Efficient and Type-Safe Generic Data Storage

Sjaak Smetsers[1] Arjen van Weelden[1] Rinus Plasmeijer[1]

Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands

Abstract

In this paper we present an elegant method for sequentializing arbitrary data using the generic language extension of the functional programming language Clean. We show how the proposed operations can be used to store values of any concrete data type in several kinds of IO containers (such as files or arrays of characters), and how to manipulate stored data efficiently. Moreover, by extending stored data with encoded type information, data manipulation will be type-safe. Defining these operations generically has the advantage that specific instances for user defined data types can be generated fully automatically. Compared to traditional sequentialization methods (or to common data manipulation, using relational data bases) our operations are an order of magnitude faster.

Keywords: Polytypic/generic programming, functional programming, data storage/sequentialization, type inference

1 Introduction

Databases are commonly used for the efficient storage and retrieval of data. Effectively, they are optimized to store (large) tables in a very efficient way. For a large class of applications this type of storage is adequate enough. However, traditional databases are commonly not suited for storing recursive data structures, such as linked lists or binary trees, or even more complex data types like abstract syntax trees. In functional languages these algebraic data structures are very common. One would like to have the ability to store these algebraic data types efficiently in some database system as well. Efficient storage not only requires fast input/output operations but also the ability to update stored data selectively, i.e. without the necessity to read in and write back the complete data structure. For example, one would like to inspect the elements of a stored tree individually and update them destructively, or, re-balance the spine of the tree without retrieving/copying the elements. Furthermore, the signature of the stored data should be encoded in the database, such that at run time the type safety of the data read in from the database is guaranteed.

[1] Email: S.Smetsers@cs.ru.nl, A.vanWeelden@cs.ru.nl, R.Plasmeije@cs.ru.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

www.manaraa.com

In this paper we will show how such efficient and flexible data manipulations can be realized using *generic* or *polytypical* programming techniques. With a generic function one can specify operations that work for (almost) *any* type. A generic function is basically defined inductively on the structure of user-defined types. Equality and map are basic examples of functions that can be defined generically. Other well-known examples are (de)serialization functions like print and parse (resulting in a string representation of data), and (un)compress (using an efficient binary representation of data). To apply a generic function on a specific type, an instance of the generic function for that type is needed. The Clean compiler will generate this instance fully automatically, but the programmer explicitly has to ask for it via a derive statement.

All proposed IO operations in this paper are specified generically. The presented generic functions are written in Clean (see [2]). Due to some language specific limitations, the presented functions cannot be converted directly to Generic Haskell. There are some syntactic differences between Haskell and Clean. The major differences will be explained on-the-fly; for a complete comparison between Clean and Haskell we refer to [1].

2 Generic Programming

Specifying generic functions resembles defining a type class and its instances. The main difference is that a generic compiler can derive most of the instances automatically, given a minimal fixed set of instances for three (generic) type constructors. The derivation of instances is based on the fact that any algebraic data type can be expressed in terms of *eithers* (for distinguishing the constructors), *pairs* (for representing the argument types of each constructor), and *units* (for representing 0-ary constructors).

This generic type representation, developed by Hinze [6], is used by Clean as well as by Generic Haskell, and is encoded by the following Clean types:

```
:: EITHER[2] a b = LEFT a | RIGHT b;    :: PAIR a b = PAIR a b;    :: UNIT = UNIT
```

To define a generic function, the programmer specifies its signature and provides instances for the generic types (`EITHER`, `PAIR`, and `UNIT`). As an example we define a generic packing program, taken from [9]. The basic idea of this program is simple: given a value, we construct a compact representation of that value in which data constructors are represented by as few bits as possible. E.g. for a simple binary tree this means that we distinguish leafs from internal nodes using a single bit. To abstract from the concrete container that is used to store the packed value, we first introduce

```
class GWrite io  where  writeC :: Char *io → *io
```

Clean uses *uniqueness types* to express sharing properties of objects. A $*$ indicates that the corresponding object will be *unique* or *single-threaded* which means that there exist no further references to it. This allows the object to be updated destructively. Hence, in Clean one can use explicit environment passing (the `io` argument

[2] Defines a new data type in Clean, Haskell uses the `data` keyword.

www.manaraa.com

in the example) instead of monads for incorporating side-effects.

We assume that writing separate bits is not directly supported. Hence, we keep track of some additional information for collecting bits until we have a complete byte that can be written. This is all hidden in the abstract data type `CSt` used in the function `compressBit :: Bool (CSt *io) → (CSt *io) | GWrite io`[3] The packing function itself is straightforward:

```
generic gCompress a :: a → (CSt *io) → (CSt *io) | GWrite io

gCompress {|UNIT|}  x                 = id
gCompress {|PAIR|}  cx cy (PAIR x y)  = cy y o cx x
gCompress {|EITHER|}  cl cr (LEFT l)  = cl l o compressBit False
gCompress {|EITHER|}  cl cr (RIGHT r) = cr r o compressBit True
gCompress {|OBJECT|}  co (OBJECT x)   = co x
gCompress {|CONS|}  cc (CONS x)       = cc x
```

Unlike instances for ordinary classes, we define type instances for `gCompress` using the special parentheses {| |}. Moreover, the actual number of arguments of each instance depends on the arity (actually the *kind*) of the used type constructor. For example, the instance for `PAIR` as well as for `EITHER` are supplied with two extra parameters that can be used to compress the arguments of these types. Finally, the (auxiliary) types `OBJECT` and `CONS`, were added to the set of generic types in Clean to give access to concrete information about the original type definition (via `OBJECT`) and data constructors (via `CONS`). In this example this information is not used, but in section 5 it plays a crucial role.

The generic instances for other types that are actually used inside a program are automatically derived. For example, if one needs an instance of `gCompress` for the following `Tree` type

```
::  Tree a = Leaf | Node a (Tree a) (Tree a)
```

it suffices to declare **derive** `gCompress Tree`. For certain types, such as function types and abstract types, the generic function cannot be derived automatically. The programmer can, however, define specialized instances for these types explicitly.

In contrast to Generic Haskell, the implementation of generics in Clean is based on type classes. Each generic function gives rise to a collection of so-called *kind indexed* type classes, i.e. classes of which the signatures are obtained from the type of the generic function via *kind indexing* [2]. These generic type classes can be used as any other type class. Therefore, they are allowed in contexts of type signatures or can be applied in expressions like any other overloaded operation.

3 Basic generic IO operations

In this section we will introduce two generic functions, `write` and `read`, that serve as a basis for defining our set of general data storage operations. In essence, the write function is an 'aligned' version of `gCompress` as defined in the previous section: for the representation of constructors `write` always uses one byte whereas `gCompress`

[3] Clean separates argument types by whitespace, instead of →, and uses a | to announce context restrictions. In Haskell this would be written as `(GWrite io)` ⇒ ···.

[4] The operator o is function composition.

www.manaraa.com

represented constructors by a bit sequence of variable size. Besides the class `GWrite` for writing, we define the class `GRead` for reading characters from an IO container.

```
class GRead st where readC :: *io → (Maybe Char, *io)
```

Two instances of both `GWrite` and `GRead` are predefined:

```
instance GWrite GString, GFile;        instance GRead  GString, GFile
```

The types `GString` and `GFile` are basically strings (arrays of characters) and files. Objects of these can be created (opened) and released (closed) by using the following functions

```
openString :: *String → *GString;                    closeString :: *GString → *String
openFile   :: !String !*World → (Maybe *GFile, *World); closeFile :: !*GFile !*World → *World
```

In Clean the (unique) `World` type is special: it has only one inhabitant which, in essence, encapsulates the complete status of the machine. This world-object corresponds to the hidden state of the `IO` monad in Haskell, and is used, for instance, to perform I/0.

The basis of our generic sequentialization operations are two generic functions:

```
generic write a :: Int a *io → *io            | GWrite io
generic read a  :: Int *io   → (Maybe a, *io) | GRead io
```

The integer parameter is an auxiliary argument. In case a constructor is written/read, this argument contains the number of that constructor (i.e. the constructor's position in the corresponding algebraic type definition). However, for top level calls the value of this argument is ignored: we will usually supply 0 as a default argument.

Suppose we want to use these operations to sequentialize a tree. The values stored in the tree are elements each consisting of an (integer) key and a record containing names and email addresses of persons.

```
::  Elem a[5]  = { key :: Int, rec :: a }
::  Person     = { name :: [Char], email :: [Char] }
```

We can now use the operations on a tree object as defined in the previous section by first declaring the necessary generic instances:

```
derive write Tree, Elem, Person, []; derive read Tree, Elem, Person, []
```

To store a tree of 4 records (created by `createTree`) to the file `fname` we simply write:

```
writeTreeToFile fname world
    # (Just file, world)[6] = openFile fname world
    = closeFile (write{|*|}[7] 0 (createTree 4) file) world
where
    createTree size = foldl (λl k → Node {key = k, rec = record k} l Leaf) Leaf [1..size]

    record k
        #[8] kl = fromString (toString k)
        = {name  = ['John Doe']++kl, email = ['john.doe']++kl++['@cs.ru.nl']}
```

[5] Record types in Clean are surrounded by { and }.
[6] This is not a cyclic `let`definition: the `#` introduces a new scope for id's on the LHS.
[7] `write{|*|}` denotes the overloaded `write` operation for kind `*`.
[8] Clean supports let-before (`#`) to increase readability.

www.manaraa.com

The generic implementations of `read` and `write` (used to generate instances for concrete types) are rather straightforward. As with `gCompress`, each constructor is represented by a number corresponding to the constructor's position in the type definition. Because `write` is almost identical to `gCompress`, we restrict ourselves to the definition of `read`. The instance for the `OBJECT` case takes care of reading the constructor's id. To get the bits of this id in the right order (such that the `EITHER` alternative can choose the correct `LEFT` or `RIGHT` branch on the basis of the least significant bit) they are swapped first before passing them to the continuation.

```
read {|UNIT|}  _ io     = (Just UNIT, io)
read {|EITHER|} read_a read_b i io
    | i bitand 1 ≠ 0 = read_b (i >> 1) io >>= RIGHT
    | otherwise      = read_a (i >> 1) io >>= LEFT
read {|PAIR|}  read_a read_b _ io
    = case read_a 0 io of (Just x, io) → read_b 0 io >>= PAIR x
                          (Nothing,io) → (Nothing, io)
read {|OBJECT of {gtd_num_conses}|}  read_a _ io
    = case readC gtd_num_conses io of
        (Just cn, io) → read_a (swap cn gtd_num_conses) io >>= OBJECT
        (_, io)       → (Nothing, io)
read {|CONS|}  read_a _ io     = read_a 0 io >>= CONS

(>>=) (mb, io) f = case mb of Just x → (Just (f x), io); Nothing → (Nothing, io)
```

4 Updatable data storage

One of the main disadvantages of the proposed IO operations is that they do not allow updates of existing data without reading in and writing back all the original information. Particularly, when large amounts of data are involved this can be very inconvenient or even unacceptable. In this section we present an extension of our library with so-called *chunks*. Chunks can be seen as references to data still residing in the IO container. By wrapping data in a chunk, the user indicates that this information remains in the container when the data structure containing this wrapped data is read into the application. As soon as this data is really needed it can be read explicitly via a special `chunkValue` operation. Moreover, and this is possibly the most important improvement, it can be selectively updated, using the `updateChunk` operation. Hence, this chunk mechanism provides selective read/write access to data without the requirement of reading/writing the entire data structure.

The implementation of chunks requires that the IO streams provide random access. This is achieved via an extra type class:

```
class GIO io | GWrite io & GRead io where
    position :: *io → (Int, *io);  seek :: Int *io → *io
```

The contexts `GWrite` and `GRead` are added for convenience. Chunks and the signatures of the operations on chunks are defined as follows.

```
:: Chunk a;        :: *CHS io[9]

openChunks  :: *io → (Bool,Chunk a,CHS *io) | GIO io
closeChunks :: (CHS *io) → *io
newChunk    :: a (CHS *io) → (Chunk a, CHS *io)   | write{|*|} a & GIO io
```

[9] An abstract data type in Clean. The * before `CHS` indicates that any `CHS`-object must always be unique. Therefore, the * can then be omitted in the rest of the code.

www.manaraa.com

```
chunkValue  :: (Chunk a) (CHS *io) → (a, CHS *io) | read{|*|} a & GIO io
updateChunk :: (Chunk a) a (CHS *io) → CHS *s     | write{|*|} a & GIO io
```

The type `Chunk` for representing chunk objects is abstract. The other abstract type, called `CHS`, is used to allocate new chunks and to sequentialize the other chunk operations. By making this type unique, single threadedness is guaranteed, hence in-place updates can be performed safely. The operation `openChunks` first inspects the container to determine whether or not it is empty. In case of an empty container, a new empty chunk (the so-called *root chunk*) is created. Otherwise, the root chunk of the container is read. The boolean result value indicates which of these possibilities occurred.

To illustrate the use of these operations we return to our `Tree` example. The idea is to wrap the stored records in chunks so that they will be left in the container when the tree data structure is read in.

First, we inject the tree built by `createTree` with chunks. By using a *generic state map* this can be specified easily in just a few lines of code.

```
addChunks :: (a (Elem b)) (CHS *c) → (a (Elem (Chunk b)), CHS *c) | write{|*|} b & GIO c &
                                                                    gMapLSt {|* → *|} a
addChunks tr io = gMapLSt {|* → *|} addChunk tr io where
    addChunk {key,rec} [10] io = let (ch, nio) = newChunk rec io in ({ key = key, rec = ch }, nio)
```

The generic function `gMapLSt` from the standard Clean library maps a state transition function (in this case `addChunk`) over the tree `tr`. The effect of applying `addChunks` is that the original records are stored in the IO container and the tree is updated with references to these records. Note that `addChunks` is overloaded in `gMapLSt {|* → *|}` which means that it actually can be applied to *any* data type of kind $* \rightarrow *$. The tree structure itself is stored in the root chunk. The complete program looks as follows:

```
writeChunkTreeToFile fname world
    # (Just file, world)  = openFile fname world
      (ok, chunk, chunks) = openChunks file
      (ch_tree, chunks)   = addChunks (createTree 100000) chunks
    = closeFile (closeChunks  (updateChunk chunk ch_tree chunks)) world
```

The tree constructed by `createTree` above is completely out of balance: all elements are stored in the right branches. The use of chunks enables us to re-balance the tree without touching the stored records. This can be achieved as follows (we left out the open and close operations which are the same as in the previous example). Below, `balance` takes an arbitrary tree and converts it into a balanced one.

```
(unbal_tree, chunks) = chunkValue chunk chunks
 chunks = updateChunk chunk (balance unbal_tree) chunks
```

Obviously, after balancing the tree we now can quickly find a particular record and update its contents directly via an `updateChunk`.

5 Type-safe IO operations

Another serious disadvantage of the sequentialization method from the previous section, and in fact of almost any compress/uncompress, pack/unpack, or read/show

[10] `{f,g}` denotes the (lazy) selection of the fields `f` and `g` from the record argument.

www.manaraa.com

operation, is that the written information is untyped. Structures written as trees can be read back as lists sometimes leading to a crash or maybe without being noticed. In this section we will show how to extend our IO operations such that they are *type safe*. Note that this problem cannot be solved statically, the data in underlying IO containers (strings, files etc) is untyped. We have to add type information about the actual data to the data that is stored. But all our operations are defined in a generic way. How do we dynamically derive types of objects such that this information can be stored in the container and retrieved from it when it is re-opened for reading?

The solution is to use generics again. Remember that the generic compiler is able to construct a version of a generic function for any concrete type. Unfortunately, the concrete type on which a generic function is applied is not known. The reason is that generic functions are defined on type *constructors*, not on concrete types. In both the generic definition on type `OBJECT` and type `CONS` we can inspect a representation of the type *definition* corresponding to the actual type constructor and concrete data constructor on which the generic function is applied. We therefore have to define a generic function `type` that uses these type definitions to construct a representation of the actual type. For non recursive types the construction is straightforward. Recursion however, requires a more subtle approach. The main problem with recursion is that, in order to derive an exact representation of the actual type, one has to traverse the generic representation of that type in such a way that all type information is indeed collected without getting into a infinite loop.

As a starting point for our description we define the signature of `type` as follows:

```
generic type a :: (Maybe a, Type)

::  TypeCons    :== String
::  TypeVar     :== Int

:: Type = TVar TypeVar | TApp  TypeCons [Type] | TArr  Type Type | TBas  BType | TEmp
```

`Type` is used for representing types as values. We use integers and strings to identify type variables and type constructors, respectively. The auxiliary `TEmp` constructor is intended to be used as an initial or default value. Note that, although we are only interested in the type, we are forced to use the generic type variable `a` in the type signature. The use of `Maybe` allows us to deliver `Nothing` as a concrete dummy value. In the library itself, this result is used to construct a default value of the requested type which can, for instance, be used to initialize new objects.

Before explaining the working of the generic type construction we illustrate its use. The aim is to define a type safe version of our basic generic IO operations. We first introduce a wrapper type to include type information in the type of the IO container, together with a function for constructing such a container. This function also takes care of the type checking.

```
:: *GStream a io = { state :: io }

openStream :: *io → (Bool, GStream a *io) | GIO io & type{|*|} a
openStream io
    # (m, t) = type{|*|}
      io = forceType io m
    | isEmptyIO io
        = (True, { state = write{|*|} 0 t io })
```


www.manaraa.com

```
        # (maybe, io) = read{|*|} 0 io
        = (isJust maybe && fromJust maybe == t, { state = io })
where
    forceType :: (GStream a *io) (Maybe a) → GStream a *io
    forceType io y = io
```

isEmptyIO checks whether io is empty. In that case the type representation (yielded by the generic type function) is written to the container. If the io is not empty, read tries to read the type of the data that has been stored previously, and compares it to the requested type. forceType is just an auxiliary function connecting the value type of the stream to the value type of m. Without forceType there would be no connection between those two, leading to a (static) internal overloading error. All other operations are simple wrappers.

```
writeValue :: a (GStream a *io) → GStream a *io | GIO io & write{|*|} a
writeValue val io=:{state} = { io & state = write{|*|} 0 val state }

readValue :: (GStream a *io) → (Maybe a,GStream a *io) | GIO io & read{|*|} a
readValue io=:{state} = let (v, nst) = read{|*|} 0 state in = (v, { io & state = nst })

closeStream :: (GStream a *io) → *io
closeStream {state} = state
```

The next small program shows how our example tree (this time without chunks) can be read from a file. Note that the additional Stream layer makes the type safety checks completely invisible.

```
readTree :: String *World → (Maybe (Tree (Elem Person)),*World)
readTree tree_file world
    # (Just file, world) = openFile tree_file world
      (ok, stream)       = openStream file
    | ok  # (val, stream) = readValue  stream
          = (val, closeFile (closeStream stream) world)
          = (Nothing, closeFile (closeStream stream) world)
```

To adjust readTree such that it can read a possibly completely different structure it suffices to change the result type only. The fact that readTree indeed reads a tree of the specified type is completely determined by that type and not by the code.

The question that remains is how to define type such that it will derive proper types. As usual, we have to provide instances for our predefined generic type constructors. This time, however, the alternative for OBJECT plays a crucial role. As said before, one of our main concerns is not getting into an infinite computation. To illustrate the danger, we start with the instance of type that will be generated for lists by writing **derive** type []. The structure of this generated function, say type_List, is completely determined by the definition of the list type.

```
::  List a = Nil | Cons a (List a)

type_List type_a = bimap (type_OBJECT (type_EITHER
                   (type_CONS type_UNIT)  (type_CONS (type_PAIR type_a (type_List type_a)))))
```

The function bimap takes care the conversions between list object and its generic representation. The precise structure of this function is irrelevant to this presentation. If, on the right-hand side of type_List, each instance function would directly call its continuation this would finally lead to a call of type_List itself and thus to an infinite computation. To prevent this we supply type with an auxiliary *history* argument containing a description of the subsequent calls done so far. The idea is to mark an

www.manaraa.com

edge in the (directed) calling graph that leads form a data constructor C to a type constructor T, say via C's n^{th} argument, by the formal type belonging to that argument. E.g. in the above list example the recursive call to `type_List` will be marked with `List (TVar n)`, i.e. the formal type of `Cons`' second argument. The number `n` of the type variable is supposed to be *fresh*, i.e. not used before. Assigning such fresh numbers to new variables requires an additional argument for `type` maintaining the variable counter.

When entering `type_List` it is checked whether the corresponding calling edge already occurs in the history. If so, the function returns immediately. Otherwise the underlying type is constructed by calling the continuation. However, in some cases a single traversal of a type is not sufficient to derive a type completely. Consider for example the following definition of an alternating list.

```
:: AList a b = ANil | ACons a (AList b a)
```

To obtain the representation of, for example, `AList Int Char` the instance of `type` for `AList` has to pass through `type_AList` twice. Our 'cycle detection mechanism' automatically takes care of that. The latter also counts for more complex types like

```
:: Rose a = Rose a (List (Rose a))
```

and even for *non-uniform recursive data types*, such as the following rather exotic type definition

```
:: Exot a  = ENil | ECons a (Exot (Exot a))
```

The fresh variables are not only used for creating auxiliary unique markers but also for the construction of the requested type. Pieces of this type are collected little by little and connected to each other via *unification*. This requires instantiation of data constructor types each time a new type constructor is encountered. All this is taken care of by the instance of `type` for `OBJECT`. One further detail of our implementation is that unification is not performed on–the–fly but deferred until all information about the type has been gathered. For this reason, all type equations resulting from the encountered dependencies are collected, and solved at the end of the type construction process. All this leads to the following signature for `type`.

```
:: History :== [TYPE]; :: Marker :== Type
:: TypeSt = { fresh :: Int, equa :: [(Type,Type)]}

generic type a :: Type History Marker TypeSt → (Maybe a, TypeSt)
```

The first argument of `type` is the type requested by the context. As soon as an instance of `type` produces a concrete type, the equation of this type and the requested one is added to the list of equations. For example, the instance of `type` for `Int` is defined as:

```
type {|Int|} rt history mark ts = (Nothing, { ts & equa = [(rt, TBas BInt) : ts.equa]})
```

From the explanation above, it should be clear that the instance for `OBJECT` is much more involved than the previous one for `Int`. Due to space limitations it is omitted, so are the almost trivial definitions for `UNIT`, `EITHER` and `PAIR`.

The generic type function is not suited for being used by programmers. First of all, there are several arguments that are only used internally. Secondly, it does not

www.manaraa.com

yield a type but a set of equalities that still have to be solved. We can hide this implementation specific information by using the following wrapper.

```
defaultType :: (Maybe a, Type) | type{|*|} a
defaultType
    # fv = TVar 0
      (m, ts) = type{|*|} fv [] TEmp { fresh = 1, equa = [] }
      subst = unify ts.equa {TEmp \\ _ ← [0 .. ts.fresh]}
    = (m, Subst fv subst)
```

We use the (fresh) variable 0 as requested type for the top call of `type`. Both history and (recursion) marker are empty, indicated by `[]` end `TEmp` respectively. After calling `type`, the collected equations are passed to `unify`. This standard unification algorithm takes a list of type equations as well as a substitution as input and produces a new substitution solving those equations. We represent the substitution by an array of types in which type variables are used as indices. We start with an empty substitution which then will be updated (destructively) by `unify`. To obtain the final type this unifier is applied to variable 0.

6 Related Work

To the best of the authors knowledge the use of generic programming techniques for the creation of a type safe destructive database system is new.

Serialization and de-serialization are standard examples in any introductory paper on generic or type driven programming techniques [8,7,10]. Data which is serialized in such a way can be stored and retrieved from persistent memory, but it lacks the ability to destructively update substructures. This is an absolutely necessary feature for any real world database system. Instead of using generic techniques one can of course also use the overloading mechanism [14]. However, this has the disadvantage that the programmer has to provide the proper instances for reading and writing of data explicitly.

In [5] a persistent storage for Haskell is presented. The basic idea is to extend the application memory with a persistent memory. When data is (explicitly) shifted from one memory to another, the internal representation of data in the application is converted to the external representation in the storage, and backwards. For the conversions support of the run time system is needed. Destructive updating is not provided, but when structures are modified, sharing is maintained in the storage in the same way as it is common in the application memory. The approach cannot be used to exchange data between different applications.

In Amber [4], CAML [11] and Clean [12] one can store and retrieve values of any type by wrapping them into a *Dynamic*. A Dynamic contains both a representation of a value and a representation of it's type. This method does enable the type safe exchange of data between applications, but one needs support from the compiler and run time system to be able to check the type consistency and take care of the conversions. In Clean it is even possible to exchange *functions* between applications [15]. Part of the run time systems is a dynamic linker which can extend a running executable with additional functionality by loading the corresponding code needed for the evaluation of new functions. However, none of the Dynamic approaches provide support for destructive updates.

www.manaraa.com

7 Conclusions

In this paper we have presented a set of generic IO operations that are flexible, efficient and type-safe. They are flexible because with a simple derive statement data of any concrete type can be stored and retrieved. They are efficient because data structures can be partially read in and stored data can be updated destructively. They are type safe because a representation of the concrete type of the data that is written is stored as well. This type is checked against the type of data requested by the reader, before the data structure can be read in.

We have tested the operations with fair-sized data sets (100,000 records) and the efficiency appeared to be sufficient for most cases. However, if performance is critical the overhead introduced by the common generic code generation scheme can be fully eliminated by using the extended fusion technique described in [3]. The resulting code appears to be very efficient. Our tests showed that our system can be one to two orders of magnitude faster than standard sequentialization methods or data base implementations.

Our system has been integrated in the iTask Workflow System [13]. It allows the high level specification of interactive workflows from which a multi-user web-based workflow system is generated fully automatically. In a workflow system, storage and retrieval of information plays a vital role. However, a programmer specifying a workflow in the iTask system does not have to worry anymore about the low level details commonly attached to database access. Any type of information can now be stored and retrieved automatically making use of our generic read and write functions. In this way database access can be completely hidden for the workflow programmer. It furthermore enables the definition of re-usable workflow schemes on a high level of abstraction. In this way one can focus on the specification of the actual workflow. Without our generic data storage approach this would not have been possible.

References

[1] Peter Achten. Clean for Haskell98 Programmers. Technical report, Computing Science Institute, Faculty of Mathematics and Informatics, University of Nijmegen, The Netherlands, 2007. `http://www.st.cs.ru.nl/papers/2007/CleanHaskellQuickGuide.pdf`.

[2] Artem Alimarine and Rinus Plasmijer. A generic programming extension for clean. In Thomas Arts and Markus Mohnen, editors, *Proceedings of the 14th International Workshop on Implementation of Functional Languages, IFL 2001*, pages 257–278, Stockholm, Sweden, September 2001. Ericsson Computer Science Laboratory.

[3] Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *The 7th International Conference, Mathematics of Program Construction*, number 3125 in LNCS, pages 16 – 31. Stirling, Scotland, UK, Springer, July 2004.

[4] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and functional programming languages : Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985*, volume 242. Springer-Verlag, 1986.

[5] A. Davie, K. Hammond, and J. Quintela. Efficient persistent haskell. In *Draft Proc. 10th International Workshop on the Implementation of Functional Languages (IFL '98)*, pages 183–194, 1998.

[6] Ralf Hinze. Generic Programs and proofs, 2000. Habilitationsschrift, Universität Bonn.

[7] Ralf 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.

www.manaraa.com

[8] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[9] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[10] Ralf Laemmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37, Charleston, January 2003. ACM.

[11] Xavier Leroy. *The Objective Caml system – release 3.08; Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, July 2004.

[12] M.R.C. Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98)*, volume 1595 of *LNCS*, pages 169–185. Springer Verlag, 1999.

[13] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proc. of the 2007 ACM SIGPLAN Intern. Conf. on Functional Programming*, pages 141–152. ACM, Oct 1-3 2007.

[14] Andr Santos and Bruno Abdon Monteiro. A persistence library for haskell. In *SBLP'2001 - V Simpsio Brasileiro de Linguagens de Programao*, May 2001.

[15] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Peña and Thomas Arts, editors, *The 14th International Workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, volume 2670 of *LNCS*, pages 101–117. Springer, September 2003.

www.manaraa.com