Staging Generic Programming

Jeremy Yallop
University of Cambridge, UK
jeremy.yallop@cl.cam.ac.uk

Abstract

Generic programming libraries such as *Scrap Your Boilerplate* eliminate the need to write repetitive code, but typically introduce significant performance overheads. This leaves programmers with the unfortunate choice of writing succinct but slow programs or writing tedious but efficient programs. We show how to systematically transform an implementation of the *Scrap Your Boilerplate* library in the multi-stage programming language MetaOCaml to eliminate the overhead, making it possible to combine the benefits of high-level abstract programming with the efficiency of low-level code.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming

Keywords multi-stage programming, generic programming, ML, MetaOCaml, partial evaluation

1. Introduction

Generic programming The promise of *generic programming* is the elimination of the tedious boilerplate code used to traverse complex data structures. For example, if you want to apply a function `munge` to all values of a particular type `t` within a larger data structure `s` you might start by writing code to traverse `s`, examining its constructors and iterating over their fields. Alternatively, you might use a generic programming library such as *Scrap Your Boilerplate* (Lämmel and Jones 2003), and write code like the following:

```
everywhere (mkT munge) s
```

This small piece of code locates all the values of type `t` within the structure `s` and transforms them with `munge`. Regardless of whether `s` is a list, a tree, a pair, or some more complex structure, `everywhere` traverses its sub-values, applying `mkT munge` to each. The application of `mkT munge` succeeds exactly when the argument has a type that matches the domain of `munge`. In this way, generic programming eliminates the need to write type-specific traversals.

Evidently, generic programming can significantly simplify certain programming tasks. However, the genericity of functions implemented using libraries such as Scrap Your Boilerplate (SYB) often comes with a severe performance cost. For example, the call to `everywhere` above may take around 15 to 20 times longer than an equivalent traversal specialized to a particular type (Section 4).

Multi-stage programming The poor performance of functions like `everywhere` is a consequence of the same genericity that makes them appealing. How might we keep the genericity but eliminate the cost? One approach to reducing the costs of abstraction is *multi-stage programming*. Multi-stage programs make use of information that becomes available between the time when a function is defined and the time when it is invoked to improve the function's efficiency. For example, the author of the `everywhere` function, who cannot possibly know the eventual types of its arguments, ought to make `everywhere` as general as possible. However, the caller of `everywhere` typically knows the types of the arguments long before the time when it is actually called. This type information can be used to instantiate `everywhere` at the call site, producing an implementation that is specialized to those argument types. In this way the overhead of genericity may be eliminated before the time comes to call `everywhere`, so that the eventual call is as efficient as possible.

Here is a call to a staged version of `everywhere` which is specialized for traversing lists of trees:

```
let transform : int tree list → int tree list =
 instantiateT (everywhere_ (mkT_ munge_))
```

The second line passes the result of the call to `everywhere_` to a function `instantiateT`, which builds a type-specialized version of the function, `transform`, without the overhead of genericity. (The convention throughout this paper is that a trailing underscore indicates a staged version of a function.) The generated function, `transform`, can be called immediately with the value `s`:

```
transform s
```

The type-specialized `transform` function behaves exactly as the original call to `everywhere`, but its performance is much closer to that of the tedious hand-written traversal which we have still managed to avoid writing. The behaviour of `instantiateT`, `everywhere_`, etc., is explained in detail in the body of this paper.

1.1 Contributions

The primary contribution of this paper is the demonstration that standard multi-stage programming techniques can eliminate the overhead of generic programming without the need to resort to extra-lingual mechanisms. In more detail, this paper includes

- a port of Scrap Your Boilerplate library to BER MetaOCaml extended with modular implicits, using extensible variant types to implement a safe type equality test (Section 2);
- a straightforward transformation of generic operations into generators of monomorphic code, eliminating the overhead of generic dispatch (Section 3.1);
- a refactoring of the SYB generic traversal schemes to use explicit fixpoints, allowing memoization of recursive calls and

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...$15.00
http://dx.doi.org/10.1145/2847538.2847546

`let` insertion, transforming open-recursive code into groups of mutually-recursive functions (Section 3.2); and

- performance measurements showing that staging eliminates the great majority of the overhead of generic traversals over hand-written code, and an analysis of the sources of the remaining overhead in the staged generic programming library (Section 4).
- Along the way are various incidental observations, including the first case study of a library written using modular implicits (White et al. 2015), an argument for adding recursive module types to OCaml (Section 2.3) and an argument for extending MetaOCaml with facilities for dynamically constructing mutually-recursive bindings.

2. Scrap Your Boilerplate, OCaml-Style

The majority of generic programming libraries, including SYB, are written for Haskell or closely-related languages. However, most research on multi-stage programming has focused on ML-family languages such as BER MetaOCaml (Kiselyov 2014). This paper uses a port of SYB to BER MetaOCaml as a starting point for demonstrating how to use staging to improve the efficiency of generic programming.

2.1 SYB Basics

The Scrap Your Boilerplate design has three principal ingredients.

The first ingredient is a type coercion that makes it possible to write functions such as the `mkT` function used in the introduction. In the original SYB implementation (Lämmel and Jones 2003) the coercion is implemented as a typed wrapper around an untyped cast. The implementation in this paper takes advantage of advances in language technology to implement the coercion more safely and efficiently as a typed equality test based around extensible GADTs (Section 2.2).

The second ingredient is a small collection of generic mapping functions over data. Each mapping function takes a generic function as argument and applies it to all the immediate subnodes of a value (Section 2.3).

The third ingredient is a collection of generic "schemes" — that is, functions which traverse data in a variety of ways. The schemes are built by recursively applying the generic mapping functions and combining them with other functions to transform, interrogate, and display data (Section 2.4).

2.2 The First SYB Ingredient: Type Equality

Type equality plays a central role in the Scrap Your Boilerplate library. The original Haskell library (Lämmel and Jones 2004) is based on a generalized cast operation with the following Haskell type:

```
gcast :: (Typeable a, Typeable b) => c a → Maybe (c b)
```

The type of `gcast` may be read as follows: for any types `a` and `b` that are instances of the `Typeable` class, `gcast` attempts to convert a value of type `a` in some context `c` (typically a `newtype`) into a value of type `b` in the same context. The `Maybe` type constructor indicates that the attempt may fail: in fact, it certainly ought to fail whenever the types `a` and `b` are not the same!

The `gcast` function is based on an unchecked coercion function, `unsafeCoerce`, which it calls after retrieving and comparing type representations of `a` and `b`, ensuring that `unsafeCoerce` is only actually invoked if the representations of `a` and `b` are the same. Convention (and, in later versions, compiler support) ensures that distinct types are given distinct representations.

Since SYB was first released, language technology has advanced, and it is now possible to implement equivalent behaviour without the need for either the low-level unsafe features or the awkwardness of type contexts encoded as `newtype`s. Recent versions of OCaml include both GADTs, which support a richer notion of type equality than `gcast`, and extensible variant types (Löh and Hinze 2006), which serve as a useful basis for building type representations.

```
type (_, _) eql = Refl : ('a, 'a) eql
```

Figure 1. Type equality

Figure 1 shows the classic equality GADT (Johann and Ghani 2008), which provides evidence that two types are equal. The `eql` type has two parameters, since it represents a relation between two types. However, the sole constructor `Refl` reveals that in the only permissible instantiation both parameters are the same, reflecting the fact that the only type equal to a type `t` is `t` itself. In code that uses pattern matching to scrutinise a value of type `(t, s) eql` to reveal a `Refl`, the compiler therefore allows the types `t` and `s` to be used interchangeably.

```
type _ type_rep = ..
```

Figure 2. An extensible type representation

Figure 2 introduces a type `type_rep`, with a single parameter and no constructors. A value of type `t type_rep` is intended to serve as a representation for the type `t`, and to support comparing two type representations to determine whether the corresponding types are equal. The dots `..` in the definition of `type_rep` indicate that it is an extensible type. Extensibility is a useful property for a type representation. Since the set of types is open — each new data type declaration generates a fresh type — it is convenient if the set of type representations is also open.

Figure 3 shows the `TYPEABLE` signature, which corresponds to a class of the same name in the original SYB implementation. A structure of type `TYPEABLE` contains a type representation for a type `t` and a function `eqty` for comparing the type representation with the representation of some other type `'s`. If the representations are determined to be equal, `eqty` returns the value `Some Refl`, which makes it possible for `t` and `'s` to be used interchangeably. (The unit argument to the `type_rep` function is there for technical reasons; it prevents difficulties in the next section, where `TYPEABLE` values are used in recursive modules, which have restrictions on non-function members (Leroy 2003).)

Figure 4 implements a binary equality function `=~~=` in terms of `eqty`, which passes the type representation of the right argument to the equality function of the left argument.

Figure 5 adds a constructor, `Int`, for representing the `int` type, to `type_rep`, and gives an instance of `TYPEABLE` for `int`. The `type_repr` constructor simply returns `Int`. The `eqty` function examines its argument of type `b type_rep`: if it is revealed to be `Int` then `b` must be equal to `int` and the type checker allows `Some Refl` to be returned; otherwise nothing more is revealed about the type and the function returns `None`.

Figure 6 extends `type_rep` with a constructor `List` with one argument for representing the one-parameter type constructor `list`, and gives an implementation of `TYPEABLE` for `list`, parameterised by an implementation of `TYPEABLE` for `list`'s type parameter. The implementation of `eqty` must examine both the `List` constructor and the argument of `List` before enough information is revealed to permit the return of `Some Refl`.

2.3 The Second SYB Ingredient: Generic Operations

The second ingredient of Scrap Your Boilerplate is a small set of generic operations over data.

```
module type TYPEABLE = sig
  type t
  val type_rep : unit → t type_rep
  val eqty : 's type_rep → (t, 's) eql option
end
```

Figure 3. The TYPEABLE interface

```
val (=~~=) : {A:TYPEABLE} → {B:TYPEABLE} →
             (A.t,B.t) eql option
let (=~~=) {A: TYPEABLE} {B: TYPEABLE} =
           A.eqty (B.type_rep ())
```

Figure 4. Equality for TYPEABLE instances

```
type _ type_rep += Int : int type_rep

module Typeable_int :
  TYPEABLE with type t = int =
struct
 type t = int
 let eqty : type b.
  b type_rep → (t, b) eql option =
   function Int → Some Refl | _ → None
 let type_rep () = Int
end
```

Figure 5. TYPEABLE for int

```
type _ type_rep += List : 'a type_rep → 'a list type_rep

module Typeable_list(A: TYPEABLE) :
  TYPEABLE with type t = A.t list =
struct
 type t = A.t list
 let eqty : type b.
  b type_rep → (t,b) eql option =
   function
     List a →
       (match A.eqty a with
         Some Refl → Some Refl
       | None → None)
   | _ → None
 let type_rep () = List (A.type_rep ())
end
```

Figure 6. TYPEABLE instances for int and list

Figure 7 gives the definition of the DATA signature[1], which supports the operations of TYPEABLE, together with two additional operations. The first operation, gmapT, is a kind of generic map, which transforms a value by applying its argument function to every subvalue. The second operation, gmapQ, is a kind of generic query, which collects the results of applying its argument function to every subvalue. (The full implementation has further operations, including support for generic folds.)

The types of gmapT and gmapQ include some elements which are not part of the current OCaml release. The brace-enclosed argument type {D: DATA} denotes an implicit module argument, which is part of a proposed extension to OCaml for overloaded function

[1] Recursive module types, introduced with module type rec, are a small, independently-useful language extension, which are macro-expressible in terms of OCaml's existing recursive modules.

```
module type rec DATA =
sig
 type t
 module Typeable : TYPEABLE with type t = t
 val gmapT : ({D: DATA} → D.t → D.t) → t → t
 val gmapQ : ({D: DATA} → D.t → 'u) → t → 'u list
end
```

Figure 7. The DATA interface

support (White et al. 2015). Modular implicits are not an essential feature of the implementation given here — it would be possible to pass explicit dictionaries in place of implicit arguments — but significantly improve usability. Figure 8 shows how various aspects of modular implicits correspond to similar features in Haskell's type classes. There are differences between implicits and type classes, but they do not play a significant role in this work.

Figure 9 defines aliases for the argument types of gmapT and gmapQ, which will appear frequently in the following pages.

Figure 10 defines two implicit instances of the DATA signature. The first instance, Data_int, gives an implementation of gmapT which simply returns its second argument and an implementation of gmapQ which returns an empty list, ignoring both its arguments — in both cases reflecting the fact that an int value has no subnodes. The second instance, Data_list, gives implementations of gmapT and gmapQ which apply the argument function f to each sub-node, reconstructing the list or collecting the results as appropriate. The generic type of f allows it to be applied to any value for which a suitable implicit argument is available; in particular, it can be applied to x, which has type A.t, since the implicit argument A is available, and to xs, which has type t, since the implicit module Data_list(A) is available.

Figure 11 defines an implicit functor, Typeable_of_data, which projects the TYPEABLE member from an implicit DATA module.

Mechanising DATA instance construction In practice, DATA instances such as Data_int and Data_list will be synthesised automatically from a type definition rather than written by hand, much as GHC automatically generates instances for the Typeable and Data classes in the original SYB library (The GHC Team 2015, 7.5.3).

To support convenient access to the gmapT and gmapQ members of a DATA module, Figure 12 defines top-level functions of the same names which simply project out the corresponding members from their implicit arguments.

Here is an example of gmapT in action:

```
# gmapT (mkT succ) [10; 20];;
- : int list = [11; 20]
```

Since gmapT applies its first argument only to immediate sub-nodes of its second — here 10 and [20] — only the first integer is incremented. (The mkT function, which builds a generic function from a non-generic function, is defined in the next section.)

2.4 The Third SYB Ingredient: Generic Traversal Schemes

The final ingredient of the Scrap Your Boilerplate library is a set of recursive schemes built atop gmapT and gmapQ.

Figure 13 shows the types of some representative schemes. Unlike the non-recursive gmapT and gmapQ functions of Section 2.3, each of the schemes everywhere, everything, and gsize involves traversing entire values.

Building the generic schemes of Figure 13 from the generic operations of Section 2.3 involves "tying the knot" — i.e. passing a function to gmapT and gmapQ that invokes those operations recursively on subvalues.

```
module type SHOW = sig
  type a
  val show : a → string
end
```

```
class Show a where
   show :: a → String
```

Declaring overloaded operations

```
implicit module Show_option {A: SHOW} =
struct
  type a = A.a option
  let show o = match o with
    Some x → "Some "^ A.show x
  | None → "None"
end
```

```
instance Show a => Show (Maybe a)
 where
  show o = case o of
    Just x → "Just " ++ show x
    Nothing → "Nothing"
```

Implementing overloaded operations

```
val print : {A:SHOW} → A.a → unit
```

```
print :: Show a => a → IO ()
```

Types of overloaded functions

```
let print {A:SHOW} (x: A.a) =
  print_string (A.show x)
```

```
print x =
  putStr (show x)
```

Defining overloaded functions

```
print 10
```

```
print 10
```

Calling overloaded functions

Figure 8. Polyglot: OCaml implicits vs Haskell typeclasses

```
type genericT = {D: DATA} → D.t → D.t
type 'u genericQ = {D: DATA} → D.t → 'u
```

Figure 9. `genericT` and `genericQ`

```
implicit module Data_int
 : DATA with type t = int =
struct
 type t = int
 module Typeable = Typeable_int
 let gmapT _ x = x
 let gmapQ _ _ = []
end

implicit module rec Data_list {A: DATA}
 : DATA with type t = A.t list =
struct
 type t = A.t list
 module Typeable = Typeable_list(A.Typeable)
 let gmapT : genericT → t → t =
     fun f l → match l with
       [] → []
     | x :: xs → f x :: f xs

 let gmapQ (f : _ genericQ) (l : t) =
   match l with
     [] → []
   | x :: xs → [f x; f xs]
end
```

Figure 10. `DATA` instances

```
implicit module Typeable_of_data{F: DATA} = F.Typeable
```

Figure 11. `TYPEABLE` from `DATA`

Writing generic traversals in this *open-recursive* style allows considerable flexibility in the form of the traversal. For example, `everywhere` traverses values in a bottom-up fashion by applying the argument function `f` to the result of the call to `gmapT`, while

```
val gmapT : genericT → genericT
let gmapT f {D: DATA} = D.gmapT f

val gmapQ : 'u genericQ → 'u list genericQ
let gmapQ f {D: DATA} = D.gmapQ f
```

Figure 12. Top-level functions `gmapT` and `gmapQ`

```
(* Apply a transformation everywhere, bottom-up *)
val everywhere : genericT → genericT

(* Summarise all nodes, top-down, left-to-right *)
val everything :
  ('r → 'r → 'r) → 'r genericQ → 'r genericQ

(* Compute size of an arbitrary data structure *)
val gsize : int genericQ
```

Figure 13. Some generic SYB schemes

```
let rec everywhere : genericT → genericT =
  fun (f : genericT) {D:DATA} x →
    f ((gmapT (everywhere f) : genericT) x)

let rec everything : 'r. ('r → 'r → 'r) → 'r genericQ →
'r genericQ =
  fun (@) g {D: DATA} x →
    fold_left (@) (g x) (gmapQ (everything (@) g) x)

let rec gsize {D:DATA} v = 1 + sum (gmapQ gsize v)
```

Figure 14. Some generic schemes (implementations)

`everything` traverses top-down, collecting results, and `gsize` combines intermediate results to compute an integer (Figure 14).

`mkT`* and *`mkQ` The `everywhere` function, like many SYB generic schemes, accepts a generic function as an argument. A common pattern when calling such schemes is to convert a non-generic func-

```
val mkT : {T:TYPEABLE} → (T.t → T.t) → genericT
val mkQ  : {T:TYPEABLE} → 'u → (T.t → 'u) → 'u genericQ
```

Figure 15. Converting functions to `genericT` and `genericQ` (types)

```
let appT (type a) (type b)
    (module A:TYPEABLE with type t = a)
    (module B:TYPEABLE with type t = b)
    (g : b → b) (x : a) : a =
  match {A} =~~= {B} with
    Some Refl → g x
  | _ → x

let mkT {T:TYPEABLE} g : genericT =
  fun {D:DATA} →
    appT (module D.Typeable) (module T) g

let appQ (type a) (type b)
    (module A : TYPEABLE with type t = a)
    (module B : TYPEABLE with type t = b)
    u (g : b → 'u) (x: a) =
  match {A} =~~= {B} with
  | Some Refl → g x
  | _         → u

let mkQ {T:TYPEABLE} u g : 'u genericQ =
  fun {D: DATA} x →
    appQ (module D.Typeable) (module T) u g x
```

Figure 16. Converting functions to `genericT` and `genericQ`

tion `f` of some type `t` → `t` to a generic function of type `genericT` that applies `f` to values of type `t` and leaves other values unchanged. The SYB function `mkT` performs this conversion. Here is an example, showing the result of applying `mkT` to the standard library function `succ`, which increments an `int`:

```
# mkT succ;;
- : Syb.genericT = <fun>
# mkT succ 10;;
- : int = 11
# mkT succ "hello";;
- : string = "hello"
```

Similarly, `mkQ` converts a function of type `a` →`b` and a default value of type `b` into a generic query of type `b genericQ`. Here is an example, showing how to use `mkQ` to convert the standard library function `string_of_int` into a generic query that turns `ints` into `strings`, and returns `"not an int"` for values of other types:

```
# mkQ "not an int" string_of_int 10;;
- : string = "10"
# mkQ "not an int" string_of_int [1;2;3];;
- : string = "not an int"
```

Figure 15 gives the types of `mkT` and `mkQ`, which indicate that the argument types of the converted functions must have `TYPEABLE` instances.

Figure 16 gives the implementations of `mkT` and `mkQ`. The implementation is given in terms of auxiliary functions `appT` and `appQ`, which compare the type representations of the function domain and the argument and optionally apply the function if the representations match. The use of an auxiliary function is for technical reasons rather than for clarity: it makes it possible to introduce the "locally abstract types" `a` and `b`, which are a prerequisite for GADT refinement.

3. Staging Scrap Your Boilerplate

SYB overhead It has frequently been observed that the SYB library has poor performance compared to handwritten code (e.g. Adams et al. (2015)). The causes of this inefficiency are various, but most can be traced back to various forms of abstraction — that is, to delaying of decisions about which code should be executed until the last possible moment. For example

- Almost every function call in an SYB traversal is a call to a polymorphic overloaded function.
- Almost every function call in an SYB traversal is indirect — through an argument rather than a statically-known function.
- Many common SYB schemes involve a runtime type equality check on almost every call.

Multi-stage programming makes it possible to systematically eliminate each of these sources of inefficiency, transforming polymorphic functions into monomorphic functions, indirect calls into direct calls, and dynamic type equality checks into static code specializers.

Here is the code from the introduction again:

```
everywhere (mkT munge) s
```

Suppose, for concreteness, that `munge` is the successor function `succ`, and that `s` has type `(int * int) tree`, where `tree` is defined as follows:

```
type 'a tree =
    Leaf
  | Bin of 'a * 'a tree * 'a tree
```

Hand-written code that performs the same function as the call above might then have the following form:

```
let rec incTree : (int * int) tree → (int * int) tree =
  fun t → match t with
  | Leaf →
    Leaf
  | Bin ((a, b), l, r) →
    Bin ((succ a, succ b), incTree l, incTree r)
```

Compared to this relatively efficient code, the call to `everywhere` performs a great many fruitless operations, attempting to apply the generic function `mkT succ` to every node, including trees and pairs, and dispatching recursive calls through polymorphic functions and `DATA` instances.

Optimising the existing inefficient SYB implementation of Section 2 to perform (almost) as well as this hand-written code does not require a complete rewrite. Sections 3.1 and 3.2 demonstrate how to transform the inefficient SYB implementation step-by-step into a code generator and specializer.

These changes to SYB involves changing two of the three ingredients in the implementation.

The code implementing the type equality test (Section 2.2) remains entirely unchanged, although it will be employed statically rather than dynamically — that is, during code generation rather than code execution — to generate code rather than to traverse values.

The generic operations of Section 2.3 are specialized to particular types. The functions `gmapT` and `gmapQ` become code generators (Section 3.1).

The recursive knot-tying schemes of Section 2.4 are transformed, first into open-recursive functions, and then into generators which can build cliques of mutually-recursive monomorphic functions (Section 3.2).

3.1 Staged Generic Operations

Staging basics The aim of *staging* a function in MetaOCaml is to change its behaviour so that rather than returning a regular value it constructs code that computes that value. Staging is accomplished by introducing quotations and antiquotations. Enclosing an expression `e` of type `t` in quotations, `.<e>.`, delays its evaluation, instead constructing a code value of type `t code`. Conversely, splicing an expression `e` of type `t code` within a quotation forces its evaluation to produce a code value which is inserted into the enclosing quotation.

3.1.1 Background: Partial Evaluation

Many forms of staging can be viewed as a kind of programmer-directed partial evaluation, and the partial evaluation literature (e.g. Jones et al. (1993)) provides useful vocabulary. Partial evaluation divides the variables in an environment into *static* — those whose values are available immediately — and *dynamic* — those whose values are not yet available — a process known as *binding-time analysis*. Binding-time analysis generalizes from variables to expressions: those expressions which involve only static variables are classified as static. Given a binding-time analysis, a term can be factored into two parts, the first of which — the *generating extension* — operates on the static data to produce the second — the *residual program*. Since only part of the evaluation remains to be performed, the residual program is typically more efficient than the original program.

The staging process is an essentially mechanical refactoring. Given a binding-time analysis which classifies each variable and each expression as static or dynamic, an unstaged expression of type `t` can be viewed without loss of generality as a function of type $\texttt{t}_{sta} \to \texttt{t}_{dyn} \to \texttt{t}$ from a static value of type $\texttt{t}_{sta}$ and a dynamic value of type $\texttt{t}_{dyn}$.

The first step changes the types of dynamic variables of type `t` to `t code`, quotes dynamic expressions, and encloses static expressions that occur within dynamic expressions in splices. (At this point there is also the option of performing *binding-time improvements* — transformations of the original program such as CPS conversion (Nielsen and Srensen 1995) or eta expansion (Danvy et al. 1996) which allow more expressions to be classified as static.) The result of this step is is a code-generating function of type $\texttt{t}_{sta} \to \texttt{t}_{dyn}$ `code` $\to$ `t code`.

The second step constructs the static data and supplies them to the function, producing a function of type $\texttt{t}_{dyn}$ `code` $\to$ `t code`. The function `back`, defined as follows

```
let back f = .< fun x → .~(f .<x>.) >.
```

turns this code generator into a residual program of type $(\texttt{t}_{dyn} \to \texttt{t})$ `code`.

Finally, *running* the residual program produces a function of type $\texttt{t}_{dyn} \to \texttt{t}$ which can be applied to the dynamic data when they become available.

3.1.2 Binding-Time Analysis and SYB

The binding-time analysis for SYB is particularly simple. The generic operations `gmapT` and `gmapQ` both take implicit arguments describing the type structure of the values passed as non-implicit arguments. The implicit parameters bound to type variables are inserted by the compiler during compilation, and so can be classified as static (but see Section 5.3 for some exceptions). The non-implicit parameters — the actual values to be traversed — are typically only available later, and so are classified as dynamic. The result is that SYB functions are changed from functions which accept both type representations and values at runtime to functions which first accept type representations, which they use to generate code representing functions which accept values.

Staging `DATA` The binding-time analysis for `DATA` transforms the signature as shown in Figure 17. Each implicit argument is classified as static, and so is left unchanged. Each non-implicit argument

```
module type rec DATA =
sig
 type t
 module Typeable : TYPEABLE with type t = t
 val gmapT_ : ({D: DATA} → D.t code → D.t code) →
              t code → t code
 val gmapQ_  : ({D: DATA} → D.t code → 'u code) →
              t code → 'u list code
end
```

Figure 17. `DATA`, staged

and each return type is classified as dynamic, and becomes a `code` value. The function arguments of `gmapT_` and `gmapQ_` are classified as static, even though in practice the functions passed might not be fully known in advance. However, Section 3.3 shows that it is still possible to use `gmapT_` and `gmapQ_` with statically-unknown functions.

```
type genericT_ = {D:DATA} → D.t code → D.t code
type 'u genericQ_ = {D: DATA} → D.t code → 'u code
```

Figure 18. `genericT` and `genericQ`, staged

The definitions of `genericT` and `genericQ` need to be modified accordingly (Figure 18).

```
implicit module Data_int = struct
  type t = int
  module Typeable = Typeable_int
  let gmapT_ _ x = x
  let gmapQ_ _ _ = .<[]>.
end

implicit module rec Data_list {A: DATA} = struct
   type t = A.t list
   module Typeable = Typeable_list(A.Typeable)
   let gmapT_ (f : genericT_) l =
     .< match .~l with
         [] → []
       | x :: xs → .~(f .<x>.) :: .~(f .<xs>.) >.

   let gmapQ_  (f : _ genericQ_) l =
     .< match .~l with
       | [] → []
       | x :: xs → [.~(f .<x>.); .~(f .<xs>.)] >.
end
```

Figure 19. `DATA` instances, staged

Applying the staging process described above to `Data_int` and `Data_list` results in the code in Figure 19. In `Data_list`, in both `gmapT` and `gmapQ`, `l` is dynamic, and must be spliced within the quotation which builds the body of the function. Similarly, the variables `x` and `xs` are dynamic, since they are only available when `l` is examined; they must be passed to `f` as quoted code values. However both `f` and its implicit argument are classified as static, and so `f` can be called immediately, generating code that is spliced into the body quotation.

Figure 20 gives staged equivalents of the top-level operations `gmapT` and `gmapQ`.

Here is the staged analogue of the example at the close of Section 2.3, illustrating the transformation of `gmapT` from a generic function to a generic code generator:

```
# (generateT (gmapT_ (mkT_ (fun x → .<succ .~x>.)))
   : (int list → int list) code);;
```

```
val gmapT_ : genericT_ → genericT_
val gmapQ_ : 'u genericQ_ → 'u list genericQ_

let gmapT_ f {D: DATA} = D.gmapT_ f
let gmapQ_ f {D: DATA} = D.gmapQ_ f
```

Figure 20. generic operations, staged

```
- : (int list → int list) code =
 .< fun x_1  →
     match x_1 with
     | [] → []
     | x_2::xs_3 → Pervasives.succ x_2 :: xs_3 >.
```

The `generateT` function instantiates a generic function of type `genericT_` to build a value of type `(t →t) code`. Its operation is described more fully in Section 3.2. The type annotation on the call to `generateT` is essential for guiding the specialization process. If the type of the result of the call is not known then the implicit arguments cannot be instantiated and no code generation takes place.

Inspecting the generated code reveals that there is no trace of either `TYPEABLE` or `DATA`. These generic signatures are now used only during generation of type-specialized traversals and are no longer needed for the traversals themselves; they can be discarded, along with all the other generic function machinery and its associated overhead, before the call to the generated function takes place.

3.2 Staged Traversal Schemes

Staging non-recursive code such as `gmapT` and `gmapQ` is straightforward. However, most useful functions in SYB are recursive, introducing new challenges for staging. In the general case, applying a generic scheme such as `everywhere` may involve traversing a number of mutually-recursive types, and so specializing a generic scheme involves generating a set of mutually-recursive functions.

```
(* Apply a transformation everywhere (bottom-up) *)
val everywhere_ : genericT_ → genericT_

(* Summarise all nodes (top-down, left-to-right) *)
val everything_ : ('r code → 'r code → 'r code) →
                  'r genericQ_ → 'r genericQ_

(* Compute size of an arbitrary data structure *)
val gsize_ : int genericQ_
```

Figure 21. Some staged generic SYB schemes

Figure 21 shows the types of the schemes of Figure 13 after applying the binding-time analysis of Section 2.4, systematically classifying implicit arguments as static and non-implicit arguments as dynamic.

Unfortunately, while updating the types of generic schemes for the multi-stage context is straightforward, naively applying the approach described in Section 3.1 is doomed to failure. The principal difficulty is that turning dynamic calls into static calls results in code generators that attempt to unroll all recursive structure. Where the static arguments represent types, unrolling the structure will never terminate, since many type definitions, such as `list`, are cyclic.

However, there are existing techniques for dealing with staging of recursive functions, which can be generalized to the current setting. Kameyama et al. (2011) outline the following three step approach.

First, the recursive function should be written in open-recursive style, and made recursive with a fixpoint combinator. For example, the `everywhere` function of Figure 14 should be rewritten to take an additional argument which is used for self calls:

```
let everywhere (f : genericT) =
  gfixT (fun self {X:DATA} x → f (gmapT self x))
```

The `gfixT` function, described below, is a fixpoint operator suitable for constructing generic functions of type `genericT`.

Second, the fixpoint combinator can be modified to support memoization, avoiding non-terminating recursion. Since `gfixT` builds functions of type `{D:DATA} →D.t →D.t`, memoization involves maintaining a heterogeneous map from `DATA` instances `D` (or equivalently, from `TYPEABLE` instances) to the corresponding functions of type `D.t →D.t`.

Third, the fixpoint combinator can be modified to support `let` insertion, avoiding duplication of computations. Generic functions require `let rec` insertion in the general case rather than simple `let` insertion, since the functions generated for mutually-recursive types must be mutually recursive.

```
let everywhere_ (f : genericT_) =
  gfixT_ (fun self {X:DATA} x →
           f (gmapT_ self x))

let everything_ (@) (g : _ genericQ_) =
  gfixQ_ (fun self {X: DATA} x →
      let f = g x in .<
      let rec crush u = function
        [] → u
      | x :: xs → crush (.~(.<u>. @ .<x>.)) xs
      in crush .~f .~(gmapQ_ self x) >.)

let gsize_ = gfixQ_ (fun self {D:DATA} v →
    .< 1 + List.fold_left (+) 0 .~(gmapQ_ self v) >.)
```

Figure 22. Some staged generic SYB schemes (implementations)

Figure 22 shows the result of staging `everywhere`, `everything` and `gsize` with fixpoint combinators `gfixT_` and `gfixQ_` that perform memoization and `let rec` insertion. The remainder of this section investigates how to define those combinators.

Generic fixpoint combinators Here is a definition of `gfixT`, an unstaged, non-memoizing fixpoint combinator for constructing `genericT` values:

```
let rec gfixT : (genericT → genericT) → genericT =
  fun f {D: DATA} (x:D.t) → f {D} (gfixT f) x
```

This definition of `gfixT` is easy to construct by starting from the equation `f = f (fix f)`, eta-expanding to account for the call-by-value setting, and then generalizing further over the implicit argument `D`.

The definition of `gfixQ` is similar:

```
let rec gfixQ : ('u genericQ → 'u genericQ) →
                'u genericQ =
  fun f {D: DATA} (x:D.t) → f {D} (gfixQ f) x
```

(Throughout this section it would be possible to combine `gfixT` and `gfixQ` into a single definition, using a functor to abstract over the differences in type, at the cost of some clarity in the code. Instead, I will focus on `gfixT`; the reader is invited to fill in the details for `gfixQ`, or consult the online code repository at the location given in Appendix A.)

These definitions are adequate for rewriting the unstaged SYB schemes in an open-recursive style.

```
type map
val new_map : unit → map ref
val lookup : {T:TYPEABLE} →
             map → (T.t → T.t) option
val push : {T:TYPEABLE} →
           map ref → (T.t → T.t) → unit
```

Figure 23. `TYPEABLE`-keyed maps: interface

Memoization via heterogeneous maps The next step involves adding support for memoization. Figure 23 gives an interface for a mutable map whose keys are `TYPEABLE` instances and whose values are functions. (This map supports memoization for `genericT` functions; the very similar map that supports `genericQ` memoization is omitted.)

```
type map =
  Nil : map
| Cons : (module TYPEABLE with type t = 'b)
       * ('b → 'b) * map → map

let new_map () = ref Nil

let rec lookup :
 {T:TYPEABLE} → map → (T.t → T.t) option =
  fun {T: TYPEABLE} → function
    Nil → None
  | Cons ((module R), f, rest) →
    match {T} =~~= {R} with
      Some Refl → Some f
    | None → lookup rest

let push {T:TYPEABLE} t c =
  t := Cons ((module T), c, !t)
```

Figure 24. `TYPEABLE`-keyed maps

Figure 24 gives an implementation of the `map` interface. A `map` value is a heterogeneous list, which is either empty (`Nil`), or a `Cons` of some `TYPEABLE` instance `T`, stored as a first-class module, together with a function whose type is shared with the type of `T`, and a further map. The `new_map` function allocates an empty map, stored in a mutable reference cell. The `lookup` function searches the list for an entry whose `TYPEABLE` component `R` matches the implicit argument `T`; if there is a match then the `Refl` GADT constructor modifies the type context with an equality that reveals that the type of the value `f` must match both `R` and the T, allowing `f` to be returned from the function. The `push` function mutates its `map` argument, adding a new entry to the map.

The typed map makes it possible to write a memoizing version of the `gfixT` combinator:

```
let gfixT (f : genericT → genericT) : genericT =
  let tbl = new_map () in
  let rec result {D: DATA} x =
    match lookup {D.Typeable} !tbl with
      Some g → g x
    | None →
      let g = f result {D} in
      let () = push tbl g in
      g x
  in result
```

This memoized definition of `gfixT` is derived directly from the earlier definition by interposing map lookups on every call to the fixpoint combinator, and returning the result of the lookup if an entry is found for the implicit argument `D.Typeable`. The table `tbl` is constructed outside of the recursive body of the combinator, but after the argument `f` has been received; thus, no state is shared between calls to `gfixT`.

```
val let_locus : (unit → 'w code) → 'w code
val genlet : 'a code → 'a code
```

Figure 25. Let insertion with MetaOCaml

Memoizing staged functions The memoizing implementation of `gfixT` above operates on unstaged functions. Supporting staged functions involves changing the type of values in the map from `'b →'b` to `('b →'b) code`:

```
type map =
  Nil : map
| Cons : (module TYPEABLE with type t = 'b)
       * ('b → 'b) code * map → map
```

and making corresponding adjustments to the types of `lookup` and `push`.

`let`- and `let rec`-insertion Memoizing with function values is straightforward: the functions are added to a table and retrieved when needed, at which point they may be called directly. Memoizing with code values is less straightforward. Since MetaOCaml code values may contain free variables, there are scoping issues to consider: it is essential to ensure that the inserting code retrieved from the table into a larger code value produces a result that is well-scoped. With insufficient care, it is possible for code which is well-scoped in the context where it is inserted into the table to be ill-scoped in the context where it is retrieved.

One tool for managing the scope of generated code is *`let`-insertion*. The goal of `let`-insertion is to float `let` bindings in generated code outwards, ensuring that the code is as efficient as possible. Figure 25 gives the interface to a generic `let`-insertion library for MetaOCaml introduced in the presentation accompanying Kiselyov (2014).

There are two operations. The first, `let_locus`, marks a point on the stack which is suitable for `let`-insertion (much as `try` marks a point where an exception may be caught). The second, `genlet`, takes a code value `.<e>.` as argument, searches all the points which have been marked by `let_locus` to find the outermost point where `.<e>.` is well-scoped, inserts a `let`-binding `.<let x = e in ...>.` there, and returns the newly-introduced bound variable `.<x>.`. The implementation of these operations requires some form of delimited control, such as the `delimcc` library (Kiselyov 2012) or the proposed algebraic effects language feature (Dolan et al. 2015).

In the current case, `let` insertion is not quite sufficient, since the generated code may be mutually recursive. For example, here is a definition of a tree as a pair of mutually-recursive type definitions:

```
type 'a branch = 'a tree * 'a tree
and 'a tree = Leaf of 'a | Branch of 'a branch
```

Traversals specialized to the mutually-recursive `branch` and `tree` types will also be mutually recursive:

```
let rec traverse_branch (l, r) =
       (traverse_tree l, traverse_tree r)
    and traverse_tree = function
       Leaf v → Leaf (succ v)
     | Branch b → Branch (traverse_branch b)
```

and so some way of generating mutually-recursive bindings is needed.

Recursion via mutable state Unfortunately, MetaOCaml does not currently support the dynamic construction of mutually-recursive binding groups, since quotations only support constructing expressions, and so an encoding is needed to achieve equivalent behaviour.

One way of simulating recursive binding groups is to use mutable state. It is not possible to dynamically construct groups of bindings such as `traverse_branch` and `traverse_tree`, but it is possible to achieve the same effect using reference cells. The idea is to start with cells that initially contain dummy functions:

```
let traverse_branch = ref (fun _ → assert false) in
let traverse_tree = ref (fun _ → assert false) in
```

With the bindings established the cells can be populated with the real function implementations. At this point, both `traverse_branch` and `traverse_tree` are in scope, and so the bodies of the functions can refer to them to achieve recursive behaviour:

```
   ...
traverse_branch :=
  fun (l, r) → (!traverse_tree l, !traverse_tree r);
traverse_tree := function
   Leaf v → Leaf (succ v)
 | Branch b → Branch (!traverse_branch b);
```

The following implementation of the `genletrec` function captures this idea. A call to `genletrec` inserts two `let` bindings. The first binding introduces a reference `r` initially bound to a dummy function, and the second assigns a value to `r`. The function `k` that constructs the right-hand side has access to the dereferenced `r`, making it possible to build recursive functions:

```
let genletrec k =
 let r = genlet (.< ref (fun _ → assert false) >.)
 in
   genlet (.<.~r := .~(k .< ! .~r >.) >.);
   .< ! .~r >.
```

Finally, the following definition of `gfixT_` combines staging, (heterogeneous) memoization and `let rec`-insertion:

```
let gfixT_ (f : genericT_ → genericT_) =
 let tbl = new_map () in
 let rec result {D: DATA} x =
   match lookup {D.Typeable} !tbl with
     Some g → .< .~g .~x >.
   | None →
      let g = genletrec
          (fun self →
             push tbl self;
            .< fun y → .~(f result .<y>.) >.)
       in .< .~g .~x >.
  in result
```

The argument to `genletrec` begins by adding an entry to the table for the fresh binding so that if the table is consulted during the call to `f` the binding will be available. Section 3.3 shows the (single) call to `let_locus` that determines where the bindings are inserted in the generated code.

This definition of `gfixT_` supports both the definitions of the staged generic schemes of Figure 22, and the other generic schemes in the SYB library.

***Staging* `mkT` *and* `mkQ`** .

It remains only to implement staged versions of the `mkT` and `mkQ` functions of Figure 16. Since these functions operate on `TYPEABLE` values, which are only used statically, the staging is entirely straightforward, and involves only the addition of the `code` type constructor to the annotations of dynamic variables in `appT` and `appQ` (Figure 26).

```
let appT_ (type a) (type b)
  (module A : TYPEABLE with type t = a)
  (module B : TYPEABLE with type t = b)
  (g : b code → b code) (x : a code) : a code =
  match {A} =~~= {B} with
  | Some Refl → g x
  | _         → x

let mkT_  {T:TYPEABLE} g : genericT_ =
 fun {D: DATA} →
   appT (module D.Typeable) (module T) g

let appQ_ (type a) (type b) (type u)
    (module A : TYPEABLE with type t = a)
    (module B : TYPEABLE with type t = b)
    u (g : b code → 'u code) (x: a code) =
  match {A} =~~= {B} with
  | Some Refl → g x
  | _         → u

let mkQ_ {T:TYPEABLE} u g : 'u genericQ_ =
  fun {D: DATA} x →
   appQ_ (module D.Typeable) (module T) u g x
```

Figure 26. Staging `mkT` and `mkQ`

3.3 Instantiating Staged Generic Functions

The staged generic schemes in Section 3.2 are built around functions from `code` to `code`. Using the code generated by such a function involves converting it to a single piece of closed code, then compiling that code into a runnable function using MetaOCaml's builtin `run` operation, which serves as a kind of `eval`.

```
val generateT : {D:DATA} → genericT_ → (D.t → D.t) code

val generateQ : {D:DATA} → 'u genericQ_ →
                 (D.t → 'u) code
```

Figure 27. Code generation: types

```
let generateT {D: DATA} (f : genericT_) =
  let_locus (fun () → .< fun x → .~(f .<x>.) >.)

let generateQ {D: DATA} (q : 'u genericQ_) =
  let_locus (fun () → .< fun x → .~(q .<x>.) >.)
```

Figure 28. Code generation: terms

Figures 27 and 28 give the types and implementations of functions `generateT` and `generateQ`, which convert staged generic schemes into code values. Both `generateT` and `generateQ` accept an implicit argument representing the type at which the generated code should be instantiated, and a generic scheme. The call to `let_locus` function of Figure 25 indicates that the point around the generated functions is a suitable point to bind the specialized schemes generated by the process described in Section 3.2. The quotations in the bodies of the functions correspond to the `back` operation described in Section 3.1, except that there is also an additional implicit argument passed to `f` and `q`.

The types of the `generateT` and `generateQ` functions indicate that `generateT {D} f` and `generateQ {D} q` specialize generic functions `f` and `q` at the type `D.t`. For example, the following call to `generateQ` generates specialized code for computing the sizes of lists of pairs of `int` and `bool`:

```
(generateQ gsize
 : ((int * bool) list → int) code)
```

```
val instantiateT : {D:DATA} → genericT_ → D.t → D.t

val instantiateQ : {D:DATA} → 'u genericQ_ → D.t → 'u
```

Figure 29. Generic function instantiation: types

```
let instantiateT {D: DATA} f =
  Runcode.run (generateT f)

let instantiateQ {D: DATA} q =
  Runcode.run (generateQ q)
```

Figure 30. Generic function instantiation

The type ascription guides the instantiation of the implicit argument to `generateQ`. In practice, the type information propagated from the surrounding context is often sufficient, in which case no ascription is needed.

Figures 29 and 30 give the types and implementations of functions `instantiateT` and `instantiateQ` which convert staged generic schemes into runnable functions by generating code using `generateT` and `generateQ` and then calling MetaOCaml's `run` function on the result.

Instantiating with unknown arguments Although most of the examples in this paper involve cases where the function argument is known at code generation time, the interface also supports generating code that is parameterised by a function. For example, the following expression generates a piece of code of type `((t →t) →(s →s)) code` for some types `t` and `s`:

```
.< fun f →
 .~(generateT
    (everywhere_ (mkT_ (fun x → .< f .~x >.))))>.
```

Generating a parameterised traversal in this way may be more space-efficient than instantiating `everywhere` separately for each argument type.

4. Performance

Figure 31 lists the performance of the staged SYB library on three representative benchmarks, which are OCaml ports of benchmarks described by Adams et al. (2015). Each row records the time taken to invoke each of three implementations of a particular function on the same data: a hand-written implementation, an implementation written using the unstaged SYB library of Section 2, and an implementation generated by the staged SYB library of Section 3.

The first benchmark, `RMWeights`, involves a transformation of a leaf-labeled weighted binary tree:

```
type ('a, 'w) wtree =
  | Leaf of 'a
  | Fork of ('a, 'w) wtree * ('a, 'w) wtree
  | WithWeight of ('a, 'w) wtree * 'w
```

The `rmWeights` operation transforms the tree to eliminate `WithWeight` nodes:

```
let rec rmWeights = function
  | WithWeight (t, w) → rmWeights t
  | Leaf x → Leaf x
  | Fork (l, r) → Fork (rmWeights l, rmWeights r)
```

An equivalent function can be written using `everywhere`, together with an auxiliary non-recursive function `rmWeight` that removes `WithWeight` from single level:

```
let rmWeight = function
  | WithWeight (t, w) → t
  | t → t

let rmWeights : {D:DATA} → D.t → D.t =
 everywhere (mkT rmWeight)
```

As Figure 31 shows, this second implementation is over 19 times slower than the hand-written version. However, switching to the staged of `everywhere` and `mkT` dramatically improves the performance so that it is only around 1.5 times slower than the hand-written version.

The majority of the remaining additional overhead in the generated code is a result of an unnecessary recursive call. In the hand-written implementation the `rmWeights` function entirely ignores the weight `w` in each `WithWeight` node. In contrast, in the implementation generated by the staged SYB library there is a call to the instantiated `gmapT` operation for each weight in the tree. Section 5.1 discusses approaches to eliminating these unnecessary calls.

The second benchmark, `SelectInt`, involves a traversal of the same `wtree` type. The function `selectInt` traverses a `wtree` value to locate and sum all the integer nodes:

```
let rec selectInt : (int, int) wtree → int =
 function
  | Leaf x → x
  | Fork (l, r) → selectInt l + selectInt r
  | WithWeight (t, x) → selectInt t + x
```

The `selectInt` traversal can be succinctly expressed using `everything` and `mkQ`:

```
let selectInt (t : (int, int) wtree) =
  everything (+) (mkQ 0 (fun x → x)) t
```

Once again, Figure 31 shows that the generic implementation comes with a significant performance penalty, running almost 15 times slower. Staging the generic implementation produces a substantial speedup, but the generated code is still over twice as slow as the hand-written version.

The remaining inefficiency in this case is primarily a result of the intermediate list generated by `gmapQ`. Improving the generated code to eliminate the intermediate lists results in a speed-up of almost double, bringing the staged version to within about 20% of the time of the hand-written code. Section 5.2 suggests how `gmapQ` might be improved to avoid the overhead introduced by list processing.

The third benchmark, `Map`, compares the performance of a hand-written map over a binary tree with an equivalent traversal generated by instantiating `everywhere`:

```
let mapTree_succ : int tree → int tree =
 fun t → everywhere (mkT succ) t
```

Once again, the generic version suffers from significantly worse performance than the hand-written code. The performance overhead of the staged version is significantly less; in fact, the only source of degraded performance is the encoding of mutual recursion using references (Section 3.2). Once these are eliminated the generated code is slightly faster than the hand-written implementation, apparently as a result of the inlining of `succ` in the generated traversal.

Figure 31 records only the performance of the generated code, and excludes the cost of code generation. However, since the time taken to generate code does not vary with the size of the data, the overhead of generating code is typically insignificant for programs which process large data sets, or which call the generated functions many times. For example, the time taken to generate code for the `Map` benchmark is less than the time taken for a single generic call to process a tree containing 3000 elements.

Benchmark	Time: hand-written code	Time: SYB (vs handwritten)	Time: SYB staged (vs handwritten)
RMWeights	93.33μs	1779.72μs (19.1x)	136.26μs (1.46x)
SelectInt	70.19μs	1032.44μs (14.71x)	158.71μs (2.26x)
Map	95.99μs	1775.17μs (18.49x)	107.62μs (1.12x)

Figure 31. Performance measurements

5. Extensions

The analysis in Section 4 identified a number of opportunities to improve the performance of the generated code.

5.1 Selective Traversal

The most significant overhead in the `RMWeights` benchmark is a result of traversing an `int` value to look for `wtree` subvalues. Improving the code generation to detect cases like this — where it is clear from the types that traversing a subtree has no possibility of success — would bring the performance of the staged SYB library significantly closer to hand-written code.

This kind of selective traversal, where static information is used to prune branches from the dynamic search, has been successfully employed by some implementors of SYB variants to improve performance (Boulytchev and Mechtaev 2011; Adams and DuBuisson 2012).

5.2 Improving the Type of `genericQ`

Almost all of the overhead in the `SelectInt` benchmark is attributable to the intermediate list in the `gmapQ` traversal. The `gmapQ` traversals use lists because the library in this paper is a direct translation of the original Haskell SYB implementation. However, whereas using lists to stream results between functions is a relatively efficient technique in a lazy language, an interface based on destructors rather than constructors would produce better performance in the MetaOCaml implementation. For example, the `gmapQ_` function (Figure 20) could be replaced with an equally-expressive but more efficient function of the following type, where the first two arguments are respectively used to combine results from different nodes and as the default value at nullary nodes:

```
val gmapQ_ : ('u code → 'u code → 'u code) →
        'u code → 'u genericQ_ → 'u genericQ_
```

5.3 Limitations

The types of all the values used as arguments of generic schemes in this paper are *regular*. The regularity of a type is a property of its definition: if every occurrence of a type constructor `t` in its own definition is applied to the type parameters in order then the type is regular; otherwise, it is irregular.

In common with many other generic programming libraries, the staged SYB implementation in this paper does not support non-regular data types. The difficulty manifests itself in the memoizing fixpoint operator (Section 3.2), where non-regular types can lead to non-termination. For example, here is a definition of a non-regular type of perfect trees:

```
type 'a perfect = Zero of 'a
                | Succ of ('a * 'a) perfect
```

A value of type `'a perfect` may contain sub-values of type `('a * 'a) perfect`. In turn, those sub-values may contain values of type `(('a * 'a) * ('a * 'a)) perfect`, and so on. It is clearly impossible to construct a recursive group of monomorphic functions which can account for all the possible sub-values.

6. Related Work

That generic programming libraries often suffer from poor performance is well known, and there have been several investigations into ways to make them more efficient.

Boulytchev and Mechtaev (2011) (with a more extensive account in Russian (Mechtaev 2011)) explore how to implement SYB efficiently in OCaml. Their implementation preceded the introduction of modular implicits and GADTs, so they use a type-passing implementation together with a type equality based on an unsafe cast. Instead of language-supported staging, they carefully refactor the SYB code to eliminate inefficiencies, translating to CPS and traversing the type structure in advance to build efficient closure-based traversals. They achieve performance fairly close to handwritten code by combining these optimisations with the selective traversal optimisation described in Section 5.1.

The work of Adams et al. (2015) (and the earlier version, Adams et al. (2014)) are a direct inspiration for the work described in this paper. Adams et al. improve the performance of the Scrap Your Boilerplate library by means of a domain-specific optimisation, implemented first as a Hermit script (Farmer et al. 2012), and then as a GHC optimisation. The optimisation seeks to eliminate expressions of "undesirable types" — that is, expressions corresponding to the dictionaries for the `Data` and `Typeable` classes, expressions of type `TypeRep`, and some associated newtypes — from code that uses SYB by various transformations on the intermediate language. The resulting improvements on several benchmarks are impressive, bringing the performance of code using SYB in line with the performance of handwritten code.

The work described in this paper improves on the work of Adams et al. in a number of ways. Staging avoids the need to go outside the language to improve performance — indeed, the semantics of the language stipulate precisely what code should be generated by the implementation of Section 3 — and so the behaviour of the staged library is not vulnerable to changes in the details of optimization passes or other internal compiler issues. Further, MetaOCaml's type system justifies a degree of confidence in the correctness of the staged code. Although there are some scope extrusion errors that can only be detected during code generation (Kiselyov 2014), any code generated by a MetaOCaml program is guaranteed to be well-typed, whereas a poorly written compiler optimisation may contain bugs that result in incorrect output. Finally, focusing on values of "undesirable" type misses some important cases. As Hinze and Löh (2006) observe, it is sometimes useful to treat type representations themselves as values to be operated on by generic functions, but classifying `TypeRep` as unconditionally undesirable necessarily misses this case.

The treatment of implicit arguments as static data in a partial evaluation goes back to Jones (1995), who applies it to the more general case of specializing overloaded functions associated with arbitrary type classes.

Magalhães (2013) applies local rewrite rules to another generic programming library for Haskell, *generic-deriving*, and with careful tuning achieves results equivalent to handwritten code. These results are encouraging, particularly since no compiler modifications are needed. Nonetheless, relying on extra-lingual annotations cannot provide strong guarantees that optimisations will continue to work with future versions of the compiler.

Finally, the staged SYB implementation in this paper can be seen as an kind of *active library* (Veldhuizen 2004) — that is, a library which interacts with the compiler in some way to improve performance. Active libraries are most commonly used in scientific programming domains where performance is critical. The implicit thesis of this paper is that the active library approach also has a role to play in significantly improving the performance of very high-level libraries such as SYB, bringing them to a point where they do not suffer significant disadvantages over hand-written code.

7. Further work

Section 5 proposed some performance improvements to the staged SYB library. Besides improving performance, there are a number of other promising avenues for future exploration.

While MetaOCaml supports generating and loading code at runtime, most of the specialization benefits of the staged SYB library come from making use of information that is available at compile time — that is, the types of the functions and values involved in a call to a generic scheme. The library is therefore likely to be a good fit for the Modular Macros proposal which extends OCaml with compile-time staged programming facilities (Yallop and White 2015), or perhaps for Typed Template Haskell, using monadic effects in place of the delimited control operators used to insert `let rec` bindings (Section 3.2).

SYB is perhaps the best-known generic programming library, but there are a wide variety of approaches to generic programming (e.g Gibbons (2007); Oliveira et al. (2007)). It would be interesting to investigate whether the kind of staging techniques applied here could also be used to improve the performance of other generic programming libraries.

References

M. D. Adams and T. M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium*, Haskell '12, pages 13–24, New York, NY, USA, 2012. ACM.

M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing SYB is easy! In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 71–82, New York, NY, USA, 2014. ACM.

M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing SYB Traversals Is Easy! *Science of Computer Programming*, 2015. To appear.

D. Boulytchev and S. Mechtaev. Efficiently scrapping boilerplate code in OCaml, September 2011. ACM Workshop on ML 2011.

O. Danvy, K. Malmkjær, and J. Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 18(6):730–751, Nov. 1996. ISSN 0164-0925.

S. Dolan, L. White, K. Sivaramakrishnan, J. Yallop, and A. Madhavapeddy. Effective concurrency through algebraic effects. OCaml Users and Developers Workshop 2015, September 2015.

A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. *SIGPLAN Not.*, 47(12):1–12, Sept. 2012. ISSN 0362-1340.

J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

R. Hinze and A. Löh. "Scrap your boilerplate" revolutions. In *Proceedings of the 8th International Conference on Mathematics of Program Construction*, MPC'06, pages 180–208, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35631-2, 978-3-540-35631-8.

P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2008. ACM, 2008.

M. P. Jones. Dictionary-free overloading by partial evaluation. *Lisp Symb. Comput.*, 8(3):229–248, Sept. 1995.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shifting the stage: Staging with delimited control. *J. Funct. Program.*, 21(6):617–662, Nov. 2011. ISSN 0956-7968.

O. Kiselyov. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, June 2012. ISSN 0304-3975.

O. Kiselyov. The design and implementation of BER MetaOCaml. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer International Publishing, 2014. ISBN 978-3-319-07150-3.

R. Lämmel and S. P. Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-649-8.

R. Lämmel and S. P. Jones. Scrap more boilerplate: Reflection, zips, and generalised casts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 244–255, New York, NY, USA, 2004. ACM. ISBN 1-58113-905-5.

X. Leroy. A proposal for recursive modules in Objective Caml. INRIA Rocquencourt, May 2003. Version 1.1.

A. Löh and R. Hinze. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 133–144, New York, NY, USA, 2006. ACM.

J. P. Magalhães. Optimisation of generic programs through inlining. In *Accepted for publication at the 24th Symposium on Implementation and Application of Functional Languages (IFL'12)*, IFL '12, 2013.

S. Mechtaev. Eliminating boilerplate code in Objective Caml programs. *System Programming*, 6(1), 2011. In Russian.

K. Nielsen and M. H. Srensen. Call-by-name CPS-translation as a binding-time improvement. In *STATIC ANALYSIS, NUMBER 983 IN LECTURE NOTES IN COMPUTER SCIENCE*, pages 296–313. Springer-Verlag, 1995.

B. C. d. S. Oliveira, R. Hinze, and A. Loeh. Extensible and modular generics for the masses. In H. Nilsson, editor, *Trends in Functional Programming*. 2007.

The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide*, 7.10.2 edition, July 2015.

T. L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

L. White, F. Bour, and J. Yallop. Modular implicits. ACM Workshop on ML 2014 post-proceedings, September 2015.

J. Yallop and L. White. Modular macros. OCaml Users and Developers Workshop 2015, September 2015.

A. Installation Instructions

The staged SYB library is written using a fork of the OCaml distribution that combines the ongoing work on BER MetaOCaml (Kiselyov 2014) and modular implicits (White et al. 2015) in a single compiler. OPAM users can install the compiler by running the following command:

```
opam switch 4.02.1+modular-implicits-ber
```

The staged SYB library implementation may be installed by running the following command:

```
opam pin add metaocaml-syb \
  https://github.com/yallop/metaocaml-syb
```

The code is available to browse and download at the same URL.