Generics for the Working ML'er

Vesa A. J. Karvonen

Department of Computer Science
University of Helsinki, Finland.
vesa.karvonen@cs.helsinki.fi

Abstract

Generic values are *type-indexed values* defined over the structure of types. A popular and pragmatic approach to type-indexed values in ML-like languages is to use a *value-dependent encoding*, where the type representations carry the values being indexed. Unfortunately, the approach has a major drawback. Because the resulting encoding is specific to a particular set of values, extending an encoding with new values requires modifying it.

In this paper, we discuss an approach to generics based on the value-dependent encoding approach. We extend upon previous work in several ways. We present a technique that allows an existing value-dependent encoding to be extended with new values. We show how to encode type representations over essentially all the ML-types. We also show how to compute fixed points over arbitrary products.

Our techniques have been implemented in Standard ML, do not compromise abstraction, and require only a fixed number of combinators.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Generic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.11 [*Software Engineering*]: Software Architectures

General Terms Design, Languages

Keywords generic programming, type-indexed values, ML

1. Introduction

A *generic value* can be used at many different types. For example, a generic *pickling* function consumes a value of a given type and produces a representation of the value as a stream of bytes. On the other hand, a generic *unpickling* function consumes a stream of bytes and produces a value of a given type or signals an error. Like pickling and unpickling, many useful generics can be specified in a straightforward manner over the structure of types.

We differentiate between *generic*, or *closed type-indexed*, and *ad-hoc polymorphic*, or *open type-indexed* values [15, 14]. A generic value is defined over the structure of types and a single, closed, definition works for multiple types. On the other hand, the definition of an ad-hoc polymorphic value is left open and is given a specialized definition for each type, or, more systematically, for each type constructor.

Unfortunately, none of the main ML-dialects, including Standard ML and O'Caml, directly support type-indexed programming. Indeed, in an implicitly typed language based on the Hindley–Milner type system, types do not even appear explicitly as parameters of values. Fortunately, it is possible to encode a family of types as values that represent the types and write type-indexed values that take such encoded type representations as parameters.

Yang [19] differentiates between *value-dependent* and *value-independent* approaches to encoding types. In a value-dependent approach, the type representation essentially carries the value being indexed by types, while a value-independent encoding is independent of any indexed value. Because each generic using a value-dependent encoding has its own type representation, different generic values can not be directly composed. Extending a value-dependent type representation with new generics requires modifying the encoding. Observing this deficiency, Yang prefers a value-independent approach, which readily allows combining different generics. However, based on recent articles [16, 8, 12, 2, 1], it would seems that most practitioners have chosen value-dependent approaches without addressing the lack of composability. Furthermore, none of the previous approaches readily deals with all ML-types, such as mutually recursive datatypes.

In this paper, we discuss an approach to generics based on the value-dependent encoding approach. In Section 2, we illustrate our basic approach to encoding generic values via a simple example. In Section 2.3, we observe the ML *value recursion challenge* [18] and present a solution to it. In Section 3, we generalize our basic approach to deal with essentially all ML-types. In Section 4, we attack the composability problem and show how to leave type representations open for extension. In Section 5, we describe some applications of our generics library. In Section 6, we briefly highlight some weaknesses of our approach. In Section 7, we discuss some related work. Section 8 concludes.

Our example code has been written in Standard ML and requires no language extensions. We make use of a number of open-source libraries, including an Extended Basis Library, that provides some basic utilities beyond the Standard ML Basis Library [9]. These libraries, as well as our library for generics, can be found from the public MLton [4] SVN repository at `svn://mlton.org/mltonlib/trunk`.

2. Towards Generics

As an illustrative example, we work through an implementation of a generic equality predicate. More precisely, we wish to implement a function eq with the type α Eq.t $\rightarrow$ α BinPr.t, where Eq.t is the type constructor for the type representation and α BinPr.t is a type abbreviation for $\alpha \times \alpha \rightarrow$ bool, which is the type of binary predicates.

The Extended Basis Library makes extensive use of type abbreviations and provides home modules for most types. In general,

α Sq.t	=	α × α
α UnOp.t	=	α → α
α BinOp.t	=	α Sq.t → α
α Fix.t	=	α UnOp.t → α
α Thunk.t	=	Unit.t → α
α Effect.t	=	α → Unit.t
α UnPr.t	=	α → Bool.t
α BinPr.t	=	α Sq.t → Bool.t
α Cmp.t	=	α Sq.t → Order.t
α Iso.t	=	(α → β) × (β → α)
α Emb.t	=	(α → β) × (β → α Option.t)

Figure 1. Type Abbreviations

for a type x, the Extended Basis library defines a structure X and a type X.t = x. Some frequently used type abbreviations are defined by the equations in Figure 1. For consistency with the libraries, we will use the same type abbreviations in this paper and, after defining types, refer to them through their home modules.

2.1 Base Types and Primitive Type Constructors

We observe that values of *equality types* like Int.t and String.t can be readily tested for equality using SML's polymorphic equality:

```
val int : Int.t BinPr.t = op =
val string : String.t BinPr.t = op =
```

Values of some non-equality types, like Real.t, can also be tested for equality in various ways. A useful notion of equality for reals is bitwise equality. Bitwise equality for reals has the pleasing property that two bitwise equal reals can always be substituted for each other without changing the meaning of any program. This is important for applications such as pickling. It is important that pickling and unpickling a data structure does not change it in subtle ways. We can implement bitwise equality through the standard PackRealBig module:

```
val real : Real.t BinPr.t =
 fn (l, r) ⇒ PackRealBig.toBytes l = PackRealBig.toBytes r
```

Using higher-order functions, we can build equality predicates for parameterized type constructors, such as Option.t and List.t:

```
val list : α BinPr.t → α List.t BinPr.t = ListPair.allEq
val option : α BinPr.t → α Option.t BinPr.t = fn eq ⇒
 fn (NONE, NONE) ⇒ true
  | (NONE, SOME _) ⇒ false
  | (SOME _, NONE) ⇒ false
  | (SOME l, SOME r) ⇒ eq (l, r)
```

As a cosmetic nicety, we have derived the names of the combinators from the names of the type constructors they support. An equality predicate for a particular type is built by transliterating the type expression into a term expression using the combinators. For example, the term

```
list (option string)
```

has the type

```
String.t Option.t List.t BinPr.t
```

and is an equality predicate for lists of optional strings.

At this point it becomes obvious that we can give trivial definitions to Eq.t and eq:

```
structure Eq = BinPr
fun eq f = f
```

In other words, the type representation for the generic eq predicate is an equality predicate. We build type representations from base type representations and type representation constructors, which are, in this case, higher-order functions for producing equality predicates. In value-dependent approaches to type-indexed values, the type representations carry the values being indexed.

2.2 Sums, Products and Isomorphisms

Continuing with the previous approach, defining a new combinator for each type constructor, we would end up with a combinator library that allows us to build equality predicates for an inductively defined family of types. This is essentially the value-dependent encoding described by Yang [19]. Furthermore, if the type representation is left concrete, the type family can be extended in an ad-hoc polymorphic fashion.

However, defining an ad-hoc equality predicate for each user defined type constructor is tedious work. Fortunately, we can avoid it. As observed by Hinze [10], in the context of Haskell, we can see the datatypes and record types of ML as producing types that are isomorphic to sums of products.

Let's first define datatypes for representing sums and products:

```
datatype (α, β) sum = INL of α | INR of β
datatype (α, β) product = & of α × β
infix &
```

We can then give a higher-order function for testing sums for equality:

```
infix 6 ⊕
val op ⊕ : α Eq.t × β Eq.t → (α, β) Sum.t Eq.t =
 fn (eqA, eqB) ⇒
    fn (INL l, INL r) ⇒ eqA (l, r)
     | (INL _, INR _) ⇒ false
     | (INR _, INL _) ⇒ false
     | (INR l, INR r) ⇒ eqB (l, r)
```

And another function for products:

```
infix 7 ⊗
val op ⊗ : α Eq.t × β Eq.t → (α, β) Product.t Eq.t =
 fn (eqA, eqB) ⇒
    fn (lA & lB, rA & rB) ⇒
       eqA (lA, rA) andalso eqB (lB, rB)
```

For working with sums, we also need a trivial equality predicate for the Unit.t type:

```
val unit : Unit.t Eq.t = fn _ ⇒ true
```

To specifically support user-defined types, we define a type constructor iso for isomorphisms:

```
type (α, β) iso = (α → β) × (β → α)
```

And a combinator iso for building comparison functions given an isomorphism:

```
val iso : β Eq.t → (α, β) Iso.t → α Eq.t =
 fn eqB ⇒ fn (a2b, b2a) ⇒ fn (lA, rA) ⇒ eqB (a2b lA, a2b rA)
```

We can now give a new definition for the option combinator through an isomorphism:

```
val option : α Eq.t → α Option.t Eq.t = fn a ⇒
    iso (unit ⊕ a)
        (fn NONE ⇒ INL () | SOME a ⇒ INR a,
         fn INL () ⇒ NONE | INR a ⇒ SOME a)
```

Or, for example, we can test triples for quality using the following definition:

```
val triple : α Eq.t × β Eq.t × γ Eq.t → (α × β × γ) Eq.t =
 fn (a, b, c) ⇒
    iso (a ⊗ b ⊗ c)
        (fn (a, b, c) ⇒ a & b & c,
         fn a & b & c ⇒ (a, b, c))
```

To recap, instead of directly implementing an ad-hoc equality predicate for a user-defined datatype, we encode the structure of the datatype as a generic sum of products, and provide an isomorphism between the datatype and the generic sum of products.

One might wonder whether this actually buys us anything. The new definition of the option combinator isn't any simpler than the old definition. However, there is an important qualitative difference between the two definitions. The new definition isn't specific to equality predicates. It merely encodes the structure of the type constructor. Changing only the definitions of iso, unit, ⊕, and ⊗, we could use the same definition of option as a type representation constructor for another generic.

2.3 The Value Recursion Challenge

The option and triple representations from the previous section were non-recursive. At first glance it might seem that we can already encode recursive datatypes such as List.t simply by using recursively defined representations:

```
val rec list : α Eq.t → α List.t Eq.t = fn a ⇒
    iso (unit ⊕ (a ⊗ list a))
        (fn [] ⇒ INL () | x::xs ⇒ INR (x & xs),
         fn INL () ⇒ [] | INR (x & xs) ⇒ x::xs)
```

While this passes the ML type-checker, applying list to any value diverges. Notice the recursive application of list. In this case, we could avoid the problem by η-expanding the definition:

```
val rec list : α Eq.t → α List.t Eq.t = fn a ⇒ fn ? ⇒
    iso (unit ⊕ (a ⊗ list a))
        (fn [] ⇒ INL () | x::xs ⇒ INR (x & xs),
         fn INL () ⇒ [] | INR (x & xs) ⇒ x::xs) ?
```

Unfortunately, η-expansion does not solve the problem in general, because it relies on the fact that our type representation is a function. In Standard ML, as noted by Kennedy [12] and Syme [18], recursive definitions are only allowed for syntactic functions. This isn't always the case for type representations. For example, the representations of picklers described by Elsman [8] and Kennedy [12] are both products of functions and the η-expansion trick can't be used. Even if we could use the η-expansion trick for defining type representations, it would violate abstraction by requiring us to reveal that type representations are functions.

2.3.1 Witnessing Fixed Points

Is it possible to compute recursive type representations without using recursion? Indeed it is. Without digressing into the theoretical foundations, a fixed point of a function $F\colon \tau \to \tau$ is a value $x\colon \tau$ such that $x = F(x)$. A fixed-point operator $\mathit{fix}_\tau\colon (\tau \to \tau) \to \tau$ has the property that $\mathit{fix}_\tau M = M(\mathit{fix}_\tau M)$. Any recursive definition $f\colon \tau = M$, where f may occur in M, can be represented by a function $F = \lambda f\colon \tau.M$. A solution to the definition may then be computed using a fixed-point operator as $f\colon \tau = \mathit{fix}_\tau F$. Of course, Standard ML does not provide such a fixed-point operator, so we need to define a suitable fixed point combinator ourselves.

While it is easy to write an ad-hoc fixed point combinator for a particular abstract type, it is more challenging to provide a general purpose framework for computing fixed points. First of all, a single combinator with a type of the form $(\tau \to \tau) \to \tau$ does not support mutual recursion over multiple values of type τ. To support mutual recursion, one might consider providing a family of fixed point combinators having types of the form $(\pi \to \pi) \to \pi$, where π is a product of the form $\tau \times \ldots \times \tau$. Unfortunately, even such a family of fixed point combinators would not support mutual recursion over multiple different abstract types. What we need is a type-indexed fixed point combinator fix : α Tie.t → α Fix.t, where Tie.t is the type constructor for the type representation for fixed point combinators. To avoid confusing the type representations of fixed point combinators and comparison functions, we refer to the type representations of fixed point combinators as *witnesses*.

In order to find a witness type for fixed point combinators, it helps to consider a general algorithm for computing fixed points. Suppose that we can simply guess a fixed point x' : α of the given function f : α → α. By definition, an application f x' will then compute a result x : α equal to x'. Of course, we can't just guess a fixed point. What we can do, however, is to create a mutable *proxy* for the fixed point and *tie* the proxy to the result *making* them equal. Assuming that the given function doesn't actually examine the value of the proxy, we have managed to compute a fixed point.

Generalizing slightly, we arrive at a definition for the fixed point combinator witness type and a definition for the type-indexed fixed point combinator:

```
structure Tie = struct
   type α t = (α × α UnOp.t) Thunk.t
end
val fix : α Tie.t → α Fix.t =
 fn wA ⇒ fn f ⇒ let val (a, tA) = wA () in tA (f a) end
```

The idea is that instantiating the witness thunk, wA, returns a proxy, a, and a function, tA, for tying the proxy to the result *and* computing the final result. Of course, whether or not an application fix a f actually computes a fixed point of f depends on both the witness a and the function f. In fact, it is not only possible, but also useful, to define witnesses that don't actually compute fixed points.

To recap, instead of directly defining fixed point combinators, we define witnesses from which fixed point combinators can be produced. Of course, by breaking up the computation of fixed points like this, we will not be able to support all possible ways of computing a fixed point of a particular type. For example, we can't use core language recursion to compute fixed points of functionals. However, we can use a hidden ref cell, like in the following witness for functions:

```
val function : (α → β) Tie.t = fn () ⇒ let
   val r = ref (fn _ ⇒ raise Fail "Fix")
in
   (fn x ⇒ !r x, fn f ⇒ (r := f ; f))
end
```

Using function, we can compute fixed points of functionals. For example, here is a definition of the factorial function:

```
val fact : Int.t UnOp.t =
   fix function (fn fact ⇒ fn 0 ⇒ 1 | n ⇒ n*fact (n−1))
```

The task of verifying that fact really is the factorial function is left as an exercise to the reader.

2.3.2 Combining Witnesses

An essential property of the witness representation is that we can combine witnesses to produce witnesses for new types. Indeed, we haven't quite solved the entire problem yet. What we really want is to seal the witness representation so that providing a witness for an abstract type does not jeopardize abstraction. To make witnesses abstract, we need to define a sufficient set of primitive combinators for manipulating them so that the representation can be hidden without loss of generality.

It turns out that we need only two primitive combinators: a product combinator and an isomorphism combinator. The product combinator allows us to compute fixed points over arbitrary products and the isomorphism combinator allows us to reuse existing witnesses at new types. Thanks to the non-recursive nature of witnesses, this turns out to be quite straightforward.

```
signature TIE = sig
   type α dom and α cod
   type α t = α dom → α cod
   val fix : α t → α Fix.t
   val pure : (α × α UnOp.t) Thunk.t → α t
   val tier : (α × α Effect.t) Thunk.t → α t
   val iso : β t → (α, β) Iso.t → α t
   val ⊗ : α t × β t → (α, β) Product.t t
   val tuple2 : α t × β t → (α × β) t
   val id : α → α t
   val function : (α → β) t
end
```

Figure 2. Signature for Fixed Point Framework

To pair two witnesses, we simply create a new witness that performs the operations of both and pairs the results:

```
val op ⊗ : α Tie.t × β Tie.t → (α, β) Product.t Tie.t =
 fn (aT, bT) ⇒ fn () ⇒ let
       val (a, fA) = aT ()
       val (b, fB) = bT ()
    in
       (a & b, fn a & b ⇒ fA a & fB b)
    end
```

The implementation of the isomorphism combinator is just as straightforward:

```
val iso : β Tie.t → (α, β) Iso.t → α Tie.t =
 fn bT ⇒ fn (a2b, b2a) ⇒ fn () ⇒ let
       val (b, fB) = bT ()
    in
       (b2a b, fn a ⇒ b2a (fB (a2b a)))
    end
```

2.3.3 Tying Representations

Figure 2 contains the complete TIE signature of the type-indexed fixed point framework, implemented as a module named Tie, which is a part of the previously mentioned Extended Basis Library. The type-indexed fixed point framework isn't specific to generics. For example, the Extended Basis library provides a module Lazy for lazy promises and the module also provides a fixed point witness for promises.

The TIE signature reveals that witnesses are functions, but leaves the domain and codomain abstract. This makes it possible to use η-expansion to work around the value restriction and implement polymorphic witnesses. The signature also provides the combinators pure and tier for introducing ad-hoc witnesses. Leaving the domain and codomain abstract does not prevent malicious code from obtaining a value of the codomain given a witness. To make the implementation safe, the codomain is the witness thunk and the combinators have an additional layer of thunking. This makes it impossible to apply an instantiated tying function multiple times, which would result in updating the corresponding proxy multiple times.

Using the Tie module, we can now give a trivial witness for the type representation of our generic equality predicate:

```
val Y : α Eq.t Tie.t = Tie.function
```

Using the witness, we can then define representations for testing recursive datatypes for equality.

Here is a new definition for the List.t datatype:

```
val list : α Eq.t → α List.t Eq.t = fn a ⇒
    Tie.fix Y (fn aList ⇒
       iso (unit ⊕ (a ⊗ aList))
           (fn [] ⇒ INL () | x::xs ⇒ INR (x & xs),
            fn INL () ⇒ [] | INR (x & xs) ⇒ x::xs))
```

The new definition no longer depends on the type representation being an arrow type and does not require such implementation details to be revealed.

The Tie module allows us to define type representations for mutually recursive datatypes. We simply need to produce a fixed point witness for a product of type representations:

```
datatype foo = FOO of bar Option.t
     and bar = BAR of foo × foo × foo
val (foo : foo Eq.t) & (bar : bar Eq.t) =
    let open Tie in fix (Y ⊗ Y) end
       (fn foo & bar ⇒
           iso (option bar) (fn FOO ? ⇒ ?, FOO) &
           iso (triple (foo, foo, foo)) (fn BAR ? ⇒ ?, BAR))
```

Note that we didn't have to implement a new ad-hoc fixed point witness for pairs of type representations. We simply used the existing witness and the ⊗ combinator.

Unlike in previously published approaches to value-dependent encodings of type-indexed values [8, 12], we are able to support mutually recursive datatypes using only a fixed number of combinators.[1]

3. Closed Generics

Having worked through the basic approach and the value recursion challenge in the previous section, we are now ready to generalize the approach to the remaining ML-types. Figure 3 shows the complete CLOSED_REP and CLOSED_CASES signatures for our approach to closed generic functions. The idea is that a generic value is defined by implementing the CLOSED_CASES signature and thereby defining the *structural cases* or type representation constructors that define the generic. The term "closed" refers to the fact that a generic implementing the signature does not support being extended with new generics. The Generics module, whose signature is shown in Figure 4, provides some auxiliary definitions for generics. The rest of this section discusses the design of the CLOSED_CASES signature.

3.1 Finer Type Representations

The signature contains several minor enhancements over the basic approach from the previous section. The enhancements make the definitions of type representations roughly isomorphic to the definitions of the corresponding types, allow a generic to better distinguish between different ML-types, and help to ensure consistency of type representation constructor definitions. In particular, the type representation, specified by the Rep substructure, uses three type constructors, t, s, and p, essentially distinguishing between the representations of complete types, incomplete sums, and incomplete products. Although the use of binary sums and products is a convenient abstraction for many generics, ML datatypes and products are not binary, and some generics need to able to distinguish between incomplete and complete specifications of a type.

[1] Pun intended!

```
signature CLOSED_REP = sig
   type α t and α s and (α, κ) p
end

signature CLOSED_CASES = sig
structure Rep : CLOSED_REP
val iso : β Rep.t → (α, β) Iso.t → α Rep.t
val isoProduct : (β,κ) Rep.p → (α,β) Iso.t → (α,κ) Rep.p
val isoSum : β Rep.s → (α, β) Iso.t → α Rep.s
val ⊗ : (α,κ) Rep.p × (β,κ) Rep.p → ((α,β) Product.t,κ) Rep.p
val T : α Rep.t → (α, Generics.Tuple.t) Rep.p
val R : Generics.Label.t → α Rep.t → (α,Generics.Record.t) Rep.p
val tuple : (α, Generics.Tuple.t) Rep.p → α Rep.t
val record : (α, Generics.Record.t) Rep.p → α Rep.t
val ⊕ : α Rep.s × β Rep.s → ((α, β) Sum.t) Rep.s
val C0 : Generics.Con.t → Unit.t Rep.s
val C1 : Generics.Con.t → α Rep.t → α Rep.s
val data : α Rep.s → α Rep.t
val unit : Unit.t Rep.t
val Y : α Rep.t Tie.t
val ⟶ : α Rep.t × β Rep.t → (α → β) Rep.t
val exn : Exn.t Rep.t
val regExn : α Rep.s → (α, Exn.t) Emb.t Effect.t
val array : α Rep.t → α Array.t Rep.t
val refc : α Rep.t → α Ref.t Rep.t
val vector : α Rep.t → α Vector.t Rep.t
val largeInt : LargeInt.t Rep.t
val largeReal : LargeReal.t Rep.t
val largeWord : LargeWord.t Rep.t
val word8 : Word8.t Rep.t
val word32 : Word32.t Rep.t
val word64 : Word64.t Rep.t
val list : α Rep.t → α List.t Rep.t
val bool : Bool.t Rep.t
val char : Char.t Rep.t
val int : Int.t Rep.t
val real : Real.t Rep.t
val string : String.t Rep.t
val word : Word.t Rep.t
end
```

Figure 3. Signatures for Closed Generics

```
signature GENERICS = sig
   structure Label : sig
      eqtype t
      val toString : t → String.t
   end
   structure Con : sig
      eqtype t
      val toString : t → String.t
   end
   structure Record : sig type t end
   structure Tuple : sig type t end
   val L : String.t → Label.t
   val C : String.t → Con.t
end
```

Figure 4. Signature for the Generics Module

Consider the following type representation constructor for the Option.t type constructor:

```
fun option a =
   iso (data (C0 (C "NONE") ⊕ C1 (C "SOME") a))
       (fn NONE ⇒ INL () | SOME a ⇒ INR a,
        fn INL () ⇒ NONE | INR a ⇒ SOME a)
```

An obvious difference to the previous definition in Section 2.2 is the addition of textual presentations for the constructors NONE and SOME. A more subtle difference is that the above explicitly specifies a single datatype with two constructors rather than a union of two single constructor datatypes.

If we would not distinguish incomplete sums from complete types, and would not have the Rep.s type constructor,[2] one could give the following inconsistent definition:

```
fun option a =
   iso (unit ⊕ C1 (C "SOME") a)
       (fn NONE ⇒ INL () | SOME a ⇒ INR a,
        fn INL () ⇒ NONE | INR a ⇒ SOME a)
```

Now a generic would not be able to distinguish the constructor NONE from the unit value (). For example, a generic function for converting values of representable types to human-readable strings corresponding to ML-syntax, would most likely convert the value NONE to the string "()" rather than the desired "NONE".

Many previously published approaches to generics allow such inconsistent definitions [5, 10]. The more refined typing of our approach does increase the surface complexity of the approach. It also doesn't rule out all inconsistencies. For example, although the type system ensures that all fields of a record must be given labels, one can still specify the same label for more than one field.

3.2 Primitive Type Constructors

The signature supports all the primitive type constructors of Standard ML. Primitive type constructors often require special consideration. As a case in point, ML allows new exception constructors to be introduced at almost any point in a program. For this reason, the signature specifies the regExn function for registering exception constructors. It takes a regular sum type representation and an embedding of the sum type into the exception type. The embedding allows the implementation of a generic to both construct and destroy registered exception values. Here is how one would register the standard Fail exception constructor:

```
regExn (C1 (C "Fail") string,
       (Fail, fn Fail m ⇒ SOME m | _ ⇒ NONE))
```

While most generics can be made to work with exceptions, the same can not be said of the other primitive type constructors. For example, there is no meaningful way to test functions for equality. Unsupported structural cases can be given fake definitions, whose application fails dynamically. For example, the generic equality provided by our library defines the arrow case as:

```
fun op ⟶ _ = fn _ ⇒ raise Fail "Eq.--> unsupported"
```

There is a way to restore type safety by adding one phantom type parameter for each primitive type constructor to the type representation type constructors. Each primitive type representation constructor would set the type parameter corresponding to its primitive type constructor to some abstract type yes and leave all other type parameters free or copy them from the argument, if any. Binary combinators would unify the phantom type parameters pointwise, effectively computing their disjunction. The isomorphism combinators would copy all the type variables from the argument to disallow cheating. A generic value that does not work with a specific

[2] Just replace all uses of Rep.s in CLOSED_CASES with Rep.t.

primitive type constructor would then specify that the corresponding phantom type parameter must unify with some abstract type no, which is not equal to yes. A generic equality function, for example, could have a specification of the form:

```
val eq : ({⟶ : no, refc : β, (* ... *)}, α) Rep.t → α BinPr.t
```

Unfortunately, while such a phantom typing scheme can certainly be made to work, it would violate abstraction by revealing structural details of abstract types. Furthermore, the value-restriction would make it quite burdensome, requiring free type variables to be explicitly grounded to no at definitions and generalized at uses.

Mutable types, references and arrays, are also often troublesome. The fundamental problem with mutable objects is that they have identity, which should ideally be preserved or observed in many cases. This can require careful coding. Mutable types, when combined with recursive types, also make it possible to create cyclic data structures. Indeed, the only way to create cyclic data structures in Standard ML is through mutable types.[3] This makes it easier to deal with cyclic data structures, in general, because it is sufficient to consider cyclicity at references and arrays. However, creating terminating algorithms that deal with cyclic data structures is, in general, difficult and costly. The difficulty is manifested when creating generic algorithms, because one has to consider, or choose to ignore, cyclic data structures right at the start—usually before one even has any cyclic data structure to use with the generic.

The bottom line is that the CLOSED_CASES signature mostly makes sense for generics that can be given meaningful definitions at all, or almost all, ML-types. Not all generic functions support all of the structural cases corresponding to primitive type constructors. For specialized generics, that only span a limited subset of types, it may be better to define specialized signatures. In such a case, the CLOSED_CASES signature can be interpreted as a design pattern or template.

3.3 Multitude of Structural Cases

Many of the structural cases specified by the CLOSED_CASES signature could also be encoded in terms of the other cases. For example, one could do with only a single word type, LargeWord.t, because a large word can represent the values of other word types. Also, List.t can be encoded as a recursive sum of products as shown previously.

One reason for the choice of several structural cases is the special treatment provided by Standard ML for the corresponding types. Strings and lists are given special syntactic support. The same goes for tuples and records. The performance characteristics of vectors can not be simulated through other types. Values of mutable types, references and arrays, have identity. Leaving any of these cases out would mean that many generics could not be given closed definitions.

However, the main reason for having so many structural cases is pragmatism. One often wants to treat many of the included types in special ways and having them in the signature makes it easier. On the other hand, having so many structural cases to support specialization of a particular generic to particular types is neither necessary nor sufficient. Type representations are first class values and it is possible to have several observably different representations of the same type. If a particular representation value doesn't provide the behavior one wants, it is possible to use a different value. An example of this is given in Section 5.

Finally, it should be noted that default encodings of non-primitive type constructors can easily be defined using functors. If one doesn't want to specialize a generic to all the structural cases, one can use such default encodings.

3.4 The Lack of Composability

It would be straightforward to implement several generic functions adhering to the CLOSED_CASES signature. For example, generic hashing, pretty printing, and pickling are definable. Unfortunately, the usefulness of such implementations would be somewhat limited. The main weakness of the CLOSED_CASES signature is that it is only sufficient for describing a single generic value.

Suppose, for the sake of argument, that we would have implemented two generics, show and some, having the following specifications:

```
val show : α Show.t → α → String.t
val some : α Some.t → α
```

The idea is that show converts any value of a representable type to a string, while some creates a value of any given representable type.

Now, consider defining and calling a function showSome in terms of show and some:

```
val showSome : α Show.t × α Some.t → String.t =
 fn (s, d) ⇒ show s (some d)
val () = print (showSome
   (Show.list (Show.refc Show.int),
    Some.list (Some.refc Some.int)))
```

The caller of showSome needs to provide two structurally identical type representations. In general, a function using n generic functions would need to be provided with n structurally identical representations. This is clearly undesirable.

4. Composable Generics

As argued by Yang [19], one should be able to decompose the task of writing complex generics into writing multiple simpler generics. The previously described approach only allows that by manual tupling of all the generics together in a non-modular fashion. In this section, we describe an approach that allows type representations to be left open in such a way that the representation can be extended later to support new generics. At the same time, the approach allows the definition of a generic to use other, previously defined, generics.

4.1 Extensible Representation

Recall the CLOSED_REP signature from Figure 3. Borrowing Berthomieu's technique for implementing inheritance and subtyping in ML [3], we derive a new signature OPEN_REP for extensible representations by adding a type parameter χ, representing an unknown value to be carried, to each type constructor in CLOSED_REP and functions for accessing the value:

```
signature OPEN_REP = sig
   type (α, χ) t and (α, χ) s and (α, κ, χ) p
   val getT : (α, χ) t → χ
   val mapT : χ UnOp.t → (α, χ) t UnOp.t
   val getS : (α, χ) s → χ
   val mapS : χ UnOp.t → (α, χ) s UnOp.t
   val getP : (α, κ, χ) p → χ
   val mapP : χ UnOp.t → (α, κ, χ) p UnOp.t
end
```

The extra type variable is reminiscent of row polymorphism. Essentially, open type representations are *open products*.

We can now give signatures for generic functions using an extensible representation. For example, our library provides the Eq

[3] Ignoring recursive functions.

generic for generic equality with the following signature:

```
signature EQ = sig
   structure Eq : OPEN_REP
   val eq : (α, χ) Eq.t → α BinPr.t
   val notEq : (α, χ) Eq.t → α BinPr.t
end
```

Another example is the signature of the Hash generic for a generic hashing function:

```
signature HASH = sig
   structure Hash : OPEN_REP
   val hash : (α, χ) Hash.t → α → Word.t
end
```

By convention, we refer to generics by the names of their type representation substructures. The signatures of all generics follow the same pattern with a type representation substructure followed by specifications of the generic values. In the rest of this paper, we will only show the value specifications, which will always include the name of the type representation substructure.

An idea behind the design of the signatures for generics is that they can be combined conveniently. Suppose that we would like to implement a generic hash set as a functor GenHashSet. We could specify the signature for the domain of the functor simply as a combination of EQ and HASH:

```
signature GEN_HASH_SET_DOM = sig
   include EQ HASH
   sharing Eq = Hash
end
```

The sharing constraint ensures that the same type representation is shared by both generic equality and hashing. The functor GenHashSet could then look like this:

```
functor GenHashSet (Arg : GEN_HASH_SET_DOM) :> sig
  type α t
  val new : (α, χ) Arg.Hash.t → α t
  val insert : (α × α t) Effect.t
  val contains : α t → α UnPr.t
  (* ... *)
```

Instead of having to provide both an equality predicate and a hashing function, the user of such a generic hash set would only provide a single type representation that provides the necessary operations.

The use of **include** to combine signatures depends on the signatures having no overlapping specifications. That is why EQ specifies a substructure named Eq and HASH specifies a substructure named Hash for the type representation. Of course, a more expressive language for signatures would eliminate the need for such trickery [17].

4.2 Extensible Structural Cases

Along with the extensible representation, we specify a signature OPEN_CASES for extensible structural cases. A snippet of the signature is shown in Figure 5. Each specification of the OPEN_CASES signature is derived from the corresponding specification of the CLOSED_CASES signature. Quite simply, each combinator now takes an extra parameter that defines what to do with the extra value carried by the type representation. This makes it possible to use existing structural cases to build additional representation values.

4.3 Composing Generics

A composable generic is implemented as a functor for extending a given type representation and structural cases with the generic. A composition of generics is defined by starting from some open generic and extending it with the desired generics. To see how this

```
signature OPEN_CASES = sig
   structure Rep : OPEN_REP
   val iso :
       (δ → (α, β) Iso.t → γ) →
       (β, δ) Rep.t → (α, β) Iso.t → (α, γ) Rep.t
   val ⊗ :
       (γ × δ → ϵ) →
       (α, κ, γ) Rep.p × (β, κ, δ) Rep.p →
       ((α, β) Product.t, κ, ϵ) Rep.p
   val Y : χ Tie.t → (α, χ) Rep.t Tie.t
   val regExn :
       (γ → (α, Exn.t) Emb.t Effect.t) →
       (α, γ) Rep.s → (α, Exn.t) Emb.t Effect.t
   val list : (γ → δ) → (α, γ) Rep.t → (α List.t, δ) Rep.t
   val int : γ → (Int.t, γ) Rep.t
   (* ... *)
end
```

Figure 5. A Snippet of the Signature for Open Structural Cases

is done, let's first look at the interfaces[4] of a couple of composable generics. In the next section we will see how the functors can be implemented.

The Eq generic from our library has the following interface:

```
signature EQ_CASES = sig
   include OPEN_CASES EQ
   sharing Rep = Eq
end
functor WithEq (Arg : OPEN_CASES) : EQ_CASES
```

A generic named TypeInfo, provides information on types:

```
val hasBaseCase : (α, χ) TypeInfo.s UnPr.t
val numAlts : (α, χ) TypeInfo.s → Int.t
val numElems : (α, κ, χ) TypeInfo.p → Int.t
```

We will later see how the information can be used in the implementations of other generics.

Unsurprisingly, the interface of TypeInfo is similar to the interface of Eq:

```
signature TYPE_INFO_CASES = sig
   include OPEN_CASES TYPE_INFO
   sharing Rep = TypeInfo
end
functor WithTypeInfo (Arg : OPEN_CASES) :
   TYPE_INFO_CASES
```

The Hash generic, already mentioned earlier, has the following interface:

```
signature WITH_HASH_DOM = TYPE_INFO_CASES
signature HASH_CASES = sig
   include OPEN_CASES HASH
   sharing Rep = Hash
end
functor WithHash (Arg : WITH_HASH_DOM) : HASH_CASES
```

As you can see, the Hash generic implementation uses the TypeInfo generic, whose implementation must be passed as an argument to WithHash.

Let's then produce a composition of the Eq, Hash, and TypeInfo generics. We start from a trivial, or identity, generic RootGeneric that *only* carries and constructs the extra value:

```
structure Root : OPEN_CASES = RootGeneric
```

[4] In Standard ML, there are no functor signatures.

Then we extend it with the TypeInfo and Eq generics:

```
structure TypeInfo = WithTypeInfo (Root)
structure Eq = WithEq (TypeInfo)
```

Then we add the Hash generic. Because WithHash depends on TypeInfo, we need to create an argument structure for it:

```
structure HashArg = struct
   open TypeInfo open Eq
   structure TypeInfo = Rep
end
structure Hash = WithHash (HashArg)
```

Note the opening of the Eq module and the binding of TypeInfo to Rep. We want WithHash to extend the Eq representation rather than the TypeInfo representation. The representation Eq.Rep and the structural cases of Eq are compatible with, but not the same as, the representation TypeInfo.Rep and the structural cases of TypeInfo. Of course, the representation Hash.Rep is still compatible with the TypeInfo.Rep representation.[5]

Then we define the combined generic as a module Open:

```
structure Open : sig
   include OPEN_CASES TYPE_INFO EQ HASH
   sharing Rep = TypeInfo = Eq = Hash
end = struct
   open TypeInfo open Eq open Hash
   structure TypeInfo = Rep and Eq = Rep and Hash = Rep
end
```

Note that the definition of Open makes all the generics use the same representation Hash.Rep and the structural cases come from Hash.

We are not quite done yet. Before we can actually use the generics, we need to close the structural cases. The functor CloseCases is provided for that purpose:

```
functor CloseCases (Arg : OPEN_CASES) :>
   CLOSED_CASES
      where type α Rep.t = (α, Unit.t) Arg.Rep.t
      where type α Rep.s = (α, Unit.t) Arg.Rep.s
      where type (α, κ) Rep.p = (α, κ, Unit.t) Arg.Rep.p
```

Using it we define the G structure:

```
structure G : sig
   include GENERIC TYPE_INFO EQ HASH
   sharing Open.Rep = TypeInfo = Eq = Hash
end = struct
   structure Open = Open
   open Open
   structure Closed = CloseCases (Open)
   open Closed
end
```

The GENERIC signature is defined as follows:

```
signature GENERIC = sig
   structure Open : OPEN_CASES
   include CLOSED_CASES
      where type α Rep.t = (α, Unit.t) Open.Rep.t
      where type α Rep.s = (α, Unit.t) Open.Rep.s
      where type (α, κ) Rep.p = (α, κ, Unit.t) Open.Rep.p
end
```

Whew! Involved as it seems, we have combined three *separately defined* generics TypeInfo, Eq, and Hash. Here is an example interactive session using the opened G module:

```
- val t = list int ;
val t = - : Int.t List.t Rep.t
- eq t ([], [1, 2]) ;
val it = false : Bool.t
- hash t [1, 2, 3] ;
val it = 0wx3AD9 : Word.t
- hash (list (list int)) [[1]] ;
val it = 0wx23 : Word.t
```

One way to use generics in a program is to simply define a combination of generics and write other modules of the program against that particular combination. Our library provides a default combination that includes several useful generics. In practice, we expect that reusable modules or libraries using generics are implemented as functors—like the GenHashSet functor hinted at earlier. A complete program then implements some particular combination of generics and instantiates the functors. This is arguably a workable, although not an ideal, solution.

```
functor WithEq (Arg : OPEN_CASES) : EQ_CASES = struct
   structure Eq : LAYERED_REP = LayerRep
      (structure Outer = Arg.Rep
       structure Closed = MkClosedRep (BinPr))
   val eq = Eq.This.getT
   fun notEq t = not ∘ eq t
   structure Layered : OPEN_CASES = LayerCases
     (structure Outer = Arg
      structure Result = Eq
      structure Rep = Eq.Closed
      (* ... closed structural cases ... *))
   open Layered
end
```

Figure 6. A Snippet of the WithEq Functor

4.4 Implementing Generics

In the previous section we simply assumed the existence of the WithEq and other functors. Let's look at their implementation in a bit more detail.

Consider the snippet of the WithEq functor shown in Figure 6. First the functor LayerRep is used to produce a layered type representation.[6] A layered representation basically contains a Closed representation sandwiched between an Outer and an Inner representation and allows it to be accessed. See Figure 7 for a snippet of the signature specifying a layered representation.

After creating the layered representation, the LayerCases functor is used to define the structural cases for the generic. The structural cases are implemented like in Section 2, but adhering to the CLOSED_CASES signature. The LayerCases functor effectively combines the given open and closed cases. The domain of the functor is shown in Figure 8.

As shown in Figure 9, the implementation of WithHash follows a similar pattern. The main difference is the use of the LayerDepCases functor to define the structural cases. It uses special structural cases to allow parameterized cases to depend on the previously defined generics. See Figure 10 for a snippet of the special structural cases. The idea is simply that each parameterized structural case is given the resulting type representation of the parameter.

[5] Here we actually pay a price for the ability to conveniently combine the generic signatures. With a more expressive language of signatures [17], we wouldn't need several bindings for the same representation structure.

[6] The MkClosedRep functor simply replicates a given unary type constructor.

```
signature LAYERED_REP = sig
 structure Outer : OPEN_REP
 structure Closed : CLOSED_REP
 structure Inner : sig
  include OPEN_REP
  val mkT : α Closed.t × χ → (α, χ) t
  (* ... *)
 end
 include OPEN_REP
  where type (α, χ) t = (α, (α, χ) Inner.t) Outer.t
  where type (α, χ) s = (α, (α, χ) Inner.s) Outer.s
  where type (α, κ, χ) p = (α, κ, (α, κ, χ) Inner.p) Outer.p
 structure This : sig
  val getT : (α, χ) t → α Closed.t
  val mapT : α Closed.t UnOp.t → (α, χ) t UnOp.t
  (* ... *)
end
```

Figure 7. Signature for Layered Representations

```
signature LAYER_CASES_DOM = sig
   structure Outer : OPEN_CASES
   structure Result : LAYERED_REP
      sharing Outer.Rep = Result.Outer
   include CLOSED_CASES
      sharing Rep = Result.Closed
end
```

Figure 8. Domain of the LayerCases Functor

```
functor WithHash (Arg: WITH_HASH_DOM): HASH_CASES =
struct
   type α t = α → {totWidth: Int.t, maxDepth: Int.t} → Word.t
   structure Hash = LayerRep
      (structure Outer = Arg.Rep
       structure Closed = MkClosedRep (type α t = α t))
   open Hash.This
   fun hash t = (* ... *)
   structure Layered = LayerDepCases
     (structure Outer = Arg and Result = Hash
      (* ... semi-closed structural cases ... *))
   open Layered
end
```

Figure 9. A Snippet of the WithHash Functor

```
signature LAYER_DEP_CASES_DOM = sig
   structure Outer : OPEN_CASES
   structure Result : LAYERED_REP
   sharing Outer.Rep = Result.Outer
   val iso : (β, 'y) Result.t → (α, β) Iso.t → α Result.Closed.t
   val ⊗ : (α, κ, χ) Result.p × (β, κ, 'y) Result.p
      → ((α, β) Product.t, κ) Result.Closed.p
   val Y : α Result.Closed.t Tie.t
   val regExn : (α, χ) Result.s → (α, Exn.t) Emb.t Effect.t
   val list : (α, χ) Result.t → α List.t Result.Closed.t
   val int : Int.t Result.Closed.t
   (* ... *)
end
```

Figure 10. A Snippet of the Domain of the LayerDepCases Functor

The extra flexibility provided by the LayerDepCases functor allows the structural case for products from the WithHash functor to use the TypeInfo generic:

```
fun op ⊗ (aT, bT) (a & b) {totWidth, maxDepth} = let
   val aN = Arg.numElems aT
   val bN = Arg.numElems bT
   val aW = Int.quot (op* (totWidth, aN), aN + bN)
   val bW = totWidth − aW
in
   getP bT b {totWidth = bW, maxDepth = maxDepth} × 0w13
+ getP aT a {totWidth = aW, maxDepth = maxDepth}
end
```

The numElems function returns the number of elements involved in a product. This allows the hashing function to balance the computation between elements of the product so that a hash of a binary tree, for example, will not be restricted to a single branch of the tree.

Notice that the LayerDepCases functor takes a part of the result, specifically the resulting type representation, as an argument. This is necessary due to the lack of higher-order functors. This is also why the type representation is constructed separately using LayerRep.

The implementations of all the layering functors, LayerRep, LayerCases, and LayerDepCases, are straightforward. Each structural case is basically just a simple one line definition. We omit the details.

5. Applications

Perhaps the main motivation behind our generics library is that Standard ML supports neither ad-hoc polymorphism nor generics. This presents a challenge to programmers. At first glance there seems to be no practical way to implement conceptually simple utility operations like a function for converting a value of any type to a string or a function for computing a hash of a value of any given type. Instead, programmers often laboriously implement many such operations for each type separately. Furthermore, functions that need such utility operations need to be parameterized and explicitly passed all the required operations. This can be quite tedious. The hope is that our library of generics eliminates much of such grunt work. Instead of having to define $O(mn)$ functions for $O(m)$ types and $O(n)$ operations, one can do with just $O(m)$ type representation constructors and $O(n)$ generic functions for a total of $O(m + n)$ definitions.

5.1 QuickCheck

Perhaps the most interesting application of our approach is a port of the Haskell QuickCheck library [6] to Standard ML. Our unit testing library provides a comprehensive set of functions for both traditional "X-Unit" style assertion based testing and QuickCheck style randomized testing of properties. Properties can be specified concisely, because a single type representation provides multiple generic functions at once. For example, here is how a property of the list reversal function described in [6] can be specified using our library:

```
all (list int) (fn xs ⇒ that (rev (rev xs) = xs))
```

The type representation, list int, provides the necessary random value generator and pretty printing function for the all combinator.

The unit testing framework is implemented as a functor that is instantiated with implementations of the required generics:

```
signature MK_UNIT_TEST_DOM = sig
  include GENERIC
  include ARBITRARY sharing Open.Rep = Arbitrary
  include EQ sharing Open.Rep = Eq
  include PRETTY sharing Open.Rep = Pretty
end
functor MkUnitTest (Arg : MK_UNIT_TEST_DOM) :>
  UNIT_TEST
    where type (α, χ) Rep.t = (α, χ) Arg.Open.Rep.t
    where type (α, χ) Rep.s = (α, χ) Arg.Open.Rep.s
    where type (α, κ, χ) Rep.p = (α, κ, χ) Arg.Open.Rep.p
```

QuickCheck style testing is mainly made possible by the two generics Pretty and Arbitrary. The Pretty generic

```
val pretty : Int.t Option.t → (α, χ) Pretty.t → α → String.t
```

implements a generic pretty printing function and allows any value of any representable type to be converted to a human-readable string. This makes it easy for the unit-testing framework to print out useful dignostics. The Arbitrary generic

```
val arbitrary : (α, χ) Arbitrary.t → α RandomGen.t
```

provides random generation of values of representable types.

As explained by Claessen and Hughes [6], custom datatypes often need custom generators. The Arbitrary generic allows the default generator stored in a type representation to be replaced by a user-defined ad-hoc generator for a particular type through the withGen function:[7]

```
val withGen : α RandomGen.t → (α, χ) Arbitrary.t UnOp.t
```

As an example of using withGen, one can implement a type representation constructor ordList

```
val ordList : α Rep.t → α List.t Rep.t = fn a ⇒ let
  val l = list a
in
  withGen (RandomGen.Monad.map
           (List.sort (ord a)) (arbitrary l)) l
end
```

for generating arbitrary ordered lists. The ordList type representation constructor first creates the default type representation l = list a. Then the default *unordered* list generator arbitrary l is extracted from the type representation and functionally updated to sort the list. Finally, the type representation l is functionally updated to use the new generator and returned.

The linear ordering for sorting generated lists in ordList is obtained using the Ord generic:

```
val ord : (α, χ) Ord.t → α Cmp.t
```

The Ord generic isn't required by the unit-testing library, but making it (and other generics) available through the same type representation can be quite useful.

An interesting algorithmic aspect of the Arbitrary implementation is that it is able to generate finite values of any representable type, including values of recursive datatypes. This is made possible by the use of the TypeInfo generic. Specifically, the structural case for sums, shown in Figure 11, uses the hasBaseCase function, which returns true only if a sum branch leads to a non-recursive variant, to pick a route that leads to a base case terminating the recursion.

[7] In fact, several generics provide a similar specialization function.

```
fun op ⊕ (aS, bS) = let
  val aGen = map INL (genS aS)
  val bGen = map INR (genS bS)
  val gen = G.frequency [(Arg.numAlts aS, aGen),
                         (Arg.numAlts bS, bGen)]
  val gen0 =
     case Arg.hasBaseCase aS & Arg.hasBaseCase bS of
        true & false ⇒ aGen
      | false & true ⇒ bGen
      | _            ⇒ gen
in
  IN {gen = G.sized (fn 0 ⇒ gen0 | _ ⇒ gen),
      cog = fn INL a ⇒ G.variant 0 ∘ cogS aS a
             | INR b ⇒ G.variant 1 ∘ cogS bS b}
end
```

Figure 11. The Sum Case of the Arbitrary Generic

5.2 Pickling

Implementation of generic pickling and unpickling functions using our approach is underway.[8] Our approach has some advantages compared to previously published Standard ML combinator libraries for pickling.

Unlike in the approach of Elsman [8], we don't have to manually tuple the pickling and unpickling functions with equality comparison and hashing functions. Indeed, the generic equality comparison and hashing functions are already available independently of pickling.

Another interesting difference is that the refCyc combinator for producing picklers for potentially cyclic references described by Elsman [8] takes a dummy value as an argument:

```
val refCyc : α → α pu → α ref pu
```

The dummy value is required for the purpose of "tying the knot" when unpickling cyclic data structures. The refc combinator specified in our signatures does not specify such an argument, because the required dummy value can be built implicitly by the Some generic

```
val some : (α, χ) Some.t → α
```

provided by our library. It can be used to generate a finite dummy value of any representable type.

6. Weaknesses

A fundamental weakness of our approach would seem to be the use of the sum-of-products representations and isomorphisms. Such an encoding tends to weaken the nominal and generative aspects of types. Distinguishing between two types, whose structural encodings are similar or the same, becomes more difficult.

The SML core language lacks first-class polymorphism and existential types. A universal type can sometimes provide a workaround. For example, to implement the Arbitrary generic, based on the `Arbitrary` type class

```
class Arbitrary a where
  arbitrary   :: Gen a
  coarbitrary :: a -> Gen b -> Gen b
```

from the Haskell QuickCheck library, would naturally require first-class polymorphism. Our type representation uses a universal type:

```
datatype α t =
  T of {gen: α Gen.t, cog: α → Univ.t Gen.t UnOp.t}
```

[8] They are likely to be available before publication.

A particular source of difficulties is mutable types. As shown by Elsman [8], representations of mutable types need to wrap mutable values in a universal type. This means that the creation of a type representation of a mutable type includes a side-effect—either the allocation of a new exception constructor or the allocation of a new ref cell! In order to detect cycles, the representations of mutable types must not be instantiated more than once.

7. Related Work

The Haskell community has developed several approaches to generic programming and Hinze et al. compare approaches to generic programming in Haskell [11]. Our approach to closed generics is similar to the "GM", or Generics for the Masses, approach [10]. The main advantage of the approach is that it is particularly lightweight and works in Haskell'98. Many of the other approaches require, or make use of, extensions beyond Haskell'98 (and SML'97), which makes the translation to ML more difficult.

It seems that Haskell's type classes can make the use of generics significantly more convenient by automatically constructing and passing around the necessary type representations or dictionaries. Recent work towards adding type classes to ML [7] would allow similar convenience. In fact, it would largely eliminate the need for combining value-dependent implementations of generics. Instead of combining generics, the type class system infers the necessary combination, expressed in the class constraints, and effectively passes the required type representations as separate arguments to functions using generics.

Several Haskell implementations provide higher-rank polymorphism. This turns out to be quite useful in generic programming as shown in several papers [5, 13]. It seems that hidden uses of existential types are relatively easy to simulate using a universal type. We have successfully implemented a toy generics library corresponding to the "LIGD" approach described in [5]. When higher-rank polymorphism appears in an interface, significant compromises must be made. While we have been successful in simulating some of the operational details of the "SYB" approach [13] in SML'97, the external interface of our implementation is not as type safe as the original Haskell version. Our toy implementations of these techniques can also be found from the MLton repository.

An important issue raised by Oliveira et al. in [15] is the need to allow customization of generics. Our approach allows ad-hoc customization, because type representations are passed explicitly. This makes it possible to functionally update a type representation and supply the desired specialization of the generic value, because the value is carried by the type representation. In contrast, it seems that any value-independent encoding of generics disallows customization. This is true of the value-independent approaches described by Yang [19] and of the LIGD approach [5], which has similar properties.

8. Conclusion

We have presented an approach to generics whose type representation can be extended. Our approach has the drawback that generics need to be combined explicitly, but this can be done in a straightforward manner. Most importantly, our approach allows generics to be developed both independently and incrementally, and to be later combined for convenient use.

Acknowledgments

The author wishes to thank Stephen Weeks for valuable feedback on early versions of the techniques presented in this article. The author is grateful to Jeremy Yallop and the anonymous referees whose comments helped to improve the presentation.

References

[1] Vincent Balat. Ocsigen: Typing Web Interaction with Objective Caml. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 84–94, New York, NY, USA, 2006. ACM Press.

[2] Nick Benton. Embedded Interpreters. *Journal of Functional Programming*, 15(4):503–542, 2005.

[3] Bernard Berthomieu. OO Programming styles in ML. Technical Report 2000111, LAAS, March 2000.

[4] Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. MLton, a whole program optimizing compiler for Standard ML. WWW: `http://www.mlton.org/`.

[5] James Cheney and Ralf Hinze. A Lightweight Implementation of Generics and Dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104, New York, NY, USA, 2002. ACM Press.

[6] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[7] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–70, New York, NY, USA, 2007. ACM Press.

[8] Martin Elsman. Type-specialized serialization with sharing. In *Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[9] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2004.

[10] Ralf Hinze. Generics for the Masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 236–243, New York, NY, USA, 2004. ACM Press.

[11] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing Approaches to Generic Programming in Haskell. In *Spring School on Datatype-Generic Programming*, 2006.

[12] Andrew Kennedy. Pickler Combinators. *Journal of Functional Programming*, 14(6):727–739, 2004.

[13] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003.

[14] Bruno C.d.S. Oliveira and Jeremy Gibbons. TypeCase: A Design Pattern for Type-Indexed Functions. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 98–109, New York, NY, USA, 2005. ACM Press.

[15] Bruno C.d.S. Oliveira, Ralf Hinze, and Andres Löh. Generics as a Library. In Henrik Nilsson and Marko van Eekelen, editors, *Seventh Symposium on Trends in Functional Programming 2006*, 2006.

[16] Norman Ramsey. Embedding an Interpreted Language Using Higher-Order Functions and Types. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 6–14, New York, NY, USA, 2003. ACM Press.

[17] Norman Ramsey, Kathleen Fisher, and Paul Govereau. An Expressive Language of Signatures. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 27–40, New York, NY, USA, 2005. ACM Press.

[18] Don Syme. Initializing Mutually Referential Abstract Objects: The Value Recursion Challenge. *Electr. Notes Theor. Comput. Sci.*, 148(2):3–25, 2006.

[19] Zhe Yang. Encoding Types in ML–like Languages. In *International Conference on Functional Programming (ICFP'98)*, September 1998.