# Scrap Your Boilerplate using Dynamic Types in Ordinary SML

Melissa E. O'Neill
Harvey Mudd College, California
oneill@acm.org

## Abstract

This paper introduces a Standard ML realization of the scrap-your-boilerplate generic-programming mechanism (first introduced by Simon Peyton Jones and Ralf Lämmel), which gives programmers tools to make writing traversal functions over complex, mutually recursive, data types considerably easier than with traditional hand-written traversal code.

The scrap-your-boilerplate approach is well-known in the Haskell community, but relies on Haskell's class system and also requires advanced features not present in standard Haskell (rank-2 polymorphism and a type-safe cast operator). None of these features are available in Standard ML.

This paper shows how to provide suitable Standard ML substitutes for this "missing functionality", and builds on those techniques to provide generic traversal-code generation in ML in a way that comes as close as possible to the original concepts as they exist in Haskell. Unlike its Haskell cousin, the ML version requires no language extensions. It also differs in its use of function composition to build efficient functions that need no dynamic type information as they run.

## 1. Introduction

Functional programming languages such as Standard ML provide powerful type-safe mechanisms to manipulate structured data. These mechanisms—algebraic datatypes and pattern matching under a static type system with type inference—are often essential to writing programs that are short and obviously correct. But sometimes these advantages become burdens. Let us examine two (related) annoyances.

A first source of frustration is the tedium, duplication, and brittleness of writing *boilerplate* code. For a classic example, consider the task of adding an underscore to all variables in a data structure representing an abstract syntax tree. The task is easy in principle, but it is made tiresome by all the traversal code we must write to delve into our structure. If we only had to write traversal code once, the task would not be so onerous, but in practice each distinct operation on the data structure frequently requires its own custom traversal code, necessitating a copy/paste/modify strategy where we grab the most similar preexisting traversal code (the "boilerplate") and modify it as necessary. All of the copied boilerplate is brittle because small changes to our data structures require all instances of this boilerplate code to be updated.



Standard ML's type system provides a second source of occasional discontent. Whereas it is frequently amply expressive, there may be occasions when we yearn for either the flexibility of dynamically typed languages or the sophistication of languages with additional features in their type system, such as classes supporting dynamic dispatch or advanced features such as rank-2 polymorphism.

Interestingly, at least some of these burdens are unique to ML-like languages. For Haskell programmers (at least those using GHC [8] or Hugs), the boilerplate issue has already been addressed by Lämmel & Peyton Jones [5]. And, of course, languages with different type systems from ML obviously face different frustrations on that front.

In this paper, I present techniques that can address these problems for ML, allowing new kinds of generic programs to be written in Standard ML. Specifically, I show how

- Dynamic types can be provided in ML without using the exception type (Section 4.2);
- Facilities exactly analogous to those presented by Lämmel and Peyton-Jones in *Scrap Your Boilerplate* [5] can be provided in ML without requiring any language extensions (Sections 5, 6, and 7);
- Specialized traversal code can be generated at runtime via function composition (Section 6), resulting in code that executes with efficiency close to that of handwritten code (Section 8).

Unusual features of my approach include its specificity to languages like ML—the techniques we will examine require language features unavailable in Haskell. The approach also requires almost no additional runtime storage overheads. Data structures continue to be stored without any type tagging (i.e., they contain no type information).

Many of the building blocks for the solution I present in this paper are well understood. My contribution is in combining those building blocks to create something that has previously not been achieved in Standard ML.

Working source code for the technique is provided at `http://www.cs.hmc.edu/~oneill/syb/`.

## 2. The Problem

Consider the task of updating the salaries in a data structure representing the organization of a company.[1] Expressed as a group of ML datatype declarations, we can describe our example company as follows:

```
datatype company = C of dept list
and      dept    = D of name * manager * subUnit list
and      subUnit = PU of employee | DU of dept
```

[1] Unsurprisingly, this example is taken directly from Lämmel & Peyton Jones [5]. See Section 10 for related work.

```
and         employee = E   of person * salary
and         person   = P   of name * address
withtype    salary   = real
and         manager  = employee
and         name     = string
and         address  = string
```

Now let us consider the task of writing a function to increase the salaries of all employees. The traditional solution would be to write code similar to that shown below:

```
fun increase k (C depts) = C (map (incD k) depts)

and incD k (D (name, manager, subunits)) =
    D (name, incE k manager, map (incU k) subunits)

and incU k (PU employee)     = PU (incE k employee)
  | incU k (DU dept)         = DU (incD k dept)

and incE k (E (name,salary)) = E (name, incS k salary)

and incS k salary            = salary * (1.0+k)
```

The form of the code mirrors that of the data structure. We traverse down to the places where the salary data is and update the salaries. The only *interesting* part of the code is incS, which actually does the increment.

It may seem as if writing this code was mindless but not especially onerous, but there are two factors to consider. First, because the structure of the code mirrors the structure of the data, the more complex our structured data, the more code we must write. Second, other tasks, such as giving a raise only to managers, finding the names of all the employees, calculating the salary bill, and so forth, each requires its own block of similar but subtly different boilerplate. As I claimed in the introduction, this code is tediously duplicative and brittle.

It would be much nicer to be able to write something much shorter. If we think about our task, what we want to create is a "salary transformer" that updates the salary part of the data structure, and then apply that transformer to the company. In Haskell, using the *Scrap Your Boilerplate* [5] approach, we can write[2]

```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))
```

where mkT means "make a generic transformation", and has the Haskell type

```
mkT :: TypeableData α =>
            (α -> α) -> (∀β. TypeableData β => β -> β)
```

and everywhere means "apply a generic transformation everywhere", and has the Haskell type

```
everywhere :: TypeableData α =>
            (∀β. TypeableData β => β -> β) -> α -> α
```

This code appears to require type classes, and some extensions beyond standard Haskell [9], including rank-2 polymorphism (although only everywhere actually requires it). Our goal is to write the same kind of code in vanilla Standard ML. This paper presents mechanisms that will allow us to write the following:

```
fun increase k =
    everywhere companyTy (mkT salaryTy (incS k))
```

We can read this code in English as "everywhere in values of type *company*, apply the *salary* transformer incS k". Mirroring the Haskell code, increase has type *real* -> *company* -> *company*. A secondary goal is for the *company* -> *company* function returned by increase k to rival the handwritten code for its execution efficiency.

## 3. Haskell-Style Classes in Standard ML

Standard ML lacks Haskell's class system, but we can often achieve the same ends by adding an additional argument corresponding to each class parameter of the function. For example, where in Haskell we might define the sort function with type Ord α => [α] -> [α], in SML we could achieve the same ends by defining sort to have the type α *ordOps* -> α *list* -> α *list*, where α *ordOps* is defined as[3]

```
abstype α ordOps = Ord of {
    cmp : α * α -> order
} with
    val mkOrdOps = Ord
    fun ordCmp (Ord {cmp, ...}) = cmp
end
```

and define *ordOps* for any type of our choosing; for example:

```
val intOrd      = mkOrdOps { cmp = Int.compare }
val stringOrd   = mkOrdOps { cmp = String.compare }
val realOrd     = mkOrdOps { cmp = Real.compare }
val intlistOrd  = mkOrdOps { cmp = List.collate Int.compare }
```

One difference between ordinary arguments and Haskell's implicit "class arguments" is that in Haskell there can be only one definition of the "class argument" for a given type. It is this very property that allows Haskell to implicitly determine these arguments, but the one-to-one correspondence can only be applied as a programming convention in SML. Arguably, SML is providing additional flexibility. For example, sorting integers according to their absolute value requires a little finesse given the fixedness of Ord Int in Haskell. In the above scheme, however, we could just as easily define a different ordering for integers and choose which to use. For the most part, however, we will eschew this flexibility and follow the convention of one tag value per type.

For implementing arbitrary classes, the above scheme does have some limitations; consider the following Haskell class:

```
class Pairable α where
    pairWith :: α -> β -> (α, β)
```

In this code, α is a free type variable, but there is no way to create an ML datatype with free type variable. We cannot write

```
abstype α pairOps = Pairable of {
    pairWith : α -> β -> α * β
} with ...
```

because we are not allowed to have a free type variable (in this case β in an ML type declaration). We will return to this problem in Section 7, but for now we can ignore such complications.

At this point, it should now be clear what companyTy and salaryTy actually were in our example at the end of the previous section—they were extra arguments to provide class-like functionality.

## 4. Two Methods for Dynamic Types in ML

To see how we may implement mkT, everywhere, and friends, we must first lay some groundwork in the form of a framework for dynamic types. In essence, the dynamic type problem is the creation of a type that can hold values of any type whatsoever. Unlike parametric polymorphic types, which represent a specific but currently

[2] I have slightly simplified the Lämmel & Peyton Jones [5] approach by using a single Haskell type class, *TypeableData* instead of two related classes *Typeable* and *Data* (originally called *Term*, but since renamed).

[3] I have used **abstype** throughout this paper, even though many ML programmers consider it somewhat archaic because it is a more concise way to show abstractions than using modules and signatures. It also has the side benefit of allowing the code to run on implementations of Standard ML that do not support the modules system.

unknown type, such a type can represent any value of any type at any time. We will call our dynamic type *any*.

For every type $t$, we will create two functions $t$2any : $t \rightarrow$ *any* and any2$t$ : *any* $\rightarrow$ $t$ *option*. For example, we could create an *any* list using

```
val l : any list = [int2any 17, int2any 42,
                    person2any (P ("Ralf", "Amsterdam"))]
```

and, after doing so, we could run `map any2int l` to get `[SOME 17,SOME 42,NONE] : int list`.

An important aspect of this scheme is that we can dynamically extend the *any* type to represent any new type of our choosing. Typically, we will extend the *any* type at the point in our code where we declare a new type. In the next two subsections, we will examine two alternative methods to create a function with the type signature

```
mkAnyFuncs : unit -> ('a -> any) * (any -> 'a option)
```

such that after defining a new type, such as *company*, we may write:

```
val (company2any : company -> any,
     any2company : any -> company option) = mkAnyFuncs ()
```

### 4.1 Using the Exception Type

One way to write mkAnyFuncs is as follows:

```
abstype any = V of exn
with
    fun mkAnyFuncs () =
        let exception Tag of 'a
            fun mkV v             = V (Tag v)
            fun getV (V (Tag v)) = SOME v
              | getV _            = NONE
        in  (mkV, getV)
        end
end
```

This implementation exploits Standard ML's exception type, and its dynamic extensibility properties. Standard ML allows a function to declare a unique exception at runtime, extending the *exn* type. Thus, each pair of converters created with mkAnyFuncs will use a unique Tag constructor, which provides a tag that is associated with the value in mkV, and removed by getV.

### 4.2 Using Entangled Functions

Interestingly, although Standard ML's exception type makes writing mkAnyFuncs *easy*, it is not actually necessary. We can achieve the same results as follows:

```
abstype any = V of {
    disclose:    unit -> unit,
    undisclose:  unit -> unit
} with
    fun mkAnyFuncs () =
        let val box = ref NONE
            fun discloser v () = box := SOME v
            fun undiscloser () = box := NONE
            fun mkV v =
                V { disclose   = discloser v,
                    undisclose = undiscloser }
            fun useV (V {disclose, undisclose}) =
              ( box := NONE;
                disclose ();
                let val v = !box in undisclose(); v end )
        in  (mkV, useV)
        end
end
```

Here, the trick is that a given mkV/getV pair share a piece of common knowledge, namely a box where they may place and find values of their particular type. This box is invisible to the outside world. For example, if we use mkAnyFuncs () to create a function pair int2any/any2int, they will share a secret box that can hold an *int*. Each *any* value conceals a value, which it can disclose by placing it into its associated box, thus an *any* made with int2any conceals an *int* that the *any* can reveal by placing it into the box for *int*s. When, say, any2int has an *any* value that was made by int2any, it can run disclose() and fetch the *int* from the box. If, however, the *any* actually represents an *employee*, running its disclose function will cause that employee to briefly appear in the *employee*s box, and any2int will find its *int* box empty.

### 4.3 Wrapping It Up in a Class

Instead of presenting users with a plethora of $t$2any/any2$t$ functions, we can wrap these functions up into a class, using the scheme outlined in Section 3. We'll provide functions with the following types:

```
mkAnyOps : unit -> 'a anyOps

mkV      : 'a anyOps -> 'a -> any
getV     : 'a anyOps -> any -> 'a option
useV     : 'a anyOps -> any -> 'a
mapV     : 'a anyOps -> ('a -> 'a) -> any-> any
```

In this arrangement, mkAnyOps creates a new instance of the "class" for a given type.[4] Class instances support the operations mkV and getV, as well as two new convenience functions useV and mapV. The useV operation is useful if we are certain that an *any* value must hold a value from a given class instance. It uses Standard ML's valOf function to turn an optional value into an actual value (if getV returns NONE, valOf will throw an exception). The mapV function is a classic map operation on the *any* type. The code is shown below:

```
abstype any = ...whatever...
and 'a anyOps = AnyOps of {
    mkV  : 'a -> any,
    getV : any -> 'a option
} with
    fun mkAnyOps () =
        let
            fun mkV v   = ...whatever...
            fun getV any = ...whatever...
        in  AnyOps {mkV = mkV, getV = getV}
        end
    fun mkV (AnyOps {mkV,...}) = mkV
    fun getV (AnyOps {getV,...}) = getV
end

fun useV anyOps = valOf o getV anyOps

fun mapV anyOps f any =
    case getV anyOps any of
        SOME v => mkV anyOps (f v)
      | NONE   => any
```

## 5. Implementing Generic Transformers

The Haskell code in Section 2 gave us a taste of mkT, a function to make a *generic transformer*. Essentially, mkT takes a function, $f$, that works on some specific type, such as *int*s or *employee*s and *extends* it to operate on *all* types. When evaluating (mkT $f$) $x$, if

[4] Value restriction in Standard ML requires that top-level values created by functions cannot have generalized type parameters (i.e., $\alpha$, $\beta$, etc.), thus a common usage pattern will be to add a type constraint when calling mkAnyOps.

$x$ matches the argument and return type of $f$, the value of $fx$ is returned, otherwise $x$ is returned.

The Haskell type for generic transformations is $\forall\beta$. TypeableData $\beta$ => $\beta$ -> $\beta$. The type $\beta$ must be an instance of *TypeableData* because we need to be able to determine whether we can apply our function or not.

It may seem that we cannot represent such a concept in ML, but the dynamic types outlined in Section 4 allow us to provide an analogous concept. Specifically, we will develop a *transform* type to represent exactly the values represented by the Haskell type given above. In the same way that the *any* type represents values of all types, the transform type will all transformation of all types.

We will develop code that supports at least the following operations:

```
mkTransformOps : unit -> 'a transformOps

idT            : transform
composeT       : transform * transform -> transform
mkT            : 'a transformOps -> ('a -> 'a) -> transform
useT           : 'a transformOps -> transform -> 'a -> 'a
```

### 5.1 A Simple But Inefficient Approach

One option for representing a transformation is as a function of type *any* -> *any* that meets the requirement that the underlying type remains the same for both argument and result.

Using our code from Section 4, we can create a pair of functions for creating and using generic *transform*s:

```
abstype    transform          = T of any -> any
and        'a transformOps    = TransOps of {
    mkT  : ('a -> 'a) -> transform,
    useT : transform -> ('a -> 'a)
} with
    val idT                        = T (fn x => x)
    fun composeT (T t1, T t2)      = T (t1 o t2)

    fun mkT (TransOps {mkT,...})   = mkT
    fun useT (TransOps {useT,...}) = useT

    fun mkTransformOps ()=
        let val anyOps       = mkAnyOps ()
            fun mkT f        = T (mapV anyOps f)
            fun useT (T f) x =
                useV anyOps (f (mkV anyOps x))
        in  TransOps { mkT = mkT, useT = useT }
        end
end
```

Consider the following example, in which we create *transformOps* for *int*s, *real*s, and *string*s, and then use them to create a generalized *double* function:

```
val intTransOps    : int transformOps    = mkTransformOps ()
val realTransOps   : real transformOps   = mkTransformOps ()
val stringTransOps : string transformOps = mkTransformOps ()

val double : transform =
    composeT (mkT intTransOps (fn x => x+x),
        composeT (mkT realTransOps (fn x => x+x),
            mkT stringTransOps (fn x=> x^x)))
```

Our example transform, double, can be applied to an *int* value using useT intTransOps double, and similarly applied to *real*s and *string*s. For other types, it will act as the identity function.

One drawback of this approach is that when we run useT intTransOps double 17, first, 17 is converted to an *any* value, then it is passed through the doubling function for strings, which returns it unchanged; then through the doubling function for *real*s; and then, through the doubling function for ints, which decodes the *any* into an int, doubles it, and then hides it again as an *any*; which is then finally decoded by useT.

In the next section, we will examine an alternate implementation of mkT and friends that avoids these deficiencies and adds additional functionality.

### 5.2 A More Powerful & Efficient Approach

Observe that for a *transform* such as double, when we apply useT intTransOps double 17, we don't care about its operations on other types, only its operation on *int*s. Thus another representation for *transform* is a list of functions, where the functions for different types occupy separate entries in the list. In this representation, useT just needs to pull the right function from the list. But how can we store this heterogeneous list of transformation functions that may operate on different types? With the *any* type, of course!

The code below shows how we may implement the generic transformations using the "list of functions" model:

```
abstype    transform          = T of any list
and        'a transformOps    = TransOps of {
    mkT    : ('a -> 'a) -> transform,
    getT   : transform -> ('a -> 'a) option
} with
    val idT                        = T []
    fun composeT (T t1, T t2)      = T (t1 @ t2)

    fun mkT (TransOps {mkT, ...})  = mkT
    fun getT (TransOps {getT, ...}) = getT

    fun mkTransformOps ()=
        let val anyOps = mkAnyOps ()
            fun mkT f  = T [mkV anyOps f]
            fun getT (T ts) =
                let val relevant = List.mapPartial (getV anyOps) ts
                    fun compose (f, NONE)   = SOME f
                      | compose (f, SOME g) = SOME (f o g)
                in  foldr compose NONE relevant
                end
        in  TransOps {mkT= mkT, getT= getT}
        end
end

fun useT tOps trans =
    case getT tOps trans of
        NONE   => (fn x=> x)
      | SOME f => f
```

The identity transform, idT, is represented by an empty list of functions to apply, and the transform composition, composeT, is simply list concatenation—this design allows us to compose transforms without any knowledge of the types on which they will ultimately operate, but it does mean that the list may contain multiple transformations that act on the same type (getT will compose those functions to return a single function).

Instances of the transform operations are created by mkTransformOps. It creates an instance of the *any* class for the particular type of transformation these operations will store and retrieve. The mkT function is simple, just calling mkV to store a single function, but getT is complicated by the need to compose functions. Finally, in the same way that we implemented useV in terms of getV, here we implement useT in terms of getT.

In this revised scheme, when we evaluate useT intTransOps double we are given back *exactly the same function* that was passed to mkT intTransOps—the returned function does not roundtrip anything through the *any* type.

It is also possible to extend this scheme in some other useful ways. For example, we can provide a function mapT that allows us to "edit" all transformations of a given type in the list. To do so,

mkTransformOps merely needs to be extended to create the following function:

```
fun mapT f (T ts) = T (map (mapV anyOps f) ts)
```

It is similarly straightforward to add a delT function to strip out transformations of a given type, and only a small amount of additional work to allow transformations to be composed with functions that are executed only for their side effect.

The *transform* type provides a powerful abstraction, but useT is a far cry from the everywhere function we mentioned in Section 2. In particular, if we create a salary transformer with mkT salaryTransOps (incS 0.25), we can apply it to a company with useT, but since the transform is for a different type, it will merely act as the identity function. But, as we see in the next section, the components we have built so far are stepping stones to everywhere.

## 6. Generic Map

Our goal in this section is to create "generic map" functionality that will allow us to create traversal functions that descend into data structures and perform some degree of rewriting on that data structure. The everywhere function mentioned in Section 2 is one example of the kind of function we would like to be able to write using a generic-map facility.

To understand how we may programatically create such functions, we shall reexamine how we code these kinds of functions by hand. Let us return to the salary-increment problem we first mentioned in Section 2. Consider the following alternate implementation of increase:

```
fun increase k =
    let fun incC (C depts) = C (map incD depts)

        and incD (D (name, manager, subunits)) =
            D (name, incE manager, map incU subunits)

        and incU (PU employee)      = PU (incE employee)
          | incU (DU dept)          = DU (incD dept)

        and incE (E (name,salary))  = E (name, incS salary)

        and incS salary             = salary * (1.0+k)
    in incC
    end
```

The functions inside the let expression have the following types:

```
incC    : company  -> company
incD    : dept     -> dept
incU    : subUnit  -> subUnit
incE    : manager  -> manager
incS    : salary   -> salary
```

Each of these functions is thus a transformation on a particular type, and could therefore be represented as a *transform*. But more importantly, because each function necessarily supports a different type, they can *all* be stored in a single *transform*. I shall call a single *transform* that holds a family of functions that together provide traversal over a data structure a *library*. Our goal is to programmatically create such a library.

Our second observation is that it is not merely sufficient to traverse the data structure, we would like to actually *do something* with each value encountered in the traversal, and we would also like to have some control over the traversal process. We can achieve both goals by providing a *modifier* that can modify each of our traversal functions before it is placed in the library. But how can we provide a single modifier that can augment all of our traversal functions, given that each of them has a different type? Again, the *transform* type can rescue us—the modifier can have type *transform -> transform*.

Thus, when creating the necessary functions to implement a map-like operation over a data structure, we need to maintain some state consisting of a library and a traversal modifier. The following functions will provide the facilities we need:

```
initML         : (transform -> transform) -> maplib
getMapML       : maplib -> 'a transformOps -> ('a-> 'a) option
setMapML       : maplib -> 'a transformOps -> ('a-> 'a) -> unit
getModifierML  : maplib -> transform -> transform
```

Here initML is passed a modifier function and creates an initial state for the code that will build our map-function library, this state consists of an empty library and the modifier function; getMapML gets a mapping function from the library if it has one; setMapML sets the mapping function for a given type; and getModifierML retrieves the modifier function. These functions can be coded as follows:

```
abstype maplib = ML of {
    modifier     : transform -> transform,
    library      : transform ref
} with
    fun initML f = ML{ modifier = f, library  = ref idT }

    fun getMapML (ML {library,...}) tOps =
        getT tOps (!library)

    fun setMapML (ML {library,...}) tOps f =
        library := composeT(mkT tOps f, !library)

    fun getModifierML (ML {modifier,...})  = modifier
end
```

With these functions in place, we can now write useMapML, a function of type

```
maplib -> 'a transformOps -> (maplib -> 'a -> 'a) -> 'a -> 'a
```

In other words, a function that is passed a library and target type (i.e., a *transformOps* value), and a function for making the mapper if it does not yet exist, and returns the resulting mapping function. The code for useMapML is as follows:

```
fun useMapML ml tOps mapper =
    case getMapML ml tOps of
        SOME f => f
      | NONE   =>
            Y (fn self =>
                let val _ = setMapML ml tOps self
                    val trans = mkT tOps (mapper ml)
                    val trans' = (getModifierML ml) trans
                in  useT tOps trans'
                end)
```

The use of the Y combinator in the above code warrants a little explanation. Because datatypes may be (mutually) recursive, the map function we are creating may (either directly, or indirectly through the functions it calls) wish to refer to itself. But at the time we are creating the code for the map function, it does not yet exist. The Y combinator, a function with type

*(('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)*

solves this problem. Y is defined as

```
exception CalledBeforeDefined
fun Y f =
    let fun fakeself _   = raise CalledBeforeDefined
        val selfref      = ref fakeself
        val self         = fn x => (!selfref) x
        val trueself     = f self
    in  selfref := trueself;
        trueself
    end
```

We now understand how useMapML operates, given suitable arguments, but the key to useMapML is clearly the function that, given

a *maplib*, makes a map function for a given type. We will now see how such functions may be written.

### 6.1 Making Mappers

The role of a map-function creation function passed to useMapML is to create a function that will traverse the structure. Its only explicit action is performing the traversal itself—all other actions are added by useMapML, using the modifier function contained in the *maplib*.

The map-function creators for types that have no substructure, such as integers and strings, simply return the identity function.

To understand how we can provide such map-function creators for types that do have some substructure, we will consider the following simple datatype:

```
datatype   foo  = FooI of int | FooB of bar
and        bar  = Bar of string * foo
```

For this type, we write the following functions:

```
fun fooMkMap ml =
    let val intMap = useMapML ml intTransOps intMkMap
        val barMap = useMapML ml barTransOps barMkMap
    in  fn (FooI i) => FooI (intMap i)
         | (FooB b)=> FooB (barMap b)
    end

and barMkMap ml =
    let val stringMap = useMapML ml stringTransOps stringMkMap
        val fooMap = useMapML ml fooTransOps fooMkMap
    in  fn (Bar (s,f)) => Z (stringMap s, fooMap f)
    end
```

Each map-function generator looks up the map functions required to descend into the substructure, and then returns a function that calls those functions. Notice that because the data structure was mutually recursive, these functions are also mutually recursive, but that their mutual recursion is between the map-function creators—recursion for the map functions themselves is provided by the Y combinator in *useMapML*.

Notice that the pattern of these definitions is very regular and closely matches the original datatype specification. It is thus possible to mechanically generate this code.

### 6.2 Using Mappers

To usefully use this generic-map functionality, we need to provide an appropriate modifier to act on the bare traversal functions. Here are two examples that perform generic preorder and postorder maps by providing a suitable modifier function to initML:

```
fun topdown tOps mkMap trans =
    let val ml = initML (fn recurse => composeT (recurse, trans))
    in  useMapML ml tOps mkMap
    end

fun bottomup tOps mkMap trans =
    let val ml = initML (fn recurse => composeT (trans, recurse))
    in  useMapML ml tOps mkMap
    end
```

We can thus make a function that increments the integer parts of a *foo* object with the code

```
preorder fooTransOps fooMkMap (mkT intTransOps (fn x => x+1))
```

which will output a function identical to the function returned by evaluating the expression

```
Y (fn fooMap =>
    (fn (FooI i) => FooI ((fn x => x+2) i)
      | (FooB b) =>
            FooB ((fn (Bar (s,f)) =>
                    Bar ((fn x => x) s, fooMap f)) b)))
```

By using more complex arguments to initML, we may create more complex maps. For example, the modifier function is passed the function that will recursively descend the structure for each type, but we are not required to call it. Thus it is possible to create map functions that only examine and transform *parts* of a structure.

### 6.3 Improving the Interface

The only annoyance in this arrangement is that we must pass two arguments to useMapML to specify the type on which we wish to act. But this issue is cosmetic, and can be fixed by creating an all encompassing "class" value that gathers together all of the useful type-related functionality in a single place. Because this umbrella class contains all the information necessary to work with a particular type, I shall call the type used to represent instances of this class *typeinfo*. With this class in place, we can now provide users with functions that satisfy a more pleasant interface; namely,

```
gmap    : 'a typeinfo -> (transform -> transform) -> 'a -> 'a
everywhere    : 'a typeinfo -> transform -> 'a -> 'a
everywhere'   : 'a typeinfo -> transform -> 'a -> 'a
```

by defining the functions

```
fun gmap (AllOps{mapper,transOps,...}) modtrans =
    useMapML (initML modtrans) transOps mapper

fun everywhere allOps t0 =
    gmap allOps (fn recurse => composeT (trans, recurse))

fun everywhere' allOps t0 =
    gmap allOps (fn recurse => composeT (recurse, trans))
```

where AllOps is the constructor for the *typeinfo* type. The everywhere and everywhere' functions mirror the topdown and bottomup functions suggested in the previous section, but with a better interface and with names matching those of Lämmel & Peyton Jones [5].

At this point we have made good on the claims we made in Section 2—we have constructed a convenient and usable mechanism for creating traversal functions.

## 7. Generic Folding

In addition to the everywhere function, Lämmel & Peyton Jones [5] provide a function everything that is equivalent to a generic fold operation. Thus, to calculate the salary bill at a company, we would wish to write

```
val addSalaries = (fn total => fn x => total+x)
val salarybill =
     everything companyTy (mkQ salaryTy addSalaries) 0.0
```

In the same way that we built generic map functionality out of *transform*s, we can build generic fold functionality out of *query*s, where the *query* type supports the following operations (each an analogue of the similarly named function for transforms):

```
mkQueryOps : unit -> 'a queryOps

idQ        : 'a query
composeQ   : 'a query * 'a query -> 'a query

mkQ        : 'a queryOps -> ('b -> 'a -> 'b) -> 'b query
getQ       : 'a queryOps -> 'b query -> ('b -> 'a -> 'b) option
useQ       : 'a queryOps -> 'b query -> 'b -> 'a -> 'b
```

Unfortunately, this function specification poses some challenges for an ML implementation, because whereas *transform*s are parameterized over a single type, and that type appears as an argument to the *transformOps* type, *query*s are parameterized over two, only one of which appears in the *queryOps* type. As we alluded to in Section 3, we cannot have free type variables in ML datatypes.

Thus, we appear to reach an impasse. If we revised the *queryOps* type to have *two* type parameters, we would solve one problem only to have it replaced by another—instead of creating a *queryOps* value for every type, we would need to produce one for every possible *pair* of types, which is ludicrous. We could instead disguise one of our two types by using an *any* value, but that approach would add additional runtime overheads when our composed function executes. So instead we take a different tack—one that will make every pure-hearted functional programmer cringe.

### 7.1 Embracing Side Effects

In Standard ML, we do not actually need the everything function at all. We can actually write the salarybill function (and, similarly, every other fold-like function) using everywhere as follows:

```
fun salarybill company =
    let val total  = ref 0.0
        fun add x  = (total := !total + x; x)
    in  everywhere companyTy (mkT salaryTy add) company;
        !total
    end
```

This code performs an identity transform on the provided company (and discards the result), but as a side effect, totals the salaries.

One drawback of this approach is that we produce an unnecessary copy of the data structure, a constant-factor overhead, but one that can be avoided. We can do so by providing generic "app" functionality (named after Standard ML's app function of type *(*$\alpha$ *-> unit) -> *$\alpha$ *list -> unit*, which applies a function to every element of a list). Using the techniques we have previously outlined in Section 6, we can define a new type analogous to *transform* called *sink* (whereas *transform*s model functions of type $\alpha$ -> $\alpha$, *sink*s model functions of type $\alpha$ -> *unit*), and use it to provide a gapp facility analogous to gmap.

The other drawback is that writing the above code is ugly, yet it need not appear so to users. It is possible to provide everything, mkQ, composeQ, and so forth such that users can remain largely unaware of the use of the *sink* type and the side-effects taking place behind the scenes. In this case, ignorance is bliss.

### 7.2 Embracing Exceptions

Another use of the generic app functionality is to produce generic find functions. Lämmel & Peyton Jones [5] do not suggest a somewhere function, but these functions occur much as maps and folds occur. The trick in this case is to traverse the structure and, when we find what we are looking for, throw an exception containing our desired result. By throwing an exception, we abort any further traversal.

## 8. Performance & Optimizations

It would be unreasonable to expect the performance of our dynamically created functions to equal that of handwritten code. First, for handwritten code, a compiler has the option to analyze and optimize the traversal functions as a group, whereas functions created at runtime by function composition may not offer the same opportunities. Second, our code must make occasional use of references, both for recursive calls (due to the implementation requirements of the Y combinator), and for maintaining state in generic folds. But, if the performance cost is sufficiently low, we will have reason to use generic traversal facilities in all but the most performance-critical situations.

Table 1 shows the performance of three functions acting on our example *company* data structure, contrasting timings for handcoded functions with those for functions created using the ML rendition of the scrap-your-boilerplate approach:

| Function | Handcoded | Generic (1) | Generic (2) |
|---|---|---|---|
| increase | 3.63 | 4.86 | 3.73 |
| salarybill | 0.15 | 0.61 | 0.35 |
| payroll | 0.66 | 1.23 | 0.86 |

**Table 1.** Performance Times (in seconds)

- increase raises everyone's salaries by a given factor (and is thus is sometimes described as the "paradise" benchmark);
- salarybill calculates the total cost of those salaries (and thus which could reasonably be described by employers as the "misery" benchmark); and
- payroll produces a list of the *name*s and *salary*s of everyone in the company.

The functions (increase and salarybill) are taken from Lämmel & Peyton Jones [5].

To obtain measurable timings, I ran these functions on a company with approximately 1.4 million employees (and almost 900,000 departments and an organizational structure that runs ten departments deep).[5,6]

The first generic column in the table shows the functions coded in the obvious way, using everywhere and mkT for increase and everything and mkQ for salarybill and payroll. For increase, the abstraction overhead is low, taking only 33% longer to execute than the original code; for salarybill the slowdown is more pronounced, running at only a quarter the speed of the handwritten code (but no slouch in real terms); for payroll, a slightly more realistic fold operation, the code executes at about half the speed of the handwritten code.

One of the reasons for the slower execution is actually that the handwritten code descends less deeply into the structure. For example, the generic code descends into the *name* field, where it seeks to apply any necessary *name* transformers and *string* transformers; and the salary field, too, is considered not only for *salary* transformers but also for *real* transformers. In these cases, the identity function is applied, but in such trivial benchmarks, those costs add up. If we recode our generic programs in a slightly smarter way—to curtail deeper traversal—we can speed things up. For example, we can alternatively write increase as

```
fun increase k =
    let fun inc (E (n,s)) = E (n, s * (1.0 + k))
    in  gmap companyTy (mapT employeeTy (fn recurse => inc))
    end
```

and apply a similar strategy to write salarybill and payroll in terms of gfold and mapQ.

The second generic column in the table shows the performance of these "smarter" generic traversal functions. As we can see from the table, these changes help to narrow the gap considerably. For increase, our dynamically composed function runs less than 3% slower than handcoded original.

These benchmarks are, of course, trivial and synthetic, but whereas this property usually risks casting a technique in a falsely

[5] The data structure was produced programmatically rather than using a real company. The statistics about the data structure were produced by querying it. Writing code to perform those queries was made considerably easier by the scrap-your-boilerplate approach.

[6] The tests were run under Standard ML of New Jersey, version 110.62, on a 2 GHz Power Mac G5, with 5.5 GB of ram running Mac OS X 10.4.9 with energy saving disabled. Times are averages from seven runs, timed using SML's Timer.startCPUTimer/Timer.checkCPUTimes. Times are the sum of user and system times including both garbage-collector and mutator overheads.

positive light, in this case, the benchmarks, if anything, *overstate* the overheads of this approach. If, as in the case of `salarybill`, the action we are performing is trivial, then the overheads of the traversing machinery will dominate, but as the action becomes more computationally demanding, these overheads fade into the background noise. Similarly, our *company* example is not an especially complex data structure, but for more complex data structures, the attraction of the scrap-your-boilerplate approach only increases.

### 8.1 Future Optimizations

Given the insight that some traversals deep into a structure are useless do-nothing operations that adversely affect time and space performance, a possible future optimization to the Standard ML implementation would be to change the way map functions are created. Instead of providing a `useMapML` that always returns a function even if it does essentially nothing, we can envision a variation that keeps track of when such functions are essentially useless and doesn't apply them.

## 9. Conclusions

This paper has shown that scrap-your-boilerplate generic programming approach can be realized in Standard ML, without requiring any additional language features. Strangely, it is Standard ML's most impure features—in particular, references—that provide us with the tools necessary to provide this facility without tools such as rank-2 polymorphism, built-in dynamic types (or a type-safe cast operator), or a class system.

That is not to say that some of these features would not be welcome additions to the language. In particular, although tricks such as our *any* type allow us to sidestep issues issues such as rank-2 polymorphism, we cannot always to so easily or efficiently.

We have also discovered how flexibly we may compose functions at runtime in Standard ML. There may be other applications besides scrapping your boilerplate where the ability to work with a library of functions of heterogeneous types and compose them together in a type-safe way may be useful.

## 10. Related Work

The foundational paper on this topic, Lämmel & Peyton Jones [5], provides the single strongest inspiration for the present work. That paper targets Haskell, and the goal throughout this paper has been to show how their ideas may be realized in Standard ML—my goal has very clearly been "if they can do it there, we must be able to do it here". Since that original work, there have been several other papers on the topic, all making progressive refinements to the original idea, and almost all of which targeted Haskell.

Lämmel & Peyton Jones expanded their original ideas in two further papers that are usually also considered foundational. Their second paper [6] adds introspection facilities, as well as generic printing, serialization, and deserialization facilities, amongst others. I believe that these ideas provide an excellent foundation for future work. Their third paper [7] focuses on improvements to the techniques to allow it to interact better with Haskell's class system, as such it is probably less relevant to an ML audience.

Cheney [1] examines the traversals involved in capture-avoiding substitution and develops extensions to the basic scrap-your-boilerplate concepts to address this specific area. The features and limitations of Haskell are fairly fundamental in Cheney's treatment, but his core ideas are no doubt also applicable to Standard ML.

Beyond these works, there are several other papers that extend or explain the basic approach [2, 3, 4] but all are closely tied to the Haskell setting.

Ren and Erwig [10] create a more customizable collection of traversal combinators than the original approach by Lämmel & Peyton Jones [5]. They also provide an excellent summary of more distant work in the field of dynamically created traversal operators.

While all of this additional work has broadened and developed the scrap-your-boilerplate idea, Standard ML programmers have been left on the sidelines, without apparent access to the original ideas (which were useful at their outset). This paper remedies the situation, and will hopefully open the door for some of these enhancements to also be provided in Standard ML in some form or another.

I believe that the techniques described in Sections 3 and 4.1 have long been part of the programming folklore for Standard ML, but I have been unable to locate any formal coverage of them in the literature. The entangled functions approach described in Section 4.2 seems to be a new alternative.

## 11. Acknowledgments

My thanks to Chris Stone, who convinced me to write this paper. And thanks to everyone who read earlier drafts, in particular Claire Connelly, Zvi Effron, Ari Wilson and Phil Miller who read some of the rough drafts.

## References

[1] J. Cheney. Scrap your nameplate (functional pearl). In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 180–191. ACM, 2005.

[2] R. Hinze and A. Löh. "Scrap your boilerplate" revolutions. In T. Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 180–208. Springer, 2006.

[3] R. Hinze, A. Löh, and B. C. D. S. Oliveira. "Scrap your boilerplate" reloaded. In M. Hagiya and P. Wadler, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming, Fuji-Susono, Japan, April 24-26, 2006,*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer, Apr. 24–26 2006.

[4] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 137–142. ACM Press, Jan. 17–19 2007.

[5] R. Lämmel and S. L. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37. ACM Press, 2003. ACM SIGPLAN Notices.

[6] R. Lämmel and S. L. Peyton Jones. Scrap more boilerplate: Reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional programming*, pages 244–255. ACM Press, Apr. 1 2004.

[7] R. Lämmel and S. L. Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 204–215. ACM, Sept. 26–28 2005.

[8] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A technical overview. In *Proceedings of the UK Joint Framework for Information Technology Technical Conference*, pages 249–257, Keele, Mar. 1993.

[9] S. L. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, F. W. Burton, J. H. Fasel, K. Hammond, R. Hinze, P. R. Hudak, T. Johnsson, M. P. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. L. Wadler. Haskell 98: A non-strict, purely functional language. Technical report, Yale University, Feb. 1999.

[10] D. Ren and M. Erwig. A generic recursion toolbox for haskell, or: Scrap your boilerplate systematically. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 13–24, New York, NY, USA, 2006. ACM Press.