baseInstance              :: TyCon → Dynamic → GenFun → GenFun
baseInstance tc dyn gf = gf {instances = addToFM (instances gf) tc dyn}
```

Finally, *specialise* replaces all type constructors in the (monomorphic) type by the corresponding instance and all type applications by *dynApply* (see section 2). This corresponds to section 5.2.

```
specialise :: GenFun → Type → Dynamic
specialise gf (TyApp t a) = dynApply (specialise gf t) (specialise gf a)
specialise gf (TyCon tc)  = case lookupFM (instances gf) tc of
                              Nothing  → derive gf tc
                              Just inst → inst
```

This is a slight simplification of the actual library function that operates on a *State* monad, adding newly derived instances to the finite map of instances in the *GenFun* record.

Now all that is left to do is implement the *derive* function. We will do so in the next section.

6.2 Functions

The dynamic function that *derive* has to construct corresponds to eq_{List} in section 5.6. Here we see the first problem: The static translation introduces new function definitions. In the dynamic setting the dynamics can contain function values and we can apply dynamics to other dynamics, but we cannot create new function definitions.

To solve this problem we enrich the term language with lambda expressions and variables.

```
data Dynamic_λ = Term Dynamic | App Dynamic_λ Dynamic_λ
               | Lambda Int Dynamic_λ | Var Int
```

In this language we can construct the derived function (the λ subscripts indicate that we are working in $Dynamic_\lambda$).

```
derive_λ       :: GenFun → TyCon → Dynamic_λ
derive_λ gf tc = foldr Lambda (adapt 'App' derived) varIds
  where
    typeDef =typeDefOf tc
    varIds  =[1..arity typeDef]
    adapt   =Term (adaptor_λ gf typeDef)
    derived =foldl App derive_λ gf typeDef) (map Var varIds)
```

The function $derive^\circ_\lambda$ constructs the derived function for the generic representation of the type definition. As we have seen in section 5.2 this is simply a matter of specialising the generic structure of the type definition. The function $adaptor_\lambda$ is more difficult and we postpone its implementation to the next subsection.

The enriched dynamics can be translated to regular dynamics by the well-known bracket abstraction algorithm that removes all lambdas and variables with the use of the S, K, and I combinators. These combinators can be defined in our term language, because dynamics can contain polymorphic functions.

$$\begin{aligned} &\mathit{derive} && :: \mathit{GenFun} \to \mathit{TyCon} \to \mathit{Dynamic} \\ &\mathit{derive}\ \mathit{gf}\ \mathit{tc} && = \mathit{bracketAbstract}\ (\mathit{derive}_\lambda\ \mathit{gf}\ \mathit{tc}) \end{aligned}$$

6.3 Pattern Matching

The function $adaptor_\lambda$ constructs the conversion function between values and their structural representation. It corresponds to $convert_{List}$ in section 5.5. Here the next problem appears.

The conversion function performs pattern matches. In the dynamic library the constructors on which we have to match are not know until run-time. In the previous function we showed how to dynamically introduce lambda expressions, but our term language does not contain pattern matching or case distinction.

Instead we use the *match* functions (see 2.1) that can be applied to the constructor info. This match function takes a value (a list in this example) and a function that should replace the constructor. If the value matches, this function is applied to the arguments of the constructor, otherwise it returns nothing. By chaining the match functions for all the constructors in a data type we can build the required conversion function.

6.4 Recursive Functions

There is one more hurdle to take. Recursive types lead to recursive functions in the translation. This means that to derive an instance for a recursive type we need the instance for this type. To escape from this loop we construct recursive functions with the use of a fix-point combinator. We could also have introduced the fix-points at the type level, this amounts to the same thing. The dynamic fix-point operator has the following definition.

$$\begin{aligned} &\mathit{fix}\ f && = \mathbf{let}\ x\ =\ f\ x\ \mathbf{in}\ x \\ &\mathit{dynFix} && :: \mathit{Dynamic} \\ &\mathit{dynFix} && = \mathbf{dynamic}\ \mathit{fix} :: \forall\alpha.(\alpha \to \alpha) \to \alpha \end{aligned}$$

Unfortunately, this fix-point combinator can only express limited forms of recursion. The type of *fix* shows that the recursive calls should all have the same type as the function itself. On the type level this means that this method does not work for non-uniform types, such as

$$\mathbf{data}\ \mathit{Nested}\ \alpha\ =\ \mathit{One} \mid \mathit{Two}\ (\mathit{Nested}\ (\alpha,\ \alpha))$$

In the static scenario instances for non-uniform types can only be expressed because Haskell supports polymorphic recursion.

Perhaps these non-uniform types can be handled with more advanced fix-point combinators, but the details have not been worked out.

7 Applications and Extensions

We present some examples of the use of the library, describe some extensions and discuss the efficiency of our solution.

7.1 Defining the Instance for Dynamic

In section 4 we saw how to derive an equality function to compare two dynamics. In the example below this example is extended to define a base instance of the static generic equality function for the type *Dynamic*.

$$
\begin{aligned}
&eq_{Dynamic}\ x@(\mathbf{dynamic}\ \_ :: a)\ y@(\mathbf{dynamic}\ \_ :: a) = eqDyn\ \lceil a \rceil\ x\ y\\
&\quad \mathbf{where}\\
&\qquad convert_{Dyn} :: \forall \alpha.\ Typeable\ \alpha \Rightarrow \alpha \rightleftarrows Dynamic\\
&\qquad convert_{Dyn} = EP\ toDyn\ fromDyn\\
&\\
&\qquad eqDyn\ type = liftDynEq\ (genEq\ type)\\
&\\
&\qquad liftDynEq :: Dynamic \rightarrow Eq\ Dynamic\\
&\qquad liftDynEq = \lambda(dynamic\ eq :: Eq\ a) \rightarrow epfrom\ (ep\{Eq\}\ convert_{Dyn})\ eq\\
&eq_{Dynamic}\ \_\ \_ = False
\end{aligned}
$$

The first alternative of $eq_{Dynamic}$ only applies if the two dynamics have a matching type. In that case the representation of this type is used to specialise the dynamic generic equality (with the function *genEq* from section 4.3). The *liftDynEq* function transforms the equality function in the dynamic (type *Eq a*) to an equality function on two values of type *Dynamic*. Such a lift function can be defined for any generic function in a similar way.

7.2 Deriving a Generic Function for the Types

So far we have only looked at how the generic function can operate on the values in the dynamics. But we also have to consider the type in the dynamic. A generic pretty printer for dynamics should not only print the value in the dynamic, but also its type.

```
  generic pprint t :: t -> String
  pprint (dynamic Cons 1 Nil :: List Int)
⇒ "dynamic Cons 1  Nil :: List Int"
```

A naive specialisation of the pretty printer for the representation of the type gives the rather unsatisfactory result `"TyApp (TyCon List) (TyCon Int)"`.

The library provides a function that helps in this situation.

$$specialiseForType :: [TypCon] \rightarrow GenFun \rightarrow Dynamic$$

In the case of the pretty printer the dynamic constructed *specialiseForType* contains a pretty printer of type $TypeRep \rightarrow String$, but it behaves as if it were defined on the type universe that is formed by the list of type constructors.

For example for the types *Int*, *Bool* and *List* this universe can be presented by the following algebraic type.

data *Type* = *Int* | *Bool* | *List Type*

Note that *Int*, *Bool* and *List* are data constructors in this type.

The library function *specialiseForType* can be used for many other generic functions. A parser for dynamics can first apply the parser generated with *specialiseForType* to parse the type string. This parser delivers a representation of the type which is then used to construct the parser for the value string. In test data generation first a type can be generated and then a value of this type. The graphical editor for dynamic values from the introduction can also be extended so that the user can also edit the type as well as the values for that type.

7.3 Error Handling

So far we have ignored the errors that can occur during the dynamic construction of generic functions. Compile-time errors from the static framework have become run-time errors in our library and this means that all the library functions we have used so far are inherently partial.

The *defineGeneric* function can fail if there is no embedding projection defined for one of the type constructors in the signature of the dynamic function. The *baseInstance* function can fail if the type of the function in the dynamic does not correspond to the type signature of the dynamic function. The *specialise* function can fail if the instance for a type cannot be derived, for example because it is an abstract type.

The library provide versions of all the functions that return proper error codes in case something goes wrong. Because of the explicit manner in which the generic functions are constructed in the library, the application programmer can use the error codes to recover from the situation.

7.4 Efficiency

The efficiency of the dynamically constructed generic functions is in the same order as the efficiency of unoptimised static generic functions. The construction of functions with combinators may seem costly, but under graph rewriting semantics each introduced combinator is only evaluated once.

A compiler does have more optimisation opportunities. Fusion for example has proved to be powerful enough to completely remove the overhead of the construction of the generic representation of values for most generic functions [5, 6]. This optimisation is not possible in our dynamic setting. The library cannot analyse the base instances that are provided by the programmer, because these dynamics contain compiled code.

8 Related Work

Earlier work by one of the authors [2] can be seen as a prequel to the present paper. In that paper the representation of types is also used implement generic function on dynamics, but it assumed compiler support to generate many of the functions that are constructed at run-time in the current approach. The system was limited to generics function with one generic variable.

Cheney and Hinze [7] combined dynamics and generics from the outset. Their implementation is lightweight in the number of language features that are used. The dynamics already contain values with the generic structure and the programmer has to write the conversions functions between values and their generic representation. The dynamics in the current paper contain the actual values with sharing fully preserved, which makes them more efficient.

The "Boilerplate" articles [11, 12] use the same run-time information about types and type definitions to build generic traversal schemes. Because this information is present in dynamics the traversal schemes can also be applied to the values in dynamics. The library presented in the current paper makes the approach from Generic Haskell or Clean available for dynamic values, but the library does require a more powerful dynamic typing system (dynamics with polymorphic types and run-time unification). Many functions can be implemented with either system and experience will have to show which approach is more convenient in what situation.

9 Conclusions and Future Work

We have developed a library in Clean that enables a programmer to create an instance for a generic function for values of type Dynamic. A dynamic can contain any value of any type which can both be inspected at run-time using a pattern match. Dynamics can be stored on disk or send to another application over the internet.

Using our new library, one is now able at run-time to apply generic functions on dynamics of any value and (almost) any type. Such a dynamic might even have been created by other applications. One cannot only apply "consuming" generic functions like equality and pretty printing, but also typical "producing" generic functions like parsers. Furthermore, one cannot only define generic functions on values but one can define generic functions on their types as well. It is possible, for example, to create a generic editor to edit a type stored in a dynamic. It can be used to compose a new type using the available ones. Now one can create another generic editor to construct a value of this newly constructed type.

The library is very easy to use for someone familiar with the static generic approach. The definition of a dynamic generic function can be given in a very mechanical way. It is even imaginable that the dynamic definition can be created automatically by a compiler from the static description.

The library is implemented in Clean. The implementation actually provides a run-time variant of the static generic transformation scheme as implemented in the Clean compiler. To realise this, one among others has to be able to construct

new functions at run-time. We have accomplished this by using bracket abstraction. For dealing with recursive types one has to be able to construct recursive functions for which we have used a fix-point combinator. Currently we can only deal with uniform recursive types.

In principle it should be possible to adopt our library for Haskell if the dynamic typing system would be more powerfull. Our solution needs dynamics that contain polymorphic types and run-time unification.

In the future we would like to investigate if it is possible to remove the current restriction that dynamic generic functions cannot be applied to non-uniform recursive types. Furthermore we want to create some larger applications to test the library. Feedback from our users is highly appreciated.

Acknowledgement

Many thanks to Artem Alimarine for valuable discussions and the anonymous referees for numerous suggestions for improvement.

References

1. M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, and D. Rèmy. Dynamic typing in polymorphic languages. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992.
2. P. Achten, A. Alimarine, and R. Plasmeijer. When generic functions use dynamic values. In R. Peña, editor, *The 14th International workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 17–33. Madrid, Spain, Springer, Sept. 2002.
3. Achten, Peter, van Eekelen, Marko and Plasmeijer, Rinus. Generic Graphical User Interfaces. In Greg Michaelson and Phil Trinder, editors, *Selected Papers of the 15th Int. Workshop on the Implementation of Functional Languages, IFL03*, volume 3145 of *LNCS*. Edinburgh, UK, Springer, 2003.
4. A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, Sept. 2002.
5. A. Alimarine and S. Smetsers. Optimizing generic functions. In D. Kozen, editor, *The 7th International Conference, Mathematics of Program Construction*, number 3125 in LNCS, pages 16 – 31. Stirling, Scotland, UK, Springer, July 2004.
6. A. Alimarine and S. Smetsers. Improved fusion for optimizing generics. In M. Hermenegildo and D. Cabeza, editors, *Proceedings of Seventh International Symposium on Practical Aspects of Declarative Languages*, number 3350 in LNCS, pages 203 – 218. Long Beach, CA, USA, Springer, Jan. 2005.
7. J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics, 2002.
8. R. Hinze. A new approach to generic functional programming. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. Boston, Massachusetts, January 2000.

9. R. Hinze and S. Peyton Jones. Derivable Type Classes. In G. Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop*, volume 41(1) of *ENTCS*. Montreal, Canada, Elsevier Science, 2001.
10. P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.
11. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
12. R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings; International Conference on Functional Programming (ICFP 2004)*. ACM Press, Sept. 2004. 12 pages; To appear.
13. S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.
14. M. Pil. Dynamic types and type dependent functions. In K. Hammond, T. Davie, and C. Clack, editors, *Implementation of Functional Languages (IFL '98)*, LNCS, pages 169–185. Springer Verlag, 1999.
15. R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.kun.nl/∼clean/contents/contents.html.
16. A. van Weelden and R. Plasmeijer. Towards a strongly typed functional operating system. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 215–231. Springer, Sept. 2003.
17. M. Vervoort and R. Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In R. Peña and T. Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 101–117. Springer, Sept. 2003.