Staging Dynamic Programming Algorithms

Kedar Swadi
Rice University, Houston, TX
kswadi@rice.edu

Walid Taha
Rice University, Houston, TX
taha@rice.edu

Oleg Kiselyov
Fleet Numerical Meteorology and Oceanography Center, Monterey, CA 93943, USA
oleg@okmij.org

Abstract

Applications of dynamic programming (DP) algorithms are numerous, and include genetic engineering and operations research problems. At a high level, DP algorithms are specified as a system of recursive equations implemented using memoization. The recursive nature of these equations suggests that they can be written naturally in a functional language. However, the requirement for memoization poses a subtle challenge: memoization can be implemented using monads, but a systematic treatment introduces several layers of abstraction that can have a prohibitive runtime overhead. Inspired by other researchers' experience with automatic specialization (partial evaluation), this paper investigates the feasibility of explicitly staging DP algorithms in the functional setting. We find that the key challenge is code duplication (which is automatically handled by partial evaluators), and show that a key source of code duplication can be isolated and addressed once and for all. The result is a simple combinator library. We use this library to implement several standard DP algorithms including ones in standard algorithm textbooks (e.g. Introduction to Algorithms by Cormen et al. [10]) in a manner that is competitive with hand-written C programs.

Our combinator library has been useful for a variety of applications beyond the scope of dynamic programming, including the generation of hardware circuits.

Keywords Dynamic programming, Staging, Program Specialization, Multi-stage Programming

1. Introduction

Abstraction mechanisms such as functions, classes, and objects can reduce development time and improve software quality. But such mechanisms often have an cumulative runtime overhead that can make them unattractive to programmers. Partial evaluation [23] 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 abstractions 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 what abstractions are eliminated in what stage is Multi-stage Programming (MSP) [46, 43]. MSP languages provide the programmer with a quotation mechanism that can be used specifically for

ICFP '05 Sep 26-28, Tallinn, Estonia.

constructing code fragments. Quotation mechanisms designed specifically for this purpose make it possible to develop both formal reasoning principles (cf. [44, 43]) and static type systems that guarantee that all generated programs would be type-safe (cf. [45]).

An important challenge for MSP is demonstrating its utility as a practical programming technique for implementing non-trivial algorithms. As a step in this direction, we consider the domain of dynamic programming (DP) (cf. [10, Chapter 16]). DP finds applications in areas such as optimal job scheduling, speech recognition using Markov models, and finding similarities in genetic sequences. Informally, the technique can be described as follows:

> "[DP], like the divide-and-conquer method, solves problems by combining the solutions to subproblems. Divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share sub-subproblems." [10, Page 301].

It is easy to see functional languages are well-suited for direct – although not necessarily efficient – implementations of DP algorithms.

Can staging give us back the desired performance?

1.1 Technical Problem

A key technical problem that arises in partial evaluation and when writing MSP programs is code duplication. In the partial evaluation literature, the problem has been addressed in a variety of different ways. For example, the FUSE system by Weise et al. [48] targets a functional subset of Scheme, and has a specializer that constructs graphs representing specialized programs. Nodes in these graphs represent computations, and out-bound edges represent their uses. Each node having multiple out-bound edges are let- bound to variables at code generation time, thus avoiding code duplication. Their system also involves sophisticated transformations graph-based 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).

The partial evaluation solutions cannot be used for MSP. First, making any transformations or analysis part of the standard semantics for MSP languages would defeat their purpose, as it means that the user has less direct control over staging. Second, giving the user access to the internal representations of quoted values would also be problematic: it would mean that the programmer can no longer reason equationally about next-stage computations [44], and would also invalidate the safety guarantees provided by the static type systems currently available for multi-stage languages.

1.2 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 had not been done before. In our previous, unstructured attempts to stage dynamic programming programs, identifying the sources of code duplication was difficult. Using the combinator approach presented here, it is possible to precisely pin-point the source of code-duplication that arises when staging memoized versions of functions, and to address it once and for all. When we make the underlying monads explicit, it also becomes possible to stage the monadic operators themselves so that all intermediate results are named, and code duplication is no longer possible.

Compared to all previous approaches to controlled naming and the use of monads in program generation, the key technical novelty in our approach is that it does not require any language extension. For example, Hatcliff and Danvy [21] 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 show a monad that combines both continuations and state and is *expressible at the object-level* in an multi-stage calculus. The calculus we use does not allow pattern matching on generated code [44], so our approach guarantees that we preserve the equational

properties of generated code fragments. By implementing all the combinators (and applications that use them) in MetaOCaml, we demonstrate that using a standard, practical static type system [45, 7] is sufficient to allow the programmer to implement several different approaches to controlling code duplication.

To test the utility of these combinators both from the programming point of view and from the point of view of the performance of the resulting implementation, 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, and that the performance of the resulting implementations is comparable to that of hand-written C implementations of these algorithms.

2. Multi-stage Programming

MSP languages [46, 43] 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 and the representation of programs, are hidden from the programmer (cf. [43]). The following minimal example illustrates MSP programming in an extension of OCaml [28] called MetaOCaml [8, 34]:

```
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, as well as 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, because 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, it is compiled and evaluated, 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 would have had to pay every time `power3` is used.

We emphasize that while MSP language constructs have a direct resemblance to LISP and Scheme's quasi-quote and eval mechanisms, the technical novelty of these languages lies in automatically avoiding accidental variable capture, supporting equational reasoning inside quotations [44], and statically ensuring that any generated programs are well-typed (c.f. [45]). This comes at the cost of not providing a mechanism for taking apart code fragments once they are constructed [44].

3. A Combinator Library for Dynamic Programming

In what follows we gradually develop the generic set of combinators that we have used for staging the dynamic programming algorithms. The simplest Y combinator can be expressed as:

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

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

Because memoization requires a notion of state, we must explicitly pass a state:

```
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)
```

A memoizing version would have an instance of this type, but would manipulate the state, which can be assumed to be the memoization table:

```
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))
```

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` that we want to memoize will be called with several arguments, and not all of them are needed to determine the key into the memoization 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 helpful 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, the DP algorithm will consist of a collection of different functions, many of which need to be memoized using separate memoization 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 Y combinator must be able to correctly lookup (or update) a state depending on which particular function it is called with. To allow this, we make another variation:

```
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))
```

This combinator takes two more arguments: `get`, and `set`, where `get` is used to identify the component of the state that the current function `f` uses, and `set` is used to update the correct state component that is used by `f`.

3.1 Staging the Y combinator

The combinators presented above introduce several layers of abstraction that are necessary to generalize these combinators for reasonable applications. But these abstractions also add a significant runtime cost.

Staging can help us avoid these costs at runtime. But direct staging does not work. A key benefit from the monadic presentation is that it allows us to explain the source of this difficulty clearly. Based on our own experience on trying to stage these algorithms effectively when they were written without using these abstractions, this can be quite difficult. The formulation above allows us to trace the problem to the behavior of the memoization scheme, and illustrate it concrete in the context of `y_memo`. In particular, the parameter `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 never problematic. In the staged setting, however, `r1` is no longer a simple value but a code fragment. When each of these code fragments is returned and then inserted into a bigger code fragment, the result will be two copies of the code fragment. A sequence of these events leads to code explosion.

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 the fragment. In the combinator above, however, direct let-insertion is not possible, because the pair that we return is in fact a partially static data-structure. This problem is addressed by another standard technique from partial evaluation, namely converting the source program into continuation-passing style. A staged, continuation-passing style version of the above combinator, `y_cps` allows 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  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>.)>.))
```

In this variation, the combinator now works with functions that take another parameter `k`, which is the continuation for the computation. As done earlier, we first check if the result has already been computed and stored in the state. If so, we pass the state `s` and the result `r` to `k`. If not, we perform the computation, but pass it a newly constructed continuation, which takes the new state `s1` and the result `v`. This continuation proceeds by first binding `v` to a fresh variable `r1`, and associates `key x` with this variable. As a result, any future lookup into the state results in a variable `r1` being passed to the continuation rather than a (possibly large) value `v`.

3.2 Monadic Encapsulation

Especially when considering the details of its type, expecting the programmer to be able to use the above Y combinator might seem impractical. Fortunately, the underlying structure is a standard monad, and is an instance of the type 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 annotation , this monad is sufficient for staging many DP algorithms where the only source of code duplication is memoization. All we have to do to DP recurrence equations is to convert them into monadic style and use open recursion. During this process, we do not need 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)
              = fun s k ->
                 a s (fun s' x -> .<let z = .~x in .~(f .<z>. s' k)>.)
```

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 Y combinator as well as the staged monadic operators will be done in the first stage. Thus, the overhead of recursion, all operations on the memoization table, and the applications of the return and bind operations will not be incurred in the generated code.

3.3 A-normal form

The above operators can be used to produce code in an approximation of A-normal form [18]. We illustrate the two cases using 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 output of this code fragment, `pow5` has the following form:

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

If we replace `ret` with `retN`, the generated code for `pow5` is now:

```
  .<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>.
```

If we use the `bindN`, 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 above two-level monad can generate unnecessary administrative redices such `let z_6 = z_5`, as it names all arguments to return and bind. Eliminating such redices does not require intensional analysis, and can be avoided using abstract-interpretation [24]. This would mean making the monad aware of the abstract domain that the problem uses to generate optimal code. The program itself, other than being written to work in the abstract domain, remains unchanged in its structure. This is achieved by declaration 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,bindA)  =
  let retA a s k =
    match a with
     Exp x -> .<let z = .~x in
                .~(k s (Val .<z>.))>.
    | _ -> k s a in
  let bindA a f =
    bind a (fun x -> bind (retA x) f) in
  (retA,bindA)

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))>.
```

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. Generating administrative variable-to-variable bindings can also be avoided using the same technique.

We thus notice that the monadic formulation is very expressive, and allows us to easily change the form of our generated code without having to make extensive changes to our generators.

4. Case Studies and Experimental Results

In this section we describe the example DP algorithms that we studied, and report on performance measurements on the generated code. MetaOCaml sources for the main recurrences of these problems are presented in the Appendix A. These examples include all the distinct DP algorithms discussed in Cormen et al's standard textbook [10]. Most of the programs are around 10 lines long (excluding standard helper functions).

Name	Unstaged	Generate	Compile	Staged Run	Speedup	BEP
forward	1.992e-01	1.125e+00	1.684e+01	7.339e-03	2.714e+01x	94
gib	1.151e-01	2.597e-01	5.067e+00	6.208e-04	1.854e+02x	47
ks	5.234e+00	3.851e+01	4.038e+02	9.716e-02	5.387e+01x	87
lcs	5.720e+00	4.691e+01	6.057e+02	1.235e-01	4.633e+01x	117
obst	3.747e+00	2.672e+01	2.862e+02	1.023e-01	3.663e+01x	86
opt	7.360e-01	1.243e+01	1.381e+02	3.011e-02	2.445e+01x	214

Figure 1. Speedup gained by staging

4.1 Test Cases and Platform

Measurements were gathered[1] for implementations of:

- `forward` is forward algorithm for Hidden Markov Models. Specialization size (size of the observation sequence) is 7.
- `gib` is the Gibonacci function, a minor generalization of the Fibonacci function. This is not a standard DP algorithm, but is useful for illustrating many of the issues that arise in staging DP algorithms. Specialization is for $n = 25$.
- `ks` is the 0/1 knapsack problem. Specialization is for size 32 (number of items).
- `lcs` is the least common subsequence problems. Specialization is for string sizes 25 and 34 for the first and second arguments, respectively.
- `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.

The full program sources used for all the measurements, as well as the automated performance measurement scripts can be downloaded from `http://www.metaocaml.org/examples/dp` [13].

Does staged memoization allow effective staging of DP problems? Figure 1 reports timings for each of the benchmarks. The first column, **Unstaged** is the average[2] time (in milliseconds) for running the unstaged version of the implementation of each problem in byte-code compiled MetaOCaml. In the case of the Gibonacci function, this would be `gib_ksp`. **Generate** is the average[3] time need to run the first-stage of the staged version. **Compile** is the average time needed to compile the program generated by the first stage into OCaml bytecode. **Staged Run** is the average execution time for the generated and compiled program. **Speedup** is Unstaged divided by Staged Run. **BEP** is the break-even point, which is the number of times that the generated program must be run to amortize the costs of generation and compilation.

The data indicates that some speedups and some potentially reasonable BEP values are attainable. Inspecting the code confirms that the code explosion problem is indeed addressed by staged memoization.

To compare our technique meaningfully with implementations of DP algorithms with memoization that are written imperatively, we used the "offshoring" technique [15] that translates our generated code to C programs. Figure 2 lists the performance we obtain from translating the generated code to C programs, compiling these

[1] Numbers were measured on a i386 3055.417MHz CPU machine 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.

[2] We use the `Trxtime.timenew` timing function found in the MetaOCaml standard library to measure execution times. This function repeatedly executes the function to be timed till the cumulative execution time exceeds 1 second. `Trxtime.timenew` reports the number of iterations and the average execution time per iteration for the argument function.

[3] To measure execution times for **Generate**, **Compile** and **Staged Run**, we first obtain the number of iterations from measuring Unstaged. Then we use the `Trxtime.time` utility to measure average execution times for each of these columns using the number of iterations $\times$ 10 obtained from `Trxtime.timenew` used earlier for the **Unstaged run**

Name	Unstaged	Generate	Compile	Staged Run	Speedup	Speedup(S)	BEP
forward	1.992e-01	1.125e+00	3.304e+01	3.548e-04	5.615e+02x	2.069e+01x	172
gib	1.151e-01	2.597e-01	1.671e+01	9.775e-05	1.178e+03x	6.351e+00x	148
ks	5.234e+00	3.851e+01	8.176e+03	1.465e-03	3.573e+03x	6.632e+01x	1570
lcs	5.720e+00	4.691e+01	6.447e+03	5.981e-03	9.563e+02x	2.064e+01x	1137
obst	3.747e+00	2.672e+01	5.290e+03	3.723e-03	1.006e+03x	2.747e+01x	1421
opt	7.360e-01	1.243e+01	4.560e+02	7.477e-04	9.844e+02x	4.027e+01x	638

Figure 2. Speedups from offshoring with `gcc`

Name	-O0	-O1	-O2	-O3	best
forward	5.881e+00	2.500e+00	1.706e+00	1.611e+00	1.812e+00
gib	2.274e+00	1.829e+01	1.933e+01	1.976e+01	1.829e+01
knapsack	8.724e+01	7.332e+01	7.265e+01	7.224e+01	7.224e+01
lcs	2.373e+00	1.556e+00	1.250e+00	1.256e+00	1.361e+00
obst	4.519e+00	4.821e+00	3.157e+00	3.115e+00	4.821e+00
opt	3.840e+00	4.441e+00	5.200e+00	5.200e+00	5.033e+00

Name	-O0	-O1	-O2	-O3
forward	9.123e-01	9.429e-01	9.363e-01	9.363e-01
gib	9.978e-01	1.028e+00	1.026e+00	1.026e+00
knapsack	2.732e-01	3.352e-01	3.171e-01	3.171e-01
lcs	2.278e-01	2.523e-01	2.347e-01	2.347e-01
obst	2.435e-01	4.018e-01	3.500e-01	3.500e-01
opt	5.672e-01	7.436e-01	7.271e-01	7.271e-01

Figure 3. Ratios of time and image size of handwritten code to generated code

C programs and then executing[4] the resulting binaries. The first three columns in this table are as in Figure 1. **Compile** lists the time to offshore and compile the generated code, **Staged Run** lists the time to execute the offshored code, **Speedup** is Unstaged divided by Staged Run, **Speedup(S)** represents the speedup of staging and offshoring over simply staging. Finally, **BEP** is the break-even point.

How do the generated C programs compare to hand-written ones? We were not able to find generic C implementations that corresponded closely to the basic DP problems that we studied. Thus, we wrote array-based, bottom-up, memoized implementations of each of these problems. Figure 3 gives the ratios of both the runtimes and the binary size for the handwritten C programs versus the generated programs using the `-O0`, `-O1`, `-O2`, and `-O3` optimization flags for `gcc`. For example, we observed that under optimization `-O0`, the binary for the compiled handwritten C program executed 5.88 times slower than the binary produced from our generated (and offshored) C program, but took up 0.91 times the corresponding space. The last column **best** lists the ratios of the best-compiled handwritten code to the best-compiled generated code. In all cases we noticed that the generated code runs faster, even when the size of the executable is not substantially larger. Generally, the optimization settings did not change the sizes of the binaries for the generated programs significantly.

[4] Each program is run multiple times using the number of iterations obtained from the measurements for the **Unstaged** run in Figure 1. The `time` bash shell function in Unix is used to measure the time for each program. This time is then divided by the number of iterations to obtain an average execution time per program. The benchmark measurement script [13] also lists the initial and final machine load to estimate and account for variances in the measurements given by the `time` utility.

5. Related Work

Monads have been shown to be useful for expressing effectful computations in purely functional languages [26], and for structuring denotational semantics [35, 37, 17], interpreters [29], compilers [20], and partial evaluators [42, 21] and program generators [19, 40].

Multi-stage programming (MSP) grew out of work in partial evaluation on two-level languages [38, 23]. Initially, however, 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 [46]. 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 focus on developing type systems that can statically ensure that the run construct is safe to use (cf. [45]).

Liu and Stoller [31] have 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 more 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 incrementalisation [30]. It would be interesting to see if the two approaches can be combined fruitfully.

McAdam [32] studies wrappers that can be joined with the standard Y 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 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 [12] illustrates how monads and aspects could both be used to add memoization to programs with minimal change to the code. Our work shows that this idea does not carry over directly to the multi-level setting, but also that this problem can be circumvented. Memoization using aspects has also been investigated by Aldrich [3] where instead of a specialized Y combinator, *around* aspects are used to capture recursive function calls, and realize memoization. In future work, we intend to investigate more closely this line of work in the AOP literature and our present work.

Consel and Danvy [9] CPS convert source programs and show that this leads to more effective partial evaluation. Many works that followed focused on improving either the performance or the clarity of the source code of the specializer [4, 27]. However, it seems that none of these works seem to 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 [22], 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[47] uses monads to capture the notion of interleaved computations for code specialization and code generation in CPS-based partial evaluation using a a type-and-effect system, where effects are used to characterize the code generation, and allow for better binding-time analysis. This use of monads is different from ours: Thiemann uses monads to represent staged computation, where as we use monads to express sharing in the second-stage computations using explicit let-statements.

Moggi and Fagorzi [36] 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 purely as a programming technique.

In the Pan system by Conal et al. [16], 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 explosion, 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.

Acar et al.[1] develop a sophisticated framework for selective memoization that uses a modal type system based language. Given the limited assumption 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 in situations where the runtime performance of the first generation is a concern.

6. Conclusion and Future Work

This paper illustrates the feasibility of explicitly staging DP algorithms in the functional setting. To do so, we propose a combinator library and a systematic approach to addressing the key challenge of code duplication. We use this library to implement several standard DP algorithms including ones in standard algorithm textbooks in a manner that is competitive with hand-written C programs.

In addition to illustrating the generality of the new technique, our used 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 there may be several instances where monads and static code types based on the next modality from linear temporal logic [11] 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 dynamic programming, including the generation of hardware circuits [24, 25]. Even though FFT does not use memoization, the a staged version of the recurrence would suffer from code explosion if the combinators presented here are not used. Thus, while the study of DP algorithms helped us identify this combinator library, the scope of its utility extends 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 multi-stage programming. In particular, there are currently are 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 [14]. 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 principle types [45, 7].

There are several important practical directions for future work. In particular, we are interested in understanding extent to which these ideas can be applied to domains other than dynamic programming. Examples of such domains include parsing [41], pretty-printing, and cryptography [33].

7. Acknowledgments

We would like to thank Stephan Ellner, Emir Pašalić, and Edward Pizzi for helpful comments on earlier drafts of this paper, and Xavier Leroy for discussing and giving us feedback on the differences in performance between the native code compiler and gcc.

References

[1] Umut Acar, Guy Blelloch, and Robert Harper. Selective memoization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2003.

[2] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast partial evaluation of pattern matching in strings. In *Michael Leuschel, editor, Proceedings of the 2003 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 3–9. ACM Press, 2003.

[3] Jonathan Aldrich. Open modules: A foundation for modular aspect-oriented programming. http://www-2.cs.cmu.edu/ aldrich/papers/tinyaspect.pdf, January 2003.

[4] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, 1992.

[5] Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. Closed types for a safe imperative MetaML. *Journal of Functional Programming*, 2003. To appear.

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

[7] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. Ml-like inference for classifiers. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[8] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[9] C. Consel and O. Danvy. For a better support of static data flow. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 496–519. Springer Verlag, June 1991.

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

[11] Rowan Davies. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)*, pages 184–195, New Brunswick, 1996. IEEE Computer Society Press.

[12] W. DeMeuter. Monads as a theoretical foundation for AOP. In *International Workshop on Aspect-Oriented Programming at ECOOP*, page 25, 1997.

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

[14] D. Dussart, R.J.M. Hughes, and P. Thiemann. Type specialization for imperative languages. In *Proceedings of the International Conference on Functional Programming (ICFP), Amsterdam, The Netherlands, June 1997*, pages 204–216. New York: ACM, 1997.

[15] Jason Eckhardt and Roumen Kaiabachev. Offshoring: Representing C and Fortran90 in OCaml, October 2004. Paper presented at the First MetaOCaml Workshop.

[16] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In *SAIG*, pages 9–27, 2000.

[17] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In *Typed Lambda Calculi and Applications: 5th International Confe rence (TLCA)*, volume 2044 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2001.

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6) of *SIGPLAN Notices*, pages 237–247. ACM Press, New York, 1993.

[19] Matteo Frigo. A Fast Fourier Transform compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 169–180, 1999.

[20] William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE International Conference on Computer Languages*, 1998.

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

[22] C. K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 223–233. ACM Press, 1991.

[23] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[24] Oleg Kiselyov, Kedar Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *the International Workshop on Embedded Software (EMSOFT '04)*, Lecture Notes in Computer Science, Pisa, Italy, 2004. ACM.

[25] Oleg Kiselyov and Walid Taha. Relating fftw and split radix. In *the International Conference on Embedded Software and Systems (ICESS '04)*, Lecture Notes in Computer Science, Hangzhou, China, 2004. Springer-Verlag.

[26] John Launchbury and Simon L. Peyton Jones. State in haskell. *LISP and Symbolic Computation*, 8(4):293–342, 1995. pldi94.

[27] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *1994 ACM Conference on Lisp and Functional Programming, Orlando, Florida, June 1994*, pages 227–238. New York: ACM, 1994.

[28] Xavier Leroy. Objective Caml, 2000. Available from `http://caml.inria.fr/ocaml/`.

[29] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of 22nd ACM Symposium on Principles of Programming Languages*, pages 333–343, New York, January 1995. ACM Press.

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

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

[32] Bruce McAdam. Y in practical programs (extended abstract). Unpublished manuscript.

[33] Nicholas McKay and Satnam Singh. Dynamic specialization of XC6200 FPGAs by partial evaluation. In Reiner W. Hartenstein and Andres Keevallik, editors, *International Workshop on Field-Programmable Logic and Applications*, volume 1482 of *Lecture Notes in Computer Science*, pages 298–307. Springer-Verlag, 1998.

[34] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.

[35] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[36] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *In A.D. Gordon, editor, Foundations of Software Science and Computational Structures - FOSSACS*, volume 2620, pages 358–374. Springer Verlag, 2003.

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

[38] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[39] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.

[40] Tim Sheard, Zine-El-Abidine Benaissa, and Emir Pasalic. DSL implementation using staging and monads. In *Domain-Specific Languages*, pages 81–94, 1999.

[41] Michael Sperber and Peter Thiemann. The essence of LR parsing. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 146–155, La Jolla, California, June 1995.

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

[43] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [39].

[44] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)*, Boston, 2000. ACM Press.

[45] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.

[46] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.

[47] Peter Thiemann. Continuation-based partial evaluation without continuations. In *Static Analysis: 10th International Symposium, R. Couset (Ed.)*, pages 366–382. Springer-Verlag Heidelberg, 2003.

[48] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 165–191. Springer-Verlag New York, Inc., 1991.

A. Programs used in the test suite

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

```
(* 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)
```