Staged Generic Programming

JEREMY YALLOP, University of Cambridge, UK

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 regrettable choice between writing succinct but slow programs and writing tedious but efficient programs.

Applying structured multi-stage programming techniques transforms *Scrap Your Boilerplate* from an inefficient library into a typed optimising code generator, bringing its performance in line with hand-written code, and so combining high-level programming with uncompromised performance.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Modules / packages*;

Additional Key Words and Phrases: multi-stage programming, generic programming, metaprogramming, partial evaluation

ACM Reference Format:
Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (September 2017), 28 pages.
https://doi.org/10.1145/3110273

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, suppose that you want to search a value `v` for every value of a certain type satisfying some predicate (e.g. even `int` values). You might start by writing code to traverse `v`, examining its constructors and iterating over their fields. Alternatively, you might use a generic programming library such as *Scrap Your Boilerplate* [Lämmel and Peyton Jones 2003] (SYB), and write code like the following:

```
listify evenp v
```

This snippet lists all even integers within `v`, whether `v` is a list, tree, pair, or some more complex structure.

Evidently, generic programming can significantly simplify certain programming tasks. However, this simplification often comes with a severe performance cost. For example, with the SYB implementation of `listify` the snippet above typically executes around 20 times slower than an equivalent hand-written traversal specialised to a particular type (Section 3.3), even if `v` is dense in integers. If the integers are sparsely distributed, performance can be significantly worse.

Multi-stage programming. The poor performance of functions like `listify` is a consequence of the same genericity that makes them appealing. How might we keep the genericity but eliminate the cost?

One approach to eliminating abstraction costs is *multi-stage programming*. Multi-stage programs improve efficiency using information that becomes available after a function is defined but before

2475-1421/2017/9-ART29
https://doi.org/10.1145/3110273

it is invoked. For example, the author of `listify` cannot possibly know the eventual types of its arguments, and so ought to make `listify` as general as possible. However, the caller of `listify` typically knows the types of the arguments before the time when the function is actually called. This type information can be used to specialise `listify` at the call site, taking advantage of type information to optimize the function implementation. In other words, multi-stage programming transforms an inefficient generic function such as `listify` into an optimising code generator.

Since the overheads of SYB are so large, even naive staging that does no more than eliminate type-passing polymorphism can achieve dramatic performance increases [Yallop 2016]. As we shall see, application of more sophisticated staging techniques can achieve more substantial improvements — eliminating branches, projections and function calls, statically merging values, and restructuring code to avoid repeated computation.

For example, here is the code generated by the staged `listify` when `v` has type `(int * string list) list` — that is, when `v` is an association list whose keys are integers and whose values are lists of strings:

```
let rec f l = match l with
  | [] → []
  | hd :: tl → let (x,y) = hd in
               if evenp x then x :: f tl else f tl
```

The code corresponds closely to what one might write by hand: a simple recursive traversal[1] that tests each key `x` in turn, consing the key onto the result if it satisfies `evenp`. The `listify` code generator has determined that the value `y` can be ignored, since a `string list` cannot contain an `int`.

1.1 Outline and Contributions

The central contribution of this work is the principled application of staging techniques to transform a generic programming library into a typed optimising code generator. Several of the techniques used have not previously appeared in the literature but, since they deal with the core elements of functional programming — algebraic data, higher-order functions, recursion, etc. — we anticipate that they will be applicable to a wide class of programs.

The next two sections recapitulate a naive staging of SYB [Yallop 2016], starting with a port of SYB to OCaml (Section 2), which is then staged to turn generic functions into code generators, largely eliminating generic dispatch overhead (Section 3).
The remainder of the paper presents new contributions:

- Building on the staging transformation of Section 3, we show how to further improve performance with an armoury of principled staging techniques: recursion analysis, `let-rec` insertion and inlining (Section 4.1), partially-static data (Section 4.2), reification and reflection (Section 4.3), case lifting (Section 4.4), branch pruning (Section 4.5), and fixed-point elimination (Section 4.6).
  Several of the techniques in Section 4 have roots in type-directed partial evaluation, normalization by evaluation, and other venerable approaches to program transformation. We show here that these techniques find happy expression in a multi-stage programming language with delimited control and advanced forms of polymorphism.
- The simple `listify` example from the introduction guides the development, as we consider techniques that improve the generated code for various types of data. However, the techniques developed for `listify` also apply directly to other generic functions (Section 5).

[1] It is common in OCaml to define functions such as `listify` without using tail recursion; the standard library has many examples, including `map`, `append`, and `split`. Like those functions, the code here runs faster than an equivalent tail-recursive definition that concludes by reversing the list, but risks overflowing the stack on extremely long input.

- An appealing property of quotation-based multistage programming is that optimizations are typically predictable — constructs that do not appear in quotations in the generated program are certain to be absent from the generated program. For this reason, a qualitative evaluation that focuses on the nature of the generated code may be more informative than a quantitative assessment of program running time. Nevertheless, we include a quantitative evaluation in Section 6 to complement the analysis of code generation, and show that the systematic staging of SYB generates code that runs 20-30× faster than the unstaged version, and that equals or outperforms handwritten code.
- Finally, this work serves to demonstrate how staged programming languages support extending libraries with the ability to optimize their own code at call sites. The staged SYB dramatically outperforms the original, but requires no compiler optimizations, no external tools, and no additional sophistication on the part of the user of the library.

2 SCRAP YOUR BOILERPLATE

We begin with an implementation of SYB in OCaml as a starting point for the staged implementation of Section 3.

2.1 SYB Basics

SYB is designed for writing generic traversals over data, with specialised behaviour at certain types. There are three key ingredients in the SYB design. First, a run-time type equality test supports examining the type of data during a traversal (Section 2.2). Second, a small collection of shallow traversals supports traversing individual nodes (Section 2.3). Third, a collection of generic "schemes" builds on these traversals to traverse entire values (Section 2.4).

2.2 The First SYB Ingredient: Type Equality Tests

Type equality tests play a central role in SYB. For example, `listify p` compares the type of each node with the argument type of the predicate `p`.

The `TYPEABLE` interface (Figure 1a) supports type equality tests[2]. Each `TYPEABLE` instance has three members: the type `t`, instantiated to a concrete type such as `int` or `bool list`; the value `tyrep`, which represents the type `t`; and the function `eqty`, which determines whether `tyrep` is equal to some other type representation.

The type `tyrep`, used for type representations, is an *open data type* [Löh and Hinze 2006]. An instance of `TYPEABLE` for a type `t` extends `tyrep` with a new constructor, which takes as many arguments as `t` has type parameters. For example, the `TYPEABLE` instance for `int` extends `TYPEABLE` with a nullary constructor `Int`, and the instance for `list` adds a unary constructor `List` (Figure 1b).

Both `tyrep` and `eql` (used in the return type of `eqty`) are *generalized algebraic data types* (*GADTs*) [Johann and Ghani 2008]. A GADT value transports information about type equalities around a program. A `match` expression that examines a value of type `t tyrep` to find `Int` also discovers that the `t` is equal to `int`, which appears in the return type of `Int`; similarly, a `match` expression that examines a `(s, t) eql` value to reveal `Refl` also reveals that the types `s` and `t` are equal.

In place of GADTs the original SYB uses a general coercion function `gcast` built on an unsafe coercion [Lämmel and Peyton Jones 2004]. The `eqty` function can be used to write `gcast` without unsafe features.

Figure 1b also makes use of *modular implicits*, an OCaml extension for overloading [White et al. 2015]. The `implicit` keyword makes a module available for implicit instantiation. Enclosing

[2] The use of laziness addresses a limitation of recursive modules, which are used for instances of recursive types e.g. `Data_list` in Figure 2a. See the "relaxed in-place update" scheme described by Leroy [2003] for details.

```
type _ tyrep = ..
module type TYPEABLE =
sig
 type t
 val tyrep : t tyrep lazy_t
 val eqty : 's tyrep → (t, 's) eql option
end
```

Fig. 1a. The TYPEABLE interface for type equality tests

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

implicit module Typeable_int =
struct
  type t = int
  let eqty = function
   | Int → Some Refl
   | _ → None
  let tyrep = lazy Int
end
```

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

implicit module Typeable_list {A:TYPEABLE} =
struct
  type t = A.t list
  let eqty = function
    | List a → (match A.eqty a with
                  | Some Refl → Some Refl
                  | None → None)
    | _ → None
  let tyrep = lazy (List (force A.tyrep))
end
```

Fig. 1b. TYPEABLE instances for `int` and `list`

braces, as in `{A:TYPEABLE}`, mark a parameter as implicit. No corresponding argument need be passed; instead, the compiler instantiates the argument automatically. Implicit arguments can also be supplied explicitly using braces: `f {Typeable_int} x`. Modular implicits are not essential to the implementation — explicit dictionaries could serve in place of implicit arguments — but significantly improve usability.

2.3 The Second SYB Ingredient: Generic Operations

The second ingredient of SYB is a small set of shallow traversals over data. The function `gmapQ` is a representative example: it accepts a function `q` and a value `v`, applies `q` to the immediate sub-values of `v`, and returns a list of the results:

$$\texttt{gmapQ q (C (x}_1\texttt{, x}_2\texttt{, ..., x}_n\texttt{)) = [q x}_1\texttt{; q x}_2\texttt{; ...; q x}_n\texttt{]}$$

Figure 2a lines 1–6 give the definition of the DATA signature[3], which includes the operations of TYPEABLE and the `gmapQ` operation. In the full SYB library DATA supports a few additional traversals. However, to simplify the exposition this paper focuses on `gmapQ`, since the staging techniques presented apply to the other traversals without significant modification.

The `gmapQ` function takes two arguments: a generic query of type `genericQ` (defined on Figure 2a line 1), and a value of type `t`. The type `genericQ` is sufficiently general that the query can be applied to any value with a DATA instance; however, the type `'u` returned by the query is the same in each case.

Figure 2a lines 8–20 define two implicit instances of the DATA signature. The first, `Data_int`, defines `gmapQ` to return an empty list, since an `int` value has no sub-values. The second, `Data_list`, defines

[3] Recursive module types, written with `module type rec`, are a language extension, but macro-expressible via OCaml's existing recursive modules.

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

implicit module Data_int = struct
  type t = int
  module Typeable = Typeable_int
  let gmapQ _ _ = []
end

implicit module rec Data_list {A:DATA} = struct
  type t = A.t list
  module Typeable = Typeable_list{A.Typeable}
  let gmapQ q l = match l with
    | [] → []
    | x :: xs → [q x; q xs]
end

let mkQ {T:TYPEABLE} u (g: T.t → u) {D:DATA} x =
  match D.Typeable.eqty (force T.tyrep) with
  | Some Refl → g x
  | _         → u

let single p x = if p x then [x] else []

let rec listify {T:TYPEABLE} p {D:DATA} x =
  mkQ [] (single p) x @ concat (D.gmapQ (listify p) x)
```

Fig. 2a. SYB: a cross-section

```
1  type 'u genericQ  = {D:DATA} → D.t code → 'u code
2  module type rec DATA = sig
3    type t
4    module Typeable : TYPEABLE with type t = t
5    val gmapQ  : 'u genericQ  → t code → 'u list code
6  end
7
8  implicit module Data_int = struct
9    type t = int
10   module Typeable = Typeable_int
11   let gmapQ _ _ = .<[]>.
12 end
13
14 implicit module rec Data_list {A:DATA} = struct
15   type t = A.t list
16   module Typeable = Typeable_list{A.Typeable}
17   let gmapQ q l = .< match .~l with
18       [] → []
19     | x :: xs → [.~(q .<x>.); .~(q .<xs>.)] >.
20 end
21
22 let mkQ {T:TYPEABLE} u (g: T.t code → u code) {D:DATA} x =
23   match D.Typeable.eqty (force T.tyrep) with
24   | Some Refl → g x
25   | _         → u
26
27 let single p x = .<if .~(p x) then [x] else []>.
28
29 let listify {T:TYPEABLE} p = gfixQ (fun self {D:DATA} x →
30  .<.~(mkQ .<[]>. (single p) x) @ concat .~(D.gmapQ self x)>.)
```

Fig. 2b. Naively staged SYB: a cross-section

`gmapQ` to apply the argument function `q` to each sub-node and collect the results. The generic type of `q` allows it to be applied to any value for which a suitable implicit argument is in scope; in particular, it can be applied to `x`, which has type `A.t` and to `xs`, which has type `t`, since the implicit modules `A` and `Data_list(A)` are in scope.

The `mkQ` function (Figure 2a lines 22–25) builds a generic query from a monomorphic function `g` and a default value `u`. The result is a generic query which accepts a further argument `x`; if `x`'s type representation is equal to `g`'s argument type representation then `g` is applied; otherwise, `u` is returned.

Here is an example of `gmapQ` in action:

```
# gmapQ (mkQ false evenp) [10;20;30];;
  - : bool list = [true; false]
```

Since `gmapQ` applies `q` only to immediate sub-values of `v` — here the head of the list, `10`, and the tail, `[20;30]` — only the result for `evenp 10` is returned. In place of the tail, `gmapQ` returns `false`, i.e. the default passed to `mkQ`.

2.4 The Third SYB Ingredient: Generic Schemes

The final ingredient of SYB is a set of recursive schemes built atop `gmapQ` and other traversals. For example, Figure 2a lines 29–30 give a definition of `listify`, which use an auxiliary function `single` (line 27), the standard list concatenation function `concat`, and the generic functions `mkQ` and `gmapQ`. The definition of `listify` may be read as follows: apply `p` to the current node, if the current node is suitably typed, and append the result of applying `listify p` to the sub-values. Thus, whereas `gmapQ` only applies to immediate sub-values, `listify` recursively traverses entire structures. This technique of building a recursive function such as `listify` from a shallow traversal such as `gmapQ` Section 2.3 is sometimes referred to as *tying the knot*.

Writing generic traversals in this *open-recursive* style allows considerable flexibility in the form of the traversal, and SYB supports a wide range of traversals besides `listify`, including `gsize` (compute the size of a value), `everywhere` (apply a function to every sub-value), `synthesize` (build values from the bottom up), and many more.

Dramatis Personæ. We pause to summarise the elements introduced so far. SYB provides interfaces for type equality (Section 2.2) and shallow traversal (Section 2.3), along with recursive schemes such as `listify`. Library authors may provide instances for the data types they define, following the pattern of `Typeable_list` and `Data_list`[4]. SYB users combine schemes and instances by calling generic schemes with suitable instances in scope. Users are, of course, also free to define their own generic schemes.

3 STAGING SYB, NAIVELY

SYB overhead. It is often observed that SYB has poor performance compared to handwritten code [Adams et al. 2015]. The causes of this inefficiency are various, but most can be traced to various forms of abstraction — that is, to delaying decisions until the last possible moment. For example,

- Most function calls in an SYB traversal involve polymorphic overloaded functions.
- Most function calls in an SYB traversal are indirect calls through arguments, not to statically-known functions.
- Many SYB schemes test for type equality at each node.

[4]Instances are often synthesized from type definitions, e.g.by GHC [The GHC Team 2015]

Multi-stage programming can 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 specialisers.

For example, compared to the relatively efficient code on page 1 that searches for even integers in a value of type `(int * bool) list`, the call to `listify` performs a great many fruitless operations, attempting to apply the generic function `mkQ [] evenp` to every node, including lists and pairs, and dispatching recursive calls through polymorphic functions and `DATA` instances.

This section shows how to narrow the performance gap between the SYB implementation (Section 2) and the hand-written code. In particular, Section 3.1 and Section 3.2 transform the inefficient SYB implementation step-by-step into a code generator and specialiser. These changes to SYB involve changing two of the three ingredients in the implementation:

The type equality code (Section 2.2) needs no changes, although it will be used statically rather than dynamically — i.e. during code generation, not during code execution.

Generic operations (Section 2.3) are specialised to particular types: `gmapQ` becomes a code generator (Section 3.1).

Recursive schemes (Section 2.4) are transformed, first into open-recursive functions, and then into generators that build groups of mutually-recursive monomorphic functions (Section 3.2).

3.1 Staged Generic Operations

Staging basics. Staging involves introducing quotations and splices to a program in order to change the program's behaviour so that rather than returning a regular value it constructs code that computes that value. Enclosing an expression `e` of type `t` in quotations:

```
.<e>.
```

delays its evaluation so that rather than evaluating to a value of type `t` it builds a code value of type `t code`. Conversely, splicing an expression `e` of type `t code` into a quotation:

```
.~e
```

indicates that `e` should be evaluated to a code value which is then inserted into the quotation.

Code generated using MetaOCaml [Kiselyov 2014] is guaranteed to be well typed; this is ensured in part by a purely generative design that provides no way to deconstruct code values.

Binding-time analysis. The first step in staging a program, known as *binding-time analysis* [Jones et al. 1993; Taha 1999], divides its free variables into *static* — those whose values are available immediately — and *dynamic* — those whose values are not yet available. The analysis extends from variables to expressions, classifying those expressions that involve only static variables as static, and other expressions as dynamic. In multi-stage languages, binding-time analysis is typically a task for the programmer, and guides the subsequent insertion of quotations and splices into a program.

SYB has a particularly simple binding-time analysis. Values that describe type structure, passed as implicit arguments, are classified as static, and values to traverse, passed as non-implicit arguments, are classified as dynamic. Consequently, SYB functions are changed from functions that accept both type representations and values at runtime to functions that first accept type representations, which they use to generate code representing functions that accept values.

Staging `DATA`. Figure 2b shows the changes to `DATA`. Both the second argument and the result type of `gmapQ` acquire a `code` constructor, since both are classified as dynamic (line 5). The argument and result type of the query passed to `gmapQ` are modified similarly (line 1). However, the query itself is typically supplied directly via a scheme such as `listify`, and so is left as static.

The implementations of `gmapQ` for the `Data_int` and `Data_list` instances follow straightforwardly from the types. For `Data_int` the list returned by `gmapQ` is dynamic (line 11). In `Data_list`'s `gmapQ`, `l` is dynamic, and must be spliced within the quotation that builds the function body (line 17). Similarly, variables `x` and `xs` are dynamic, since they are only available when `l` is examined; they are passed to `q` as quoted code values. However both `q` and its implicit argument are static, and so `q` is called immediately, generating code to be spliced into the body quotation (line 19).

Since `mkQ` operates on `TYPEABLE` values, which are only used statically, only the type of `mkQ`, not its definition, needs to change to give dynamic variables type `code` (line 22).

The staged `mkQ` function examines type representations during code generation to determine how to generate code. Here is the code generated at type `int → bool` for the example from Section 2.3:

```
.< fun y → evenp y >.
```

At type `int list → bool`, the type representations are incompatible and so the generated code is even simpler:

```
.< fun y → false >.
```

In both cases, the generated code reveals no trace of either `TYPEABLE` or `DATA`. These generic signatures are now used only during traversal generation and are no longer needed for the traversals themselves; they can be discarded, along with the other generic function machinery and its overhead, before the call to the generated function takes place.

3.2 Staged Traversal Schemes

Staging non-recursive code such as `gmapQ` and `mkQ` is straightforward. However, the recursive schemes in SYB introduce new challenges for staging. Applying a generic scheme such as `listify` may involve traversing a number of mutually-recursive types, and so specialising a generic scheme involves generating a set of mutually-recursive functions (an instance of so-called *polyvariant specialization* [Hughes 1999; Launchbury 1991]).

Here is the type of `listify` following binding-time analysis, classifying implicit arguments static and others dynamic:

```
val listify : {T:TYPEABLE} → (T.t code → bool code) → T.t list genericQ
```

Staging `listify` involves deciding whether the recursion should be considered static or dynamic. Unfortunately, neither option seems to be what is needed. Since the recursion performed by SYB traverses values, which are dynamic, it is clear that the generated code must be recursive. However, if all recursion is left until the dynamic phase then `listify` will be unable to statically discover the type structure, which is used to generate monomorphic code.

The solution is found in the literature [Kameyama et al. 2011]: replacing `let rec` with a fixed-point operator that traverses the static data (i.e. `DATA` instances) to generate recursive code. Two features of SYB constrain the required behaviour of the fixed-point operator. First, it must perform memoization to avoid non-termination, since SYB type descriptions may contain cycles; for example, the `gmapQ` instance for `Data_list` invokes `q` with the `Data_list` instance. Second, it must be able to generate arbitrary-size recursive groups, since an `SYB` traversal may involve any number of types.

Figure 2b lines 29–30 show the result of staging `listify` with a fixed-point combinator `gfixQ` that performs memoization and `let rec` insertion. The remainder of this section shows how to define that combinator.

Generic fixed-points with memoization. The following definition provides a starting point for `gfixQ`:

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

val new_map: unit → 'a map ref
val lookup: {T:TYPEABLE} → 'a map → (T.t → 'a) option
val push: {T:TYPEABLE} → 'a map ref → (T.t → 'a) → unit
```

Fig. 3. TYPEABLE-keyed maps: interface

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

Fig. 4a. the let insertion interface [Kiselyov 2014]

```
let_locus (fun () → κ[genlet e])
  ⇝
.<let x = .~e in
   .~(let_locus (fun () → κ[.<x>.]))>.
```

Fig. 4b. let insertion: basic operation

```
effect GenLet : 'a code → 'a code
```

Fig. 5a. GenLet, an effect for let insertion

```
let genlet v = perform (GenLet v)
```

Fig. 5b. genlet, a performer of algebraic effects

```
let let_locus body =
  try body ()
  with effect (GenLet v) k →
    .< let x = .~v in .~(continue k .< x >.)>.
```

Fig. 5c. let_locus, a handler of algebraic effects

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

This definition is derived from the standard fixed-point equation `fix f = f (fix f)` by η-expanding to adapt to the call-by-value setting, then generalising over the implicit argument `D`.

Updating `gfixQ` to support memoization requires a suitable memo table. Figure 3 gives a definition: each entry in a `map` value is a pair of an instantiated generic scheme of type `'b → 'a` and a `TYPEABLE` instance for `'b`. The operations `new_map`, `lookup` and `push` define creation, resolution and extension operations for `map`; their implementations are straightforward and omitted.

The `map` type serves as the basis for a memoizing `gfixQ`, which interposes map lookups on every recursive call and adds an entry when lookup fails:

```
let gfixQ f =
  let m = new_map () in
  let rec h {D:DATA} x = match lookup {D.Typeable} !m with
    | Some g → g x
    | None → let g = f h {D} in push m g; g x
  in h
```

let insertion. We take a moment to review standard techniques for let insertion in multi-stage programming as introduced by Kiselyov [2014]. Figure 4a gives the interface, with two operations: `let_locus` marks a point on the stack which is suitable for `let`-insertion, and `genlet` requests that its argument `.<e>.` be inserted at that point. Figure 4b shows the behaviour: a call to `genlet` in a context κ sends the argument `e` to an enclosing `let_locus`, which `let`-binds `e` to `x` and continues with `x` as the argument to κ, enclosing the context with `let_locus`. (In fact, the behaviour is more sophisticated: `genlet` searches the stack for the highest insertion point at which its argument is well-scoped.)

```
let genlet v =
 try perform (GenLet v) with Unhandled → v
```

Fig. 6a. Improved `genlet`: inline when no handler is found

```
let let_locus body =
 try body () with
 effect (GenLet v) k when is_well_scoped v →
  match perform (GenLet v) with
  | v → continue k v
  | exception Unhandled →
   .< let x = .~v in .~(continue k .< x >.)>.
```

Fig. 6b. Improved `let_locus`: insert as high as scoping permits

```
let genrec k =
 let r = genlet (.<ref dummy>.) in
 genlet (.<.~r := .~(k .< !.~r >.) >.);
 .< !.~r >.
```

Fig. 7. The `genrec` combinator

```
let gfixQ f =
 let m = new_map () in
 let rec h {D:DATA} x =
  match lookup {D.Typeable} !m with
  | Some g → .< .~g .~x >.
  | None → let g = genrec @@ fun j →
                    push m j;
                   .<fun y → .~(f h .<y>.)>.
           in .< .~g .~x >.
 in h
```

Fig. 8. The staged `gfixQ` fixed-point combinator

As the use of contexts suggests, the implementation of these operations typically involves requires some form of delimited control, such as the `delimcc` library [Kiselyov 2012] or algebraic effects [Bauer and Pretnar 2012; Dolan et al. 2015], although implementations based on state are also possible [Filinski 2001]. Figure 5a, Figure 5b and Figure 5c give a simple implementation of `let` insertion in terms of algebraic effects: the `genlet` function (Figure 5b) performs the effect `GenLet` (Figure 5a) to transfer control to the handler in the body of `let_locus` (Figure 5c), which builds a `let` quotation, and passes the bound variable `.<x>.` via the continuation `k` back to the context surrounding `genlet`.

Figures 6a and 6b give a more sophisticated implementation. If no handler is in scope then the `genlet` of Figure 6a fails gracefully, returning its argument directly to the calling context instead. The handler in `let_locus` (Figure 6b) is more sophisticated, too: it checks whether the code passed by `getlet` would be well-scoped in the current context; if so, it first tries to forward the request to the surrounding handler, only generating a `let`-binding if forwarding fails.

In each of these implementations, the captured continuation `k` that appears in an `effect` case within the body of a `match` or `try` handler extends to include the handler itself — that is, OCaml implements so-called *deep* handlers.

`let rec` *insertion.* It remains to stage `gfixQ` and add support for `let rec` insertion. Staging `gfixQ` requires a straightforward modification to `map` and its operations to give the stored functions type `('b → 'a) code`.

MetaOCaml does not support generating recursive binding groups of arbitrary size. However, Landin's observation that such groups may be simulated without recursion using mutable references [Landin 1964] suggests an encoding in terms of `let`.

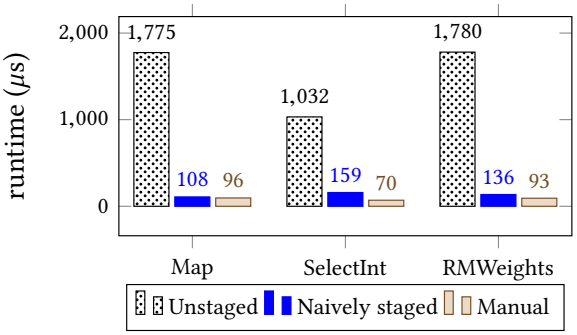


Fig. 9. Naively staged SYB Performance: standard benchmarks

Figure 7 defines a function `genrec`, which adds a binding to a recursive group by inserting two `let` bindings: one to introduce a reference `r`, and a second to assign a value to `r`. The function `k` that constructs the right-hand side has access to `r`, making it possible to build recursive functions:

Finally, Figure 8 gives a new definition of `gfixQ` that combines staging, heterogeneous memoization and `let rec`-insertion. The argument to `genrec` 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. The full implementation of the library provides a function `instantiate` (not shown here) that combines the instantiation of a generic function with a call to `let_locus`, ensuring that bindings are correctly grouped.

This definition of `gfixQ` supports both the staged `listify` (Figure 2b), and the other generic schemes in the SYB library.

3.3 Performance of the Naive Staging

Figure 9 summarises the performance of the naively-staged SYB on a set of benchmarks described by Adams et al. [2015]. For each benchmark the graph records the running time of three implementations of a function on the same data: a hand-written implementation, an unstaged SYB implementation (Section 2), and an implementation generated by staged SYB (Section 3).

Each benchmark has a straightforward implementation as a hand-written recursive function, and a succinct implementation as a generic function. For example, here is the hand-written function for SelectInt, which finds and sums all the weights in a weighted tree:

```
let rec selectInt t = match t with
  | Leaf x → x
  | Fork (l, r) → selectInt l + selectInt r
  | WithWeight (t, x) → selectInt t + x
```

And here is an SYB implementation of the same function:

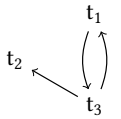
```
let selectInt = everything (+) (mkQ 0 id)
```

As Figure 9 shows, the SYB implementations are between 14 and 19 times slower than equivalent handwritten functions on the test data, and the staged SYB implementations eliminate the majority of this overhead. Yallop [2016] gives further details.

Yallop [2016] goes on to identify three remaining sources of overhead. First, there is an unnecessary `gmapT` call for each weight in the generated code for `rmWeights`. Second, the generated code for `selectInts` builds an intermediate list, unlike the handwritten code. Third, recursive calls through references are slower than direct calls.

```
type t1 = A of t3
 and t2 = B
 and t3 = C of t1
        | D of t2
```

(a) type recursion



(b) type DAG

```
let () = f1 := fun x → e1
let () = f2 := fun x → e2
let () = f3 := fun x → e3
```

(c) Landin-style recursion

```
let f1 x = e1
let rec f2 x = e2
    and f3 x = e3
```

(d) Precise recursion

```
let rec f2 x = e2[f1(x):=e1]
    and f3 x = e3[f1(x):=e1]
```

(e) Precision + inlining

Figure 10. Improved code with simpler recursion

4 STAGING SYB, CAREFULLY

The naively staged SYB shows dramatic improvements over the unstaged library on standard benchmarks (Section 3.3). However, there are two reasons to pause before celebrating.

First, the code generated by the staged version is still significantly slower than hand-written code — over twice as slow on the SelectInt benchmark.

Second, none of the benchmarks represent situations where the hand-written code ignores large parts of the structure, and in such situations the naively staged code exhibits very poor performance. For example, when `listify evenp` is applied to a `string list` the resulting list is certain to be empty, and the optimal code is the constant function:

```
fun _ → []
```

However, the naively staged SYB generates code that traverses the entire list, recursing via references and appending empty lists:

```
let () = f := (fun _ → [])
let () = g := (fun l → match l with
                       | [] → []
                       | h :: t → !f h @ !g t)
```

There is evidently substantial room for improvement here.

The effectiveness of staging is often improved by applying *binding-time improvements* — program transformations such as CPS conversion [Lawall and Danvy 1994; Nielsen and Sørensen 1995] or eta expansion [Danvy et al. 1996] that allow more expressions to be classified as static. This section explores a succession of binding-time improvements to the staged SYB, resulting in a second round of drastic performance improvements and bringing the generated output much closer to hand-written code.

4.1 Recursion and Inlining

The implementation in Section 3 inherits SYB's agnosticism about recursion in type definitions, treating all types involved in a traversal as though they were defined in a mutually recursive group, and so generating a mutually-recursive group of bindings. This is a sound over-approximation, but has two drawbacks for performance. First, there is a certain overhead in each function call, particularly when every function call goes through a mutable cell. Second, the scheme is unfavourable to further optimizations, since it makes inlining difficult; each function definition consequently serves as a boundary between static and dynamic code.

```
let gfixQ f =
 let m = new_map () in
 let rec h {D:DATA} x =
  match lookup {D.Typeable} !m, peers {D} with
  | Some g, _ → R.dyn .< .~g .~x >.
  | None, lazy [] →
    f h {D} x
  | None, lazy [_] →
    let g = genrec1 (fun j → push m j;
                             fun y → (f h y))
     in .< .~g .~x >.
  | None, _  →
    let g = genrec (fun j →
        push m j; .< fun y → .~(f h .<y>.) >.)
     in .< .~g .~x >.
 in h
```

Fig. 11. A `gfixQ` for more precise recursion

```
module type PS =
sig
  type t and sta
  val dyn : sta code → t
  val cd : t → sta code
  val now : t → sta option
end
```

Fig. 12. An interface to partially-static data

Furthermore, it is the over-approximation that makes the encoding with reference cells necessary. In practice, mutual recursion between large numbers of types is rare: the common case is for cycles in type definitions to involve only a single type. There is no difficulty in generating a single recursive binding using `let rec`; it is only when the number of bindings is unknown that difficulties arise.

Figure 10 outlines a superior approach: starting from the types involved in a traversal (Figure 10(a)), the staged code should make use of the dependency graph between types to determine where the cycles occur (Figure 10(b)); then, rather than a single large group tied together with references (Figure 10(c)), the generated code should consist of a sequence of mutually-recursive groups (Figure 10(d)). This approach may be further refined by inlining all the functions generated from non-recursive type definitions (Figure 10(e)).

The improved scheme avoids the performance cost arising from over-approximating recursion between types. More importantly, inlining enables many more opportunities for optimization. (This is no surprise: exposing optimization opportunities is generally the main benefit of inlining [Peyton Jones and Marlow 2002].)

For example, generating code for `listify evenp` at the type `bool * string` involves generating functions at the types `bool` and `string`. For both `bool` and `string` the generated code ignores its argument, returning `[]`, the default value passed to `mkQ`. The generated code for `bool * string` calls the generated functions for `bool` and `string`, and appends the results:

```
fun (x, y) → !f x @ !f y
```

This is clearly sub-optimal: both lists are statically known to be empty and so there is no need for the generated code to perform an append. Inlining alone is not sufficient to eliminate the dynamic append, but it is an essential prerequisite both for that and for many more powerful optimizations described in the following sections (Section 4.2–Section 4.6).

Figure 11 gives an implementation of an improved `gfixQ` that takes proper account of the recursive structure of types. The key new detail is the use of the function `peers` that returns a list of those types that partake in a cycle with `D.t`. (It is perhaps worth noting that `peers` may itself be defined as a function using the SYB framework, since `gmapQ` may be used together with delimited control to retrieve a list of the types immediately referenced by a type with a `DATA` instance, which is all that

is needed to find the cycles.) There are then four cases to consider: either the function is already in the table (line 5); or there are no peers (line 6), in which case the function is non-recursive and is applied directly (i.e. inlined), not inserted in the table; or there is a single peer (line 8), in which case the function `genrec1` (not shown) inserts a single `let rec` binding; or there are multiple peers (line 11), in which case `gfixQ` falls back to the old scheme.

It would, of course, be easy to add a few more cases to handle other small fixed-size recursive groups.

With this improved `gfixQ`, the generated code for `listify evenp` at type `(int * string) list` is as follows:

```
let rec evenslist x = match x with
 | [] → []
 | h::t → let (i, s) = h in
            (if evenp i then [i] else []) @ [] @ evenslist t
```

The generated code for `listify evenp` at the types `int` and `string` has been inlined, and there is an obvious opportunity for optimization (i.e. eliminating the append of `[]`), which the next section considers.

4.2 Partially-Static Data

Several of the code excerpts seen so far have involved dynamic values with some statically-known structure. For example, Figure 2b includes quotations involving list literals of known length such as `.<[]>.` and `.<[x]>.`. Quoted literals often suggest missed opportunities for static computation; for example, in the closing example of Section 4.1, the subexpression `(if evenp i then [i] else []) @ []` can be simplified to eliminate the append.

As with the over-approximation of recursive structure, the root cause is that the binding-time analysis is too crude, marking an entire expression as dynamic whenever any sub-expression is dynamic. Every implementation of the `gmapQ` function must have the same return type (`'u code`); the result is that even fully-static expressions such as `[]` are marked as dynamic and quoted (Figure 2b, line 11).

One solution to this problem is *partially-static data*, well known in the partial evaluation literature as a binding-time improvement [Bondorf 1992; Thiemann 2013], and often used in writing staged programs [Carette et al. 2009; Inoue 2014; Kaloper-Meršinjak and Yallop 2016; Kiselyov et al. 2004; Sheard and Diatchki 2002]. As the name suggests, partially-static data are built partly from statically-known values, and partly from unknown dynamic values, and typically support some form of computation on the static portions.

Figure 12 defines a basic interface to partially-static data. The `PS` interface exposes two types: `t`, the partially-static type, and `sta`, the fully-static type. The functions `dyn` and `cd` convert back and forth between partially-static and fully-dynamic values; the function `now` attempts to extract a fully-static value from a partially-static one. Instances of `PS` typically expose additional constructors for injecting static data into `t`, along with operations for computing with partially-static values.

There are many useful instances of `PS`. The binding-times of `listify` may be improved by defining a type of partially-static lists, whose elements are dynamic, and whose spines may be partly static and partly dynamic (Figure 13). The result of appending two such partially-static lists, `l <+> r`, results in static reduction in the case where `l` has a static suffix and `r` a static prefix (Figure 14). Furthermore, the static value `[]` acts as a left and right identity for append, which is needed to simplify the code from the end of Section 4.1.

Partially-static data and algebraic structure. The discussion above suggests a connection between partially-static data and algebraic structure that often proves useful. Several other generic schemes

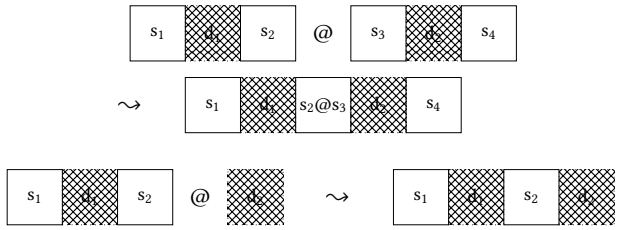


Fig. 13. Partially-static lists: adjacent static parts merge

```
type 'a ps_list =
     Empty
   | SCons of 'a code list * 'a ps_list
   | DCons of 'a list code * 'a ps_list

let rec (<+>) l r = match l with
 | Empty → r
 | SCons (s, tl) → (match tl <+> r with
                  | SCons (u, tl) → SCons (s @ u, tl)
                  | Empty | DCons _ as r → SCons (s, r))
 | DCons (s, tl) → (match tl <+> r with
                  | DCons (u,tl) → DCons (.<.~s @ .~u>.,tl)
                  | Empty | SCons _ as r → DCons (s, r))
```

Fig. 14. Partially-static lists with append

benefit from partially-static data designed to take advantage of the algebraic laws of the underlying structure: as with partially-static lists, the string monoid proves useful when defining the generic pretty-printer `gshow`; commutative monoids appear when defining `gsize`; tropical semirings (formed from addition and `max`) arise in the definition of `gdepth` (Section 5).

Incorporating partially-static data into the staged SYB implementation involves a few changes. First, the original flexibility in the return type of `genericQ` is restored; a generic query no longer unconditionally returns dynamic values:

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

Second, `gfixQ` acquires a `PS` constraint, requiring that the return type of a query satisfies the partially-static interface:

```
val gfixQ : {P:PS} → (P.t genericQ → P.t genericQ) → P.t genericQ
```

In practice the constraint is no hardship, since partially-static data subsume the old fully-dynamic data. The body of `gfixQ` must be updated with `dyn` and `cd` accordingly. Finally, schemes such as `listify` must be written in terms of partially-static data; in practice this means replacing `list` operations such as `concat` with corresponding operations for `ps_list` (Figure 14).

With these changes the superfluous appends disappear from generated code. Here is the updated output of `listify evenp` at the type `(int * string) list`:

```
let rec evenslist x = match x with
 | [] → []
 | h::t → let (i, s) = h in
           (if evenp i then [i] else []) @ evenslist t
```

```
1  module type rec DATA =
2  sig
3    type t and t_
4    module Typeable : TYPEABLE with type t = t
5    val gmapQ  : 'u genericQ  → t_ → 'u list
6    val reify : t code → (t_ → 'a code) → 'a code
7  end
8
9  type ('a, 'r) list_ =
10       Nil
11     | Cons of 'a code * 'r
12 implicit module rec Data_list {A: DATA} =
13 struct
14   type t = A.t list and t_ = (A.t, t code) list_
15   let gmapQ q l = match l with
16                   | Nil → []
17                   | Cons (h, t) → [q h; q t]
18
19   let reify c k = .< match .~c with
20                     | [] → .~(k Nil)
21                     | h :: t → .~(k (Cons (.<h>., .<t>.)))>.
22 end
```

Fig. 15. Extended `DATA` with `reify`

4.3 Reification (and Reflection)

The staged `gmapQ` (Figure 2b, lines 17–19) combines two operations: it builds a code value that examines a datum, and it applies a generic query `q` to the sub-values of that datum. These operations need not always occur together; in fact, it is useful to decompose `gmapQ` into its constituent parts.

Figure 15 extends the `DATA` interface with a function `reify` that serves the first function, and a type `t_` that describes the input to the second argument of `reify`. More precisely, `t_` represents the *one-step unrolling* of `t`, with a constructor corresponding to each constructor of `t`, but with dynamic rather than static arguments. For example, the type `A.t list` (Figure 15, line 10) has a constructor `::` (pronounced "cons") with arguments of type `A.t` and `A.t list`. Accordingly, the type `(A.t, t code) list_` (also Figure 15, line 10) has a constructor `Cons` with arguments of type `A.t code` and `A.t list code`.

With the addition of `reify`, the body of `gmapQ` is identical to the unstaged version, except for the names of the constructors. (However, the types differ.)

The "continuation" argument to `reify` offers considerable flexibility. For example, `reify c (gmapQ q)` behaves like the staged `gmapQ` from Figure 2b, while `reify c id`, where `id` is the identity function, is a function that can be used as the `dyn` injection to treat a `DATA` instance as partially static. The `reflect` function (not shown) is an inverse to `reify`, satisfying `reify c reflect` ≡ `c` up to observational equivalence. While `reify` is useful for analysing values described by `DATA`, `reflect` is useful for constructing values. A staged version of the generic traversal `gmapT`, which rebuilds a term after transforming its sub-values, may be conveniently expressed by pre-composing `reify` and post-composing `reflect`.

The sections that follow give further uses for `reify`, which provides a basis for code motion (Section 4.4) and for statically exploring and eliminating dynamic branches (Section 4.5, Section 4.6).

```
κ[dyn                                    dyn
   .< match x with                         .< match x with
    | [] → .~(cd e1)          ⇝             | [] → .~(cd κ[e1])
    | h::t → .~(cd e2)>.]                   | h::t → .~(cd κ[e2])>.
```

Fig. 16. Lifting `match` across continuation frames

```
val case_locus: {P:PS} → (unit → P.t) → P.t
val reify: {P:PS} → {D:DATA} → D.t code → (D.t_ → P.t) → P.t
```

Fig. 17. case insertion, with partially-static data

`reify`*, eta, and The Trick.* The one-step dynamic unrolling that provides access to the top-level static structure of a dynamic value corresponds directly to a well-known binding-time improvement known in the partial evaluation community as The Trick [Jones 1995]. From another perspective, the `reify` function for a `DATA` instance simply performs the appropriate `eta` expansion for the type `t`, enabling a more favourable binding-time analysis [Danvy et al. 1995].

4.4 Case Lifting

The `reify` function (Section 4.3) exposes the top-level structure of data to allow static computation with dynamic values. However, the inserted `match` expressions form a boundary between the static and dynamic parts of the program. For example, consider the code generated for `listify evenp` at the type `int * int`:

```
.< let (x, y) = p in
    if evenp .~x then .~(cd [x]) else .~(cd [])
  @ if evenp .~y then .~(cd [y]) else .~(cd [])>.
```

There are four possible outcomes for the two `evenp` tests, and for each outcome the length of the result list is statically known. However, there is no static context that encloses enough data to build any of the lists statically.

Transforming the program to lift one of the `if` expressions to the highest-possible point, just below where `x` and `y` are bound, exposes more opportunities for optimization:

```
.< let (x, y) = p in
   if evenp .~x then
    .~(.< if evenp .~y then .~(cd ([x] @ [y])) else .~(cd [x])>.)
    .~(.< if evenp .~y then .~(cd [y]) else .~(cd [])>.) >.
```

This is not an anomalous example; the *case lifting* transformation increases the scope of static variables, which is generally a binding-time improvement. Figure 16 shows the interaction with contexts in the general case: the context κ will not further reduce the fully-dynamic `match` expression, but lifting the `match` across the context plugs in the possibly-static values e_1 and e_2, possibly exposing new redexes.

Interface and implementation. Figure 17 gives an interface for case lifting which resembles the `let`-insertion interface of Figure 4a. The `case_locus` function, analogous to `let_locus`, marks a place where bindings may be inserted. The reify function is a generic version of the `reify` member of `DATA` (Figure 15); it interacts with the `reify` member through the continuation argument, hoisting the binding generated by `reify` to a `case_locus` point and interposing the context κ to shuffle the

```
let f x = match x with
  Left x → []
| Right y → []
```

a. before pruning

```
let f x = []
```

b. after pruning

Fig. 18. Pruning `listify evenp` at `(bool,string) either`

```
let rec r l = match l with
   [] → []
 | h::t → r t
```

a. before recursive pruning

```
let r l = []
```

b. after recursive pruning

Fig. 19. Pruning `listify evenp` at `bool list`

continuation frames as depicted in Figure 16. Since κ is duplicated in each branch the implementation makes use of multi-shot delimited control.

Case lifting elsewhere. While `let`-insertion is a common feature of staged programs, full case lifting appears much rarer in practice, although less powerful forms of `if` insertion are occasionally seen [Kameyama et al. 2011, 2014]. One possible reason is that duplicating continuations is often unwise; while in our setting it is commonly the case that static computation will substantially reduce the resulting code, that may not be true elsewhere. Second, while `let` insertion is polymorphic in the type of the expression, case lifting is data-specific. The `DATA` constraint provides a convenient basis for lifting arbitrary `match` expressions; without `DATA`, case lifting would have to be defined separately for every type.

Case lifting for binary sums is, however, found in the broader literature relating to normalization by evaluation and type-directed partial evaluation [Balat et al. 2004; Lindley 2007].

4.5 Branch Pruning

Here is a definition of binary sums in OCaml:

```
type ('a, 'b) either =
    Left of 'a
  | Right of 'b
```

The generic function `listify evenp`, applied to a value `v` of type `(a, b) either`, must first determine whether `v` is `Left x` or `Right y` and then return a list of even integers found in `x` or `y`. If both `x` and `y` are types such as `bool` or `string` that cannot contain integers, then the list will be empty in both cases (Figure 18a).

In such cases, where it is statically apparent that all branches of a `match` are equal and independent of the variables bound by the match, the `match` may be eliminated altogether, and replaced with the value of the branch (Figure 18b) (provided, of course, that the scrutinee is free of effects!).

This is an instance of eta *contraction*, which applies to all algebraic data types, not just to simple binary sums.

This branch pruning optimization enjoys two favourable interactions with inlining (Section 4.1) and partially-static data (Section 4.2). First, reducing partially static data makes it more likely that the branches of the match will produce evidently-equal values. Second, branch pruning eliminates the dynamic matches introduced by reification (Section 4.3), making their static results available for further computation.

The eta rule therefore appears twice in converting a fully dynamic value to a fully static value: first, reification statically exposes the top-level structure of a dynamic value. Here is an example, with a dynamic value `.<v>.` of type `(string, bool) either`

```
listify evenp .<v>.
```

The effect of reification is to eta-expand `.<v>.`:

```
dyn .< match v with Left x → .~(listify evenp .<x>.)
                  | Right y → .~(listify evenp .<y>.) >.
```

Next, computation with partially-static data simplifies the branches:

```
dyn .< match v with Left x → .~(sta [])
                  | Right y → .~(sta []) >.
```

Finally, the branches are determined to be equal, and the result is eta reduced again, leaving only a static value.

```
sta []
```

Implementation. A natural implementation of pruning is a wrapper around `reify` that passes into `reify` a continuation that accumulates each value `k v`. When all the values have been accumulated they are compared for equality; if they can be determined to be static and equal then the whole expression built by `reify` is replaced with the static value.

4.6 Recursive Branch Elimination

The code in Figure 19(a), generated by the instantiation of `listify evenp` at the type `bool list`, is an example of an unnecessary branch that is not eliminated by the branch pruning optimization of Section 4.5.

The simplicity of the code is a consequence of the optimizations introduced so far: inlining brought the instantiation of `listify` for `bool` into the body of `r`; reification expanded the value to examine the `true` and `false` cases; branch pruning eliminated the `match`, having determined that both branches were static empty lists, and partially-static data eliminated the resulting append of an empty list.

However, the recursive data type (`list`), which led to the generation of the recursive function, prevented further optimizations from taking place. The difficulty arises because the recursive call `r t` is not static, and so the approach in Section 4.5 is not sufficiently powerful to detect that the code can be simplified. Nevertheless, it is both evident that the function always returns the empty list, and important that the code should be simplified, since the list it traverses may be arbitrarily long.

How might Section 4.5 be generalized to the recursive case? The solution is to determine a static fixed point. Starting from the assumption that `r` always returns the empty list, every recursive call to `r` is replaced with the static empty list value. If the branches are then all statically equal to the empty list then it is legitimate to eliminate the whole recursion (Figure 19(b)). This approach naturally generalises both to mutual recursion, where the starting assumption is that all the functions in a recursive binding group return zero, and to monoids other than lists. (Strictly speaking, it is also necessary to check that the recursion is well-founded, or we will make the embarrassing optimization of replacing a non-terminating loop with a constant. However, it is likely that all recursion in the SYB setting is well-founded.)

```
let rec gshow {D:DATA} v =
  show_constructor (constructor v) (gmapQ gshow v)
```

Fig. 20a. gshow, unstaged

```
let gshow = gfixQ (fun f {D:DATA} v →
                     (show_constructor (constructor v) (gmapQ f v)))
```

Fig. 20b. gshow, staged

```
let gshow_list_bool =
  let sl = ref dummy in let sb = ref dummy in
  let () = sb := fun b → apply_constructor (string_of_bool b) [] in
  let () = sl := fun l → apply_constructor
                          (match l with [] → "[]" | _::_ → "::")
                          (match l with [] → []
                                      | h::t → [!sb h; !sl t])
  in fun x → !sl x
```

Fig. 20c. gshow, naively staged: generated code (instantiated at `bool list`, slightly simplified)

```
let rec r l = match l with
  | [] → "[]"
  | h::t → if h then "(true :: "^ r t ^")"
           else "(false :: "^ r t ^")"
```

Fig. 20d. gshow, carefully staged: generated code (instantiated at `bool list`)

Implementation. The implementation is broadly similar to the implementation of branch pruning (Section 4.5): the $\texttt{gfixQ}_{\texttt{Mon}}$ combinator, an extension of `gfixQ` with an additional `MONOID` constraint (ensuring that the result type has a zero element) interacts with `reify` to extract the results of each branch in an environment where all functions in the local recursive group return zero. If the results of every branch are zero for every function in the group then (under the well-foundedness assumption mentioned above) the functions in the group may be replaced with constant zero functions and inlined in any bindings subsequently generated.

Here is a simple example: when `listify evenp` is instantiated with argument type `int * (bool list)`, the fixed-point calculation determines that traversing the second component of the pair always returns an empty list and generates the following efficient code:

```
let evenspair p =
   let (i, l) = p in
   if evenp i then [i] else []
```

5 EVALUATION: APPLICABILITY

We conclude the technical development by evaluating the more carefully staged library with a variety of examples. This section shows the dramatic improvements to the generated code, which becomes simpler, shorter, and more obviously correct. The following section (6) shows the consequent positive impact on performance.

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

Fig. 21a. `gsize`, unstaged

```
let gsize = gfixQ (fun self {D:DATA} v →
                     sta 1 <+> fold_left (<+>) zero (gmapQ self v))
```

Fig. 21b. `gsize`, staged

```
let gsize_list =
  let sl = ref dummy in let sp = ref dummy in
  let ss = ref dummy in let si = ref dummy in
  let sb = ref dummy in
  let () = si := fun y → 1 + sum [] in
  let () = sb := fun y → 1 + sum [] in
  let () = ss := fun y →
                   1 + (sum (match y with
                             | Left x → [!sb x]
                             | Right y → [!si y])) in
  let () = sp := fun y →
                   1 + (sum (let (x,y) = y in [!sb x; !ss y])) in
  let () = sl := fun y →
                   1 + (sum (match y with
                             | [] → []
                             | h::t → [!sp h; !sl t])) in
  fun x → !sl x
```

Fig. 21c. `gsize`, naively staged: generated code
(instantiated at `(int*(int,string) either) list`)

```
let rec r l = match l with
  | [] → 1
  | h::t → 5 + r t
```

Fig. 21d. `gsize`, carefully staged: generated code
(instantiated at `(int*(int,string) either) list`)

`gshow`. Figures 20a and 20b show unstaged and staged implementations of the `gshow` function, defined in terms of the SYB functions `constructor` and `gmapQ`, and an auxiliary function `show_constructor`. The changes needed for staging are minimal: besides the custom fixed point operator `gfixQ`, the implementations are identical.

Figures 20c and 20d show the code generated by the naive and carefully staged libraries when `gshow` is instantiated at argument type `bool list`.

Several of the optimisations developed in Section 4 improve the generated code. Recursion analysis and inlining have reduced the code to a single recursive function (Section 4.1), as is also the case in the examples that follow. Since `gshow` inspects `v` twice, case lifting (Section 4.4) avoids the multiple dynamic tests (`match` expressions). The strings built by `gshow`, like the lists built by `listify`, are partially-static (Section 4.2), enabling the static concatenation of strings in the generated code: the boolean constructor names appear in the same string literals as the infix `cons` constructor.

```
let gtypecount {X:TYPEABLE} x = gcount (mkQ false (fun _ → true))
```

Fig. 22a. gtypecount, unstaged

```
let gtypecount {X:TYPEABLE} x = gcount (mkQ ff (fun _ → tt))
```

Fig. 22b. gtypecount, staged

```
let rec crush u = function
 | [] → u
 | h::t → crush (u + h) t in
let tl = ref dummy in let te = ref dummy in
let tp = ref dummy in let tb = ref dummy in
let ti = ref dummy in
let () = tb := fun y → crush (if false then 1 else 0) [] in
let () = ti := fun y → crush (if true then 1 else 0) [] in
let () = tp := fun p → crush (if false then 1 else 0)
                  (let (a,b) = p in [!ti a; !tb b]) in
let () = te := fun y →
                crush (if false then 1 else 0)
                 (match y with Left x → [!ti x]
                             | Right y → [!tp y]) in
let () = tl := fun y →
                crush (if false then 1 else 0)
                 (match y with [] → []
                               | h::t → [!te h; !tl t]) in
fun x → !tl x
```

Fig. 22c. gtypecount, naively staged: generated code
(instantiated at `(int, int * bool) either list`)

```
let rec r l = match l with
  | [] → 0
  | h::t → 1 + r t
```

Fig. 22d. gtypecount, carefully staged: generated code
(instantiated at `(int, int * bool) either list`)

`gsize`. Figures 21a and 21b show unstaged and staged implementations of the `gsize` function, which computes the size of a value as the successor of the sum of the sizes of its sub-values. Partially-static data changes the code slightly from the unstaged version: arithmetic with `<+>` and `zero` replaces arithmetic with standard integers. The larger structure of the code is unchanged.

Figures 21c and 21d show the generated code for the naive and carefully staged libraries when `gsize` is instantiated with argument type `(int*(int,string) either) list`. Once again, various optimizations from the preceding pages significantly simplify the generated code .

Since `int` and `string` both have size 1, a combination of partially-static data (Section 4.2) and branch pruning (Section 4.5) reduces the computation of `gsize` at type `(int,string) either` to `3`, and `int*((int,string) either)` to `5`.

gtypecount. Figures 22a and 22b show staged and unstaged implementations of a function `gtypecount` in terms of another generic scheme, `gcount`. Since `gtypecount` is not defined recursively

```
let rec gdepth {D: DATA} x = succ (maximum (gmapQ gdepth x))
```

Fig. 23a. gdepth, unstaged

```
let gdepth_ = gfixQ (fun self {D:DATA} x →
                     sta 1 <+> fold_left max zero (gmapQ self x))
```

Fig. 23b. gdepth, staged

```
let dp = ref dummy in let di = ref dummy in
let tq = ref dummy in let dl = ref dummy in
let () = di := fun y → succ (maximum []) in
let () = dl := fun y → succ (maximum (match y with
                                      | [] → []
                                      | h::t → [!di h; !dl t])) in
let () = tq := fun y → succ (maximum (let (a,b) = y in
                                      [!dl a; !di b])) in
let () = dp := fun y → succ (maximum (let (a,b) = y in
                                      [!tq a; !di b])) in
fun x  → !dp x
```

Fig. 23c. gdepth, naively staged: generated code
(instantiated at `((int list * int) * int) → int`)

```
let rec r x = match x with
  | [] → 1
  | h::t → 1 + max 1 (r t)
let f p = let (x,y) = p in
          let (a,b) = x in
            2 + max 1 (r a)
```

Fig. 23d. gdepth, carefully staged: generated code
(instantiated at `((int list * int) * int) → int`)

there is no need for custom fixed points, and so the two implementations are identical except for the use of partially-static booleans `ff` and `tt` in place of `false` and `true`.

Figures 22c and 22d show the output for `gtypecount` when it is instantiated to count the number of `int` values in a value of type `(int, int * bool) either list`.

As with `gsize`, partially-static data (Section 4.2) and branch pruning (Section 4.5) have significantly simplified the body of the generated function (Figure 22d), so that it simply adds a known integer for each element in the list, having determined that a value of type `(int, int * bool) either` always contains exactly one `int`.

gdepth. Finally, Figures 23a and 23b show unstaged and staged versions of the generic function `gdepth`, a generic function for computing the longest path from a value to one of its leaves. The implementations are similar except for the use of a custom fixed point `gfixQ` and partially static data (`<+>`, `zero`, `max`) in the staged version.

The implementation of `gdepth` is similar to `gsize`, except that the results of traversing the subvalues are combined with `max`, not with addition; however, both addition and `max` are needed, and the partially static data form a tropical semiring.

Figures 23c and 23d show the code generated when `gdepth` is instantiated with argument type `((int list * int) * int)`.

As the figures show, the advanced staging techniques of Section 4 have significantly improved the output. Recursion analysis has determined that the generated function `r` for traversing `int list` values should be recursive, and that the code that follows should be non-recursive. Examining the code for `r` reveals a remaining opportunity for improvement: since `r` always returns at least `1`, the call to `max` in the cons branch is unnecessary. However, the simple fixed-point analysis in Section 4.6 is not sufficiently powerful enough to detect the redundancy.

6 EVALUATION: PERFORMANCE

The code generated by the improved staging library is evidently clearer, shorter, and simpler than the code generated by the naive staging. We now confirm that it also performs better.

Figure 24 compares the performance on five representative benchmarks of the unstaged SYB (Section 2), the naively-staged version (Section 3), the more carefully staged version (Section 4), and a hand-written version. The first three benchmarks, `Map`, `SelectInt`, and `RMWeights`, are the same as those used to evaluate the naive staging in Section 3.3. The two additional benchmarks, `Size` and `Show`, are introduced in this work; they are drawn from Section 5, and illustrate how the more sophisticated optimizations of Section 4 improve the performance of traversals that do not simply visit each node in a data structure.

All measurements, both for these benchmarks and those in Section 3.3, were made using the `4.02.1+modular-implicits-ber` fork of MetaOCaml, which is the most recent available version with support for implicits. The benchmarks were run on an AMD FX 8320 machine with 16GB of memory running Debian Linux. With the exception of the `Map` benchmark, whose times have 95% confidence intervals within ±5%, all timings have 95% C.I. within ±2%. The measurements were taken using *core-bench*, a sophisticated micro-benchmarking library that accounts for garbage collection overheads and automatically computes the number of runs needed to eliminate the cost of the timing function from the measurements [Hardin and James 2013]. Each measurement reported in Figures 9 and 24 is thus computed by *core-bench* from thousands of runs of the specified function.

The hand-written code for each benchmark is written in idiomatic functional style, prioritizing modularity over low-level performance tricks. For example, code for the `Map` benchmark first defines a function `mapTree`, which is then applied to the successor function (rather than, say, inlining `succ` within `mapTree`):

```
let rec mapTree f t = match t with
  | Leaf → Leaf
  | Bin (v, l, r) → Bin (f v, mapTree f l, mapTree f r)
```

Similarly, the hand-written code for the printing benchmark `Show` is written using a combinator per type constructor (cf. e.g. work described by Kennedy [2004]; Yallop [2007]) rather than fusing the code together as in Figure 20d. Since the output of a multi-stage program is OCaml code, it is always possible in principle, but rarely advisable in practice, to manually write identical code to the output of a staged library, and so achieve identical performance.

The performance of `Map` with the carefully staged library is almost 20× faster than the unstaged generic version, slightly improved over the naively staged version, and a little faster than the handwritten code, apparently because of the inlining of the successor function by the library.

The improvement in `SelectInt` is more dramatic: it is over 22× as fast as the unstaged generic version, over twice as fast as the naively-staged version, and there is no remaining overhead compared to handwritten code.

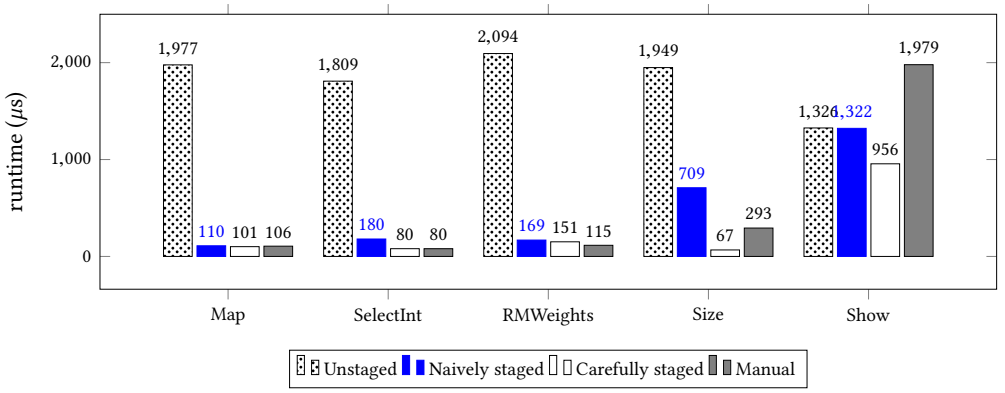


Fig. 24. Enhanced staged SYB Performance

The `RMWeights` benchmark shows improvements over the naive version; there is still a little residual overhead compared to the handwritten version.

The results of the two new benchmarks are more notable. For `Size`, the carefully staged version is almost 30× as fast as the unstaged version; more remarkably, it is over 4× as fast as hand-written code, due to the fusing together of generated functions and shifting of arithmetic work to compile-time (Figure 21d).

The carefully staged version of `Show` is also faster than handwritten code, but there is an additional surprise: the unstaged generic version also beats the handwritten entry! Examining the handwritten code for printing lists uncovers the reason: the string concatenation operator is right associative in OCaml, and so the naive implementation copies the long strings on the right, generated by `show_list`, many times.

```
let rec show_list f l = match l with
    [] → "[]"
  | h::t → "("^ f h ^" :: " ^ show_list f t ^")"
```

Parenthesizing to avoid right nesting brings the running time down from 1979μs to 1022μs, almost as fast as the generated code that fuses together the printers for lists and bools (Figure 20d).

7 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 hand-written code by combining these transformations with an additional optimisation whose effects are similar to the selective traversal optimisation described in Section 4.5 and Section 4.6.

The work of Adams et al. [2015] (and the earlier version, [Adams et al. 2014]) are comparable to the earlier attempt to stage SYB described in Section 3 [Yallop 2016]. 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 are impressive, bringing the performance of the SYB benchmarks in Section 3.3 in line with handwritten code. (However, the additional benchmarks introduced in Section 6, which do not visit every node, have no direct counterpart in the work of Adams et al. [2015], and the critical optimizations that eliminate unnecessary traversals (Sections 4.2 and 4.6) are not supported by their implementation.)

The work described in this paper improves on the work of Adams et al. in several ways. First, as the examples in this paper demonstrate, focusing on values of "undesirable" type is not always sufficient to achieve reasonable performance; in particular, it does not help with avoiding fruitless traversals of sub-values, such as searching for integers within the list of strings in the `listify` example of Section 1. Second, 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 staged SYB library — and so the behaviour of our implementation is not vulnerable to changes in the details of optimization passes or other internal compiler issues. As Adams et al. [2015] note, the success of their optimizations depends critically on the details of GHC's inlining behaviour, and so optimizations that are performed successfully with one version of GHC are found to fail with a later version. Finally, MetaOCaml's type system justifies a degree of confidence in the correctness of the staged code that is not available in a compiler pass. The types of the staged SYB library are fully integrated with the rest of the program; in contrast, it is easy in a compiler optimization pass to inadvertently generate ill-typed code.

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.

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.

Finally, there is an increasing body of evidence that staging can significantly improve the performance of elegant but inefficient libraries such as SYB. Two recent examples are given by Jonnalagedda et al. [2014], who present a staging transformation of high-level parser combinators using Scala's Lightweight Modular Staging [Rompf and Odersky 2010], and Kiselyov et al. [2017], who use staging techniques to implement a stream library with a high-level interface that generates low-level code with strong performance guarantees. The latter paper implements equivalent staging transformations in two languages (Scala LMS and MetaOCaml); we expect the techniques developed in the present paper to be similarly transferable to a variety of systems, including LMS and the forthcoming Typed Template Haskell [Peyton Jones 2016], which adds MetaOCaml-style typed quotations to the currently untyped Template Haskell system.

8 CONCLUSION

We have shown how to apply existing and novel multi-stage programming techniques to transform a popular generic programming library into an optimising code generator. Our staging of SYB combines the following attributes:

Efficient: while generic programming libraries often suffer from poor performance, the output of the staged library is comparable to hand-written code.

Incremental: the staging is decomposed into a series of local changes with virtuous interactions, maintaining the original structure of the library.

Type-safe: MetaOCaml's type safety properties ensure that the staged library never generates ill-typed code.

Reusable: the staging techniques presented here deal with the core elements of functional programming: algebraic data, higher-order functions, recursion, etc. We anticipate that these techniques will apply to a wide class of programs.

REFERENCES

Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. 2014. Optimizing SYB is Easy!. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM '14)*. ACM, New York, NY, USA, 71–82.

Michael D. Adams, Andrew Farmer, and José Pedro Magalhães. 2015. Optimizing SYB traversals is easy! *Science of Computer Programming* 112, Part 2 (2015), 170 – 193. https://doi.org/10.1016/j.scico.2015.09.003 Selected and extended papers from Partial Evaluation and Program Manipulation 2014.

Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. 2004. Extensional Normalisation and Type-directed Partial Evaluation for Typed Lambda Calculus with Sums. In *POPL '04*. ACM, New York, NY, USA, 64–76. https://doi.org/10.1145/964001.964007

Andrej Bauer and Matija Pretnar. 2012. Programming with Algebraic Effects and Handlers. *CoRR* abs/1203.1539 (2012). http://arxiv.org/abs/1203.1539

Anders Bondorf. 1992. Improving Binding Times Without Explicit CPS-conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/141471.141483

Dmitri Boulytchev and Sergey Mechtaev. 2011. Efficiently scrapping boilerplate code in OCaml. (September 2011). ACM Workshop on ML 2011.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543.

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1995. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation* 8, 3 (1995), 209–227. https://doi.org/10.1007/BF01019004

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (Nov. 1996), 730–751.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. (September 2015). OCaml Users and Developers Workshop 2015.

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. 2012. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2364506.2364508

Andrzej Filinski. 2001. Normalization by Evaluation for the Computational Lambda-calculus *(TLCA'01)*. Springer-Verlag, Berlin, Heidelberg, 151–165. http://dl.acm.org/citation.cfm?id=1754621.1754638

Christopher S. Hardin and Roshan P. James. 2013. Core bench: micro-benchmarking for OCaml. OCaml Users and Developers Workshop. (September 2013).

John Hughes. 1999. *A Type Specialisation Tutorial.* Springer Berlin Heidelberg, Berlin, Heidelberg, 293–325. https://doi.org/10.1007/3-540-47018-2_12

Jun Inoue. 2014. Supercompilation via staging. In *Fourth International Valentin Turchin Workshop on Metacomputation*.

Patricia Johann and Neil Ghani. 2008. Foundations for structured programming with GADTs *(POPL 2008)*. ACM.

Mark P. Jones. 1995. Dictionary-free Overloading by Partial Evaluation. *Lisp Symb. Comput.* 8, 3 (Sept. 1995), 229–248.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming*

Systems Languages & Applications (OOPSLA '14). ACM, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

David Kaloper-Meršinjak and Jeremy Yallop. 2016. Generic Partially-static Data (Extended Abstract). In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, New York, NY, USA, 39–40. https://doi.org/10.1145/2976022.2976028

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011. Shifting the Stage: Staging with Delimited Control. *J. Funct. Program.* 21, 6 (Nov. 2011), 617–662.

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2014. Combinators for Impure Yet Hygienic Code Generation. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM '14)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2543728.2543740

Andrew J. Kennedy. 2004. Functional Pearl: Pickler Combinators. *Journal of Functional Programming* 14, 6 (November 2004).

Oleg Kiselyov. 2012. Delimited Control in OCaml, Abstractly and Concretely. *Theor. Comput. Sci.* 435 (June 2012), 56–76.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Lecture Notes in Computer Science, Vol. 8475. Springer International Publishing, 86–102.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *POPL 2017*. ACM.

Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. 2004. A Methodology for Generating Verified Combinatorial Circuits. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04)*. ACM, New York, NY, USA, 249–258. https://doi.org/10.1145/1017753.1017794

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming *(TLDI '03)*. ACM, New York, NY, USA, 26–37.

Ralf Lämmel and Simon Peyton Jones. 2004. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP '04)*. ACM, New York, NY, USA, 244–255.

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:http://comjnl.oxfordjournals.org/content/6/4/308.full.pdf+html

J. Launchbury. 1991. *Project Factorisations in Partial Evaluation*. Cambridge University Press. https://books.google.co.uk/books?id=B1UTK2j8rksC

Julia L. Lawall and Olivier Danvy. 1994. Continuation-based Partial Evaluation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP '94)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/182409.182483

Xavier Leroy. 2003. A proposal for recursive modules in Objective Caml. INRIA Rocquencourt. (May 2003). Version 1.1.

Sam Lindley. 2007. Extensional Rewriting with Sums. In *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA'07)*. Springer-Verlag, Berlin, Heidelberg, 255–271. http://dl.acm.org/citation.cfm?id=1770203.1770222

Andres Löh and Ralf Hinze. 2006. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*. ACM, New York, NY, USA, 133–144.

José Pedro Magalhães. 2013. 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)*.

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

Kristian Nielsen and Morten Heine Sørensen. 1995. Call-By-Name CPS-Translation As a Binding-Time Improvement. In *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*. Springer-Verlag, London, UK, UK, 296–313. http://dl.acm.org/citation.cfm?id=647163.717677

Simon Peyton Jones. 2016. Template Haskell, 14 years on. Talk given at the International Summer School on Metaprogramming, Cambridge, UK. (August 2016). https://www.cl.cam.ac.uk/events/metaprog2016/Template-Haskell-Aug16.pptx.

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.* 12, 5 (July 2002), 393–434.

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

Tim Sheard and Iavor S. Diatchki. 2002. Staging Algebraic Datatypes. Unpublished manuscript. (2002). http://web.cecs.pdx.edu/~sheard/papers/stagedData.ps.

Walid Mohamed Taha. 1999. *Multistage Programming: Its Theory and Applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology. AAI9949870.

The GHC Team. 2015. *The Glorious Glasgow Haskell Compilation System User's Guide* (7.10.2 ed.).

Peter Thiemann. 2013. Partially Static Operations *(PEPM '13)*. ACM, New York, NY, USA, 75–76. https://doi.org/10.1145/2426890.2426906

Todd L. Veldhuizen. 2004. *Active Libraries and Universal Languages.* Ph.D. Dissertation. Indiana University Computer Science.

Leo White, Frédéric Bour, and Jeremy Yallop. 2015. Modular Implicits. ACM Workshop on ML 2014 post-proceedings. (September 2015).

Jeremy Yallop. 2007. Practical Generic Programming in OCaml. In *ACM SIGPLAN Workshop on ML*, Derek Dreyer (Ed.). Freiburg, Germany.

Jeremy Yallop. 2016. Staging Generic Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16).* ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/2847538.2847546