A Monadic Approach for Avoiding Code Duplication when Staging Memoized Functions [*]

Kedar Swadi [†] Walid Taha
Rice University, Houston, TX, USA
{kswadi, taha}@rice.edu

Oleg Kiselyov
FNMOC, Monterey, CA, USA
oleg@okmij.org

Emir Pasalic
Rice University, Houston, TX, USA
pasalic@cs.rice.edu

Abstract

Building program generators that do not duplicate generated code can be challenging. At the same time, code duplication can easily increase both generation time and runtime of generated programs by an exponential factor. We identify an instance of this problem that can arise when memoized functions are staged. Without addressing this problem, it would be impossible to effectively stage dynamic programming algorithms. Intuitively, direct staging undoes the effect of memoization. To solve this problem once and for all, and for any function that uses memoization, we propose a staged monadic combinator library. Experimental results confirm that the library works as expected. Preliminary results also indicate that the library is useful even when memoization is not used.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program synthesis

General Terms Multi-stage Programming, Program Specialization

Keywords Staging, Monads, Fixed points, Code duplication, Program specialization, Multi-stage programming, Partial evaluation, Program generation

1. Introduction

Abstraction mechanisms such as functions, modules, and objects can reduce development time and improve software quality. But such mechanisms often have a cumulative runtime overhead that can make them unattractive to programmers. Partial evaluation [25] encourages programmers to use abstraction mechanisms by trying to ensure that such overheads can be paid in a stage earlier than the main stage of the computation. But because of the subtleties of the process of separating computations into distinct stages, a partial evaluator generally cannot guarantee all abstraction overheads will be eliminated in an earlier stage. Furthermore, it is not always intuitive for a novice user to guess which ones will be eliminated. One approach to dealing with the issues of explicit guarantees about which abstraction overheads are eliminated is Multi-stage Programming (MSP) [48, 44]. MSP languages provide the programmer with a quotation mechanism designed specifically for constructing code fragments. Such quotation mechanisms enable the development of both the formal reasoning principles (cf. [45, 44]) and the static type systems that guarantee that all generated programs would be type-safe (cf. [47]).

1.1 Code Duplication

A key technical problem that arises in source-level program generation (including partial evaluation and MSP) is code duplication. The problem can be illustrated using a trivial generator that represents programs as strings:

```
let a = "x*y" in
let b = a ^ " + " ^ a in ...
```

Here string quotations are used to delay some sub-computations. This is often called staging. It is common in MSP to derive staged programs like the one above from an unstaged counterpart [48] such as:

```
let a = x*y in
let b = a + a in ...
```

Clearly, the unstaged program involves no code generation, so there is no duplicated computation. When the unstaged program is executed, the multiplication is performed and the result is stored in `a`. This result is simply a number. Computing `b` only involves adding two numbers to produce a third number. In the staged variant, computing `a` involves no computation (it is already a string value representing a code fragment). Computing `b` also involves computing two concatenation operations to produce the string value `"x*y + x*y"`. Thus, the length of the string `b` will be a bit more than twice as much as `a`. In addition, performing the computation represented by `b` will also take a bit more than twice as much as that for `a`.

In the example above, code duplication leads to doubling the amount of work required both for generating and for executing the resulting program. By the Master Theorem, this constant-factor degradation in performance can also lead to degradation in the order of complexity. In particular, it is often the case that code generation is done in iterative or recursive functions where the number of repetitions depends on an input value. These repetitions often compound the duplication of code (see the example of the power function in Section 3.5). This combination of events is common in practice, and easily leads to exponential degradation in the order of complexity.

[*] Supported by NSF ITR-0113569 “Putting Multi-Stage Annotations to Work”, Texas ATP 003604-0032-2003 “Advanced Languages Techniques for Device Drivers”, and NSF SOD-0439017 “Synthesizing Device Drivers”.

[†] Current affiliation: Persistent Systems Pvt. Ltd., Pune, India

PEPM ’06 January 9–10, Charleston, South Carolina, USA.

1.2 Problem

The motivation for the present work comes from ongoing efforts to stage various dynamic programming (DP) algorithms. DP algorithms are memoized versions of recursive equations. Without memoization, evaluating these recursive equations directly would require an exponentially large number of steps. However, the call graph for such a computation involves a significant number of repeated calls to the same function with the same inputs. Memoization makes it possible to avoid this redundancy, and often leads to algorithms that are sub-exponential. This paper shows how direct staging of memoized functions has the effect of undoing what is achieved by memoization, and that the runtime complexity of the resulting solution degenerates back to exponential time. Local transformations, such as inserting let statements into the generated code, appear to be insufficient. A common global transformation in the partial evaluation literature is to convert the program into continuation-passing style (CPS). Because the use of memoization means that a notion of state is already present, precisely how the CPS transform should be carried out is not obvious. Furthermore, the resulting code (and types) from these transformations obfuscate the basic recurrence underlying the algorithm being staged, and complicate the task of programming.

Can we resolve these issues, and package the results in a form that would help keep the task of staging a memoized function manageable?

1.3 Contributions

After a brief review of MSP (Section 2), we present a generic account of memoization using fixed-point combinators and monads (Section 3). Somewhat surprisingly, this has not been done before. In our previous, ad hoc attempts to stage memoized functions, identifying the sources of code duplication was difficult. To help programmers avoid this difficulty, we propose a combinator-based formulation of memoized functions. This formulation allows us to precisely characterize a pervasive source of code-duplication for this class of functions. It also allows us to show that the problem we identify can be addressed once and for all by staging the combinators in a particular manner. An interesting byproduct of this formulation is that it allows us to show how the combinators can also be staged so that all intermediate results are named, thus ensuring that any generated programs would be in A-normal form [19].

Compared to previous approaches to controlled naming and the use of monads in program generation, the key technical novelty is that our approach does not require any language extensions. For example, Hatcliff and Danvy [22] use a monad augmented with special-purpose rewriting rules that can be used at the meta-level to specify continuation-based partial evaluators. In contrast, we present a monad that combines both continuations and state and is *expressible at the object-level* in a multi-stage calculus. The calculus we use does not allow pattern matching on generated code [45], so our approach guarantees that we preserve the equational properties of generated code fragments. By implementing all the combinators in MetaOCaml [8, 35], we also demonstrate that using a standard, practical static type system [47, 7] is sufficient to allow the programmer to implement several different approaches to controlling code duplication.

To test the utility of these combinators from the point of view of both the performance of the resulting implementation and the difficulty of programming, we have used them to implement and study several standard DP algorithms (Section 4). Our experiments show that the combinators can be used without change for these different examples, and that the performance of the resulting implementations is comparable to that of hand-written (unspecialized) C implementations of these algorithms.

2. Multi-stage Programming

MSP languages [48, 44] provide three high-level constructs that allow the programmer to break down computations into distinct stages. These constructs can be used for the construction, combination, and execution of code fragments. Standard problems associated with the manipulation of code fragments, such as accidental variable capture, are eliminated (cf. [44]). The following minimal example illustrates MSP programming MetaOCaml:

```
let rec power n x =
   if n=0 then .<1>.
   else .< .~x * .~(power (n-1) x)>.
let power3 = .! .<fun x -> .~(power 3 .<x>.)>.
```

Ignoring the staging constructs (brackets `.<`e`>.`, escapes `.~`e, and run `.!` e) the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^3. Without staging, the last step simply returns a function that would invoke the `power` function every time it gets invoked with a value for x. In contrast, the staged version builds a function that computes the third power directly (that is, using only multiplication). To see how the staging constructs work, we can start from the last statement in the code above. Whereas a term `fun x ->` e `x` is a value, an annotated term `.<fun x -> .~(`e `.<x>.)>.` is not: the outer brackets contain an escaped expression that still needs to be evaluated. Brackets mean that we want to construct a future stage computation, and escapes mean that we want to perform an immediate computation *while* building the bracketed computation. In a multi-stage language, these constructs are not hints, they are imperatives. Thus, the application e `.<x>.` must be performed even though `x` is still an uninstantiated symbol. In the `power` example, `power 3 .<x>.` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the function `power`, the recursive application of `power` is escaped to ensure its immediate execution in the first stage. Evaluating the definition of `power3` first results in the equivalent of

```
.! .<fun x -> x*x*x*1>.
```

Once the argument to run (`.!`) is a code fragment that has no escapes, this code fragment is compiled and evaluated. In this case, evaluation returns a function that has the same performance as if we had explicitly coded the last declaration as:

```
let power3 = fun x -> x*x*x*1
```

Applying this function does not incur the unnecessary overhead that the unstaged version pays every time `power3` is used.

Finally, while MSP language constructs resemble LISP and Scheme's quasi-quote and eval mechanisms, the technical novelty of these languages lies in automatically avoiding accidental variable capture [8], supporting equational reasoning inside quotations [45], and statically ensuring that any generated programs are well-typed (c.f. [47]). This comes at the cost of not providing a mechanism for taking apart code fragments once they are constructed [45].

3. A Monadic Combinator Library

In what follows we gradually develop the set of generic combinators that we use to characterize memoizing functions, to point out a pervasive source of code duplication when staging such programs, and to stage these combinators in a way that deals with this problem once and for all.

3.1 Fixed-point Combinators

In a call-by-name (CBN) setting, the simplest fixed-point combinator can be expressed as:

```
y : (’a -> ’a) -> ’a

let rec y f = f (y f)
```

In a call-by-value (CBV) language, such a definition does not work, because the inner application would always be evaluated, leading to divergence of any application of the fixed-point combinator. This problem is often addressed by restricting the type of the fixed-point combinator, and using eta-expansion to delay the inner application. For function types, we can use the following definition[1]:

```
y : ((’a -> ’b) -> (’a -> ’b)) -> (’a -> ’b)

let rec y f = f (fun x -> y f x)
```

3.2 Memoization

Memoization requires that whenever we apply a function to some arguments, we store an association of the arguments with the result of the application. This way, successive applications of the function to the same arguments need not be recomputed. To memoize recursive functions, this store and lookup work can be performed in the fixed-point combinator itself, so that the programmer can get the benefits of memoization without having to make changes to the code for the core algorithm.

The fact that memoization involves a notion of state suggests that we need a variant of the above combinator where the types are specialized as follows:

```
y_state : ((state -> ’a -> (state * ’b)) ->
           (state -> ’a -> (state * ’b))) ->
           (state -> ’a -> (state * ’b))

let rec y_state f = f (fun s x -> y_state f s x)
```

The memoizing fixed-point combinator that we will use is an instance of this type, where state is a memo table, and the combinator itself manipulates the state as follows:

```
y_memo :
  (((’a,’b) table -> ’a -> (’a,’b) table * ’b) ->
   ((’a,’b) table -> ’a -> (’a,’b) table * ’b)) ->
   ((’a,’b) table -> ’a -> (’a,’b) table * ’b)

let rec y_memo f =
  f (fun s x ->
       match (lookup s x) with
       | Some r -> (s,r)
       | None ->
             let (s1, r1) = (y_memo f s x) in
               ((ext s1 (x,r1)), r1))
```

where `ext` and `lookup` are functions for extending the table and for looking up values in the table, respectively.

This combinator first checks if the result of `f` applied to its argument `x` was already computed. If so, we return the stored result `r`, along with the unmodified table `s`. If not, we perform the application `y_memo f s x`, and add the result to the new table.

In practice, the function `f` to be memoized will be called with several arguments, and not all of them are needed to determine the key into the memo table. To support this common situation we pass a `key` function that extracts only the relevant components of `x` as used in the next variation, `y_key`.

```
y_key :
 (’a -> key) ->
 (((key,’b) table -> ’a -> (key ,’b) table * ’b) ->
  ((key,’b) table -> ’a -> (key ,’b) table * ’b))->
  ((key,’b) table -> ’a -> (key ,’b) table * ’b)

let rec y_key key f =
  f (fun s x ->
       match (lookup s (key x)) with
       | Some r -> (s,r)
       | None ->
             let (s1, r1) = (y_key key f s x) in
               ((ext s1 ((key x),r1)), r1))
```

The `key` is also useful when arguments might be grouped into equivalence classes such that all elements of a class yield the same result when called with `f`. In such a scenario, `key` can be used to translate the arguments into some representative of the class to which they belong, thus allowing for more effective memoization.

In general, a DP algorithm will consist of a collection of different functions, many of which need to be memoized using separate memo tables. Furthermore, because all of these tables must be maintained throughout the full computation, it might be necessary to memoize the results of more than one recursive function that might be used together. In such a case, the fixed-point combinator must be able to correctly lookup (or update) a state depending on which particular function it is called with. To do this, we make one more generalization:

```
y_mult : (’a -> key) *
         (tables -> (key * ’b) table) *
         ((key * ’b) table -> tables -> tables) ->
         ((tables -> ’a -> tables * ’b) ->
          (tables -> ’a -> tables * ’b)) ->
          (tables -> ’a -> tables * ’b)

let rec y_mult (key, get, set) f =
  f (fun s x ->
      match (lookup (get s) (key x)) with
       | Some r -> (s,r)
       | None ->
          let (s1, r1) =
             (y_mult (key, get, set) f s x)
          in ((set (ext (get s1) ((key x),r1)) s1),
              r1))
```

Here, `get` is used to identify the component of the state that the current function `f` uses, and `set` is used to update the state component used by `f`.

3.3 Staging the Fixed-point Combinator

The combinators presented above allow us to characterize a large class of memoized functions. Now we turn to the question of staging such functions when the parameter that determines the recursion structure is known statically.

3.3.1 The Source of Code Duplication

Staging the function to be memoized and not the fixed-point combinator leads to a code duplication problem. In particular, it generates code that repeats all the work that memoization saved. Consider the function `y_mult` presented above: the variable `r1`, used to name the result of the computation when it is first computed, is copied both into the table and returned back as the result value. In the unstaged setting, this is not problematic. In the staged setting, `r1` is not a simple value but a code fragment. If the variable `r1` has more than one usage occurrence, the code fragment carried by `r1` will then be

[1] Following a suggestion from a reviewer, we use more parenthesis than strictly needed so that the types easier to read.

inserted into a bigger code fragment multiple times. A sequence of these events leads to exponential code growth.

3.3.2 Where to Generate Let-statements

In the partial evaluation literature, the standard technique for avoiding code duplication is inserting let-statements that provide a name for the fragment and allow us to return several copies of the name rather than of the fragment. Code duplication can be avoided if the combinator `y_mult` can be modified to generate a let-statement that names the result of evaluating `r1` and then to allow us to refer to it in the rest of the code by that name. However, with `y_mult`, it is not clear that this can be done by a local addition of a let-statement and staging annotations to its definition. In particular, inserting a second-stage let-statement naming the result of evaluating `r1` requires inserting brackets around the final state returned by the computation, as well as name of the result. The difficulty here is that we must *simultaneously* return these two values as a static (first-stage) pair, and not a dynamic pair. The staged computation needs to process each of the two elements of the pair separately in the rest of the computation. The name of the result will be inserted into (and looked up from) the memo-table, while the state will be passed around and accessed statically during the first-stage computation. The type system simply does not allow an expression consisting of brackets to have any type other than a code type (and so it cannot have a pair type). In essence, the computation needs to return a dynamic let as a “side-effect”, without altering the type of the return value of the computation.

3.3.3 Conversion to CPS

When direct insertion of let statement seems to be impossible, another standard technique from the partial evaluation literature is to convert the source program into CPS. The following CPS version of the above combinator will allow us to do just that:

```
y_cps :
 ('a -> key) *
 (tables -> (key * 'b) table) *
 ((key * 'b) table -> tables -> tables) ->
   (('a -> tables -> (tables -> 'b -> 'c) -> 'c) ->
    ('a -> tables -> (tables -> 'b -> 'c) -> 'c)) ->
    ('a -> tables -> (tables -> 'b -> 'c) -> 'c)

let rec y_cps (key, get, set) f =
  f (fun s x k ->
       match (lookup (get s) (key x)) with
     | Some r -> k s r
     | None -> y_cps (key, get, set) f s x
         (fun s1 -> fun r1 ->
              k (set (ext (get s1) ((key x),r1)) s1)
                r1))
```

The function `f` is assumed to be in CPS, taking a continuation `k` that consumes the memo tables, a value of type `'b`, and to return a final answer of type `'c`. The added flexibility that we get form conversion from CPS (and that we do not seem to have before conversion to CPS) is that the type `'c` is unconstrained. The type of the return context for the computation does not have to be the same as `'b`. The staged fixed-point combinator can now produce new let statements that are added to the context of the computation but without interfering with the type or value being returned by the computation. The CPS fixed-point combinator can be staged as follows:

```
let rec y_cps (key,get,set) f =
  f
   (fun  x s k ->
     match (lookup (get s) (key x)) with
     | Some r -> k s r
     | None ->  y_cps (key,get,set) f x s
         (fun s1 -> fun v ->
           .<let r1 = .~v
             in .~(k (set (ext (get s1) ((key x),
                 .<r1>.)) s1) .<r1>.)>.))
```

Now, when we have to perform a function call, the continuation proceeds by first binding `v` to a fresh variable `r1`, and associating `key x` with this variable. As a result, any future lookup into the state results in the *name* `r1` being passed to the continuation rather than a (possibly large) value `v`.

Staging the fixed-point combinator in this fashion constraints the type of the memoized value and the answer type of the continuation to both be of code type. Each is a (different) polymorphic code type, but both are constrained to have the same environment classifier.

3.4 Monadic Encapsulation

Especially because of the complexity of its type, expecting the programmer to be able to use the above fixed-point combinator might seem impractical. Fortunately, the structure underlying our fixed-point combinator is a standard monad. Monads have been found useful for expressing effectful computations in purely functional languages [28], and for structuring denotational semantics [36, 37, 18], interpreters [30], compilers [21], partial evaluators [43, 22] and program generators [20, 41]. Several excellent introductions to monads exist in the published literature [50, 24].

For the purposes of this paper, a sufficient monad is an instance of the state-continuation monad:

```
'a monad =
  state -> (state -> 'a -> answer) -> answer
```

The operators for this monad are simply:

```
let ret a    = fun s k -> k s a
let bind a f =
  fun s k -> a s (fun s' x -> f x s' k)
```

Without any staging annotations, this monad is sufficient for staging many DP algorithms where the only source of code duplication is memoization. All we have to do is to convert the DP recurrence equations into monadic style and use open recursion (several examples are presented in Appendix A). During this process, we do not need to worry about the details of the monad.

If, however, there are other sources of duplication in the algorithm itself, the type of the monad can be restricted, and we have the choice of using either the standard operators above or the following staged version:

```
let retN a    =
  fun s k -> .<let z = .~a in .~(k s .<z>.)>.
let bindN a f =
  bind a (fun x -> bind (retN x) f)
```

Alternatively, `bindN` can be defined first:

```
let bindN a f =
  fun s k ->
   a s (fun s' x ->
         .<let z = .~x in
           .~(f .<z>. s' k)>.)
let retN a    = bindN (ret x) ret
```

The combinators always name the monadic value in their argument, thus ensuring that using this argument multiple times cannot lead to code duplication.

It is important to note that programs generated using these combinators can only contain the program fragments that are in brackets. This means that most of the computation in the staged, CPS fixed-point combinator as well as the staged monadic operators will be done in the first stage. Thus, the overhead of recursion, all operations on the memo table, and the applications of the return and bind operations will not be incurred in the generated code.

3.5 Aside: A-normal Form

With some care, the above operators can be used to produce code in A-normal form [19]. Consider a variant of the power function staged as follows:

```
let power f (n, i) =
  if (n = 0) then ret .<1>.
  else if (even (n)) then
    bind (f (n / 2, i)) (fun x ->
    ret (.<.~x * .~x>.))
  else
    bind (f (n-1, i)) (fun x ->
    ret (.<.~i * .~x>.))

let pow5 = .<fun i ->
     .~((y_sm power) (5, .<i>.)
         [] (fun s x -> x))>.
```

The combinator `y_sm` is a variant of `y_cps` with the `key`, `get`, and `set` parameters instantiated so that `n`, the first argument to `power`, is the key and there is only one table in the global store.

Evaluating the expression for `pow5` leads to code duplication. In particular, the code we get is:

```
  .<fun i_1 ->
     (i_1 * (((i_1 * 1) * (i_1 * 1)) *
             ((i_1 * 1) * (i_1 * 1))))>.
```

Code duplication here has nothing to do with memoization, because no memoization is needed for computing this function. In situations like this, the combinators can be used inside the memoized function (as opposed to the fixed point combinator) to avoid code duplication in the body of the staged function.

It should be noted that if we replace `ret` with `retN`, the generated code for `pow5` becomes:

```
  .<fun i_1 ->
   let z_2 = (i_1 * 1) in
   let z_3 = (z_2 * z_2) in
   let z_4 = (z_3 * z_3) in
   let z_5 = (i_1 * z_4) in z_5>.
```

This modification is sufficient for solving the problem for this particular function. In general, however, computations done in the body of a `bindN` operation can introduce duplication. Thus, in general, it is more appropriate to use `bindN` everywhere rather than `retN`.

Using `bindN` introduces a subtle problem: If we use the `bindN` in the function above, we get:

```
  .<fun i_1 ->
   let z_2 = 1 in
   let z_3 = (i_1 * z_2) in
   let z_4 = z_3 in
   let z_5 = (z_4 * z_4) in
   let z_6 = z_5 in
   let z_7 = (z_6 * z_6) in
   let z_8 = z_7 in
   let z_9 = (i_1 * z_8) in z_9>.
```

The generated code contains several unnecessary administrative redexes such `let z_6 = z_5`.

Unnecessary redexes can be avoided using abstract interpretation [26]. This involves making the monad aware of the abstract domain that the program uses to generate optimal code. Other than being written to work in the abstract domain, the staged program itself remains unchanged in its structure.

The use of abstract interpretation can be illustrated by declaring the following datatype (and auxiliary function), and modifying the body of the `power` function:

```
type ’a exp =
    Val of (’a, int) code
  | Exp of (’a, int) code

let retA a s k =
  match a with
  | Exp x -> .<let z = .~x in .~(k s (Val .<z>.))>.
  | _ -> k s a
let bindA a f =
  bind a (fun x -> bind (retA x) f)

let conc z =
  match z with
    Val x -> x
  | Exp x -> x

let power f (n, i) =
  if (n = 0) then
    retA (Val .<1>.)
  else if (even (n)) then
    bindA (f (n / 2, i)) (fun x ->
    retA (Exp .<.~(conc x) * .~(conc x)>.))
  else
    bindA (f (n-1, i)) (fun x ->
    retA (Exp .<.~i * .~(conc x)>.))

let pow5 =
    .<fun i -> .~((y_sm power) (5, .<i>.)
                  [] (fun s x -> conc x))>.
```

The type `exp` is an abstract domain that discriminates simple expressions (`Val`s) such as constants or variables that are cheap to duplicate from complex expressions (`Exp`s) that should not be duplicated, but instead bound to variables that can then be used multiple times without resulting in code-size explosion. While `Val` and `Exp` construct values of an abstract domain from a concrete domain (`int code` in this case), the function `conc` allows us to recover the concrete values from abstract values. This allows the construction of larger abstract values from smaller ones. Evaluating `pow5` yields

```
  .<fun i_1 ->
   let z_2 = (i_1 * 1) in
   let z_3 = (z_2 * z_2) in
   let z_4 = (z_3 * z_3) in
   let z_5 = (i_1 * z_4) in z_5>.
```

which has no unnecessary name bindings.

4. Case Studies and Experimental Results

The memoizing combinators presented above were used with several standard example DP algorithms and various timings were recorded. MetaOCaml sources for the main recurrences of these problems are presented in the Appendix A. They include all the example DP algorithms presented in Cormen et al’s standard textbook [11]. Most of the programs are around 10 lines long (excluding standard helper functions).

4.1 Example Algorithms

Our study used the following algorithms specialized to various first-stage parameters:

- `fwd` is forward algorithm for Hidden Markov Models. Specialization is for the size of the observation sequence equal to 7.
- `gib` is the Gibonacci function, a minor generalization of the Fibonacci function. Specialization is the 25th number in the sequence.
- `ks` is the 0/1 knapsack problem. Specialization is for 32 items with statically known weights.
- `lcs` is the longest common subsequence problem. Specialization is for strings lengths of 25 and 34.
- `obst` is the optimal binary search tree algorithm. Specialization is a leaf-node-tree of size 15.
- `opt` is the optimal matrix multiplication problem. Specialization is for 18 matrices.

4.2 Platform

To facilitate the gathering of timing results, a timing library is included in the MetaOCaml distribution.[2] Benchmarks were timed using the `Trxtime.timenew` function which takes a function from unit to an arbitrary type and executes it repeatedly until the total execution time exceeds one second. The number of iterations as well as the average time is reported to the user. Timings for the unstaged versions of the benchmark algorithms are computed in this way. Since the number of iterations for the unstaged functions is computed automatically by the timing function that ensures that the function is run enough times so that the total time is more than one second, this number of iterations is reported to the benchmark script and used when measuring execution times for staged and generated C programs. Using the same number of iterations helps ensure that the total number of runs outside MetaOCaml gives a total time that can still be measured reliably. The timings for C programs executed outside of the (Meta)OCaml runtime were collected using the bash shell `time` utility.

The experiments were performed by a fully-automated set of scripts that execute the benchmark algorithms and produce detailed reports, making our experiments easily reproducible and verifiable. The benchmark is available online [14]. The full set of raw data summarized in this paper as well as a detailed report on the execution platform can be found in a sample report named `v1.20-ex2.tex` in the `runs/` directory of the distribution of the benchmark.

Measurements were made on an Intel Pentium 4 machine (3055MHz clock speed) with 512 KB cache size and 1GB main memory running Linux 2.4.20-31.9. Objective Caml bytecode and native code compilers version 3.08, and gcc version 2.95.3 were used to compile source code files. Results reported in the paper were produced with version 1.20 of the benchmark.

4.3 Results

We have argued that the proposed combinators avoid the duplication problem that arises when we stage memoized functions. The goal of this section is to confirm experimentally that when the combinators are used there is no exponential increase in runtime performance or in the size of the generated code.

The execution times for the OCaml code generated by staged versions of the algorithms appear to be within a constant factor of the runtime of the unstaged OCaml versions. Second, if we translate the generated code to C using offshoring [16], the execution times appear to be within a constant factor of those for correspond-

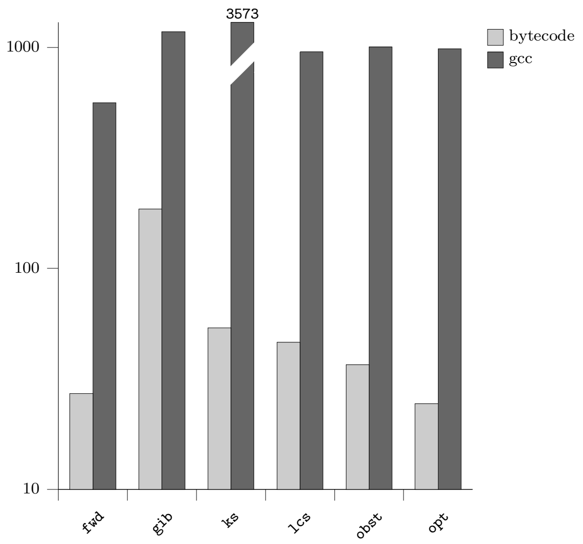

Figure 1. Speedup from staging and from staging plus offshoring.

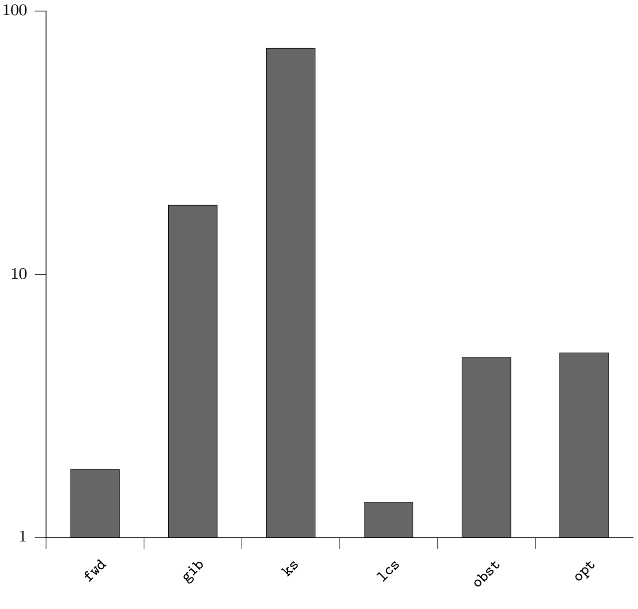

Figure 2. Speedups of generated C code over handwritten (unspecialized) C programs.

[2] MetaOCaml is a conservative extension of the OCaml compiler itself, and is thus "binary compatible" with it: it generates the same code for non-MSP programs, and executes them in the same runtime environment as OCaml 3.08.

ing hand-written (unspecialized) C implementation of the example algorithms.

The handwritten C implementations have no problem-specific optimizations. The array-based bottom-up memoizing approach was one that is suggested as an efficient approach for DP problems. In our code-generation approach, the implementer of a DP algorithm would not have to deal with memoization explicitly, but neither would our approach add any optimizations in the generated code that are not available to the handwritten implementations. We therefore feel that this comparison suggests that the approach proposed here is promising.

Both the generated OCaml and C programs are considerably faster than the unstaged OCaml versions. Figure 1 plots the speedups of staged (and offshored) OCaml implementation over unstaged DP programs executed using the bytecode compiler. The C programs achieve an order of magnitude increase in speedup because they are compiled using native code compiler for C (gcc), and because such compilers often perform more optimizations aimed at numerically intensive code. In the case of `ks` the gains are particularly high, mainly because the original benchmark does not make provisions for ensuring that the OCaml numerical comparison operators are monomorphically typed. As a result, OCaml uses a much slower polymorphic comparison operation.[3] Experimental results confirming this explanation, as well as a more detailed analysis of the effect of offshoring on performance can be found elsewhere [16].

The generated C programs were also faster than the handwritten (unspecialized) C programs (Figure 2).[4] We were not able to find generic C implementations that corresponded closely to the DP algorithms that we studied. Thus, we wrote array-based, bottom-up, memoized implementations of each of these algorithms. These were compared to the generated C programs (performing a role analogous to the unstaged implementations in OCaml).

Because staged programs are significantly faster than the unstaged memoized programs, the exponential increase in execution times due to the undoing of effects of memoization by staging (Section 3.3.1) does not occur with our combinators. Because the specialization sizes are all greater than ten, and because the exponential explosion would be a function of this parameter, the generate programs would have been at least 1024 (2 to the power of 10) times slower if there was any such effect.

The generated code is still quite large. As a result, the break-even points for the number of times the generated code must be used in order to amortize the cost of generation ranges from 50 to 300 times in the case of OCaml, and from 170 to 1800 times in the case of offshoring. The size of the generated code is an area where we expect future work to lead to significant improvements. Nevertheless, for all benchmark examples, experiments showed that we can increase the size of the problem until the number of characters in the generated code is less than 2 raised to the power of this size. This confirms experimentally that the combinators also avoid any exponential increase in the size of the generated code.

5. Related Work

MSP grew out of work in partial evaluation on two-level languages [39, 25]. Initially, two-level languages were intended only as a model for the internal language of an offline partial evaluator, and not as languages for writing multi-stage programs [48]. The work described in this paper can be carried out in the context of a two-level language extended with a run construct. Much of the work on MSP languages focused on developing type systems that can statically ensure that the run construct is safe to use (cf. [47]).

Liu and Stoller [32] studied the generation of efficient algorithms from DP problems specified as recurrence equations. Whereas our approach is largely extensional and uses a set of monadic and fixed-point combinators for all problems, their approach is intensional in that it focuses on the use of a variety of program transformation techniques to extract invariants from the recurrence equations and to exploit opportunities for incrementalization [31]. It will be interesting to see if the two approaches can be combined fruitfully.

McAdam [33] studied wrappers that can be joined with the standard fixed-point combinator to allow programmers to manipulate the workings of functions in an extensional manner. This work is carried out in an unstaged, imperative setting.

Ager, Danvy and Rohde [2] show how to derive string matchers specialized with respect to a particular pattern. Their work also focuses on speeding up the program generator, and controlling its space usage when the size and time complexity of the generated program is already under control.

De Meuter [13] outlines how monads and aspects can both be used to add memoization to programs with minimal change to the code. The approach taken is to define a non-standard semantics for an object-oriented language in Scheme. The semantics is discussed but not presented in the paper. Memoization using aspects has also been investigated by Aldrich [3] where instead of a specialized fixed-point combinator, *around* aspects are used to capture recursive function calls, and realize memoization.

As mentioned earlier in the paper, code duplication is addressed in the context of partial evaluation either by modifying the semantics of specialization (from the simple semantics of two-level languages) or by post-processing. For example, the FUSE system by Weise et al. [51] targets a functional subset of Scheme, and has a specializer that constructs graphs representing specialized programs. Nodes in these graphs represent computations, and outbound edges represent their uses. Each node having multiple outbound edges are let- bound to variables at code generation time, thus avoiding code duplication. FUSE also involves sophisticated graph-based transformations that rely on data flow analysis. This approach is typical in partial evaluation systems, in that it solves the problem at the meta-level, often using an analysis that looks for cases when generated code is being duplicated (by insertion into multiple different contexts in bigger code fragments). These two approaches are not well-suited for MSP: First, making any transformations or analysis part of the standard semantics for MSP languages defeats their goal of giving the programmer full control over staging. Second, giving the user access to the internal representations of quoted values is problematic as it means the programmer can no longer reason equationally about next-stage computations [45], and invalidates the safety guarantees provided by the static type systems currently available for multi-stage languages.

Consel and Danvy [10] showed that converting source programs into CPS leads to more effective partial evaluation. Several works that followed focused on improving either the performance or the clarity of the source code of the specializer [4, 29]. But it seems that none of these works indicate that CPS conversion improves opportunities for controlling code duplication using let insertion. A related idea to continuation-based partial evaluation was explored by Holst and Gomard [23], who do not CPS convert, but rather, rewrite each source program to push the syntactic context of every let expression into its body. So, in contrast to Consel and Danvy's idea of CPS converting the source program, the scope of the continuation is limited to each source program procedure.

Thiemann [49] uses monads to capture the notion of interleaved computations for code specialization and code generation in CPS-

[3] We thank Xavier Leroy for pointing this out.

[4] Both C programs were compiled with all available gcc optimization options, and the two best runs were used in the comparison.

based partial evaluation using a type-and-effect system, where effects are used to characterize the code generation, and allow for better binding-time analysis. This work uses monads to represent staged computation, whereas we use monads to express sharing in the second-stage computations.

In a study on using partial evaluation to optimize imperative programs, Debois also encounters the code duplication problem, and uses a bisimulation-based post-processing transformation that he calls "rewinding" to address it [12]. Debois notes that "a different direction is further studying the rewinding transformation, in particular finding out whether it can somehow be integrated with specialization. Currently, we may produce enormous residual programs only to cut them back down with rewinding, incurring correspondingly enormous time consumption." It will be interesting to see if the monadic combinator library approach proposed here can be used to avoid the need for a posteriori rewinding.

Moggi and Fagorzi [38] describe a monadic multi-stage metalanguage. In this work, monads are used to separate code generation from the computational effects and to give a simpler operational semantics for multistage programming languages. This is orthogonal to our work which uses monads as a programming technique.

In the Pan system by Elliott et al. [17], all function definitions are inlined and function applications are beta-reduced. This results in the generated code having a lot of replication. To remove this replication, they use CSE to identify the replicated program fragments and make let-bindings for them. In contrast to our approach of avoiding code duplication, they use sophisticated intensional analysis-based post-processing (whose correctness must be proved separately) to first generate large-sized code and then reduce its size. It will be interesting to see if the techniques presented here can be used to achieve the same runtime performance as their system.

Acar et al.[1] develop a sophisticated framework for selective memoization that uses a language based on a modal type system. Given the limited assumptions that we make about the notion of memoization that we address, we expect that the techniques we propose here are compatible with their more refined notions of memoization. It will be interesting to use their techniques to improve the performance of the generation phase.

6. Conclusion and Future Work

This paper proposes a systematic approach to avoiding code duplication when staging memoized functions. A generic combinator library supporting this approach is presented. To confirm experimentally that the library does achieve the goal of avoiding duplication, it is used to implement several standard DP algorithms including ones in standard algorithm textbooks. The generated implementations are competitive with hand-written (unspecialized) C programs.

Our use of monads gives rise to what may be the first example of two-level monads and two-level monadic fixed points in the literature. It seems reasonable to expect that there may be other instances where monads [36] and code [47] can be used synergistically.

Since the completion of the core of this work, the combinator library has found a variety of applications beyond the scope of memoized functions, including the generation of hardware circuits for FFT [26, 27] as well as specialized Gaussian elimination algorithms [9]. Even though FFT does not use memoization, a staged version of the recurrence would suffer from code duplication if the combinators presented here are not used. Thus, while the proposed combinator library addressed our initial problem of staging memoized functions, the scope of its utility seems to extend beyond this domain.

6.1 Future Work

In terms of theoretical exploration, we are interested in investigating the extent to which the two-level monadic fixed point approach can be used to define patterns of staging problems and their solution. It will also be interesting to see if the monad used here can help in the development of more expressive type systems for imperative MSP. In particular, there are currently no static type systems for imperative multi-stage languages that allow storing open terms in the store (and retrieving them). State of the art type systems use closedness types [6, 5]. Binding time analysis for imperative languages use the idea of regions (which do not have principal types), and still do not allow the storing of open terms in memory [15]. The combinator library presented in this paper uses a two-level monad that stores open values in an explicit state, and is fully expressible in a standard multi-stage type system with principal types [47, 7].

The proposed library provides the user with a mechanism for blocking specific sources of code duplication in a program, one by one. An interesting open question is whether all code duplication can be avoided *without* changing the semantics of the language. This open question can be viewed in two ways: First, there may be applications where code duplication poses difficulties for staging, and more sophisticated approaches (possibly based on other combinator libraries) may be needed. Second, some partial evaluators (like Tempo) are said to never introduce any code duplication. Such a property is not expressed in current static type systems for two level languages. In the future, it will be interesting to whether this gap can be bridged.

There are several important practical directions for future work. We are interested in understanding extent to which the combinators can be applied to domains other than DP. Examples of such domains include parsing [42], pretty-printing, and cryptography [34].

7. Acknowledgments

We thank Stephan Ellner, and Edward Pizzi for helpful comments on earlier drafts, and Xavier Leroy for discussing and giving us feedback on the differences in performance between the native code compiler and gcc. We thank the PEPM reviewers and Peter Sestoft for helpful comments.

References

[1] U. Acar, G. Blelloch, and R. Harper. Selective memoization. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM Press, Jan. 2003.

[2] M. S. Ager, O. Danvy, and H. K. Rohde. Fast partial evaluation of pattern matching in strings. In M. Leuschel, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '03)*, pages 3–9. ACM Press, 2003.

[3] J. Aldrich. Open modules: A foundation for modular aspect-oriented programming. Available online from `http://www-2.cs.cmu.edu/~aldrich/papers/tinyaspect.pdf`, Jan. 2003.

[4] A. Bondorf. Improving binding times without explicit CPS-conversion. In *The Proceedings of the ACM Conference on Lisp and Functional Programming (LFP '92)*, pages 1–10. ACM Press, June 1992.

[5] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 13(3):545–571, 2003. In [46].

[6] C. Calcagno, E. Moggi, and W. Taha. Closed types as a simple approach to safe imperative multi-stage programming. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP '00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, July 2000.

[7] C. Calcagno, E. Moggi, and W. Taha. ML-like inference for classifiers. In *Proceedings of the European Symposium on Programming (ESOP '04)*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, Mar. 2004.

[8] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In K. Czarnecki, F. Pfenning, and Y. Smaragdakis, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT International Conference on Generative Programming and Component Engineering (GPCE '03)*, volume 2830 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2003.

[9] J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In R. Glück and M. Lowry, editors, *Proceedings of the ACM International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes In Computer Science*. Springer-Verlag, Sept. 2005.

[10] C. Consel and O. Danvy. For a better support of static data flow. In R. Hughes, editor, *Proceedings of the ACM SIGPLAN/SIGARCH Functional Programming Languages and Computer Architecture (FPCA '91)*, volume 523 of *Lecture Notes in Computer Science*, pages 496–519. ACM Press, Springer-Verlag, Aug. 1991.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 14th edition, 1994.

[12] S. Debois. Imperative program optimization by partial evaluation. In N. Heintze and P. Sestoft, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '04)*, pages 113 – 122. ACM Press, Aug. 2004.

[13] W. DeMeuter. Monads as a theoretical foundation for AOP. In *Proceedings of the International Workshop on Aspect-Oriented Programming at ECOOP*, volume 1357 of *Lecture Notes in Computer Science*, page 25. Springer-Verlag, 1997.

[14] Dynamic Programming Benchmark. Available online from `http://www.metaocaml.org/examples/dp`, 2005.

[15] D. Dussart, R. Hughes, and P. Thiemann. Type specialization for imperative languages. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 204–216. ACM Press, June 1997.

[16] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, and W. Taha. Implicitly heterogeneous multi-stage programming. In R. Glück and M. Lowry, editors, *Proceedings of the ACM International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes In Computer Science*. Springer-Verlag, September 2005.

[17] C. Elliott, S. Finne, and O. de Moore. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, May 2003. In [46].

[18] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In S. Abramsky, editor, *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA '01)*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, May 2001.

[19] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI '93)*, pages 237–247. ACM Press, June 1993.

[20] M. Frigo. A Fast Fourier Transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 169–180. ACM Press, May 1999.

[21] W. L. Harrison and S. N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE International Conference on Computer Languages (ICCL '98)*. IEEE Computer Society Press, May 1998.

[22] J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.

[23] C. K. Holst and C. K. Gomard. Partial evaluation is fuller laziness. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*, pages 223–233. ACM Press, June 1991.

[24] R. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[26] O. Kiselyov, K. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In *Proceedings of the ACM SIGBED International Workshop on Embedded Software (EMSOFT '04)*, pages 249–258. ACM Press, Sept. 2004.

[27] O. Kiselyov and W. Taha. Relating FFTW and split radix. In Z. Wu, M. Guo, C. Chen, and J. Bu, editors, *Proceedings of the International Conference on Embedded Software and Systems (ICESS '04)*, volume 3605 of *Lecture Notes in Computer Science*. Springer-Verlag, Dec. 2004.

[28] J. Launchbury and S. L. Peyton Jones. State in Haskell. *LISP and Symbolic Computation*, 8(4):293–342, 1995.

[29] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *Proceedings of the ACM Conference on Lisp and Functional Programming (LFP '94)*, pages 227–238. ACM Press, June 1994.

[30] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL '95)*, pages 333–343. ACM Press, Jan. 1995.

[31] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*, pages 73–82. ACM Press, Jan. 2000.

[32] Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. *Higher-Order and Symbolic Computation (HOSC)*, 16(1-2):37–62, Mar. 2003.

[33] B. McAdam. Y in practical programs (extended abstract). Unpublished manuscript.

[34] N. McKay and S. Singh. Dynamic specialization of XC6200 FPGAs by partial evaluation. In R. W. Hartenstein and A. Keevallik, editors, *Proceedings of the International Workshop on Field-Programmable Logic and Applications (FPL '98)*, volume 1482 of *Lecture Notes in Computer Science*, pages 298–307. Springer-Verlag, Aug. 1998.

[35] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`.

[36] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS'89)*, pages 14–23. IEEE Computer Society Press, June 1989.

[37] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[38] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In A. Gordon, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS '03)*, volume 2620 of *Lecture Notes in Computer Science*. Springer-Verlag, Apr. 2003.

[39] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[40] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `http://www.cse.ogi.edu/tech-reports/`.

[41] T. Sheard, Z.-e.-A. Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Proceedings of the ACM USENIX Conference on Domain-Specific Languages (DSL'99)*, pages 81–94. ACM Press, October 1999.

[42] M. Sperber and P. Thiemann. The essence of LR parsing. In

Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '95), pages 146–155. ACM Press, June 1995.

[43] E. Sumii and N. Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[44] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [40].

[45] W. Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00)*. ACM Press, Jan. 2000.

[46] W. Taha, editor. *Journal of Functional Programming, Special Issue on 'Semantics, Applications, and Implementation of Programming Generation (SAIG)'*, volume 13(3). Cambridge University Press, May 2003.

[47] W. Taha and M. F. Nielsen. Environment classifiers. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM Press, Jan. 2003.

[48] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM '97)*, pages 203–217. ACM Press, 1997.

[49] P. Thiemann. Continuation-based partial evaluation without continuations. In R. Cousot, editor, *Proceedings of the International Symposium on Static Analysis (SAS '03)*, volume 2694 of *Lecture Notes in Computer Science*, pages 366–382. Springer-Verlag, 2003.

[50] P. Wadler. The essence of functional programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '92)*, pages 1–14. ACM Press, Jan. 1992.

[51] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Proceedings of the ACM SIGPLAN/SIGARCH Conference on Functional programming languages and computer architecture (FPCA '91)*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191. Springer-Verlag New York, Inc., Aug. 1991.

A. Programs used in the test suite

We give below the code fragments corresponding to the core recursive algorithms in each of the DP problems that were used in the test suite. These fragments illustrate the concise high-level style that can be used by a programmer type implement basic dynamic programming algorithms. The use some unlisted simple helper functions such as `for_lm` (to produce a list of results) and `sum_lms` (to sum up elements in a list).

```
(* Forward algorithm *)
let rec alpha_ms  f
                  ((time,state, (hma, hmb, pi),
                    stateSize), obs) =
  match time with
    1 -> let p1 = pi.(state) in
         ret .<(p1 *. (.~hmb).(state).((.~obs).(1)))>.
  | _ -> bind (for_lm 1 stateSize (fun kstate ->
                 bind
                   (f ((time-1, kstate, (hma, hmb, pi),
                        stateSize), obs))
                                         (fun r1 ->
                 let p2 = (hma).(kstate).(state) in
                 ret (.<.~r1 *. p2>.)))) (fun r2 ->
         bind (sum_lms r2)               (fun r3 ->
         ret (.<(.~hmb).(state).((.~obs).(time)) *.
                .~r3>.)))

(* Gibonacci *)
let gib_ms  f (n, (x, y)) =
  match n with
  | 0 -> ret  x
  | 1 -> ret  y
  | _ -> bind (f  ((n-2), (x, y))) (fun r1  ->
         bind (f  ((n-1), (x, y))) (fun r2 ->
         ret .<.~r2 + .~r1>.))

(* Longest Common Subsequence *)
let lcs_mks f ((i,j), (x,y)) =
  if (i=0 || j=0) then ret .<0>.
  else
   bind (f ((i-1, j-1),(x,y))) (fun r1 ->
   bind (f ((i, j-1),(x,y)))   (fun r2 ->
   bind (f ((i-1, j),(x,y)))   (fun r3 ->
   ret
    .<if ((.~x).(i) = (.~y).(j)) then .~(r1) + 1
      else max .~(r2)  .~(r3)>.)))

(* 0/1 Knapsack *)
let ks_sm  f ((i,w,wt), vl) =
  match (i,w) with
    (0,_) -> ret .<0>.
  | (_,0) -> ret .<0>.
  | (ni,nw) ->
     if (wt.(i) > nw) then
       f ((i-1, nw, wt), vl)
     else
      bind (f ((i-1, nw - wt.(i), wt), vl))
           (fun r1 ->
      bind (f ((i-1, nw, wt), vl))
           (fun r2 ->
      ret (.<max ((.~vl).(i) + .~r1) .~r2>.)))

(* Optimal Binary Search Tree *)
let obst_sm  f ((j, k), p) =
  if (j = k) then ret .<(.~p).(j)>.
  else
   if (k < j) then ret .<(0.0)>.
   else
     bind (for_lm j k (fun x -> ret .<(.~p).(x)>.))
          (fun r0 ->
     bind (msum_ls_memo_y ((r0, j, k),0))
          (fun r1 ->
     bind (for_lm j k (fun i ->
            bind (f ((j, i-1), p))
                 (fun r2 ->
            bind (f ((i+1, k), p))
                 (fun r3 ->
            retN .<(.~r2 +. .~r3)>.))))
          (fun r4 ->
     bind (min_lms r4)
          (fun r5 ->
      ret .<.~r1 +. .~r5>.))))

(* Optimal Matrix Multiplication Order *)
let optmult_sm f ((i,j),p) =
  if i=j then ret .<0>.
  else bind (for_lm i (j-1)
             (fun k ->
              bind (f((i,k),p))   (fun r1 ->
              bind (f((k+1,j),p)) (fun r2 ->
              retN (.<.~r1 + .~r2 +
                      (.~p).(i-1) *
                      (.~p).(k)   *
                      (.~p).(j)>.)))))
       (fun r3 -> min_lms r3)
```