When Generic Functions Use Dynamic Values

Peter Achten, Artem Alimarine, and Rinus Plasmeijer

Computing Science Department, University of Nijmegen, 1 Toernooiveld, 6525 ED, Nijmegen, The Netherlands

Abstract. Dynamic types allow strongly typed programs to link in external code *at run-time* in a type safe way. Generic programming allows programmers to write code schemes that can be specialized *at compile-time* to arguments of arbitrary type. Both techniques have been investigated and incorporated in the pure functional programming language Clean. Because generic functions work on all types and values, they are the perfect tool when manipulating dynamic values. But generics rely on compile-time specialization, whereas dynamics rely on run-time type checking and linking. This seems to be a fundamental contradiction. In this paper we show that the contradiction does not exist. From any generic function we derive a function that works on dynamics, and that can be parameterized with a dynamic type representation. Programs that use this technique combine the best of both worlds: they have concise universal code that can be applied to any dynamic value regardless of its origin. This technique is important for application domains such as type-safe mobile code and plug-in architectures.

1 Introduction

In this paper we discuss the interaction between two recent additions to the pure, lazy, functional programming language Clean 2.0(.1) [5, 10, 14]:

Dynamic types Dynamic types allow strongly typed programs to link in external code (*dynamics*) *at run-time* in a type safe way. Dynamics can be used anywhere, regardless from the module or even application that created them. Dynamics are important for type-safe applications with *mobile code* and *plug-in* architectures.

Generic programming enables us to write general function schemes that work for any data type. From these schemes the compiler can derive automatically any required instance of a specific type. This is possible because of Clean's strong type system. Generic programs are a compact way to elegantly deal with an important class of algorithms. To name a few, these are *comparison*, *pretty printers*, *parsers*.

In order to apply a generic function to a dynamic value in the current situation, the programmer should to do an exhaustive type pattern-match on all possible dynamic types. Apart from the fact that this is impossible, this is at odds with the key idea of generic programming in which functions *do* an exhaustive distinction on types, but on their finite (and small) structure.

One would imagine that it is alright to apply a generic function to any dynamic value. Consider for instance the application of the generic equality function to two dynamic values. Using the built-in dynamic type unification, we can easily check the equality of the *types* of the dynamic values. Now using a generic equality, we want to check the equality of the *values* of these dynamics. In order to do this, we need to know at *compile-time* of which type the instance of the generic equality should be applied. This is not possible, because the type representation of a dynamic is only known at *run-time*.

We present a solution that uses the current implementation of generics and dynamics. The key to the solution is to guide a generic function through a dynamic value using an explicit type representation of the dynamic value's type. This guide function is predefined once. The programmer writes generic functions as usual, and in addition provides the explicit type representation.

The solution can be readily used with the current compiler if we assume that the programmer includes type representations with dynamics. However, this is at odds with the key idea of dynamics because these already store type representations with values. We show that the solution also works for conventional dynamics if we provide a low-level access function that retrieves the type representation of any dynamic.

Contributions of this paper are:

– We show how one can combine generics and dynamics in one single framework in accordance with their current implementation in the compiler.
– We argue that, in principle, the type information available in dynamics is enough, so we do not need to store extra information, and instead work with conventional dynamics.
– Programs that exploit the combined power of generics and dynamics are universally applicable to dynamic values. In particular, the code handles dynamics in a generic way without precompiled knowledge of their types.

In this paper we give introductions to dynamics (Section 2) and generics (Section 3) with respect to core properties that we rely on. In Section 4 we show our solution that allows the application of generic functions to dynamic values. An example of a generic pretty printing tool is given to illustrate the expressive power of the combined system (Section 5). We present related work (Section 6), our current and future plans (Section 7), and conclude (Section 8).

2 Dynamics in Clean

The Clean system has support for *dynamics* in the style as proposed by Pil [12, 13]. Dynamics serve two major purposes:

Interface between static and run-time types: Programs can convert values from the statically typed world to the dynamically typed world and back without loss of type security. Any Clean expression e that has (verifiable or inferable) type t can be formed into a value of type `Dynamic` by: `dynamic` e `::` t, or: `dynamic` e. Here are some examples:

```
toDynamic :: [Dynamic]
toDynamic = [e1, e2, e3, dynamic [e1,e2,e3]]
where  e1 = dynamic 50                 :: Int
       e2 = dynamic reverse            :: A.a: [a] -> [a]
       e3 = dynamic reverse ['a'..'z'] :: [Char]
```

Any `Dynamic` value can be matched in function alternatives and case expressions. A 'dynamic pattern match' consists of an expression pattern *e-pat* and a type pattern *t-pat* as follows: `(`*e-pat*`::`*t-pat*`)`. Examples are:

```
dynApply :: Dynamic Dynamic -> Dynamic
dynApply (f::a -> b) (x::a) = dynamic (f x) :: b
dynApply _            _      = abort "dynApply: arguments of wrong type."

dynSwap :: Dynamic -> Dynamic
dynSwap ((x,y) :: (a,b)) = dynamic (y,x) :: (b,a)
```

It is important to note that *unquantified* type pattern variables do not indicate polymorphism. Instead, they are bound to the offered type, and range over the full function alternative.
Finally, *type-dependent* functions are a flexible way of *parameterizing* functions with the type to be matched in a dynamic. Type-dependent functions are overloaded in the `TC` class, which is a built-in class that basically represents all *type codeable* types. The overloaded argument can be used in a dynamic type pattern by postfixing it with `^`. Typical examples that are also used in this paper are the packing and unpacking functions:

```
pack :: a -> Dynamic | TC a
pack x = dynamic x::a^

unpack :: Dynamic -> a | TC a
unpack (x::a^) = x
unpack _       = abort "unpack: argument of wrong type."
```

Serialization: At least as important as switching between compile-time and run-time types, is that dynamics allow programs to *serialize* and *deserialize* values without loss of type security. Programs can work safely with data and code that do not originate from themselves.
Two library functions store and retrieve dynamic values in named files, given a proper unique environment that supports file I/O:

```
writeDynamic :: String Dynamic
                       *env -> (Bool,*env)         | FileSystem env
readDynamic  :: String *env -> (Bool,Dynamic,*env) | FileSystem env
```

> Making an effective and efficient implementation is hard work and requires careful design and architecture of the compiler and run-time system. It is not our intention to go into any detail of such a project, as these are presented in [15]. What needs to be stressed in the context of this paper is that dynamic values, when read in from disk, contain a binary representation of a complete Clean computation graph, a representation of the compile-time type, and references to the related rewrite rules. The programmer has no means of access to these representations other than those explained above.

At this stage, the Clean 2.0.1 system restricts the use of dynamics to *basic*, *algebraic*, *record*, *array*, and *function* types. Very recently, support for polymorphic functions has been added. Overloaded types and overloaded functions have been investigated by Pil [13]. Generics obviously haven't been taken into account, and that is what this paper addresses.

3 Generics in Clean

The Clean approach to generics [3] combines the polykinded types approach developed by Hinze [7] and its integration with overloading as developed by Hinze and Peyton Jones [8]. A generic function basically represents an infinite set of overloaded classes. Programs define for which types instances of generic functions have to be generated. During program compilation, all generic functions are converted to a finite set of overloaded functions and instances. This part of the compilation process uses the available compile-time type information.

As an example, we show the generic definition of the ubiquitous equality function. It is important to observe that a generic function is defined in terms of *both* the type *and* the value. The signature of equality is:

```
generic gEq a :: a a -> Bool
```

This is the type signature that has to be satisfied by an instance for types of kind $\star$ (such as the basic types `Bool`ean, `Int`eger, `Real`, `Char`acter, and `String`). The generic implementation compares the values of these types, and simply uses the standard overloaded equality operator `==`. In the remainder of this paper we only show the `Int`eger case, as the other basic types proceed analogously.

```
gEq{|Int|} x y = x == y
```

Algebraic types are constructed as sums of pairs – or the empty unit pair – of types. It is useful to have information (name, arity, priority) about data constructors. For brevity we omit record types. The data types that represent sums, pairs, units, and data constructors are collected in the module `StdGeneric.dcl`:

```
::  EITHER a b = LEFT  a | RIGHT b
::  PAIR   a b = PAIR  a b
::  UNIT       = UNIT
::  CONS   a   = CONS  a
```

The built-in function type constructor `->` is reused here. The kind of these cases (`EITHER`, `PAIR`, `->` : $\star \to \star \to \star$, `UNIT` : $\star$, and `CONS` : $\star \to \star$) determines the number and type of the higher-order function arguments of the generic function definition. These are used to compare the sub structures of the arguments.

```
gEq{|UNIT|}          UNIT         UNIT         = True
gEq{|PAIR|}   fx fy (PAIR x1 y1) (PAIR x2 y2) = fx x1 x2 && fy y1 y2
gEq{|EITHER|} fx fy (LEFT  x1)   (LEFT  x2)   = fx x1 x2
gEq{|EITHER|} fx fy (RIGHT y1)   (RIGHT y2)   = fy y1 y2
gEq{|EITHER|} _  _  _            _            = False
gEq{|CONS|}   f     (CONS  x)    (CONS  y)    = f x y
```

The only case that is missing here is the function type `->`, as one cannot define a feasible implementation of function equality.

Programs must ask explicitly for an instance of type T of a generic function g by: `derive` g T. This provides the programmer with a *kind-indexed* family of functions $\mathtt{g}_{\star}$, $\mathtt{g}_{\star\to\star}$, $\mathtt{g}_{\star\to\star\to\star}$, ... The function g_κ is denoted as: g`{|`κ`|}`. The programmer can parameterize g_κ for any $\kappa \neq \star$ to customize the behaviour of g. As an example, consider the standard binary tree type *:: MyTree a = Leaf* | *Node (MyTree a) a (MyTree a)* and let *a = Node Leaf 5 (Node Leaf 7 Leaf)*, and *b = Node Leaf 2 (Node Leaf 4 Leaf)*. The expression ($\mathtt{gEq}_{\star}$ *a b*) applies integer equality to the elements and hence yields false, whereas ($\mathtt{gEq}_{\star\to\star}$ *(const o const True) a b*) applies the binary true constant function, and hence yields true.

4 Dynamics + Generics in Clean

In this section we show how we made it possible for programs to manipulate *dynamics* by making use of *generic* functions. Suppose we want to apply the generic equality function of Section 3 to two dynamics, as mentioned in Section 1. One would expect the following definition to work:

```
dynEq :: Dynamic Dynamic -> Bool    // This code is incorrect.
dynEq (x::a) (y::a) = gEq{|*|} x y
dynEq _ _           = False
```

However, this is not the case because at compile-time it is impossible to check if the required instance of `gEq` exists, or to derive it automatically simply because of the absence of the proper compile-time type information.

In our solution, the following code is used:

```
dynEq :: Dynamic Dynamic -> Bool    // This code is correct.
dynEq x=:(_::a) y=:(_::a)    = gEq_dyn (dynTypeRep x) x y
dynEq _ _                    = False
```

Two new functions have come into existence: $\mathtt{gEq}_{\mathtt{dyn}}$ and `dynTypeRep`. The first is a derived function of type *Type Dynamic Dynamic* $\to$ *Bool*; the second is a low-level access function of type *Dynamic* $\to$ *Type*. The crucial difference with the incorrect program is that $\mathtt{gEq}_{\mathtt{dyn}}$ works on the complete dynamic.

We obtain `gEq`$_{\texttt{dyn}}$ by *specializing* `gEq` for a certain type τ. Specialization can be done by a single function *specialize* that is parameterized with a generic function and a type, and that returns the instance of the function for the given type, packed as a dynamic. This requires a suitable representation of types and generic functions. We encode types with a new type *TypeRep* and pack it in a dynamic such that all values $(t :: \mathit{TypeRep}\ \tau) :: \mathit{Dynamic}$ satisfy the invariant that t is the type representation of τ (Section 4.1). For readability we introduce the synonym type *Type* for these dynamics. Generic functions are really code schemes, and therefore we need to wrap them in a suitable form in order to pass them to the specialization function. We show how to do this in Section 4.2. A wrapped generic function is of type *GenRec*. The result of $\mathit{specialize} :: \mathit{GenRec}\ \mathit{Type} \rightarrow \mathit{Dynamic}$ (Section 4.3) is easily transformed to one that works on dynamics (Section 4.4). For our `gEq` case, this is `gEq`$_{\texttt{dyn}}$.

In Section 4.5 we show that *specialization* is sufficient to handle all generic and non-generic functions on dynamics. However, it forces programmers to work with *Dynamic*s that are extended with the proper *Type*. An elegant solution is obtained with the low-level access function `dynTypeRep` which retrieves *Type*s from *Dynamic*s, and can therefore be used instead (Section 4.6).

We want to stress the point that except for the `dynEq` function all code can be derived automatically by the compiler. However, this is not currently incorporated, so for the time being the result code needs to be included manually.

The remainder of this section fills in the details of the scheme as sketched above. We continue to illustrate every step with the `gEq` example. When speaking in general terms, we assume that we have a function g that is generic in argument a and has type $(G\ a)$ (so $g =$ `gEq`, and $G\ a =$ `Eq a` defined as `:: Eq a :== a a -> Bool`). We will have a frequent need for conversions from type a to b and vice versa. These are conveniently combined into a record of type *Bimap* $a\ b$ (see Appendix A for its type definition and the standard bimaps that we use).

4.1 Dynamic type representations

Dynamic type representations are dynamics of synonym type *Type* containing values $(t :: \mathit{TypeRep}\ \tau)$ such that t represents τ, with *TypeRep* defined as:

```
::  TypeRep t
 =  TRInt
 |  TREither Type Type | TRPair Type Type | TRUnit | TRArrow Type Type
 |  TRCons String Int Type | TRType [Type]     // [TypeRep a1,... TypeRep an]
                                      Type     // (TypeRep T°)
                                    Dynamic    // (Bimap T T°)
```

For each data constructor (`TR`$C\ t_1 \ldots t_n$) ($n \leq 0$) we provide a n-ary *constructor* function `tr`C of type $\mathit{Type} \ldots \mathit{Type} \rightarrow \mathit{Type}$ that assembles the corresponding alternative, and establishes the relation between representation and type. For basic types and the cases that correspond with generic representations (*sum*, *pair*, *unit*, and *arrow*), these are straightforward and proceed as follows:

```
trInt :: Type
trInt = dynamic TRInt :: TypeRep Int

trEither :: Type Type -> Type
trEither tra=:(_::TypeRep a) trb=:(_::TypeRep b)
      = dynamic (TREither tra trb) :: TypeRep (EITHER a b)

trArrow :: Type Type -> Type
trArrow tra=:(_::TypeRep a) trb=:(_::TypeRep b)
      = dynamic (TRArrow tra trb) :: TypeRep (a -> b)
```

The last two alternatives of the dynamic type representation handle all custom type definitions. This is necessary because our solution relies on the fact that every dynamic value includes a dynamic type representation of the value's type. Suppose we have a type constructor $T\ a_1 \ldots a_n$ with a data constructor $C\ t_1 \ldots t_m$. The *TRCons* alternative collects the *name* and *arity* of its data constructor. This is the same information a programmer might need when handling the `CONS` case of a generic function (although in the generic equality example we had no need for it). The *TRType* alternative keeps track of the type representations of the type arguments a_i (*TypeRep* a_i), the dynamic type representation of the 'standard' generic representation (*TypeRep* T°), and the bimap of T and T°. The corresponding constructor functions follow from the type definition:

```
trCons :: String Int Type -> Type
trCons name arity tra=:(_:: TypeRep a)
    = dynamic (TRCons name arity tra) :: TypeRep (CONS a)

trType :: [Type] Type Dynamic -> Type
trType args tg=:(_::TypeRep tgen) conv=:(_::Bimap t tgen)
    = dynamic (TRType args tg conv) :: TypeRep t
```

As a first example, consider the Clean *list* type constructor. Clean lists are defined internally as `:: [] a = _Cons a [a] | _Nil`. Generically speaking they are a *sum* of: *(a)* the data *cons*tructor (`_Cons`) of the *pair* of the element type and the list itself, and *(b)* the data *cons*tructor (`_Nil`) of the *unit*. This structure is reflected by the generated dynamic type representation of the list data constructor *trListG* below, which is also used by the generated dynamic type representation of the list type constructor *trList*.

```
trListG :: Type -> Type
trListG tra
    = trEither  (trCons "_Cons" 2 (trPair tra (trList tra)))      (a)
                (trCons "_Nil" 0 trUnit)                          (b)

::  ListG a :== EITHER (CONS (PAIR a [a])) (CONS UNIT)

trList :: Type -> Type
trList tra=:(_::TypeRep a)
    = trType [tra] (trListG tra) (dynamic epList :: Bimap [a] (ListG a))
```

```
where epList = { map_to = map_to, map_from = map_from }
      map_to [x:xs] = LEFT  (CONS (PAIR x xs))
      map_to []     = RIGHT (CONS UNIT)
      map_from (LEFT  (CONS (PAIR x xs))) = [x:xs]
      map_from (RIGHT (CONS UNIT      ))  = []
```

As a second example, we show the dynamic type representation for our running example, the equality function which has type `Eq a`:

```
trEq :: Type -> Type
trEq tra=:(_::TypeRep a)
    = trType [tra]
             (trArrow tra (trArrow tra trBool))
             (dynamic bimapId :: Bimap (Eq a) (Eq a))
```

4.2 First-class generic functions

In this section we show how to turn generic functions, that are really compiler schemes, into first-class values that can be passed to functions for inspection. The first-class representation of g is obtained simply by taking the desired instance of g per case, so the *Int* case is g_{Int}, *UNIT* is g_{UNIT}, and so on. Each of these cases is packed into a dynamic, and they are collected in the *GenRec* record. (The compiler will actually *inline* the corresponding right-hand side of g.) In general, this results in:

```
genrec_g :: GenRec
genrec_g
    = { genConvert= dynamic convertG     (Section 4.3)
      , genType   = trG                  (Section 4.1)
      , genInt    = dynamic g_Int        :: G Int
      , genUNIT   = dynamic g_UNIT       :: G UNIT
      , genPAIR   = dynamic g_PAIR       :: A.a b: G a -> G b -> G (PAIR   a b)
      , genEITHER = dynamic g_EITHER     :: A.a b: G a -> G b -> G (EITHER a b)
      , genARROW  = dynamic g_→          :: A.a b: G a -> G b -> G ((->)   a b)
      , genCONS   = \n a -> dynamic g_CONS
                                         :: A.a  : G a -> G (CONS a)
      }
```

The generated code for `gEq` is:

```
genrec_gEq :: GenRec
genrec_gEq
    = { genConvert= dynamic convertEq
      , genType   = trEq
      , genInt    = dynamic gEq{|*|}        :: Eq Int
      , genUNIT   = dynamic gEq{|*|}        :: Eq UNIT
      , genPAIR   = dynamic gEq{|*->*->*|}  :: A.a b: Eq a -> Eq b
                                                      -> Eq (PAIR   a b)
      , genEITHER = dynamic gEq{|*->*->*|}  :: A.a b: Eq a -> Eq b
```

```
                                                      -> Eq (EITHER a b)
, genARROW  = dynamic gEq{|*->*->*|}  :: A.a b: Eq a -> Eq b
                                                      -> Eq ((->)   a b)
, genCONS   = \n a -> dynamic gEq{|*->*|}
                                      :: A.a  : Eq a -> Eq (CONS   a  )
}
```

4.3 Specialization of first-class generics

Every generic function g can be passed around to functions as $genrec_g$, and every type τ as $(t :: \mathit{TypeRep}\ \tau) :: \mathit{Type}$. This puts us in the position to provide a function, called *specialize*, that takes such a generic function representation and a dynamic type representation, and which yields $g_\tau :: G\ \tau$, packed in a conventional dynamic. This function has type $\mathit{GenRec}\ \mathit{Type} \rightarrow \mathit{Dynamic}$. Its definition is a case distinction based on the dynamic type representation. The basic types and the generic *unit* case are easy:

```
specialize genrec (TRInt :: TypeRep Int)  = genrec.genInt
specialize genrec (TRUnit:: TypeRep UNIT) = genrec.genUNIT
```

The generic cases (*sum*, *pair*, *arrow*, and *cons*tructor) specialize $genrec_g$ with the dynamic type representations of the argument types. These are obtained via the `genType` field of the `genrec` structure. The recursive specialization of $genrec_g$ to the sub structures of the generic cases yield the higher-order arguments of the corresponding alternatives of $genrec_g$. The definition is surprisingly elegant. (The constructor case deviates a little because it passes around the name and arity information.)

```
specialize genrec ((TREither tra trb) :: TypeRep (EITHER a b))
    = applyGenCase2 (genrec.genType    tra) (genrec.genType    trb)
                    genrec.genEITHER
                    (specialize genrec tra) (specialize genrec trb)
specialize genrec ((TRPair tra trb) :: TypeRep (PAIR a b))
    = applyGenCase2 (genrec.genType    tra) (genrec.genType    trb)
                    genrec.genPAIR
                    (specialize genrec tra) (specialize genrec trb)
specialize genrec ((TRArrow tra trb) :: TypeRep (a -> b))
    = applyGenCase2 (genrec.genType    tra) (genrec.genType    trb)
                    genrec.genARROW
                    (specialize genrec tra) (specialize genrec trb)
specialize genrec ((TRCons name arity tra) :: TypeRep (CONS a))
    = applyGenCase1 (genrec.genType    tra)
                    (genrec.genCONS name arity)
                    (specialize genrec tra)

applyGenCase1 :: Type Dynamic Dynamic -> Dynamic
applyGenCase1 (a :: TypeRep fa) (fta :: fa -> fta) dfa
    = dynamic fta (unwrapTR a dfa) :: fta
```

```
applyGenCase2 :: Type Type Dynamic Dynamic Dynamic -> Dynamic
applyGenCase2 (trfa :: TypeRep fa) (trfb :: TypeRep fb)
              (ftab :: fa fb -> ftab)
              dfa dfb
    = dynamic ftab (unwrapTR trfa dfa) (unwrapTR trfb dfb) :: ftab

unwrapTR :: (TypeRep a) Dynamic -> a | TC a
unwrapTR _ (x :: a^) = x
```

Type constructors are more complicated because the packed bimap between a and a° needs to be transformed to $(G\ a)$. This conversion is done by *convertG* below, and is also included in the generic representation of g in the `genConvert` field (Section 4.2). Let `dynApply2` be the 2-ary version of `dynApply`, then:

```
specialize genrec ((TRType args tr ep) :: TypeRep a)
    = dynApply2 genrec.genConvert ep (specialize genrec tr)
```

The definition of `convert`G has a standard form, namely:

`convert`G `:: (Bimap a b)` $\rightarrow (G\ \ b) \rightarrow (G\ \ a)$
`convert`G `ep = (bimap`G `ep).map_from`

The function body of `bimap`G `a` is derived from the structure of the type term $G\ a$: `bimap`G `a =` $\langle G\ \ a\rangle$ with the following rules:

$$
\begin{array}{lll}
\langle x\rangle & = x & \text{(type variables, including } a\text{)} \\
\langle t_1 \rightarrow t_2\rangle & = \langle t_1\rangle \texttt{ --> } \langle t_2\rangle & \\
\langle c\ t_1 \ldots t_n : \kappa\rangle & = \texttt{bimapId} & \textbf{if } a \notin \bigcup Var(t_i) (n \geq 0) \\
 & = \texttt{bimapId\{|}\kappa\texttt{|\}}\ \langle t_1\rangle \ldots \langle t_n\rangle & \textbf{otherwise}
\end{array}
$$

Appendix A defines `-->` and `bimapId`; *Var* yields the variables of a type term. The generated code for `convert`Eq and `bimap`Eq is:

```
convertEq :: (Bimap a b) -> (Eq b) -> (Eq a)
convertEq ep = (bimapEq ep).map_from

bimapEq :: (Bimap a b) -> Bimap (a -> a -> c) (b -> b -> c)
bimapEq ep = ep --> ep --> bimapId
```

4.4 Generic dynamic functions

In the previous section we have shown how the *specialize* function uses a dynamic type representation as a 'switch' to construct the required generic function g, packed in a dynamic. We now transform such a function into a function of type (G *Dynamic*), using a function g_{dyn}. This function g_{dyn} takes the same dynamic type representation argument as *specialize*. Its body invariably takes the following form (`bimapDynamic` and `inv` are included in Appendix A):

```
g_dyn :: Type -> G Dynamic
g_dyn tr = case specialize genrec_g tr of
           (f::G a)    -> convertG (inv bimapDynamic) f
```

As discussed in the previous section, `convert`G transforms a *(Bimap a b)* to a conversion function of type $(G\ b) \rightarrow (G\ a)$. When applied to $(\mathit{inv\ bimapDynamic}) :: (\mathit{Bimap\ Dynamic}\ a)$, it results in a conversion function of type $(G\ a) \rightarrow (G\ \mathit{Dynamic})$. This is applied to the packed generic function $f :: G\ a$, so the result function has the desired type $(G\ \mathit{Dynamic})$.

When applied to our running example, we obtain:

```
gEq_dyn :: Type -> Eq Dynamic
gEq_dyn tr = case specialize genrec_gEq tr of
           (f::Eq a)    -> convertEq (inv bimapDynamic) f
```

4.5 Applying generic dynamic functions

The previous section shows how to obtain a function g_{dyn} from a generic function g of type $(G\ a)$ that basically applies g to dynamic arguments, assuming that these arguments internally have the same type a. In this section we show that with this function we can handle all generic and non-generic functions on dynamics. In order to do so, we require the programmer to work with *extended* dynamics, defined as:

```
::  DynamicExt = DynExt Dynamic Type
```

An extended dynamic value $(\texttt{DynExt}(v :: \tau)(t :: \mathit{TypeRep}\ \tau))$ basically is a pair of a *conventional Dynamic* $(v :: \tau)$ *and* its dynamic type representation $(t :: \mathit{TypeRep}\ \tau)$. Note that we make effective use of the built-in unification of dynamics to enforce that the dynamic type representation really is the same as the type of the conventional dynamic.

For the running example `gEq` we can now write an equality function on extended dynamics, making use of the generated function $\texttt{gEq}_{\texttt{dyn}}$:

```
dynEq :: DynamicExt DynamicExt -> Bool
dynEq (DynExt x=:(_::a) tx) (DynExt y=:(_::a) _) = gEq_dyn tx x y
dynEq _ _                                        = False
```

It is the task of the programmer to handle the cases in which the (extended) dynamics do not contain values of the proper type. This is an artefact of dynamic programming, as we can never make assumptions about the content of dynamics.

Finally, we show how to handle *non-generic* dynamic functions, such as the `dynApply` and `dynSwap` in Section 2. These examples illustrate that it is possible to maintain the invariant that extended dynamics always have a dynamic type representation of the type of the value in the corresponding conventional dynamic. It should be observed that these non-generic functions are basically *monomorphic* dynamic functions due to the fact that unquantified type pattern variables are implicitly existentially quantified. The function `wrapDynamicExt` is a predefined function that conveniently packs a conventional dynamic and the corresponding dynamic type representation into an extended dynamic.

```
dynApply :: DynamicExt DynamicExt -> DynamicExt
dynApply (DynExt (f::a->b) ((TRArrow tra trb) :: TypeRep (a->b)))
         (DynExt (x::a) _)
    = wrapDynamicExt (f x) trb

dynSwap :: DynamicExt -> DynamicExt
dynSwap (DynExt ((x,y)::(a,b)) ((TRType [tra,trb] _ _) :: TypeRep (a,b)))
    = wrapDynamicExt (y,x) (trTuple2 trb tra)

wrapDynamicExt :: a Type -> DynamicExt | TC a
wrapDynamicExt x tr=:(_::TypeRep a^) = DynExt (dynamic x::a^) tr
```

4.6 Elimination of extended dynamics

In the previous section we have shown how we can apply generic functions to conventional dynamics if the program manages *extended* dynamics. We emphasized in Section 2 that every conventional dynamic stores the representation of all compile-time types that are related to the type of the dynamic value [15]. This enables us to write a low-level function *dynTypeRep* that computes the dynamic type representation as given in the previous section from any dynamic value. Informally, we can have:

```
dynTypeRep :: Dynamic -> Type
dynTypeRep (x::t) = dynamic (tr::TypeRep t)
```

If we assume that we have this function (future work), we do not need the extended dynamics anymore. The `dynEq` function can now be written as:

```
dynEq :: Dynamic Dynamic -> Bool
dynEq x=:(_::a) y=:(_::a)    = gEq_dyn (dynTypeRep x) x y
dynEq _ _                    = False
```

The signature of this function suggests that we might be able to derive dynamic versions of generic functions automatically as just another instance. Indeed, for type schemes $G\ a$ in which a appears at an argument position, there is always a dynamic argument from which a dynamic type representation can be constructed. However, such an automatically derived function is necessarily a *partial* function when a appears at more than one argument position, because one cannot decide what the function should do in case the dynamic arguments have non-matching contents. In addition, if a appears only at the result position, then the type scheme is not an instance of $G\ Dynamic$, but rather $Type \to G\ Dynamic$.

5 Example: a pretty printer

Pretty printers belong to the classic examples of generic programming. In this section we deviate a little from this well-trodden path by developing a program that sends a graphical version of any dynamic value to a user-selected printer.

The generic function `gPretty` that we will develop below is given a value to display. It computes the bounding box (*Box*) and a function that draws the value if provided with the location of the image (*Point2 Picture* → *Picture*). Graphical metrics information (such as text width and height) depends on the resolution properties of the output environment (the abstract and unique type **Picture*). Therefore `gPretty` is a state transformer on *Picture*s. *Picture* is predefined in the Clean Object I/O library [2], and so are *Point2* and *Box*.

```
generic gPretty t :: t -> St Picture (Box,Point2 Picture -> Picture)
::  Point2 = { x     :: Int, y     :: Int }
::  Box    = { box_w :: Int, box_h :: Int }
```

The key issue of this example is how `gPretty` handles dynamics. If we assume the existence of the derived code of `gPretty` as presented in Section 4 (that is either generated by the compiler or manually included by the programmer), then this code does the job:

```
dynPretty :: Dynamic -> St Picture (Box,Point2 Picture -> Picture)
dynPretty dx = gPretty_dyn (dynTypeRep dx) dx
```

It is important to observe that the program contains no *derived* instances of the generic `gPretty` function. Still, it can display every possible dynamic value.

We first implement the `gPretty` function and then embed it in a simple GUI. In the example we use a monadic programming style. Clean has no special syntax for monads, but the standard combinators are easily defined. For the synonym type *:: St s a :== s → (a,s)*, we assume the functions *return :: a → St s a* and the infix binding operator `>>=` *:: (St s a) (a → St s b) → St s b* in the usual way.

Basic values simply refer to the string instance that does the real work. It draws the text and the enclosing rectangle (we assume that the `getMetricsInfo` function returns the width and height of the argument string, proportional margins, and base line offset of the font):

```
gPretty{|Int|}    x = gPretty{|*|} (toString x)
gPretty{|String|} s
    = getMetricsInfo s >>= \(width,height,hMargin,vMargin,fontBase) ->
      let bound = { box_w=2*hMargin + width, box_h=2*vMargin + height }
      in  return ( bound
                 , \{x,y} -> drawAt {x=x+hMargin, y=y+vMargin+fontBase} s
                           o drawAt {x=x+1,y=y+1}
                                    {box_w=bound.box_w-2,box_h=bound.box_h-2}
                 )
```

The other cases only place the recursive parts at the proper positions and compute the corresponding bounding boxes. The most trivial ones are `UNIT`, which draws nothing, and `EITHER`, which continues recursively (poly)typically:

```
gPretty{|UNIT|}         _          = return (zero,const id)
gPretty{|EITHER|} pl pr (LEFT  x) = pl x
gPretty{|EITHER|} pl pr (RIGHT x) = pr x
```

PAIRs are drawn in juxtaposition with top edges aligned. A CONS draws the recursive component below the constructor name and centres the bounding boxes.

```
gPretty{|PAIR|} px py (PAIR x y)
    = px x >>= \({ box_w = wx, box_h = hx}, fx) ->
      py y >>= \({ box_w = wy, box_h = hy}, fy) ->
      let bound = { box_w = wx + wy, box_h = max hx hy }
      in  return ( bound, \pos -> fy {pos & x=pos.x+wx} o fx pos )
gPretty{|CONS of {gcd_name}|} px (CONS x)
    = gPretty{|*|} gcd_name >>= \({ box_w = wc, box_h = hc}, fc) ->
      px x                  >>= \({ box_w = wx, box_h = hx}, fx) ->
      let bound = { box_w = max wc wx, box_h = hc + hx }
      in  return ( bound, \pos -> fx (pos + {x=(bound.box_w-wx)/2, y=hc})
                                o fc (pos + {x=(bound.box_w-wc)/2, y=0 })
                 )
```

This completes the generic pretty printing function. We will now embed it in a GUI program. The Start function creates a GUI framework on which the user can drop files. The program response is defined by the ProcessOpenFiles attribute function which applies showDynamic to each dropped file path name.

```
module prettyprinter

import StdEnv, StdIO, StdDynamic, StdGeneric

Start :: *World -> *World
Start world = startIO SDI Void id
                [ ProcessClose     closeProcess
                , ProcessOpenFiles (\fs pSt -> foldr showDynamic pSt fs)
                ] world
```

The function showDynamic checks if the file contains a dynamic, and if so, sends it to the printer. This job is taken care of by the print function, which takes as third argument a Picture state transformer that produces the list of pages. For reasons of simplicity we assume that the image fits on one page.

```
showDynamic :: String (PSt Void) -> PSt Void
showDynamic fName pSt
    = case readDynamic fName pSt of
          (True,dx,pSt) = ( snd
                          o uncurry (print True False (pages dx))
                          o defaultPrintSetup
                          ) pSt
          (_,   _, pSt) = pSt
where pages :: Dynamic PrintInfo -> St Picture [IdFun Picture]
      pages dx _ = dynPretty dx >>= \(_,draw_x) -> return [draw_x zero]
```

6 Related work

Cheney and Hinze [6] present an approach that unifies dynamics and generics in a single framework. Their approach is based on explicit type representations for every type, which allows for *poor man's dynamics* to be defined explicitly by pairing a value with its type representation. In this way, a generic function is just a function defined by induction on type representations. An advantage of their approach is that it reconciles generic and dynamic programming right from start, which results in an elegant representation of types that can be used both for generic and dynamic programming. Dynamics in Clean have been designed and implemented to offer a *rich man's dynamics* (Section 2). Generics in Clean are schemes used to generate functions based on types available at compile-time. For this reason we have developed a first-class mechanism to be able to specialize generics at run-time. Our dynamic type representation has been inspired by Cheney and Hinze, but is less verbose since we can rely on built-in dynamic type unification.

Altenkirch and McBride [4] implement generic programming support as a library in the dependently typed language OLEG. They present the generic specialization algorithm due to Hinze [9] as a function *fold*. For a generic function (given by the set of base cases) and an argument type, *fold* returns the generic function specialized to the type. Our *specialize* is similar to their *fold*; it also specializes a generic to a type.

7 Current and future work

The low-level function *dynTypeRep* (Section 4.6) has to be implemented. We expect that this function gives some opportunity to simplify the *TypeRep* data type. Polymorphic functions are a recent addition to dynamics, and we will want to handle them by generic functions as well. The solution as presented in this paper works for generic functions of kind $\star$. We want to extend the scheme so that higher order kinds can be handled as well. In addition, the approach has to be extended to handle generic functions with several generic arguments. The scheme has to be incorporated in the compiler, and we need to decide how the derived code should be made available to the programmer.

8 Summary and Conclusions

In this paper we have shown how generic functions can be applied to dynamic values. The technique makes essential use of dynamics to obtain first-class representations of generic functions and dynamic type representations. The scheme works for all generic functions. Applications built in this way combine the best of two worlds: they have compact definitions and they work for any dynamic value even if these originate from different sources and even if these dynamics rely on alien types and functions. Such a powerful technology is crucial for type-safe mobile code, flexible communication, and plug-in architectures. A concrete application domain that has opportunities for this technique is the functional operating system *Famke* [16] (parsers, pretty printers, tool specialization).

References

1. Achten, P. and Hinze, R. Combining Generics and Dynamics. *Technical Report NIII-R0206*, July, 2002, Nijmegen Institute for Computing and Information Sciences, Faculty of Sciences, University of Nijmegen, The Netherlands.
2. Achten, P.M. and Wierich, M. A Tutorial to the Clean Object I/O Library - version 1.2. *Technical Report CSI-R0003*, February 2, 2000, Computing Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen, The Netherlands.
3. Alimarine, A. and Plasmeijer, M. A Generic Programming Extension for Clean. In Arts, Th., Mohnen M., eds. *Proceedings of 13th International Workshop on the Implementation of Functional Languages (IFL2001)*, Selected Papers, Älvsjö, Sweden, September 24-26, 2001, Springer-Verlag, LNCS **2312**, pp.168-185.
4. Altenkirch, T. and McBride, C. Generic Programming Within Dependently Typed Programming. To appear in *Proceedings Working Conference on Generic Programming*, Dagstuhl, Castle, Germany, July 11-12, 2002.
5. Brus, T., Eekelen, M.C.J.D. van, Leer, M.O. van, and Plasmeijer, M.J. Clean: A Language for Functional Graph Rewriting. In Kahn. G. ed. *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, USA, LNCS **274**, Springer-Verlag, pp. 364-384.
6. Cheney, J. and Hinze, R. A Lightweight Implementation of Generics and Dynamics. *Work in progress.*
7. Hinze, R. Polytypic values possess polykinded types. In Roland Backhouse and J.N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), July 3-5, 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag, July 2000.
8. Hinze, R. and Peyton Jones, S. Derivable Type Classes. In Graham Hutton, ed., *Proceedings of the Fourth Haskell Workshop*, Montreal, Canada, September 17, 2000.
9. Hinze, R. *Generic Programming and Proofs.* Habilitationsshrift, Universität Bonn, 2000.
10. Nöcker, E.G.J.M.H., Smetsers, J.E.W., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. Concurrent Clean. In Aarts, E.H.L., Leeuwen, J. van, Rem, M., eds., *Proceedings of Parallel Architectures and Languages Europe*, June, Eindhoven, The Netherlands. LNCS **506**, Springer-Verlag, pp. 202-219.
11. Peyton Jones, S. and Hughes, J. eds. *Report on the Programming Language Haskell 98 – A Non-strict, Purely Functional Language*, 1 February 1999.
12. Pil, M.R.C., Dynamic types and type dependent functions. In Hammond, Davie, Clack, eds., *Proc. of Implementation of Functional Languages (IFL '98)*, London, U.K., Springer-Verlag, Berlin, LNCS **1595**, pp.169-185.
13. Pil, M. *First Class File I/O*, PhD Thesis, *in preparation.*
14. Plasmeijer, M.J. and van Eekelen, M.C.J.D. *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Publishing Company, 1993.
15. Vervoort, M. and Plasmeijer, R. Lazy Dynamic Input/Output in the lazy functional language Clean. In Peña, R. ed. *Proc. of the 14th International Workshop on the Implementation of Functional Languages (IFL 2002)*, Madrid, Spain, September 16-18 2002, Technical Report 127-02, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, pp. 404-408.
16. van Weelden, A. and Plasmeijer, R. Towards a Strongly Typed Functional Operating System. In Peña, R. ed. *Proc. of the 14th International Workshop on the Implementation of Functional Languages (IFL 2002)*, Madrid, Spain, September 16-18 2002, Technical Report 127-02, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, pp. 301-319.

A Bimap combinators

A *(Bimap a b)* is a pair of two conversion functions of type $a \rightarrow b$ and $b \rightarrow a$. The trivial *Bimap*s `bimapId` and `bimapDynamic` are predefined:

```
::  Bimap a b   = { map_to :: a -> b, map_from :: b -> a }

bimapId :: Bimap a a
bimapId         = { map_to = id, map_from = id }

bimapDynamic :: Bimap a Dynamic | TC a
bimapDynamic    = { map_to = pack, map_from = unpack }
```
(Section 2)

The bimap combinator `inv` swaps the conversion functions of a bimap, `oo` forms the sequential composition of two bimaps, and `-->` obtains a functional bimap from a domain and range bimap.

```
inv :: (Bimap a b) -> Bimap b a
inv {map_to, map_from} = {map_to = map_from, map_from = map_to}

(oo) infixr 9 :: (Bimap b c) (Bimap a b) -> Bimap a c
(oo) f g = { map_to   = f.map_to   o g.map_to
           , map_from = g.map_from o f.map_from
           }

(-->) infixr 0 :: (Bimap a b) (Bimap c d) -> Bimap (a -> c) (b -> d)
(-->) x y = { map_to   = \f -> y.map_to   o f o x.map_from
            , map_from = \f -> y.map_from o f o x.map_to
            }
```