Supercompiling with Staging

Jun Inoue

INRIA Paris-Rocquencourt
DI École Normale Supérieure, Paris
Jun.Inoue@inria.fr

Abstract. Supercompilation is a powerful program optimization framework which Sørensen et al. showed to subsume, and exceed, partial evaluation and deforestation. Its main strength is that it optimizes a conditional branch by assuming the branch's guard tested true, and that it can propagate this information to data that are not directly examined in the guard. We show that both of these features can be mimicked in multi-stage programming, a code generation framework, by modifying metadata attached to generated code in-place. This allows for explicit, programmer-controlled supercompilation with well-defined semantics as to where, how, and whether a program is optimized. Our results show that staging can go beyond partial evaluation, with which it originated, and is also useful for writing libraries in high-level style where failing to optimize away the overheads is unacceptable.

Keywords: Supercompilation, Multi-Stage Programming, Functional Programming

1 Introduction

Supercompilation is a powerful metacomputation framework known to subsume other systems like deforestation and partial evaluation [13]. A key benefit of such frameworks is to enable the use of abstractions without runtime penalties. For example, functional programs often split a loop into a function that produces a stream of items and another function that performs work on each item. This abstraction with streams greatly improves modularity, at the cost of more allocation and time spent inspecting the stream. Supercompilation can eliminate the stream, resolving the tension between abstraction and performance.

Supercompilation is usually studied as a fully automatic optimization, but this approach has pros and cons. In exchange for the convenience of automation, programmers lose control over when and how optimization happens, and it can be difficult to tell whether supercompilation will eliminate a specific abstraction used in a given source program. This can be problematic if failing to optimize is unacceptable, as in embedded systems and high-performance computing.

Multi-stage programming (MSP) [17] has evolved as a tool to solve similar problems with automation in the context of partial evaluation (PE). For instance, the MSP language MetaOCaml can optimize the power function as follows.

```
let rec power n x = if n = 1 then x else x * power (n-1) x
let rec genpow n x = if n = 1 then x else .<.~x * .~(genpow (n-1) x)>.
let rec stpow n = !. .<fun x → .~(genpow n .<x>.)>.
```

`power` computes x^n, while `stpow` generates loop-unrolled `power` for concrete values of `n` using MetaOCaml's three *staging constructs*. Brackets `.<`e`>.` delay the expression e. An escape `.~`e must occur inside brackets and instructs e to be evaluated without delay. The result must be of the form `.<`e'`>.`, and e' replaces `.~`e. Run `!.`e compiles and runs the delayed expression returned by e. These constructs are like LISP's quasiquote, unquote, and eval but are hygienic, i.e. preserves static scope [1]. In this example, `genpow n .<x>.` generates the code `.<x*x*...x>.` with `n` copies of `x`, while `stpow` places that inside a binder to get `.<fun x → x*x*...x>.` and compiles it by `!.`. It is evident from the source code that `genpow` completely unrolls the recursion seen in `power` and produces code containing only `*`, because that's the only operation occurring inside brackets.

In this paper, we bring this kind of explicit programmer control to supercompilation through MSP techniques that mimic positive supercompilation [12,13]. Most importantly, we express the introduction and propagation of assumptions under conditionals that Sørensen et al. [13] identified as the key improvements that supercompilation makes over PE and deforestation. For example, in

```
let rec contrived zs =
  let f xs ys = match xs with [] → length xs + length ys
                            | w::ws → length xs
  in f zs (1::zs)
and length ws = match ws with [] → 0
                            | _::ws → 1 + length ws
```

positive supercompilation optimizes the first branch of the `match` by *assuming* `xs = []` and simplifying the branch body, which gives `0 + length ys`. Moreover, noting `ys` shares a substructure with `xs`, it *propagates* the assumption `xs = []` to `ys = [1]`, optimizing the whole branch to just `1`. By pattern-matching on `xs`, we learn something about `ys`, and the supercompiler tracks this knowledge.

In our technique of *delimited assumptions*, we manipulate not raw code values like `.<x>.` in the power example, but a wrapper that attaches information about the value `x` must have. We update this metadata when we generate a branch of a conditional to reflect any new assumptions. We modify the metadata in place to propagate the change to all copies of the data. This modification is undone by a dynamic wind when that branch is left, giving the assumption a (dynamic) scope delimited by the conditional. We show in this paper how this technique, combined with auxiliary techniques for ensuring termination of the generator, can achieve a great deal of the effects of supercompilation.

1.1 Contributions

We will use the `contrived` function above as a running example to illustrate the main techniques. Specifically, after reviewing in more detail how the positive supercompiler works (Section 2):

– We introduce the technique of delimited assumptions, which combines partially static data [11] with judicious uses of mutation to achieve the introduction and propagation of assumptions explained above (Section 3).
– We show memoization techniques for ensuring termination of the generator, explaining unique challenges posed by supercompilation (Section 4). Briefly, conditionals are converted to functions and invoked whenever a conditional of the same form is encountered, where the criterion for sameness must be modeled after α-invariant folding.
– We show that the techniques in this paper are sufficient to specialize a naïve string search algorithm to the Knuth-Morris-Pratt (KMP) algorithm [8], which is a staple test case for supercompilation (Section 5). This example motivates a technique called delimited aliasing which ensures static information is properly retained during memoization.

A heavily commented MetaOCaml source file containing all nontrivial code of this paper is available from the author's homepage. However, note that some parts of the code were shortened or omitted due to space limitations.

2 Background: Supercompilation

In this section we briefly review Sørensen et al.'s positive supercompiler [13]. We use the `contrived` function from the introduction as a running example. When asked to optimize the `contrived` function, the supercompiler starts a process called *driving* on the body of the function, reducing it as much as possible:

```
let f xs ys = match xs with [] → length xs + length ys
                          | w::ws → length xs
in f zs (1::zs)
⇓
match zs with [] → length zs + length (1::zs)
            | w::ws → length zs
```

where ⇓ denotes reduction – note that an open term is being reduced, with `zs` free. Now the code is at a pattern-match that cannot be resolved statically. In that case, driving replaces the scrutinee in each branch with the corresponding pattern:

```
match zs with [] → length [] + length (1::[])
            | w::ws → length (w::ws)
```

Note that `zs` is replaced by `[]` in the first branch but by `w::ws` in the second.

This substitution implements the *introduction of assumptions* mentioned in the introduction: the supercompiler rewrites each branch with the knowledge that the scrutinee must have a particular form in order for that branch to be entered. Furthermore, both calls to `length` in the first branch benefit by introducing the assumption `zs = []`. In the original source program, the scrutinee was `xs`, whereas the second call's argument was `ys`; the β substitution during the reduction step (shown as ⇓ above) has exposed the sharing of the substructure `zs` in these two variables, so that the assumption introduced on (what used to be) `xs` *propagates* to (what used to be) `ys`. Put another way:

– Assumptions are introduced by replacing the scrutinee with patterns.
– Assumptions are propagated by sharing substructures.

The main idea behind delimited assumptions is that we can imitate both of these mechanisms by mutating metadata on a delayed variable.

After assumptions are introduced and propagated, the rewritten branch bodies are driven separately; however, blindly doing so can lead to non-termination. For example, driving the second branch by unrolling `length` gives

```
length (w::ws)
⇓
match w::ws with [] → 0
              | _::ws' → 1 + length ws'
⇓
1 + length ws
```

Note the `match` statement can be resolved statically, so no assumptions are introduced. The supercompiler at this point drives each argument of `+` separately. The left operand is in normal form, so it turns to the right operand, `length ws`.

```
length ws
⇓
match ws with [] → 0
           | _::ws' → 1 + length ws'
```

But the second branch is in a form already encountered before, so this unrolling can go on forever.

To avoid infinite unrolling, the positive supercompiler *lambda-lifts* and memoizes each statically unresolvable `match`. After introducing assumptions, but before driving each branch, the supercompiler places the whole `match` expression in a new top-level function whose parameters are the free variables of the expression.

```
let rec newfun xs =
  match xs with [] → 0
             | _::ws' → 1 + length ws'
```

When the supercompiler encounters the same `match` while driving the branches of `newfun`, where two terms are the "same" iff lambda-lifting them gives α-equivalent functions, then it emits a call to `newfun` instead of driving the same term again. For example, driving the `length ws'` in the second branch of the `match` in `newfun` replaces it by `newfun ws'`.

Put together, the supercompiler compiles the `contrived` function into

```
let rec contrived zs =
  match zs with [] → 0 + (1 + 0)
             | w::ws → 1 + length ws
and length ws = match ws with [] → 0
                           | _::ws → 1 + length ws
```

```
type ('s,'d) sd =
   { mutable dynamic : 'd code;
     mutable static : 's option; }
type ('s,'d) ps_cell =
 | Nil
 | Cons of ('s,'d) sd * ('s,'d) psl
and ('s,'d) psl =
 (('s,'d) ps_cell, 'd list) sd
(*unknown : 'd code → ('s,'d) sd*)
let unknown x =
   { dynamic = x; static = None }
(*forget : ('a,'b) sd → 'b code*)
let forget x = x.dynamic

(* assuming_eq : ('a, 'b) sd → 'a
   → (unit → 'c) → 'c *)
let assuming_eq x v thunk =
  let saved = x.static in
  try x.static <- Some v;
      let ret = thunk () in
      x.static <- saved; ret
  with e → x.static <- saved;
           raise e

(*dfun : (('a, 'b) sd → 'c code)
      → ('b → 'c) code*)
let dfun f =
 .<fun x → .~(f (unknown .<x>.))>.

(* match_ls :
     (('a,'b) ps_cell,'b list) sd
     → (unit → 'c code)
     → (('a,'b) sd → ('a,'b) psl
         → 'c code)
     → 'c code
 *)
let match_ls ls for_nil for_cons =
  match ls.static with
   | Some Nil → for_nil ()
   | Some (Cons (x,xs)) →
       for_cons x xs
   | None →
     .<match .~(forget ls) with
       | [] → .~(assuming_eq ls Nil
                  for_nil)
       | x::xs →
        .~(let x = unknown .<x>.
           and xs = unknown .<xs>.
           in assuming_eq
                ls (Cons (x,xs))
                (fun () →
                   for_cons x xs))>.
```

Fig. 1: Data types and functions implementing delimited assumptions.

In general, driving stops when the term under consideration reaches either a normal form or a memoized form. This heuristic is called *α-invariant folding*. Stronger termination heuristics are possible and implemented usually as *generalization*, but we will not deal with that aspect in this paper.

3 Delimited Assumptions

Driving follows the execution of its input program with three mechanisms: reduction of open terms, introduction of assumptions, and propagation of assumptions. As seen in the `power` example from the introduction, reduction of open terms is handled very naturally with MSP, as delayed variables can be manipulated like values and injected into generated code. Effectively, inserting brackets and escapes to force evaluation under binders corresponds to implementing the reduction part of driving. The trickier part is the handling of assumptions.

Figure 1 shows types and functions used to handle assumptions with MSP. Whereas the `power` example directly manipulated raw code values of the form `.<x>.`, the delimited assumption technique uses *static-dynamic values*, of type sd. Here, "dynamic" means delayed by brackets, and "static" means not delayed. The sd type carries a dynamic value .<x>. and a static description of x's dynamic value (i.e. of the value x will have when the generated code is run). The type of x's value is 'd, and the type of the static description is 's. Static-dynamic values are created by unknown, which attaches void static information to a dynamic value, and cast back to a dynamic value with forget, which discards static information. An example is seen in dfun, which generates a dynamic **fun**, wraps the parameter in unknown, and passes that to a callback to generate the body.

Static knowledge is often partial. For example, we might know that a dynamic list xs must be a cons cell x::xs' but not the value of x or whether xs' is also a cons cell. We need to mix in sd throughout data structures to represent such partial knowledge, which for the list type gives the partially static list type, psl. The ps_cell type encodes one cell worth of static information: empty or not, and if nonempty, the static-dynamic representations of the head and tail. The psl type is a static-dynamic type whose dynamic component is a list and whose static component is ps_cell.

The static information is manipulated during a call to match_ls, which looks deliberately like a **match** on a list:

```
match_ls xs (fun () → .<"empty">.) (fun x xs' → .<"nonempty">)
```

Conceptually, this function is a dynamic **match** whose branches are generated by the two callbacks, but it avoids generating a **match** at all if the static information on xs tells us the outcome, e.g. whether the list is empty or a cons cell. This optimization is implemented in the first half of match_ls – if static information is available, match_ls calls only one of the callbacks. However, if static information is unavailable, match_ls generates a dynamic **match**, then wraps pattern variables (if any) in sd and invokes the callbacks. Moreover, the scrutinee's static information is destructively updated to reflect which branch was taken: to Nil in the [] branch, and to Cons in the x::xs branch. This update is undone when the callback returns, so the assumption's lifetime is delimited by the **match** branch in which it was introduced – hence the name *delimited assumption*. This modification and restoration of static information is done in assuming_eq.

Note that the update by assuming_eq is done by mutation. By destructively updating static information, all copies of the data see the update. For example, the contrived function in the introduction can be staged as follows.

```
let rec gen_contrived () = dfun (fun zs →
  let f xs ys = match_ls xs
               (fun () → .<.~(gen_len xs) + .~(gen_len ys)>.)
               (fun _ ws → .<(* discussed later *)>.)
  in f zs (cons (known 1 .<1>.) zs)))
and gen_len ws = match_ls ws (fun () → .<0>.)
                             (fun _ ws → .<1 + .~(gen_len ws)>.)
```

Basically, we just replaced **fun** by dfun and **match** by match_ls. The dfun wraps the generated parameter in void static information, so zs.static = None and zs.dynamic = .<v_zs>. for some (dynamically bound) variable. The cons operator is just :: for partially static lists, and known creates sd with the specified static information (definitions omitted), so when f is entered, we have[1]

```
zs = { dynamic = .<v_zs>.; static = None }
xs == zs (* NB: physical equality *)
ys = { dynamic = .<1::v_zs>.; static = Cons (1, zs) }
```

representing the fact that we have no knowledge of the dynamic value of zs while we do know xs = zs and ys = 1::zs. Most importantly, ys shares the zs node with xs, so that any changes to zs are visible from both xs and ys. When the match_ls in f introduces the assumption xs = [] by modifying xs, that change also happens on zs (because they're physically equal), and this change is visible from ys. After introducing the assumption, the data look like

```
zs = { dynamic = .<zs'>.; static = Some Nil }
xs == zs (* NB: physical equality *)
ys = { dynamic = .<1::zs'>.; static = Cons (1, zs) }
```

representing the updated, local knowledge zs = [] and xs = [] and ys = [1], as desired. Subsequent match_ls on ys can avoid generating any dynamic **match** using this static information.

Overall, the generated code is

```
.<fun zs → match zs with [] → 0 + (1 + 0)
                       | x::xs → (* discussed later *)>.
```

Both calls to length have been completely optimized away. This would not have happened if the assumption about xs didn't propagate to ys.

Thus, the techniques in this section suffice to imitate driving, including open-term reduction, introduction of assumptions, and propagation to all copies. We should note that not all open-term reductions are easily simulated this way. For example, in the f function above, (+) is hard-coded inside brackets, so it's not optimized away, whereas an automated supercompiler might reduce it as well. For this example, if we really need to optimize that addition, we can still do so by making the returned integer partially static. Such a workaround may or may not be so obvious in general; however, experience with more traditional, PE-like uses of MSP suggests that this is not a significant issue.

4 Ensuring Termination

The previous section deliberately ignored a part of contrived that involves a termination issue. In this section, we explain how to simulate α-invariant folding to ensure termination. The most obvious way to fill in the expression marked *(∗discussed later ∗)* in the staged code above is to put gen_len xs there, following the structure of the original, unstaged code. Alas, this call never finishes. The input xs is not completely statically known, so gen_len eventually runs out of

[1] Pedantically, the first argument of the Cons in ys.static should be another sd, but we simply write the static representation 1 for the sake of conciseness.

```
(* State monad. *)
type ('a,'st) monad = 'st → ('a * 'st)
(* memoize : 'key
   → ('a code, ('key,'a code) table) monad
   → ('a code → ('b code, ('key,'a code) table) monad)
   → ('b code, ('key,'a code) table) monad *)
let memoize key fcn call =
  bind get (fun table →
  match lookup key table with
  | Some f → call f
  | None → bind get (fun table →
           ret .<let rec f = .~(run_monad fcn (add key .<f>. table))
                  in .~(run_monad (call .<f>.) table)>.))

(* Fix the table type for brevity. *)
type 'a table_monad =
  ('a, ((int, int) psl, (int list → int) code) table) monad
(* gen_contrived : unit → (int list → int) code table_monad *)
let rec gen_contrived () = dfun (fun zs →
  let f xs ys = match_ls xs
                (fun () → gen_len xs +! gen_len ys)
                (fun _ ws → gen_len xs)
  in f zs (cons (known 1 .<1>.) zs))
(* gen_len : (int,int) psl → (int code) table_monad *)
and gen_len ws =
  memoize (freeze ws)
   (dfun (fun ws' → alias ws (forget ws') (fun () →
           match_ls ws
           (fun () → ret .<0>.)
           (fun _ ws → ret .<1>. +! gen_len ws))))
   (fun f → return .<.~f .~(forget ws)>.)
```

Fig. 2: Staged `contrived` function with memoization.

static information to act on. This means the `match_ls` in `gen_len` generates a dynamic **`match`**, whose cons-branch is generated by creating a fresh `ws`, again with no static information. This is then passed recursively to `gen_len`, which repeats the same process.

This situation is analogous to driving without folding. With `match_ls`, we are forcing the evaluation of branch bodies of statically unresolvable pattern-matches by making deeper and deeper assumptions about the input list, but there is no bound on the depth of this assumption. This leads to non-termination, because unlike the driving process described in Section 2, the code shown here doesn't generate a (recursive) function that can be reused later when an identical `match_ls` is reached. Generating and memoizing those functions is an integral part of the positive supercompiler's termination heuristic, and we need to simulate this in MSP as well.

Figure 2 shows a terminating generator which memoizes the pattern-match in `gen_len`, keyed with the scrutinee (since that's the only free variable in the `match` statement). Following Swadi et al. [15], we thread the memo table by a state monad; `ret`, `bind`, and `get` are the usual state monad operations, and `match_ls` and `dfun` are updated to work inside the monad. Similarly, `(+!)` generates a dynamic `(+)` inside the monad. Other than that, the only change is the addition of a call to `memoize`, which takes a `key`, a monadic action `fcn` that generates a function, and `call` which maps a dynamic function to some code invoking that function. If `key` is not in the table, `memoize` dynamically binds the function returned by `fcn` and generates a call to it with `call`. The `fcn` is run on a state extended with the mapping `key` $\mapsto$ `.<f>.`, where `f` is the newly generated function. If `memoize` is invoked again with the same key while `fcn` generates the body of `f`, then only `call` is invoked, without generating a new function. Thus, the code in Figure 2 terminates and generates

```
.<fun zs → match zs with [] → 0 + (1 + 0)
                       | w::ws → 1 +
                               (let rec len ws =
                                  match ws with [] → 0
                                              | _::ws' → 1 + len ws'
                                in len ws)>.
```

This memoization scheme has several subtleties, two of which are explained here, while the last one is explained in the next section using the more sophisticated KMP example. The first subtlety is that memoization keys must be deep-copied before inserting into the table, because subsequent introduction of assumptions can change their static information. The `freeze` function in Figure 2 performs this deep copy. The second subtlety is that key comparison cannot be simple equality. For example, if `gen_len` is called on the partially static datum

```
xs = { dynamic = .<v_xs>.; static = None }
```

for some dynamic variable `v_xs` bound on the caller's side, then a new entry is created in the memo table with `xs` as the key (assuming it's not already there). However, the second branch of `match_ls` calls `gen_len` on

```
ws = { dynamic = .<v_ws>.; static = None }
```

where `v_ws` is the symbol freshly generated by `match_ls`. If these keys were compared with `(=)`, then the lookup would fail, resulting in non-termination.

This shows that key comparison should ignore differences in names of dynamic variables. However, it should *not* ignore differences in sharing. Although not an issue for `gen_len`, if a function with two arguments `xs` and `ys` introduces assumptions on `xs` and then pattern-match on `ys`, then a memo entry created when `xs` and `ys` are physically equal must not be used at a call site where they are not equal. In general, the keys must be compared under DAG isomorphism – they are equal iff they have the same shape (same number of cons cells with the same heads, a.k.a. car's, linked together in the same manner), but not the same names on the leaves where static information is `None`.

This keying discipline is not so mysterious if we consider the connection to α-invariant folding in positive supercompilation. A static-dynamic datum with void static information is like a variable in the object term of supercompilation, whereas a static-dynamic datum with, say `Cons(1,xs)` as static information is like an open object term `1::xs` in supercompilation. The function generated during memoization is the lambda-lifting of the **`match`** that is memoized, and the memo keys are the collection of all partially static data manipulated inside that **`match`**. Hence, if a **`match`** statement on a particular source location is executed multiple times, each execution instance is uniquely identified by the key. The lambda-lifted function `f` is reusable precisely when supercompiling a term whose lambda-lifting is α-equivalent to `f`, which is necessary and sufficient for the lookup key to be graph-isomorphic to the key found in the table.

5 Case Study: KMP

In this section, we show that our MSP techniques suffice to pass the "KMP test" for supercompilation [13]. In this test case, we explain the final subtlety in implementing α-invariant folding with memoization, which motivates one final technique which we call delimited aliasing.

Figure 3a shows a function `search` that tests if a pattern string `p` occurs in a subject string `s`.[2] It checks if `p` is a prefix of `s` by character-wise comparison, and upon a mismatch, drops the head of `s` and starts over. If `p`,`s` have lengths m, n, respectively, this takes $O(mn)$ comparisons. The objective is, given a concrete pattern, to generate the efficient KMP algorithm in Figure 3c which performs only $O(m+n)$ comparisons (not counting generation cost).

Specializing `search` to a fixed pattern `"aab"` with PE gives more or less Figure 3b, where the `[]`-cases of **`match`**es are omitted due to space limitations. The **`match`**es on the pattern are statically resolved, but the subject is still rewound to the beginning upon a mismatch, resulting in $O(mn)$ comparisons. We can do better. If the third character mismatched, the subject must start with `"aa"`, so we know the first comparison of the next round will return `true`. We can therefore skip that comparison. Eliminating such redundant comparisons gives the KMP algorithm in Figure 3c. Note the failing branch of the comparison in `kmp_b` jumps to `kmp_ab` instead of `kmp_aab`.

This optimization happens by noting static information learned about `os` due to pattern matches and comparisons on `ss`. It's by following the **`match`** `ss` and **`if`** `s = 'a'` that we learn (or assume) that the subject starts with `"aa"`, and `os` is never inspected; nonetheless, this information should propagate to `os` and be used to skip (or statically perform) redundant comparisons. This is just what positive supercompilation does, as do our MSP techniques. Figure 3d demonstrates a staged version of the matcher. It is fairly straightforward, with

[2] Strings are represented as `char list` rather than `string`, but for brevity we write literals `"like this"` where convenient.

```
let rec search p s = loop p s p s
and loop pp ss op os =
  match pp with
   | [] → true
   | p::pp' →
    match ss with
     | [] → false
     | s::ss' →
      if s = p
      then loop pp' ss' op os
      else next op os
and next op = function
   | [] → false
   | s::ss → loop op ss op ss
(* Mnemonics for variable names:
   p, pp -- Pattern to search for
   s, ss -- Subject to search over
   op -- Original Pattern
   os -- Original String *)
```

(a) Generic version.

```
let rec naive_aab ss = aab ss ss
and aab ss os =
  match ss with
   | x::xs →
      if x = 'a' then ab xs os
      else next os
and ab ss os =
  match ss with
   | x::xs →
      if x = 'a' then b xs os
      else next os
and b ss os =
  match ss with
   | x::xs →
      if x = 'b' then true
      else next os
and next = function
   | _::xs → aab xs xs
```

(b) Naïvely specialized to “aab”.

```
let rec kmp_aab = function
  | x::xs →
    if x = 'a' then kmp_ab xs
    else kmp_aab xs
and kmp_ab = function
  | x::xs →
    if x = 'a' then kmp_b xs
    else kmp_aab xs
```

```
and kmp_b = function
  | x::xs →
    if x = 'b' then true
    else if x = 'a' then kmp_b xs
         else kmp_ab xs
```

(c) Hand-written KMP for “aab” (split in two columns).

```
let rec loop pp ss op os =
  match pp with
  | p::pp' →
    memoize (freeze (pp,ss,op,os))
      (dfun (fun ss' → alias ss (forget ss') (fun () →
              match_ls ss (fun () → ret .<false>.)
                          (fun s ss →
                             ifeq s (known p .<p>.)
                             (fun () → loop pp' ss op os)
                             (fun () → next op os)))))
      (fun f → ret .<.~f .~(forget ss)>.)
and next op os () = match_ls os (fun () → ret .<false>.)
                                (fun s ss → loop op ss op ss ())
```

(d) Staged string search with memoization (one column).

Fig. 3: String matcher. Suffixes in specializations indicate remaining pattern.

`match` replaced by `match_ls` and `if s = c` replaced by `ifeq`, a combinator similar to `match_ls` but generating equality tests with constants.

The one aspect in which this example differs significantly from `contrived` is the use of the combinator

```
alias : ('a,'b) sd → 'b code → (unit → ('c,'d) monad) → ('c,'d) monad
```

which is almost the same as `assuming_eq` but updates the dynamic value instead of the static information. For example, if we reach the `memoize` in Figure 3d when

```
pp = "aab"
op = "aab"
ss = { dynamic = .<v_ss>.; static = None }
os = { dynamic = .<v_os>.; static = Some ('a', ss) }
```

then `memoize` calls back the generator of the memoized body (i.e. the part that starts out with `dfun`), and the `dfun` creates a new static-dynamic value

```
ss' = { dynamic = .<v_ss'>.; static = None }
```

Then `alias ss (forget ss')` modifies the dynamic variable associated to `ss` to make it an alias for `ss'`, hence

```
ss = { dynamic = .<v_ss'>.; static = None }
```

All other static-dynamic values remain unchanged. Just like `assuming_eq`, this mutation is undone when the thunk (the last argument to `alias`) returns.

The reason we need this is because, by making `v_ss` an argument to the function generated by `memoize`, we're effectively renaming the dynamic variable `v_ss`. The whole point of generating a function is to have its body process the parameter `v_ss'` instead of `v_ss`, so that this body becomes reusable. However, in the case of KMP, the body must also process `os`, which would still refer to `v_ss` instead of `v_ss'`; mutating the `ss` structure ensures that both `os` and `ss` are updated to point to `v_ss'`.

This scheme once again corresponds to α-invariant folding, where free variables are captured and consistently renamed. Mutating the dynamic variables on leaf nodes of static-dynamic values corresponds to renaming the dynamic variable associated with that value across the board.

It should be noted that to be faithful to the α-invariant folding heuristic, `alias` should only be used on leaf nodes, whose static information is `None`. This ensures maximum retention of static information, because `alias`'ed nodes must have void static information (since the new dynamic variables have no static information). Thus, a combinator would be helpful that traverses static-dynamic data and collects such nodes, generating a function with as many arguments as needed. This will be fairly tricky to type in (Meta)OCaml, since we need to traverse arbitrary data structures while managing a heterogeneous collection of dynamic variables to eliminate duplicates. We leave the pursuit of such a combinator for another occasion.

With these mechanisms in hand, the generator in Figure 3d produces more or less the KMP code in Figure 3c, with two minor differences. Firstly, as positive supercompilation tracks equalities but not disequalities, we have redundant comparisons of the form

```
if x = 'a' then ...
else if x = 'a' then ... else ...
```

This can be eliminated by maintaining richer static information. For example, a dynamic value can be tagged with the set of values it can have, rather than a single value. The other difference is that the generated code nests `let rec`s like

```
let rec f1 =
  let rec f2 = bar
  in baz
in foo
```

instead of having a single, flat `let rec`. Hence, only functions generated in the direct ancestors of a `memoize` call can be reused, which is both a good safety precaution and a limitation. Reusing functions from a different conditional branch runs the risk of invoking code that relies on assumptions valid only in that branch, but if used properly it can reduce generated code size. Current MetaOCaml provides no way to generate `let rec` with a variable number of bindings, but a new primitive allowing that is expected in a future release.[3] It would be interesting to see if they enable notable improvements.

6 Related Work

Supercompilation was devised by Turchin for Refal [19] and later adapted to more standard functional languages by Glück and Klimov [3]. Sørensen et al. placed this on the same theoretical footing as PE, deforestation, and generalized partial computation (GPC), and showed that supercompilation subsumes PE and deforestation [13]. We have drawn heavily from this work: [13] effectively identified all the key ingredients for supercompilation, in terms that are transferable to MSP. Supercompilation has been extended by distillation [4], but it remains unclear what the differences are, in terms that can be mapped to MSP.

GPC [2,18] is an extension of PE that uses a theorem prover to manage static information. While the use of a theorem prover makes it harder to predict how it performs on any given task, we remark that the delimited assumption technique can be used to simulate GPC as well, by simply taking the static information to be variables in the theorem prover. Compared to GPC, however, our techniques perform the very stylized information propagation of supercompilation, which behaves more predictably than if the bookkeeping is delegated to a black-box solver. In this way, our techniques might be useful to lighten the load on the prover.

MSP was originally a notation for PE [10] but was later developed into a programming language feature by Taha and Sheard [17]. Its main advantages are the existence of a well-behaved metatheory [5] and type systems that make strong

[3] Private communication with the maintainer.

guarantees about generated code [6, 16, 20]. MetaOCaml statically prevents the construction of ill-formed or ill-typed code values, with the one exception that effects can cause scope extrusion, where a dynamic variable is floated out of its scope. Much effort has been expended on catching this problem early, resulting in static type systems [6,20] and dynamic checks [7], the latter of which MetaOCaml already implements. The present paper adds to the motivation for these efforts by offering a new, important use for effectful MSP.

Partially static data types were known in PE circles starting perhaps with Mogensen [9], but Sheard and Diatchki [11] seem to be the first to use it as a staging technique. However, they duplicated constructors instead of pairing dynamic values with optional static information, which made their code generally more verbose than ours. The pairing technique itself appears in earlier PE works, for example [14]. The observation that mutating components of these pairs can simulate supercompilation appears to be new.

7 Conclusion

We showed that MSP can achieve a good deal of the effects of positive supercompilation. The central idea is to update the static portion of partially static data structures upon entering a dynamic conditional, and to do this with mutation. This arrangement ensures that the assumption is propagated to all copies of the data, allowing smart handling of nonlinear code. As an auxiliary technique, a fairly nonstandard memoization scheme may be required to ensure termination, namely comparing partially static data with graph isomorphism. Taken together, these techniques can specialize a naïve string matcher to a KMP matcher.

The techniques in this paper should be thought of as low-level groundwork for realizing supercompilation by staging. It is fairly technical and we can't expect most MetaOCaml programmers to be apply this easily, without making mistakes. A well-designed combinator library should be able to alleviate this problem. An important goal for such a library is to offer a richer `memoize` combinator that collects leaf nodes from its key and generates a function with as many parameters as are needed, performing delimited aliasing as well. This would make the techniques much more straightforward to understand.

Finally, this paper's purpose is to demonstrate techniques that are useful in expressing supercompilation-like optimizations in MSP, and not to lay down a formal analysis. We did not attempt to define precisely what class of programs can be supercompiled, but as mentioned earlier, not all driving trees are naturally expressed with MSP. It would be interesting to see what kinds of driving trees are beyond MSP in its current form (if any).

Acknowledgment. We thank Oleg Kiselyov for his insightful comments and encouragement to publish this work.

References

1. Dybvig, R.K.: Writing hygienic macros in scheme with syntax-case. Tech. Rep. TR356, Indiana University Computer Science Department (1992)
2. Futamura, Y.: Program evaluation and generalized partial computation. In: FGCS. pp. 685–692 (1988)
3. Glück, R., Klimov, A.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Static Analysis, Lecture Notes in Computer Science, vol. 724, pp. 112–123. Springer Berlin Heidelberg (1993)
4. Hamilton, G.W.: Distillation: Extracting the essence of programs. In: PEPM. pp. 61–70. ACM, New York, NY, USA (2007)
5. Inoue, J., Taha, W.: Reasoning about multi-stage programs. In: ESOP. pp. 357–376 (2012)
6. Kameyama, Y., Kiselyov, O., Shan, C.c.: Combinators for impure yet hygienic code generation. In: PEPM. pp. 3–14 (2014)
7. Kiselyov, O.: The design and implementation of BER MetaOCaml: System description. In: FLOPS, to appear (2014)
8. Knuth, D.E., Morris, J., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal of Computing 6(2), 323–350 (1977)
9. Mogensen, T.Æ.: Efficient self-interpretations in lambda calculus. Journal of Functional Programming 2(3), 345–363 (1992)
10. Nielson, F., Nielson, H.R.: Two-level functional languages. Cambridge University Press (1992)
11. Sheard, T., Diatchki, I.S.: Staging algebraic datatypes. Unpublished manuscript, `http://web.cecs.pdx.edu/~sheard/papers/stagedData.ps`
12. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School. pp. 246–270 (1999)
13. Sørensen, M.H., Glück, R., Jones, N.D.: Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In: ESOP. pp. 485–500 (1994)
14. Sperber, M.: Self-applicable online partial evaluation. In: Partial Evaluation, LNCS, vol. 1110, pp. 465–480. Springer (1996)
15. Swadi, K., Taha, W., Kiselyov, O., Pašalić, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: PEPM. pp. 160–169. ACM (2006)
16. Taha, W., Nielsen, M.F.: Environment classifiers. In: POPL. pp. 26–37. ACM (2003)
17. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: PEPM. pp. 203–217. ACM (1997)
18. Takano, A.: Generalized partial computation using disunification to solve constraints. In: CTRS. pp. 424–428 (1993)
19. Turchin, V.: A supercompiler system based on the language Refal. SIGPLAN Notices 14(2), 46–54 (1979)
20. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: PLDI (2010)