Dynamic Values Need Polytypic Functions
– Draft –

Peter Achten, Artem Alimarine, and Rinus Plasmeijer

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

Abstract. To do.

1 Introduction

In this paper we discuss the interaction between two recent additions to the pure, lazy, functional programming language Clean 2.0(.1) [4, 9, 13]:

Dynamic types enable us to write programs that convert values (data and functions) from the compile time world to the run time world in serialized format and vice versa. Dynamics work over module and application boundaries.

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 at compile time all types can be derived or verified.

In order to apply a generic function to a dynamic value in the current situation, the programmer needs to do an exhaustive type pattern-match on the dynamic type. Apart from the fact that this is impossible, this is at odds with the key idea of generic programming, because the programmer already did that in the generic function definition!

One would imagine that it is alright to apply a generic function to any dynamic value. Because generic functions are functions that are defined on the structure of types and their values they are ideal for application to dynamic values. A dynamic value contains a complete *type representation* and a properly linked-in Clean graph. Therefore, knowing the generic function to apply, one should, *in principle*, be able to determine the correct instance of the generic function.

The main problem is that the type representation of a dynamic is only known at *run time*, while the instantiation of generic functions needs a compiler-type that is known only at *compile time*. One obvious solution is to force the programmer not to use generic functions on unpacked dynamic values, but instead extend the dynamic with the 'generic' functions that have to be applied to it. When packing such a dynamic, adding the generic functions will suffice. This is unfeasible, as it requires knowledge of all (future) contexts a dynamic value will ever be applied in.

We show how to solve this problem *without changing the compiler*, and relying only on the current implementation of generics and dynamics. The key to the solution is to find a *fixed* representation for types that can be used both at compile time (to work with compile time type representations), and at run time (to work with dynamic type representations). Dynamics are always created in a context in which their compile time type is known. We seize this opportunity to additionally store conversion functions between the type of the dynamic value and a generic dynamic type. The conversion functions are predefined once *generically*. For types of kind $\star$ stored in a dynamic, there is a fixed transformation rule that delivers the code for applying the generic function to the dynamic content. In principle, this can be added in a compiler. As a consequence, the programmer only needs to write a generic function definition (as usual), and the compiler can, in principle, derive the dynamic version from it (but, as said, only for kind $\star$).

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 give examples of programs that exploit the combined power of generics and dynamics. The main characteristic of these programs is that they are universally applicable to dynamic values without preprogrammed knowledge of specific types.
- The solution that we show can be implemented in any functional language that has support for dynamics and generics. It also works for a language that supports dynamics and overloading, at the expense of writing most of the functions that we present manually.

We start the remainder of this paper with introductions to dynamics (Section 2) and generics (Section 3) with respect to core properties that we rely on. We then show how the generic dynamic extension works, based on a small set of type and function definitions (Section 4). A number of examples are given to demonstrate the expressive power of the combination of generics and dynamics (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 [11, 12]. Dynamics serve two major purposes:

Interface between static and run time types: Programs can convert values from the *compile time* world to the *run time* world and back again 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.

```
toDynamic :: [Dynamic]
toDynamic = [e1, e2, e3, dynamic [e1,e2,e3]]
where  e1 = dynamic 50                :: Int
       e2 = dynamic reverse           :: [Real] -> [Real]
       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*). After successful matching, *e-pat* has become a Clean expression, and *t-pat* a compile time type. It is important to note that type variables in a type pattern do not indicate polymorphism. Instead, they are bound to the offered type.

```
fromDynamic :: [Dynamic] -> (Int,[Real] -> [Real],[Char])
fromDynamic [ e1 :: Int, e2 :: [Real] -> [Real], e3 :: [Char] ]
    = (e1, e2, e3)

dynApply :: Dynamic Dynamic -> Dynamic
dynApply (f :: a -> b) (x :: a) = dynamic (f x) :: b
```

Serialization: Being able to store values in a program in a dynamic format is already useful: it allows flexible manipulation of data and functions. However, the main virtue of dynamics is that it allows 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.
Making an effective and efficient implementation is hard work and requires careful design and architecture of 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 explained above.
Two library functions store and retrieve dynamic values in named files:

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

The `*` in front of `env` is its *uniqueness attribute* which indicates that this environment value will be passed around single-threadedly. It is an example of the *uniqueness type system* [14] of Clean to handle safe destructive updates in a pure functional language. The two functions also demonstrate the explicit environment passing style of handling I/O in Clean.

At this stage, the Clean 2.0.1 system restricts the use of dynamics to *basic*, *algebraic*, *record*, *array*, and *function* types. Polymorphism is not supported, so functions such as the identity function (`id :: a -> a`) can not be stored as such in a dynamic. Overloaded types and overloaded functions are still an open issue, although it has been investigated by Pil [12]. 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 [6] and its integration with overloading as developed by Hinze and Peyton Jones [7]. 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 makes use of the compile time type information that is available.

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. Equality compares two arguments of the same (unifiable) type:

```
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 `Int`eger, `Real`, `Bool`ean, `Char`acter, and `String`). The generic implementation compares the values of these types, and simply uses the appropriate predefined instance for basic types of the overloaded equality operator `==`.

```
gEq{|Int|}    x y = x == y
gEq{|Char|}   x y = x == y
gEq{|Bool|}   x y = x == y
gEq{|Real|}   x y = x == y
gEq{|String|} x y = x == y
```

Non-basic types are constructed as sums (`EITHER`) of pairs (`PAIR`) – or empty pair (`UNIT`) – of types. It is useful to have information about data constructors (`CONS`) and record fields (`FIELD`), such as names, arity, priority, and so on. These data types are predefined in Clean, and are collected in the module `StdGeneric.dcl`.

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

The kind of these cases (`UNIT` : $\star$, `CONS`, `FIELD` : $\star \to \star$, and `EITHER`, `PAIR` : $\star \to \star \to \star$) 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 (note that for equality, the additional information is not really used).

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

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

The elements of the infinite set of overloaded functions represented by a generic function are indexed with the kind, so for equality we have $\mathtt{gEq}_{\star}$, $\mathtt{gEq}_{\star\rightarrow\star}$, $\mathtt{gEq}_{\star\rightarrow\star\rightarrow\star}$, and so on. The generic function f of kind κ is denoted as: f{| κ |} and can be used as any other overloaded function in a Clean program. Programs must ask explicitly for an instance of type T of a generic function f by: `derive` f T. The kind of T is derived by the compiler. Here is an example of a complete Clean program:

```
module myTree

import StdEnv, StdGeneric

::  MyTree a = Leaf | Cons (MyTree a) a (MyTree a)

derive gEq MyTree   // Generate the instance of gEq for MyTree

tree1 = Cons Leaf 5 (Cons Leaf 7 Leaf)
tree2 = Cons Leaf 2 (Cons Leaf 4 Leaf)

Start = ( gEq{|*|}    tree1 tree2          // Value     equality: False
        , gEq{|*->*|} (const o const True)
                      tree1 tree2          // Structure equality: True
        )

generic gEq a :: a a -> Bool        // Definition as in this section
...
```

4 Dynamics + Generics in Clean

In this section we show how we enable programs that manipulate *dynamics* to use any *generic* definition on any dynamic they have obtained. The current implementation of dynamics and generics in Clean imposes the following restrictions on such a solution:

1. Generic functions are really ‘schemes’ from which instances of overloaded functions can be generated. For this we need to know the type of the instance.

2. Clean expressions are unpacked from a dynamic using a type pattern. The only static type information that is available at compile time are the constants of the type pattern, the values of type pattern variables are only known at run time after unification.
3. When packing a value in a dynamic, the type information is available (provided by program or infered by compiler). The constituents, however, may result from dynamic type pattern variables (see `dynApply` in Section 2).
4. Generic functions should be 'polymorphic' for dynamic values. The opaque dynamic type representation contains sufficient information to distil a run-time type out of.

The main problem occurs when applying a generic function to a dynamic value: we need to have or generate the proper instance of an overloaded function, but we do not know what the dynamic type pattern variables will be unified with *statically*.

The key idea of our solution is to include *conversion functions* to and from a *generic dynamic type representation* whenever a value is packed into a dynamic. This can be done automatically by a *generic function*, because when a value is packed into a dynamic, its type is known. The generic dynamic type representation is given in Section 4.1, and the conversion function in Section 4.2. When unpacking a dynamic value, we can then extract both the dynamic type representation and the generic dynamic type conversion functions. The first is required for our 'normal' dynamic programming (to enforce unification between the dynamic type representation and the static dynamic type pattern), and the second is required to convert the dynamic value to the generic representation.

Our solution relies on the ability of the dynamics implementation that it is able to store arbitrary Clean expressions, but does not change it. The definition of generic functions also does not change in this framework. The programmer writes a function that has the same type as the generic signature, except that the overloaded type variables are replaced by `Dynamics`. We show this in Section 4.3. (We also show how to do this for functions with argument dynamics (Section 5.1) and result dynamics (Section 5.2.))

The ideal situation is that generic functions behave 'polymorphically': they should work for any dynamic value. This means that we do not want the programmer to write his functions for dynamic values, but rather simply apply a generic function to a dynamic value and let the compiler sort things out. In the current proposal we can do this for generic functions of kind $\star$. This is presented in Section 4.4.

4.1 A generic dynamic type representation

The first thing we need is a fixed representation for generic dynamic values:

```
::  GenRep
 =  GRInt Int | GRReal Real | GRBool Bool | GRChar Char | GRString String
 |  GRUnit
```

```
| GRPair  GenRep GenRep
| GRLeft  GenRep | GRRight GenRep
| GRCons  GenericConsDescriptor  GenRep
| GRField GenericFieldDescriptor GenRep
```

This type is very similar to the types that we used in Section 3. The main difference is that `GenRep` is a sum type, rather than a collection of type constructors. The key advantage is that we can easily specialize the generic functions to `GenRep` in one go (Section 4.3).

The `GenericConsDescriptor` and `GenericFieldDescriptor` types are predefined in `StdGeneric`. They contain the additional information a programmer might need when handling the `CONS` and `FIELD` cases of a generic function. In the generic equality example we had no need for them.

4.2 Conversion functions

Because we will have to do a great deal of back and forth conversion, it is convenient to have the two conversion functions at hand. A value of type (`Bimap` a b) is a pair of conversion functions of type $a \to b$ and $b \to a$. A number of predefined bimaps and standard combinators are provided, see Appendix A.

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

Because we will need to convert any compile time type a to `GenRep` and back again, we must have a *(Bimap a GenRep)* for this type. For this we define a generic function `genRep`. (Note that of the basic types, we only show the `Int` case, as the others are similar. In addition, note that the `map_from` functions are partial functions.

```
generic genRep a :: Bimap a GenRep
genRep{|Int|}                = {map_to = map_to, map_from = map_from}
where map_to   x             = GRInt x
      map_from (GRInt x)     = x
genRep{|UNIT|}               = {map_to = map_to, map_from = map_from}
where map_to   UNIT          = GRUnit
      map_from GRUnit        = UNIT
genRep{|PAIR|} fx fy         = {map_to = map_to, map_from = map_from}
where map_to   (PAIR   x y)  = GRPair (fx.map_to   x) (fy.map_to   y)
      map_from (GRPair x y)  = PAIR   (fx.map_from x) (fy.map_from y)
genRep{|EITHER|} fl fr       = {map_to = map_to, map_from = map_from}
where map_to   (LEFT    x)   = GRLeft  (fl.map_to   x)
      map_to   (RIGHT   x)   = GRRight (fr.map_to   x)
      map_from (GRLeft  x)   = LEFT    (fl.map_from x)
      map_from (GRRight x)   = RIGHT   (fr.map_from x)
genRep{|CONS of d|} fx       = {map_to = map_to, map_from = map_from}
where map_to   (CONS     x)  = GRCons d (fx.map_to   x)
      map_from (GRCons _ x)  = CONS     (fx.map_from x)
genRep{|FIELD of d|} fx      = {map_to = map_to, map_from = map_from}
where map_to   (FIELD     x) = GRField d (fx.map_to   x)
      map_from (GRField _ x) = FIELD     (fx.map_from x)
```

When packing and unpacking values of type a to a dynamic we also store and read the corresponding *(Bimap a GenRep)*.

4.3 Generic dynamic function definitions

The generic equality function gEq remains the same as in Section 3. Given this generic function, we can ask the compiler to derive the proper instance for *GenRep* values. Because derived instances have to be asked for explicitly, also instances of the type constructors on which *GenRep* relies need to be instances of the generic function. In general, for an arbitrary generic function g this would result in:

```
derive g GenRep,
         GenericConsDescriptor,
         GenericFieldDescriptor,
         GenConsPrio,
         GenConsAssoc,
         GenericTypeDefDescriptor,
         GenType
```

However, for gEq, only a derived instance for GenRep will do, given the following name equality for the extra CONS and FIELD descriptors:

```
derive gEq GenRep
gEq{|GenericConsDescriptor |} x y = gEq{|*|} x.gcd_name y.gcd_name
gEq{|GenericFieldDescriptor|} x y = gEq{|*|} x.gfd_name y.gfd_name
```

Now if we assume that we use the extended way of storing dynamics, then for each dynamic value of type a we also have the corresponding *(Bimap a GenRep)*. The equality operation on dynamics uses this *Bimap* to convert dynamic values to generic dynamic values, for which a generic equality function is available (above we have just asked the compiler to derive it). Here is the definition of the equality on dynamics:

```
dEq :: Dynamic Dynamic -> Bool
dEq ((x::a,epx)::(Dynamic,Bimap a GenRep)) ((y::a,epy)::(Dynamic,Bimap a GenRep))
    = adaptEq (epx oo inv bimapDynamic) gEq{|*|} x y
where bimapEq a  = a --> a --> bimapId
      adaptEq ep = (bimapEq ep).map_from
dEq _ _
    = False
```

To understand this quite concise function, let us dissect it:

- Let f = (epx oo inv bimapDynamic). Because (inv bimapDynamic) :: Bimap Dynamic a and epx :: Bimap a GenRep, we have f :: Bimap Dynamic GenRep. In other words, f transforms dynamic values to generic dynamic representations and vice versa via the additionally stored bimap epx.

– `bimapEq :: (Bimap a b)` $\rightarrow$ `Bimap (a` $\rightarrow$ `a` $\rightarrow$ `c) (b` $\rightarrow$ `b` $\rightarrow$ `c)`, so `bimapEq` *f* `:: Bimap (Dynamic` $\rightarrow$ `Dynamic` $\rightarrow$ `c) (GenRep` $\rightarrow$ `GenRep` $\rightarrow$ `c)`.
– `(bimapEq` *f*`).map_from` is a conversion function of type `(GenRep` $\rightarrow$ `GenRep` $\rightarrow$ `c)` $\rightarrow$ `(Dynamic` $\rightarrow$ `Dynamic` $\rightarrow$ `c)`.
– `((bimapEq` *f*`).map_from gEq{|*|}) :: Dynamic` $\rightarrow$ `Dynamic` $\rightarrow$ `Bool` is the desired function.
– We use the opaque dynamic type representation of `x` and `y` to enforce unification. If type pattern matching succeeds, we are ensured that `x` and `y` will have identical types.

4.4 Integration in Clean compiler

In this section we discuss the transformation rules that can be implemented in the compiler to derive most of the code above automatically. **To be done.**

5 Examples of dynamics and generics

In this section we give two examples that exploit the combined power of dynamics and generics. Both examples are useful tools when having a disk filled with dynamic files. The first is a *pretty printer* of dynamic values, and the second is a *parser generator* for dynamic values. In each example, we first give the generic function definition, then present the code of the dynamic version, and finally apply the dynamic version to an actual dynamic value.

In order to have a user-friendly tool, we make use of the Clean Object I/O library to create a simple GUI. The module `simpleGUI` exports the function `simpleGUI` that is parameterized with a function of type *String* $\rightarrow$ *Dynamic* $\rightarrow$ *env* $\rightarrow$ *env*. Whenever a user drops a file on the GUI framework, it is checked if that file contains a dynamic value, and if this is the case, the argument function is evaluated. It is applied to the full path name of the file, dynamic content of the file, and the GUI environment. We will not discuss this module any further, as it is an easy exercise in Object I/O. For completeness, its code is presented in Appendix C. The relevant type and function are:

```
::  DynamicIO env :== String -> Dynamic -> IdFun env

simpleGUI :: (DynamicIO (PSt Void)) -> IdFun *World
```

(`IdFun` is a predefined synonym type with: `IdFun x :== x -> x`. `Void` is equivalent with Haskell's `()`. A discussion of the `PSt` is out of scope, but it can be considered as a full-fledged environment specialized for GUI operations.)

5.1 Pretty printer

Pretty printers belong to the classic examples of generic programming. In this example 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 The generic function `pretty`, given a value to display, computes the bounding box (`Box`) and the function that actually draws the value, if it is given the left-top corner of the bounding box (`Point2 *Picture -> *Picture`). Because graphical metrics information depend on the resolution properties of the output environment, the function is a state transformer on `*Picture`s. Therefore, `pretty` has the following type:

```
generic pretty t :: t *Picture
                 -> ((Box,Point2 *Picture -> *Picture),*Picture)
```

(The `Picture` environment can be used to draw in any visual component or the printer. It is predefined in the Clean Object I/O library [2], and so are `Point2` and `Box`:

```
::  Picture  // abstract data type
::  Point2 = { x     :: !Int, y     :: !Int}
::  Box    = { box_w :: !Int, box_h :: !Int }
```

In Clean, record types are surrounded by `{}`. Data type declarations start with `::`. The sort of type declaration is usually indicated by the separator, which is `=` for algebraic and record types, and `:==` for synonym types.)

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 `getMetrics-Info` function returns the width and height of the argument string, proportional margins, and base line offset of the font):

```
pretty{|Int|}    x picture = pretty{|*|} (toString x) picture
pretty{|Real|}   x picture = pretty{|*|} (toString x) picture
pretty{|Char|}   x picture = pretty{|*|} (toString x) picture
pretty{|Bool|}   x picture = pretty{|*|} (toString x) picture
pretty{|String|} s picture
    # ((width,height,hMargin,vMargin,fontBase),picture)
            = getMetricsInfo s picture
    # bound = { box_w=2*hMargin + width, box_h=2*vMargin + height }
    = ( ( 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}
        )
      , picture
      )
```

(In Clean, `#` allows convenient reuse of (in particular environment) names. If r is a record value, and f one of its fields, then $r.f$ selects the field value of r. In a pattern match, $\{f\}$ can be used to select the field value. Function composition is predefined as `o`.)

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:

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

PAIRs are drawn in juxtaposition with top edges aligned. A CONS draws the constructor name on top, and puts the recursive component below the constructor. The bounding boxes are centred. (Note that FIELDs are handled the same way as CONSs are.)

```
pretty{|PAIR|} px py (PAIR x y) picture
    # ((bx,fx),picture) = px x picture
    # ((by,fy),picture) = py y picture
    # bound             = { box_w =     bx.box_w + by.box_w
                          , box_h = max bx.box_h   by.box_h
                          }
    = ( ( bound, \pos -> fy {pos & x=pos.x+bx.box_w} o fx pos )
      , picture
      )
pretty{|CONS of {gcd_name}|} px (CONS x) picture
    # ((bc,fc),picture) = pretty{|*|} gcd_name picture
    # ((bx,fx),picture) = px x picture
    # bound             = { box_w = max bc.box_w   bx.box_w
                          , box_h =     bc.box_h + bx.box_h
                          }
    = ( ( bound
        , \pos -> fx {pos & x=pos.x + (bound.box_w-bx.box_w)/2
                          , y=pos.y+bc.box_h
                     }
                o fc {pos & x=pos.x + (bound.box_w-bc.box_w)/2}
        )
      , picture
      )
```

(If r is a record value, and v a new value for the field f, then {r & f=v} is a new record value, equal to r, but with value v for field f.)

The dynamic function If we follow the implementation scheme for dynamic functions (Section 4.3) we obtain the following code:

```
derive pretty GenRep
pretty{|GenericConsDescriptor|}  _ picture = ((zero,const id),picture)
pretty{|GenericFieldDescriptor|} _ picture = ((zero,const id),picture)

dpretty :: Dynamic *Picture -> ((Box,Point2 *Picture -> *Picture),*Picture)
dpretty ((x::a,epx)::(Dynamic,Bimap a GenRep)) picture
    = adaptPretty (epx oo inv bimapDynamic) pretty{|*|} (dynamic x::a) picture
where bimapPretty a  = a --> bimapId --> bimapId
      adaptPretty ep = (bimapPretty ep).map_from
```

However, this function pretty prints generic dynamic representations of dynamic values instead of the dynamic values. This is obvious, because we asked the compiler to derive `pretty` for `GenRep`. In this case, we need to define our own instance of `pretty`. We can do this by making good use of the generic `pretty` function:

```
pretty{|GenRep|} v p = fp v p
where fp :: GenRep *Picture
         -> ((Box,(Point2,Point2) *Picture -> *Picture),*Picture)
      fp (GRInt   x)     p = pretty{|*|} x p
      fp (GRReal  x)     p = pretty{|*|} x p
      fp  GRUnit         p = pretty{|*|}             UNIT        p
      fp (GRCons  gcd x) p = pretty{|*->*|}    fp    (CONS  x)   p
      fp (GRField gfd x) p = pretty{|*->*|}    fp    (FIELD x)   p
      fp (GRPair  x y)   p = pretty{|*->*->*|} fp fp (PAIR  x y) p
      fp (GRLeft  l)     p = pretty{|*->*->*|} fp fp (LEFT  l)   p
      fp (GRRight r)     p = pretty{|*->*->*|} fp fp (RIGHT r)   p
```

Embedding in GUI The dynamic pretty printing function can now be used in the argument function `prettyprinter` of `simpleGUI`. It sends the dynamic content of any dropped file to the printer. For this we use the following functions of the Object I/O library: `defaultPrintSetup` reads in the default printer setup, and the function `print` actually does the printing. The third argument of `print` is a `Picture` state transformer that produces the pages as a list of drawing functions. Note that for reasons of simplicity we assume that the image will fit on one page.

```
prettyprinter :: (DynamicIO *env) | FileEnv, PrintEnvironments env
prettyprinter
    = \_ x -> snd o uncurry (print True False (pages x)) o defaultPrintSetup
where
    pages :: Dynamic PrintInfo *Picture -> ([IdFun *Picture],*Picture)
    pages dx _ picture
        # ((boundBox_x,draw_x),picture) = dpretty dx picture
        = ([draw_x zero],picture)

Start :: *World -> *World
Start world = simpleGUI prettyprinter world
```

5.2 Parser generator

Just as pretty printers, *parsers* also belong to the classic repertoire of generic programming. In this section we develop an application that generates a parser for expressions that have the same type as any dynamic value that is dropped on it. Of course, we store the generated parser as a dynamic itself.

The generic function The generic parser function uses a small *combinator parser* library [8]. We refer to Appendix B for a brief description of the combinators and functions. What we need to know right now is that parsers are constructed by composition of basic *continuation* parsers of type (CParser *s a t*). Such a continuation parser consumes (part of) an input sequence of symbols of type *s*, and returns a result of type *a*. We will use a Character input sequence, hence our generic parser generating function must be a (CParser Char a t):

```
generic parser a :: CParser Char a t
```

We adopt the convention that every continuation parser skips whitespace first. The basic types have straightforward definitions: the Int and Real instances use int and real, Char accepts any character between '' (we do not handle escape characters), and Booleans accept either True or False.

```
parser{|Int|}  = sp int
parser{|Real|} = sp real
parser{|Char|} = sp (symbol '\'') &> satisfy (const True) <& symbol '\''
parser{|Bool|} = (sp (token ['True'])  &> yield True)
                     <!>
                 (sp (token ['False']) &> yield False)
```

The standard polytypic cases are also not hard to figure out: UNITs require no parsing at all and simply yield UNIT, parsing PAIRs parses the elements in order and returns their results, and parsing the alternatives of EITHER simply uses the <!> operator.

```
parser{|UNIT|}         = yield UNIT
parser{|PAIR|}   fx fy = sp fx <&> \x -> sp fy <&> \y -> yield (PAIR x y)
parser{|EITHER|} fl fr = (sp fl <&> \x -> yield (LEFT  x))
                                 <!>
                         (sp fr <&> \x -> yield (RIGHT x))
```

In order to create a parser for data constructors (CONS), we need access to the actual name of the constructor. This is provided by the gcd_name field that can be requested from the CONS instance.

```
parser{|CONS of {gcd_name}|} fx
    = sp (symbol '(')                &>
      sp (token (fromString gcd_name)) &>
      sp fx                         <&> \x ->
      sp (symbol ')')                &> yield (CONS x)
```

Predefined type constructors, such as tuples, lists, and records, have special syntax, so we need to specialize these cases ourselves. We only show the list instance: lists can be empty [], or contain a number of elements [e_1,e_2,... e_n]. If an initial element is parsed, we use the <*?> combinator to find the maximum occurrences of list elements.

```
parser{|[]|} fx
    = sp (symbol '[') &> ((sp (symbol ']') &> yield [])
                              <!>
                           (sp fx                              <&> \head ->
                            (<*?> (sp (symbol ',') &> sp fx)) <&> \tail ->
                            sp (symbol ']') &> yield [head:tail]
                           )
                         )
```

The dynamic function The dynamic version of the parser generator follows a slightly different scheme than presented in Section 4.3. The reason is that a parser is a function that is overloaded in the right-hand-side. This implies that we do not have a bimap argument to work with. We solve this pragmatically by adding an additional dynamic argument to these functions that contains the bimap. This yields:

```
derive parser GenRep

dparser :: Dynamic -> CParser Char Dynamic Dynamic
dparser (ep::Bimap a GenRep)
    = adaptParser (ep oo inv bimapDynamic) parser{|*|}
where bimapParser a  = bimap{|*->*->*|} (bimap{|*->*->*|} a bimapId) bimapId
      adaptParser ep = (bimapParser ep).map_from
```

Embedding in GUI We can now embed `dparser` in `simpleGUI`, as the function `storeParser`. It is applied to a dynamic with file path name *path*, and dynamic value (x `::` a, `bimap`) where `bimap :: Bimap` a `GenRep`. It creates a new file named (*path* `+++ "parser"`) in which a parser `p` is stored that parses values of type a. The parser `p` is the value `(begin (dparser bimap)) :: Parser Char Dynamic`. As promised, we also store the corresponding bimap, automatically generated by `genRep{|*|} :: Bimap (Parser Char Dynamic) GenRep`. The `Start` rule of the program simply passes `storeParser` to `simpleGUI`.

```
storeParser :: (DynamicIO *env) | FileSystem env
storeParser
    = \path (((x::a),bimap)::(Dynamic,Bimap a GenRep))
      -> snd o (writeDynamic
                 (path+++"parser")
                 (dynamic (dp,genRep{|*|})
                             ::
                         (Dynamic, Bimap (Parser Char Dynamic) GenRep)
               ))
where p  = begin (dparser (dynamic bimap :: Bimap a GenRep))
      dp = dynamic p :: (Parser Char Dynamic)

Start :: *World -> *World
Start world = simpleGUI storeParser world
```

6 Related work

Cheney and Hinze [5] present a *poor man's dynamics* that marries generic programming with dynamics. Their solution is more verbose for the programmer who needs to apply explicit unifications to dynamic types. Clean dynamics have been designed and implemented to offer a *rich man's dynamics*, taking care of separate compilation issues, efficient graph and type representations, and version management. 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.

7 Current and future work

To do.

8 Conclusions

To do.

References

1. Achten, P. and Hinze, R. Combining Generics and Dynamics. To appear as *Technical Report NIII-R0206*, Computing Science Institute, Faculty of Mathematics and Informatics, 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. 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.
5. Cheney, J. and Hinze, R. A Lightweight Implementation of Generics and Dynamics. *Work in progress.*
6. 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.
7. Hinze, R. and Peyton Jones, S. Derivable Type Classes. In Graham Hutton, ed., *Proceedings of the Fourth Haskell Workshop*, Montreal, Canada, September 17, 2000.
8. Koopman, P. and Plasmeijer, M.J. Efficient Combinator Parsers. In Hammond, K., Davie, A.J.T., and Clack, C. eds., *Proc. of Implementation of Functional Languages (IFL '98)*, London, U.K., Springer Verlag, LNCS **1595**, pp. 120-136.

9. 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.
10. Peyton Jones, S. and Hughes, J. eds. *Report on the Programming Language Haskell 98 – A Non-strict, Purely Functional Language*, 1 February 1999.
11. 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.
12. Pil, M. *First Class File I/O*, PhD Thesis, *in preparation.*
13. Plasmeijer, M.J. and van Eekelen, M.C.J.D. *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Publishing Company, 1993.
14. Smetsers, J.E.W., Barendsen, E., Eekelen, M.C.J.D. van, and Plasmeijer, M.J. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In Schneider, H.J., Ehrig, H. eds. *Proceedings Workshop Graph Transformations in Computer Science*, Dagstuhl Castle, Germany, January 4-8, 1993, LNCS **776**, Springer-Verlag, Berlin, pp. 358-379.
15. Vervoort, M. and Plasmeijer, M.J. Lazy dynamic Input/Output in the compiled lazy functional language Clean. *In these proceedings.*

A Bimap combinators

The *Bimap* type is a simple pair of two conversion functions. Two *Bimap*s are predefined: `bimapId` is the pair of identity functions, and `bimapDynamic` packs and unpacks values to and from dynamics. The combinators that we use in this paper are: `inv`, which swaps the conversion functions of a `(Bimap` a b`)` to `(Bimap` b a`)`, `oo`, which forms the sequential composition `(Bimap` a c`)` of two arguments `(Bimap` b c`)` and `(Bimap` a b`)`, and finally, `-->`, which converts functions $(a \to c)$ $(b \to d)$ if given conversion pairs `(Bimap` a b`)` and `(Bimap` c d`)`.

```
::  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 = map_to, map_from = map_from}
where map_to    x        = dynamic x :: a^
      map_from (x :: a^) = x

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
            }
```

B Combinator parsers

Here is a brief summary of combinator parsers à la Koopman [8]. A *parser* is a function of type (`Parser` s r) that reads an input sequence of s symbols, and returns a list of successful parses (`ParsResult` s r). Each element is a pair of the remaining input [s] and a value of type r:

```
::  Parser     s r :== [s] -> ParsResult s r
::  ParsResult s r :== [([s],r)]
```

Parsers are constructed by glueing *continuation* parsers of type (`CParser` s r t). Such a parser consumes an input sequence of s symbols, and returns results of type r that contribute to the total result of type t. It uses continuations to prevent the construction of intermediate data structures, and hence improve the performance. Given any continuation parser p, (`begin` p) turns it into a parser.

```
::  CParser s r t :== (Suc s r t) -> (Xor s t) -> (Alt s t) -> Parser s t
::  Suc     s r t :==         r    -> (Xor s t) -> (Alt s t) -> Parser s t
::  Xor     s   t :== (Alt s t) -> ParsResult s t
::  Alt     s   t :== ParsResult s t

begin :: (CParser s r r) -> Parser s r
```

We use the following combinators to construct continuation parsers: (`yield` r) simply returns r without consuming input; (`satisfy` c) consumes and yields the input that satisfies c; (`token` s) and (`symbol` s) consume and return their argument; (`sp` p) first consumes whitespace and continues as p; (`<*?>` p) applies p as many times as possible, returning the parsed results in a list; `int` and `real` parse and return any integer and real value.

```
yield   ::  r           -> CParser s r t
satisfy :: (s -> Bool) -> CParser s s t
token   :: [s] -> CParser s [s] t | == s
symbol  ::  s  -> CParser s  s  t | == s
sp      :: (CParser Char r t) -> CParser Char r  t
<*?>    :: (CParser s    r t) -> CParser s   [r] t
int     ::  CParser Char Int  t
real    ::  CParser Char Real t
```

(p `<&>` f) is the standard way of combining two parsers in the expected way: first parse as p, and pass its result to f which returns a new parser that is evaluated. Two useful variants are (p_1 `&>` p_2) and (p_1 `<&` p_2) which parse as p_1 and p_2 subsequently, but ignore the result of p_1 and p_2 respectively. Finally, (p_1 `<!>` p_2) proceeds as p_1 if successful or p_2 if not successful.

```
(<&>) infixr 6 :: (CParser s u t) (u -> CParser s v t) -> CParser s v t
( &>) infixr 6 :: (CParser s u t)      (CParser s v t) -> CParser s v t
(<& ) infixr 6 :: (CParser s u t)      (CParser s v t) -> CParser s u t
(<!>) infixr 4 :: (CParser s u t)      (CParser s u t) -> CParser s u t
```

C simpleGUI Framework

```
implementation module simpleGUI

import StdEnv, StdIO, StdDynamic

::  DynamicIO env :== String -> Dynamic -> IdFun env

simpleGUI :: (DynamicIO (PSt Void)) -> IdFun *World
simpleGUI f
    = startIO SDI Void id
        [ ProcessClose     closeProcess
        , ProcessOpenFiles (\fs pState -> foldr (toDynamic f) pState fs)
        ]
where
    toDynamic :: (DynamicIO (PSt Void)) String (PSt Void) -> PSt Void
    toDynamic f fName pState
        = case readDynamic fName pState of
            (True,dyn,pState)
                = case dyn of
                    ((x::a),bimap)::(Dynamic,Bimap a GenRep)
                         = f fName dyn pState
                    _    = pState
            (_,_,pState) = pState
```