Practical Aspects of Multi-stage Programming

Jason L. Eckhardt[1], Roumen Kaiabachev[1], Kedar N. Swadi[1], Walid Taha[1], and Oleg Kiselyov[2]

[1] Rice University, Houston, TX, USA
{jle,roumen,kswadi,taha}@cs.rice.edu
[2] Fleet Numerical Meteorology and Oceanography Center, Monterey, CA 93943
oleg@okmij.org

Abstract. High-level languages offer abstraction mechanisms that can reduce development time and improve software quality. But abstraction mechanisms often have an accumulative runtime overhead that can discourage their use. Multi-stage programming (MSP) languages offer constructs that make it possible to use abstraction mechanisms without paying a runtime overhead. This paper studies applying MSP to implementing dynamic programming (DP) problems. The study reveals that staging high-level implementations of DP algorithms naturally leads to a code explosion problem. In addition, it is common that high-level languages are not designed to deliver the kind of performance that is desirable in implementations of such algorithms. The paper proposes a solution to each of these two problems. Staged memoization is used for code explosion, and a kind of "offshoring" translation is used to address the second. For basic DP problems, the performance of the resulting specialized C implementations is almost always better than the hand-written generic C implementations.

1 Introduction

Abstraction mechanisms such as functions, classes, and objects can reduce development time and improve software quality. But such mechanisms often have an accumulative runtime overhead that can make them unattractive to programmers. The key goal of multi-stage programming (MSP) languages [32] is to encourage programmers to use abstraction mechanisms. This is achieved by offering the programmer a mechanism for ensuring that the overhead of an abstraction can be paid in a stage earlier than the main phase of the computation. While preliminary studies on MetaOCaml (an MSP extension of a functional language) have shown that staging can be used to improve the performance of OCaml programs [6], in some cases improvement was not as high as is demonstrated by the Tempo [27] partial evaluator for C programs. If we are to show that MSP is relevant to mainstream programming, there is a need to show that MetaOCaml can be used to generate artifacts with acceptable performance when compared with implementations written in languages like C.

As a first step in this direction, this paper focuses on dynamic programming (DP) problems (c.f. [8, Chapter 16]). DP finds applications in areas such as op-

timal job scheduling, speech recognition using Markov models, and finding similarities in genetic sequences. Two problems are encountered in directly applying MSP to DP problems. The first is algorithmic, and the second is technological.

Problem 1: Staging DP algorithms easily leads to code explosion. Probably the most declarative implementation of a DP problem is as a recurrence equation. Evaluated directly, such a recurrence equation would take exponential time, because it is typical that such problems have a high degree of redundancy between internal calls. Using memoization, such recurrence equations can often be solved in polynomial time. A natural question to ask is whether MSP can be used to further improve the performance of these implementations when partial information about the problem (such as its size) is known ahead of time. Unfortunately, direct staging of such memoized algorithms *does not* preserve the sharing obtained by memoizing, and leads to a severe code explosion problem. Furthermore, the standard technique of let-insertion [16] from partial evaluation does not apply directly.

Problem 2: High-level languages are generally not designed for numerical computation. Even if the first problem is overcome, the runtime performance does not measure up to handwritten C programs. While C programs do not give the high-level abstractions found in languages like (Meta)OCaml, they are sufficient for expressing numerical computations, and these computations are generally well-optimized by standard compilers such as GNU Compiler Collection `gcc`. To get both the benefits of high-level language abstractions and at the same time the performance of lower-level languages, it would be useful to have the flexibility to run the code on a computational platform other than what is available for full OCaml. We would also like to do this with minimal modification to the MSP framework, which currently consists only of *three* staging annotations.

1.1 Contributions

From the practical point of view, the main contribution of this paper is to show that it is possible to write concise and at the same time efficient implementations of DP problems in MetaOCaml. From the technical point of view, there are two key contributions: First, the paper shows that the code explosion problem can be addressed without making any changes to the operational semantics of the MSP language (and in turn, without any changes to the implementation of MetaOCaml). This is achieved using a general technique that we call **staged memoization**. From the partial evaluation point of view, the existence of this technique illustrates that the *pending list* [16] in offline partial evaluators can in fact be viewed as a *binding-time improvement* [16]. From the DP point of view, we show that memoized implementations of DP problems are amenable to specialization. This is achieved by showing that staged memoization is a high-level concept that can be expressed as a monadic fixed-point operator.

Second, the paper proposes a kind of **offshoring** technique to allow the MSP programmer to take advantage of implementations of programming languages other than the host language. In particular, one possible approach to producing a second-stage program in a language other than OCaml is to introduce explicit

facilities for both pattern matching on the generated OCaml code and constructing C code. If we are developing staged programs in a language like Scheme, this approach works fine (c.f. [10]). But from the MSP point of view, adding such facilities complicates the equational theory of the language [33], and can also complicate the type system [32]. Instead, the offshoring alternative uses a specialized implementation of the run construct that executes an OCaml program by first translating it into C and then compiling it using a standard C compiler. As is known from the partial evaluation literature [21], translating the output of *one* program *does not* require building an OCaml to C *compiler*. For MSP, rather than focusing on the form of the output of one generator, we observe that it can be useful to ensure that a large subset of the *target* language (in this case, C) is expressible as OCaml.

Experimental results indicate that a staged memoization and offshoring can be sufficient for building implementations for a number of DP problems that have promising performance characteristics.

1.2 Organization of this Paper

The rest of this paper is organized as follows. Section 2 reviews the basic features of an MSP language, and constitutes a brief introduction to MetaOCaml. Section 3 explains the code explosion problem that arises when staging recurrence equations that are typical of DP problems, and presents the staged-memoization technique. Section 4 describes the subset of C that the offshoring translation targets, and briefly summarizes the issues that arise in representing such programs in OCaml. Section 5 presents performance results for a set of example DP problems. Section 6 describes related work, and Section 7 concludes.

2 Multi-stage Languages and MetaOCaml

MSP languages [36, 32] 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 (c.f. [32]). The following minimal example illustrates MSP programming in an extension of OCaml [18] called MetaOCaml [6, 24]:

```
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 annotations (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 annotations 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 annotations are not hints, they are imperatives. Thus, the application `e .<x>.` must to 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.

3 Implementing DP Algorithms in an MSP Language

We begin with a review of a basic approach for implementing DP problems in a functional language, and then show that a direct attempt to stage such an implementation is problematic. We use a running example and present its development in an iterative-refinement style, only to expose the key ingredients of the class of implementations we are concerned with. In the conclusion of the section we illustrate the generality of these steps, and show how, in practice, they can be reduced to one standard refinement step.

3.1 Implementing DP Algorithms in a Functional Language

"Dynamic programming, 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." [8, Page 301].

From the above standard description it is easy to see functional languages are well-suited for direct – although not necessarily efficient – implementations of DP algorithms. We will illustrate this concretely with a standard generalization of the Fibonacci function called Gibonacci (c.f. [4]).

Recursion As a recurrence, this function can be implemented in OCaml as:

```
let rec gib n (x,y) =
   match n with
     0 -> x
   | 1 -> y
   | _ -> gib (n-2) (x,y) + gib (n-1) (x,y)
```

Of course, this direct implementation of Gibonacci suffers the same inefficiency as a similar implementation of Fibonacci would and it runs in exponential time.

Memoization To allow the unstaged program `gib` to reuse results of subcomputations, we modify it to obtain another program `gib_m` which takes a store `s` as an additional parameter, where all partial results would be memoized. Before we attempt to compute the result for `gib_m n`, we check whether it has already been computed and stored in `s` using `lookup s n`. If not, we perform the computation to obtain the answer a, and cache it in the store using `ext s (n,` a`)`. To be able to use the store, the `gib_m` function must return both the result, as well as an updated store. Furthermore, recursive calls to `gib_m` must be explicitly ordered so that the results computed by the call to `gib(n-2)` are available to the call to `gib(n-1)` via the updated store `s1`. If we assume that we have an implementation of a hash table with two functions `lookup` and `ext`end (c.f. [28]), the complexity of this first implementation can be reduced by using memoization:

```
let rec gib_m n s (x,y) =
  match (lookup s n) with
    Some z -> (z,s)
  | None -> match n with
      0 -> (x,s)
    | 1 -> (y,s)
    | _ -> let (a1, s1) = gib_m (n-2) s (x,y) in
           let (a2, s2) = gib_m (n-1) s1 (x,y) in
           (a1+a2, ext s2 (n, a1+a2))
```

Given an almost constant-time hash table implementation, this algorithm will run in time almost linear in `n`.

3.2 Direct Staging of Memoized Functions

The Gibonacci function above is generic, in that it will compute an expansion of order `n`. If we are interested in having specialized implementations of DP algorithms that work for a particular problem size, this would correspond (in many cases) to fixing the order `n`. It is natural in this case to ask if staging can be used to produce such specialized implementations automatically. Indeed, the above memoizing definition of Gibonacci would result in the following definition:

```
let rec gib_sm n s (x,y) =
  match (lookup s n) with
```

```
  Some z -> (z,s)
| None -> match n with
    0 -> (x,s)
  | 1 -> (y,s)
  | _ ->
      let (a1, s1) = (gib_sm (n-2) s (x,y)) in
      let (a2, s2) = (gib_sm (n-1) s1 (x,y)) in
      (.<.~a1+ .~a2>., ext s2 (n, .<.~a1+ .~a2>.))
```

Evaluating `.<fun x -> fun y-> .~(gib_sm 5 (.<x>.,.<y>.))>.` yields:

```
.<fun x -> fun y -> ((y + (x + y)) + ((x + y) + (y + (x + y))))>.
```

which is a more specialized function that takes the two remaining parameters to Gibonacci and produces the final answer without the overhead of recursion calls or recursive testing on the parameter `n`.

Code Explosion If we consider the general pattern for programs generated as above, we find that they are exponential in the size of the input. In fact, the generated code is a symbolic trace of the computation that would have been carried out by a call to the non-memoizing, unstaged definition of Gibonacci. The generated code is also syntactically the same as if we had directly staged the un-memoizing definition. It might seem as though introducing staging had completely undone the effect of memoization. This is not quite the case: running the staged definition above *without* printing the output runs in polynomial (linear) time. Memoization actually continues to have an effect, in that it allows the generator to run in polynomial time. However, the generator produces a directed-acyclic graph (DAG) as a parse tree. This DAG is exponentially compact, just like a binary-decision diagram (BDD). So, in fact the first stage continues to execute in polynomial time, and it is only printing that takes exponential time. So, if we had pretty-printers and compilers that handled such compact representations, it seems that the general problem of code explosion could be reduced or eliminated. It also suggests that other implementation strategies for MSP languages, such as runtime-code generation (RTCG), might not suffer from this kind of code explosion problem. In this paper, we will show how this problem can be overcome without changes to the implementation.

The Desired Output Before attempting to address the code explosion problem, it is instructive to be more explicit about what would be an acceptable solution. The code explosion problem described above would be solved if we are able to generate code that binds the result of each memo table entry to a named variable, and this variable (rather than the code for the computation) is then used in the solution of bigger sub-problems. For example, it would be acceptable if instead of generating the code above we got:

```
.<fun x ->
  fun y ->
    let z2 = (x + y) in
```

```
      let z3 = (y + z2) in
      let z4 = (z2 + z3) in (z3 + z4)>.
```

Direct Let-Insertion A standard remedy that we inherit from the partial evaluation literature is to add explicit let-statements to the generated code so that code duplication can be avoided, and involves replacing a term of the form

```
fun x -> ... .< ... .~x ... .~x ...>.
```

where the double insertion of the value of `x` is the source of the problem, by a term of the form

```
fun x -> ... .< let y = .~x in ... y ... y ...>.
```

This works well for a variety of cases when code is duplicated. But for the staged, memoizing Gibonacci function, this transformation cannot be applied directly. In particular, the two uses of $\mathtt{a}_i$ occur within different pairs of brackets. For the transformation to be used, there must be one dynamic context that surrounds both uses of $\mathtt{a}_i$. In the current form of the program, it is not clear how such a dynamic context can be introduced without changing the rest of the program.

CPS Conversion of Direct Staged Function Generating the desired let-statements requires introducing a new dynamic context that encapsulates both code fragments where each use of $\mathtt{a}_i$ occurs. To do this, we must have a handle on how these two code fragments will be eventually combined. This suggests a need for an explicit representation of the continuation of the computation. It is well-known in the partial evaluation literature that this technique can be used to improve opportunities for staging [9]. Here, however, we are CPS converting a two-level program. If we rewrite the last definition of Gibonacci into continuation passing style (CPS), we find that there is a natural place to insert such a let-statement:

```
let rec gib_ksp (n,x,y) s k =
  match (lookup s n) with
    Some z -> k s z
  | None -> match n with
      0 -> k s x
    | 1 -> k s y
    | _ ->
       gib_ksp (n-2, x, y) s  (fun s1 a1 ->
       gib_ksp (n-1, x, y) s1 (fun s2 a2 ->
       k (ext s2 (n, .<.~a1+ .~a2>.)) .<.~a1+ .~a2>.))
```

Given the appropriate initial continuation, this version behaves exactly as the last one.

Staged Memoization Now, we can introduce a dynamic context that was not expressible when the staged memoizing Gibonacci was written in direct style. This dynamic context is a legitimate place to insert a let statement that would avoid the code duplication. This is achieved by changing the last line in the code above as follows:

```
.<let z= .~a1+ .~a2 in .~(k (ext s2 (n, .<z>.)) .<z>.)>.
```

Running this function, which we can call `gib_lksm`, produces exactly the desired code described above.

3.3 A Monadic Recount

The above exposition was intended to motivate the details necessary to achieve staged memoization. To summarize the key ideas in a manner that emphasizes their generality rather than their details, we recast the development using a monadic interface [25] between the implementation of staged memoization and the DP problem being solved. Staged memoization is implemented as a monadic fixed-point combinator [13].

The underlying monad combines aspects of both state and continuations:

```
type ('a, 's, 'c) M = 's  -> ('s -> 'a -> 'c) -> 'c
let (ret,bind) =
   let ret a = fun s k -> k s a in
   let bind a f = fun s k -> a s (fun s' b -> f b s' k)
   in (ret,bind)
```

The `ret`urn of this monad takes a store `s` and a continuation `k`, and passes both the state and `a` to the continuation. The `bind` of this monad passes to the monadic value `a` a store `s` and a new continuation. The new continuation first evaluates the function `f` using the new store store `s'` and continues with `k`.

Explicit fixed-point, monadic style was used to rewrite all DP algorithms that we studied. This style can be illustrated with the Gibonacci function as follows:

```
let gib_ym f (n, x, y) =
    match n with
    | 0 -> ret x
    | 1 -> ret y
    | _ -> bind (f ((n-2), x, y)) (fun y2  ->
           bind (f ((n-1), x, y)) (fun y1 ->
           ret .<.~y1 + .~y2>.));;
```

Note that this is simply the original function written in monadic style, and that it does not expose any details about how the staged memoization is done. From the point of view of iterative-refinement, going from original definition to the monadic one has the advantage of being one, well-defined step [25].

The staged memoization fixed-point operator is defined as follows:

```
let rec y_sm f =
   f (fun x s k ->
     match (lookup s x) with
     | Some r -> k s r
     | None -> y_sm f x s (fun s' -> fun v ->
                .<let z = .~v in
                  .~(k (ext s' (x,.<z>.)) .<z>.)>.))
```

Here, continuing the recurrence first checks if the argument `x` was encountered before, and the result was computed and stored in `s`. If it was, we pass that value along with the current state to the current continuation. Otherwise, we compute the rest of the computation in a dynamic context where the result of computing the function on the current argument is placed in a let-binding, and the memo table is extended with the name of the variable used in the let-binding, rather than the value itself. The way the let-statement is presented here (compared to the previous subsection) demonstrates that the idea of staged memoization is independent of the details of the function being considered. Compared to the pending list used in offline partial evaluators [11, 16], this definition shows that staged memoization is a notion expressible within an MSP language, and does not require changing the standard operational semantics of such a language.

Staged memoization vs. a-normal form Staged memoization is not the same as generating code in a-normal form [14]. First, staged memoization does not – and is not intended to – generate programs in a-normal form. Staged memoization is in the first place a memoization technique, and it tries to name only the results of recursive function calls that are memoized. In contrast, a-normal form names all intermediate values. It is instructive to note that in the framework we present above, staged memoization must be expressed in the fixed-point combinator, whereas generating a-normal form can be done by changing the underlying monad.

4 Avoiding Limits of Standard Implementations of High-level Languages

Functional programming languages like OCaml provide powerful abstraction mechanisms that are not conveniently available in mainstream languages like C. In the previous section we showed how these mechanisms can be used to implement DP problems almost directly as recurrence equations. We also showed how, using MSP constructs and the staged memoization technique, we can generate code for specialized instances of such DP implementations. As was the case with Gibonacci, the generated code will generally not contain uses of the more sophisticated abstraction mechanisms. Furthermore, the code generated by the first stage of MSP typically uses a limited subset of the object language, with less expressiveness. The generated code also exhibits high regularity. This is a standard observation from the partial evaluation literature. Unfortunately,

it is unlikely that standard implementations of high-level languages like OCaml would compile such code as effectively as would implementations of lower-level languages like C. Furthermore, it was noted in previous work that MetaOCaml sometimes does not demonstrate as high a speedup with staging than has been shown in the context of C using partial evaluation [6].

There are a number of possible reasons why standard implementations of functional languages are not the most appropriate platforms for running programs generated using MetaOCaml. First, standard implementations of functional languages tend to focus primarily on the efficient implementation of the various abstraction mechanisms they provide. Second, the limited form of automatically generated code is not typical in hand-written code. To the contrary: because generated code often makes less use of higher-level abstraction mechanisms, it is probably an example of the kind of code that programmers are discouraged from writing. Third, extending an implementation of a language like OCaml to perform the same optimizations that a C compiler does would probably double the size of (and as we will see in the next section, significantly slow down) the compiler.

4.1 Offshoring

Implementations of functional languages that work by translation to C (c.f. [17, 3, 15, 37]) do not address the problem described above. When compiling a full functional language, direct transcription of simple computations is generally not sound, because the translation for each construct must deal with issues such as automatic memory management and boxing and unboxing optimizations. Type-based compilation techniques [30], in principle, can alleviate this problem. But as the performance of current implementations of this technology on numerical computations is limited due to legacy issues, we cannot make direct use of it in our work.

Instead, we propose the use of a specialized translation from a subset of OCaml to C. Unlike compilers, which should translate the full source language, the design goal for the translation we describe is to cover as much as possible of the target language. The idea is that we want the programmer to be able to express C computations as OCaml computations, and to be able to get the same performance on this subset of OCaml computations as C. Because the goal of this translation is to allow the programmer to take certain (and not all) tasks to be done outside the native language, we can view this technique as a kind of *offshoring*.

The translation described in this section was in fact designed and implemented independently of the DP application. Therefore, having just one such translation indicates that it may be possible to strengthen an observation from the partial evaluation literature, which is that specialized programs often have a grammatical form that can be determined statically [21]. The strengthening that we suggest is that one, relatively small grammar may be sufficient to capture the structure of many specialized programs. This makes it possible to have one offshoring translation that would provide the MSP programmer with efficient implementations of many specialized programs without the need to extend the MSP language with support for pattern matching over code. Such pattern

matching over generated code is known to weaken the equational theory of an MSP programming language [33], and make statically ensuring the safety of dynamically generated programs more difficult [32].

User View To use an offshoring implementation, the user replaces the standard MetaOCaml run construct `.!` by an alternative, specialized run construct `.!{C}`. Going back to the power function from the introduction, the user would write:

```
let power3 = .!{C} .<fun x -> .~(power 3 .<x>.)>.
```

First, the type checker ensures statically that the type of the argument is within the set of types that are allowed for this particular offshoring implementation of run. This is necessary to ensure that the marshalling and unmarshalling code (c.f. [5]) that is needed can be generated statically.

At runtime, if the term passed to this run construct is outside the scope of the translation, an exception is raised. Otherwise, translation is performed, and the resulting C code is compiled. If the return value is a ground type, the compiled C code is executed directly. Otherwise, it is of function type, and the resulting image is dynamically loaded into memory, and a wrapper is returned that makes this C function available as an OCaml function.

4.2 Overview of the Translation

The BNF of the target subset of C and the source subset of OCaml, along with the offshoring translation, are provided in Appendix A. In what follows we outline the high-level features of the translation.

The current translation covers basic C types and one- and two-dimensional arrays:

$$
\begin{array}{lll}
\textit{Base type} & b \in & \{\mathsf{int}, \mathsf{double}, \mathsf{char}\} \\
\textit{Reference type} & r ::= & *\, b \mid *\, *\, b \\
\textit{Array type} & a ::= & b\,[\,] \mid b\,[\,]\,[\,] \\
\textit{Type} & t ::= & b \mid r \mid a
\end{array}
$$

The translation is currently not intended to cover pointers, but it is convenient to include them in a limited form to make marshalling and unmarshalling of OCaml values easier. At the term level, all built-in operators on the basic types, as well as conditionals, while-loops, switch statements, and assignments are covered. Naturally, imperative languages such as C use l-values to support assignment. There is no notion in OCaml that corresponds directly (for all values), but references can be used to represent this notion fairly directly. The subset also includes C function declarations.

From the point of view of the types allowed in the OCaml subset being translated, we noted earlier that the subset only needs to be large enough to cover the aspects of C that we are interested in. For example, it does not include dynamic datatypes, closures and higher-order functions, polymorphism, exceptions, objects, or modules. Function types are allowed, but only if they are first-order (specifically, they can only take a tuple of arguments of base types, and return a value of base type). To support assignment in C, the subset does

include OCaml's `ref` type and its associated operators. It also includes OCaml's `array` type to represent one- and two-dimensional arrays:

$$
\begin{array}{lll}
\textit{Base type} & \hat{b} & \in \{\texttt{int}, \texttt{bool}, \texttt{float}, \texttt{char}\} \\
\textit{Reference type} & \hat{r} & ::= \hat{b}\ \texttt{ref} \\
\textit{Array type} & \hat{a} & ::= \hat{b}\ \texttt{array} \mid \hat{b}\ \texttt{array array} \\
\textit{Argument type} & \hat{p} & ::= \hat{b} \mid \hat{a} \\
\textit{Type} & \hat{t} & ::= \hat{b} \mid \hat{r} \mid \hat{a} \\
\textit{Functiontype} & \hat{u} & ::= (\hat{p}, \ldots, \hat{p}) \rightarrow \hat{b}
\end{array}
$$

The translation of the supported OCaml types into C is as follows:

$$
\begin{array}{c}
[\![\texttt{int}]\!] = \mathsf{int},\ [\![\texttt{float}]\!] = \mathsf{double},\ [\![\texttt{bool}]\!] = \mathsf{int},\ [\![\texttt{char}]\!] = \mathsf{char} \\
[\![\hat{b}\ \texttt{ref}]\!] = [\![\hat{b}]\!],\ [\![\hat{b}\ \texttt{array}]\!] = [\![\hat{b}]\!]\,[\,],\ [\![\hat{b}\ \texttt{array array}]\!] = [\![\hat{b}]\!][n][m]
\end{array}
$$

In the last case of the translation, the values n and m are extracted from the declaration in the term being translated. Currently, only arrays declared locally to the generated C code are represented as C arrays. If an array is marshalled in from the OCaml world, for simplicity, it is currently represented as an array of pointers to arrays (which is a more direct interpretation of OCaml arrays). Note that this does not affect the term translation. At the term level, the OCaml subset is artificially divided into expressions and statements so as to have a direct correspondence with these notions in the grammar of C. The OCaml subset consists of restricted forms of a number of standard OCaml constructs. For example, a function application must be an explicit application of a variable to a tuple of expressions. Operations on mutable values are represented in OCaml as operations using OCaml reference, dereferencing and assignment. Declarations of local variables are represented by restricted forms of OCaml let-statements. For example, such statements can only bind the result of OCaml expressions to variables. C variable declarations are represented by let bindings which introduce a local variable that is explicitly bound to an application of the OCaml `ref` constructor to an initial value for that variable. C array and function declarations are represented by similarly stylized OCaml `let` declarations.

Example Below we present a C program and its representation in OCaml. The translation automatically produces the C program from the OCaml program.

```
let power (n,x) =


 let t = ref 1 in
  for i=1 to n do
    t:=!t * x
  done;
  !t
```

```
int t;
int power(int n, int x)
{
  int i;
  t = 1;
  for(i = 1; i <= n; i++) {
    t = t * x;
  }
  return t;
}
```

Marshalling and Unmarshalling There are at least two possible approaches to marshalling and unmarshalling of values as they are moved from the OCaml world to the C world. The simpler one is to copy arguments and copy them back at end of computation, and the more sophisticated one is to share the same data structures. For simplicity, our initial implementation uses the copying approach. Blume [5] studies the second approach more closely, and we intended to explore applying his ideas to our implementation in future work.

4.3 Limitations

The current prototype of the translation is still in its early stages, and we expect to make a number of extensions in the future. Such extensions will include targeting general C types (structures, unions, pointers), and goto-statements. To represent certain goto-statements, a standard technique is to use function calls in tail position in the source language (c.f. [17, 3, 15, 37]). It will also be interesting to explore the use of continuations and other control operators to express more general patterns of goto-statements in target programs.

Some features of C we do not intend to target. One of them is pointer-arithmetic. We hypothesize that it will not be a crucial performance bottleneck for the programs we generate. Other features like the sizeof operator in C have an indeterminate status from the point of view of offshoring. Frequently sizeof can be computed statically, but that is not always the case. The translation also does not support the C do-while, break and continue constructs, and the fall-through semantics of the C switch statement. As we use offshoring for more applications, we expect to learn more about the importance of handling these features.

5 Experimental Results

This section summarizes some empirical measurements that we have gathered to validate the techniques proposed in this section. The specific questions that these experiments address are:

1. Does staged memoization allow staging DP problems?

2. What is the effect of translating to C?
3. What is the marshalling/unmarshalling overhead?
4. How does the native-code OCaml compiler compare to gcc?
5. How do the generated C programs compare to hand-written ones?

Test Cases and Platform Measurements were gathered for implementations of:

- `forward` is forward algorithm for Hidden Markov Models. Specialization size is 7.
- `gib` is the Gibonacci function used as the running example in this paper. Specialization is for $n = 25$.
- `knapsack` is the 0/1 knapsack problem. Specialization is for size 16.
- `lcs` and `lcs-array` are the least common subsequence problems. The second uses cache results. Specialization is for string sizes 24 and 35 for the first and second arguments, respectively.
- `obst` is the optimal binary search tree algorithm. Specialization is a leaf-node-tree of size 11.
- `opt-mult` is the optimal matrix multiplication problem. Specialization is for 10 matrices.

All performance measurements were collected by running programs on a 3.05GHz Intel Pentium 4 with 512KB cache and 1GB RAM running Linux 2.4.18. Measurements were carried out on a modified version of MetaOCaml, based on OCaml 3.06. We used `gcc` version 2.96 and `ocamlopt` version 3.06 for the benchmark tests.

Does staged memoization allow staging DP problems? The first column, **Unstaged Run** is the time 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`. The times reported in seconds (**s**) are the total time for which this benchmark is run. It is followed by the number of iterations of the basic function. The result of the division is the average time per function call. Timings in the table that are followed by (**x**) are reported as normalized with respect to this time. **Generate** is the time need to run the first-stage of the staged version. **Compile** is the time needed to compile the program generated by the first stage into OCaml bytecode. **Run** is the time needed to run this compiled program. **Speedup** is Unstaged Run divided by 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.

Program	Unstaged Run(O)	Generate (O)	Compile (O)	Run (O)	Speedup (O)	BEP (O)
forward	2.29s/10^4=1x	4.83x	92.9x	0.0646x	15.4	105
gib	5.95s/10^5=1x	3.16x	39.3x	0.0071x	139.4	43
knapsack	1.25s/10^3=1x	6.63x	44.5x	0.0186x	53.5	53
lcs	1.171s/10^4=1x	32.28x	530.0x	0.1419x	7.0	656
lcs-array	6.91s/10^2=1x	1.84x	13.5x	0.0053x	186.7	16
obst	3.40s/10^3=1x	0.57x	7.9x	0.0064x	154.1	9
opt-mult	1.09s/10^3=1x	4.75x	60.3x	0.0266x	37.5	67

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

What is the effect of translating to C? The first two columns in this table are as above. The next five are similar to the ones above, but are for an implementation of MetaOCaml's run that calls the `gcc` compiler. The last column is the ratio between Speedup with this implementation and Speedup with the standard table (above).

Program	Unstaged Run (O)	Generate (O)	Compile (C)	Run (C)	Speedup (C)	BEP (C)	Speedup (C/O)
forward	2.29s/10^4=1x	4.83x	402.7x	0.00333x	299.9	409	19.4
gib	5.95s/10^5=1x	3.16x	244.2x	0.00171x	581.6	248	4.1
knapsack	1.25s/10^3=1x	6.63x	207.1x	0.00132x	756.0	214	14.1
lcs	1.171s/10^4=1x	221.05x	35177.3x	0.04830x	20.7	37195	2.9
lcs-array	6.91s/10^2=1x	1.84x	139.9x	0.00012x	7811.4	148	41.8
obst	3.40s/10^3=1x	0.57x	86.8x	0.00063x	1577.4	89	10.2
opt-mult	1.09s/10^3=1x	4.75x	8070.0x	0.00080x	1235.8	8082	32.9

While the ratio between speedups is always greater than one for these tests, it should be noted that we are comparing runtimes in bytecode OCaml to compiled C. Also, there is a practical risk to translating to C, which is that compilation times are much higher for `gcc`. This means that the BEP values become higher. But the unusually high compilation times for `lcs` are possibly due to a known issue with `gcc` taking a long time to compile machine-generated code.

What is the marshalling/unmarshalling overhead? The following table compares the runtime for the generated C programs when called from within MetaOCaml against the runtime for the same programs when called from C.

Program	From OCaml ($x10^{-3}$ms)	From C ($x10^{-3}$ms)	Difference ($x10^{-3}$ms)	From OCaml(%)
forward	0.763	0.458	0.305	40%
gib	0.102	0.017	0.085	83%
knapsack	1.653	1.145	0.508	31%
lcs	8.260	4.860	3.400	41%
lcs-array	8.846	5.829	3.017	34%
obst	2.455	2.314	0.141	6%
opt-mult	0.882	0.509	0.373	42%

Generally, the overhead is significant, and can be considered a bottleneck. This suggests that it maybe useful to explore a marshalling strategy that allows sharing of the same data-structures across the OCaml/C boundary, rather than copying them. For example, OCaml supports a `Bigarray` type that allows sharing of arrays with C or FORTRAN.

How does the native-code OCaml compiler compare to gcc? The motivation for offshoring is that standard implementations of high-level languages are unlikely to compile unstructured computations such as the ones generated by staged programs as effectively as standard implementations for lower-level languages such as C. The following table presents measurements for runtimes of the generated programs when executed using the native-code compiler for OCaml and the best runtime when running the result of their translation in C. **Speedup** compares the best OCaml time with the best gcc time.

Programs	normal	unsafe	inline 2	inline 10	Best gcc	Speed Up
forward	2.432	2.102	2.107	2.107	0.375	5.605
gib	0.139	0.141	0.141	0.141	0.287	0.484
knapsack	4.936	4.920	4.906	4.828	1.201	4.019
lcs	4.164	3.734	3.715	3.635	0.635	5.724
lcs-array	4.480	4.492	4.479	4.477	0.764	5.859
obst	0.982	0.898	0.898	0.898	0.457	1.964
opt-mult	6.711	6.924	6.896	6.879	1.068	6.283

Only in one instance, native-code OCaml out-performs gcc. In half the cases, C is much faster.

How do the generated C programs compare to hand-written ones? We were not able to find generic C implementations of the basic DP problems that we studied. Thus, we wrote array-based, bottom-up, memoized implementations of each of these problems. The table below 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`.

Program	`-O0`		`-O1`		`-O2`		`-O3`	
	Time	Space	Time	Space	Time	Space	Time	Space
forward	4.87	1.12	1.43	1.09	0.50	1.10	0.52	1.10
gib	4.63	1.04	9.89	1.03	7.98	1.03	7.55	1.02
knapsack	28.40	2.26	10.08	1.75	9.33	1.70	9.24	1.70
lcs	3.15	7.22	1.35	4.91	1.32	4.69	1.39	4.69
lcs-array	2.56	5.91	1.20	3.56	1.10	3.55	1.15	3.55
obst	1.04	2.44	0.62	1.55	0.22	1.74	0.22	1.73
opt-mult	5.88	1.76	6.78	1.34	5.86	1.38	5.90	1.38

With the most aggressive optimization settings `-O3` for `gcc`, in all but two cases the generated code runs faster, even when the size of the executable is not substantially larger. In the case of `forward` and `obst`, the generated code actually ran faster with `-O0` and `-O1` flags than with `-O2` and `-O3` flags. With this optimization setting, the difference between the sizes of the binaries is also the smallest.

Generally, the optimization settings did not change the sizes of the binaries for the generated programs significantly.

6 Related Work

Multi-stage programming (MSP) grew out of work in partial evaluation on two-level languages [26, 16]. 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 [36]. 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 (c.f. [35]).

Liu and Stoller [19] 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 [20]. It would be interesting to see if the two approaches can be combined fruitfully.

McAdam [22] 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 [1] show how to derive string matchers specialized with respect to a particular pattern. Their work focuses on speeding up the program generator rather than on controlling the size of the generated program.

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 [2] 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.

The array optimization paper by Clarkson and Vaish [7] provided us with some information about arrays in OCaml and the fact that when they are rendered in C, they will have to be dynamically allocated.

7 Conclusion and Future Work

The paper reports on experience using MetaOCaml to generate specialized solutions for dynamic programming problems. We show that code explosion arises naturally when staging dynamic programming problems, but that it can also be avoided using staged memoization. To avoid unnecessary performance overheads from using OCaml as the second stage language, we use a specialized translator from a well-defined subset of OCaml into C. The guiding principle for the design of this translation is not the compilation of OCaml into C, but rather, the representation of key constructs in C as OCaml constructs. This approach allows for an implicit form of heterogeneous MSP, which has the advantage of allowing reuse of second-stage code across different target platforms, and avoids extending the language with a capability for pattern matching on the MetaOCaml code type.

There are a number of important directions for future work. First, it would be interesting to study the extent to which offshoring can be applied to domains other than dynamic programming. Examples of such domains include parsing [31], pretty-printing, and cryptography [23]. Second, we would like to continue the development of the internal OCaml-to-C translator to cover a larger subset of C. We also plan to build translators to other targets, especially languages designed for high-performance computing.

Acknowledgments We would like to thank Stephan Ellner for helpful comments. The motivation for the proposed offshoring extension came directly from the students taking a graduate course on Multi-stage Programming at Rice University.

References

1. 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.
2. Jonathan Aldrich. Open modules: A foundation for modular aspect-oriented programming. http://www-2.cs.cmu.edu/ aldrich/papers/tinyaspect.pdf, jan 2003.
3. Joel F. Bartlett. Scheme->c: A portable Scheme-to-C compiler. Technical Report 89/1, DEC Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA 94301 USA, jan 1989.
4. Arthur T. Benjamin and Jennifer J. Quinn. *Proofs that Really Count: The Art of Combinatorial Proof*. Mathematical Association of America, 2003.
5. Matthias Blume. No-longer-foreign: Teaching an ml compiler to speak c "natively.". In *BABEL'01: First Workshop On Multi-Language Infrastructure And Interoperability*, Firenze, Italy, sep 2001.

6. 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.
7. Michael Clarkson and Vaibhav Vaish. Array optimizations in ocaml, may 2001.
8. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 14th edition, 1994.
9. O. Danvy. Semantics-directed compilation of non-linear patterns. Technical Report 303, Indiana University, Bloomington, Indiana, USA, 1990.
10. Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. In *[34]*, page 108, 2001.
11. Olivier Danvy, Karoline Malmkjaer, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, 1(19), 1995.
12. W. DeMeuter. Monads as a theoretical foundation for AOP. In *International Workshop on Aspect-Oriented Programming at ECOOP*, page 25, 1997.
13. Levent Erkök and John Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, September 2000.
14. 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.
15. The GHC Team. The glasgow haskell compiler user's guide, version 4.08. Available online from `http://haskell.org/ghc/`. Viewed on 12/28/2000.
16. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
17. R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *ACM Symposium on Principles of Programming Languages*, pages 181–192, January 1989.
18. Xavier Leroy. Objective Caml, 2000. Available from `http://caml.inria.fr/ocaml/`.
19. Y. A. Liu and S. D. Stoller. Dynamic programming via static incrementalization. volume 16, pages 37–62, mar.
20. 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.
21. K. Malmkjær. On static properties of specialized programs. In M. Billaud et al., editors, *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991 (Bigre, vol. 74)*, pages 234–241. Rennes: IRISA, 1991.
22. Bruce McAdam. Y in practical programs (extended abstract). Unpublished manuscript.
23. 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.
24. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2003.

25. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.
26. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.
27. François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 132–142. IEEE Computer Society Press, 1998.
28. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.
29. 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`.
30. Zhong Shao and Andrew Appel. A type-based compiler for Standard ML. In *Conference on Programming Language Design and Implementation*, pages 116–129, 1995.
31. 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, 21-23 June 1995.
32. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [29].
33. 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.
34. Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, Firenze, 2001. Springer-Verlag.
35. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
36. 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.
37. David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

A Current Offshoring Translation

This appendix presents the BNF for the source and target languages as well as the definition for the offshoring translation.

A.1 Target Subset of C

$$
\begin{array}{lll}
\textit{Program} & p & ::= d_1, \ldots, d_i\ g_1, \ldots, g_i \\
\textit{Type keyword} & t & \in \{\mathsf{int}, \mathsf{double}, \mathsf{char}\} \\
\textit{Constant} & c & \in \textit{Chars} \cup \textit{Ints} \cup \textit{Floats} \\
\textit{Variable} & x & \in X \\
\textit{Basic definition} & d & ::= t\ x \mid t * x \mid t\ x[\mathsf{i}] \mid t\ x[\mathsf{i}][\mathsf{i}] \\
\textit{Function definition} & g & ::= t\ x\ (\ t\ x\ , t_1\ x_1\ , \ldots, t_n\ x_n\ )\ b \\
\textit{Block} & b & ::= \{d_1, \ldots, d_i\ s_1, \ldots, s_i\} \\
\textit{Unary operator} & f^{(1)} & \in \{(\mathsf{float})\ ,\ (\mathsf{int})\ ,\ \cos\ ,\ \sin\ ,\ \mathsf{sqrt}\} \\
\textit{Binary operator} & f^{[2]} & \in \{+, -, *, /, \%, \&\&, ||, \&, |, \wedge, <<, \\
 & & \quad >>, ==, !=, <, >, <=, >=\} \\
\textit{Expression} & e & ::= c \mid x \mid e \mid f^{(1)}\ e \mid e\ f^{[2]}\ e \mid x\ (e, e_1, \ldots, e_i) \mid e{+}{+} \\
 & & \mid e{-}{-} \mid x[e] \mid x[e][e] \mid x{=}e \mid x[e]{=}e \mid x[e][e]{=}e \mid e\ ?\ e : e \\
\textit{Switch branch} & w & ::= \mathsf{case}\ c{:}\ s\ \mathsf{break};\\
\textit{Statement} & s & ::=\ e \mid b \mid \mathsf{if}\ (e)\ s\ \mathsf{else}\ s\ \mid \mathsf{while}\ (e)\ s \mid \mathsf{for}\ (e; e; e)\ s \mid \mathsf{return}\ e \\
 & & \mid \mathsf{switch}\ (e)\ \{d_1, \ldots, d_i\ w_1, \ldots, w_i\ \ \mathsf{default:}\ \ \mathsf{s;}\} \mid \mathsf{assert}\ e
\end{array}
$$

A.2 OCaml Subset to Represent Target

The subset of OCaml that we allow is contained in the following grammar:

$$
\begin{array}{lll}
\textit{Constant} & \hat{c} & \in \textit{Bools} \cup \textit{Chars} \cup \textit{Ints} \cup \textit{Floats} \\
\textit{Variable} & \hat{x} & \in \hat{X} \\
\textit{Unary operator} & \hat{f}^{(1)} & \in \{\mathtt{cos}, \mathtt{sin}, \mathtt{sqrt}, \mathtt{float\_of\_int}, \mathtt{int\_of\_float}\} \\
\textit{Limit operator} & \hat{f}^{(2)} & \in \{\mathtt{min}, \mathtt{max}\} \\
\textit{Binary operator} & \hat{f}^{[2]} & \in \{+, -, *, /, +., -., *., /., **, \mathtt{mod}, \mathtt{land}, \mathtt{lor}, \mathtt{lxor}, \\
 & & \mathtt{lsl}, \mathtt{lsr}, \mathtt{asr}, =, <>, <, >, \leq, \geq, \&\&, ||\} \\
\textit{Expression} & \hat{e} & ::= \hat{x} \mid \hat{x}\ (\hat{e}_0{,}\hat{e}_1, \ldots{,}\hat{e}_n) \mid \hat{f}^{(1)}\ \hat{e} \mid \hat{f}^{(2)}\ \hat{e}\ \hat{e} \mid \hat{e}\ \hat{f}^{[2]}\ \hat{e} \mid \\
 & & \mid \mathtt{if}\ \hat{e}\ \mathtt{then}\ \hat{e}\ \mathtt{else}\ \hat{e} \mid !\hat{x} \mid \mathtt{Array.get}\ \hat{x}\ \hat{e} \mid \hat{x}.(\hat{e}) \mid \hat{x}.(\hat{e}).(\hat{e}) \\
\textit{Statement} & \hat{s} & ::= \hat{e} \mid \hat{s}; \hat{s} \mid \mathtt{let}\ \hat{x} = \hat{e}\ \mathtt{in}\ \hat{s} \mid \mathtt{let}\ f(\hat{x}_0{,}\hat{x}_1, \ldots{,}\hat{x}_n) = \hat{s}\ \mathtt{in}\ \hat{s} \\
 & & \mid \mathtt{let}\ \hat{x} = \mathtt{ref}\ \hat{c}\ \mathtt{in}\ \hat{s} \mid \mathtt{let}\ \hat{x} = \ \mathtt{Array.make}\ \hat{c}\ \hat{c}\ \mathtt{in}\ \hat{s} \\
 & & \mid \mathtt{let}\ \hat{x} = \ \mathtt{Array.make\_matrix}\ \hat{c}\ \hat{c}\ \hat{c}\ \mathtt{in}\ \hat{s} \\
 & & \mid \hat{x}{:=}\ \hat{e} \mid \hat{x}.(\hat{e}) \leftarrow \hat{e} \mid \ \mathtt{Array.set}\ \hat{x}\ \hat{e}\ \hat{e} \mid \hat{x}.(\hat{e}).(\hat{e}) \leftarrow \hat{e} \\
 & & \mid \mathtt{if}\ \hat{e}\ \mathtt{then}\ \hat{s}\ \mathtt{else}\ \hat{s} \mid \mathtt{while}\ \hat{e}\ \mathtt{do}\ \hat{s}\ \mathtt{done} \\
 & & \mid \mathtt{for}\ \hat{x}{=}\hat{e}\ \mathtt{to}\ \hat{e}\ \mathtt{do}\ \hat{s}\ \mathtt{done} \mid \mathtt{for}\ \hat{x}{=}\hat{e}\ \mathtt{downto}\ \hat{e}\ \mathtt{do}\ \hat{s}\ \mathtt{done} \\
 & & \mid \mathtt{match}\ \hat{e}\ \mathtt{with}\ (\hat{c}_1\ \rightarrow\ \hat{s}_1 \mid , \ldots, \hat{c}_i\ \rightarrow\ \hat{s}_i \mid \_ \rightarrow\ \hat{s}) \\
 & & \mid \mathtt{assert\ false} \mid \mathtt{assert}\ \hat{e}
\end{array}
$$

A.3 Translation

$$\boxed{[\![\hat{f}^*]\!] \equiv f^*}$$

$$[\![+]\!] \equiv [\![+.]\!] \equiv +\ ,\ [\![-]\!] \equiv [\![-.]\!] \equiv -\ ,\ [\![*]\!] \equiv [\![*.]\!] \equiv *\ ,\ [\![/]\!] \equiv [\![/.]\!] \equiv /\ ,\ [\![\&\&]\!] \equiv \&\&$$

$$[\![\mathtt{mod}]\!] \equiv \%\ ,\ [\![\mathtt{land}]\!] \equiv \&\ ,\ [\![\mathtt{lor}]\!] \equiv |\ ,\ [\![\mathtt{lxor}]\!] \equiv \hat{}\ ,\ [\![||]\!] \equiv ||$$

$$[\![\texttt{lsl}]\!] \equiv << \ , \ [\![\texttt{lsr}]\!] \equiv [\![\texttt{asr}]\!] \equiv >> \ , \ [\![=]\!] \equiv == \ , \ [\![<>]\!] \equiv ! =$$
$$[\![<]\!] \equiv < \ , \ [\![>]\!] \equiv > \ , \ [\![<=]\!] \equiv <= \ , \ [\![>=]\!] \equiv >=$$

$\boxed{[\![\hat{e}]\!] \equiv e}$

$$[\![\hat{c}]\!] \equiv c \ , \ [\![\hat{x}]\!] \equiv x \ , \ [\![!\hat{x}]\!] \equiv x \ , \ [\![\ \hat{f}^{(1)} \ \hat{e}]\!] \equiv [\![\hat{f}^{(1)}]\!]([\![\hat{e}]\!])$$
$$[\![\hat{e}' \ \hat{f}^{[2]} \ \hat{e}'']\!] \equiv [\![\hat{e}']\!] \ [\![\hat{f}^{[2]}]\!] \ [\![\hat{e}'']\!] \ when \ \hat{f} \neq ** \ , \ [\![\hat{e}' * * \hat{e}'']\!] \equiv \mathsf{pow}([\![\hat{e}']\!], [\![\hat{e}'']\!])$$
$$[\![\hat{x}.(\hat{e})]\!] \equiv [\![\texttt{Array.get} \ \hat{x} \ \hat{e}]\!] \equiv x[\ [\![\hat{e}]\!] \ ]$$
$$[\![\hat{x} := \hat{e}]\!] \equiv x = [\![\hat{e}]\!] \ , \ [\![\hat{x}.(\hat{e}').(\hat{e}'')]\!] \equiv x[\ [\![\hat{e}']\!] \ ][\ [\![\hat{e}'']\!] \ ]$$
$$[\![\texttt{Array.set} \ \hat{x} \ \hat{e}' \ \hat{e}'']\!] \equiv [\![\hat{x}.(\hat{e}') \leftarrow (\hat{e}'')]\!] \equiv x[\ [\![\hat{e}']\!] \ ] = [\![\hat{e}'']\!]$$
$$[\![\hat{x}.(\hat{e}').(\hat{e}'') \leftarrow (\hat{e}''')]\!] \equiv x[\ [\![\hat{e}']\!] \ ][\ [\![\hat{e}'']\!] \ ] = [\![\hat{e}''']\!]$$
$$[\![\texttt{if} \ \hat{e}' \ \texttt{then} \ \hat{e}'' \ \texttt{else} \ \hat{e}''']\!] \equiv [\![\hat{e}']\!] \ ? \ [\![\hat{e}'']\!] \ : \ [\![\hat{e}''']\!]$$
$$[\![\texttt{max} \ \hat{e}' \ \hat{e}'']\!] \equiv [\![\hat{e}']\!] > [\![\hat{e}'']\!] \ ? \ [\![\hat{e}']\!] : [\![\hat{e}'']\!] \ , \ [\![\texttt{min} \ \hat{e}' \ \hat{e}'']\!] \equiv [\![\hat{e}']\!] < [\![\hat{e}'']\!] \ ? \ [\![\hat{e}']\!] : [\![\hat{e}'']\!]$$
$$[\![\hat{x} \ (\hat{e}_0, \hat{e}_1, \ldots, \hat{e}_n)]\!] \equiv x \ (e_0, e_1, \ldots, e_n) \ , \ [\![\hat{x} \ \hat{e}]\!] \equiv x([\![\hat{e}]\!])$$

$\boxed{[\![\hat{s}]\!] \equiv s}$

$$\frac{[\![\hat{s}']\!] \equiv (D_1, e_1) \quad [\![\hat{s}'']\!] \equiv (D_2, e_2)}{[\![\hat{s}'; \hat{s}'']\!] \equiv (D_1; D_2 \vdash e_1; e_2)} \qquad \frac{}{[\![\texttt{let} \ \hat{x} : \hat{b} = \texttt{ref} \ \hat{c} \ \texttt{in} \ \hat{s}]\!] \equiv [\![\hat{b}]\!] \ x = c \vdash \ [\![\hat{s}]\!]}$$

$$\frac{}{[\![\texttt{let} \ \hat{x} : \hat{b} = \texttt{Array.make} \ \hat{c}_1 \ \hat{c}_2 \ \texttt{in} \ \hat{s}]\!] \equiv [\![\hat{b}]\!] \ x \ [c_1] \vdash \ [\![\hat{s}]\!]}$$

$$\frac{}{[\![\texttt{let} \ \hat{x} : \hat{b} = \texttt{Array.make\_matrix} \ \hat{c}_1 \ \hat{c}_2 \ \hat{c}_3 \ \texttt{in} \ \hat{s}]\!] \equiv [\![\hat{b}]\!] \ x \ [c_1] \ [c_2] \vdash \ [\![\hat{s}]\!]}$$

$$\frac{[\![\hat{s}]\!] \equiv (D, e)}{[\![\texttt{let} \ \hat{x} : \hat{b} = \hat{e} \ \texttt{in} \ \hat{s}]\!] \equiv ([\![\hat{b}]\!] \ x; D \vdash \ x = [\![\hat{e}]\!]; e)}$$

$$\frac{[\![\hat{s}']\!] \equiv (D_1, e_1) \quad [\![\hat{s}'']\!] \equiv (D_2, e_2)}{[\![\texttt{if} \ \hat{e} \ \texttt{then} \ \hat{s}' \ \texttt{else} \ \hat{s}'']\!] \equiv (D_1; D_2 \vdash \ \mathsf{if} \ ([\![\hat{e}]\!]) \ \{e_1\} \ \mathsf{else} \ \{e_2\})}$$

$$\frac{[\![\hat{s}]\!] \equiv (D, e)}{\begin{array}{c}[\![\texttt{let} \ \hat{x} : \hat{b} \ (\hat{x}_0 : \hat{p}_0, \hat{x}_1 : \hat{p}_1, \ldots, \hat{x}_n : \hat{p}_n) = \hat{s}' \ \texttt{in} \ \hat{s}'']\!] \equiv \\ (D; [\![\hat{b}]\!] \ x([\![\hat{p}_0]\!] \ x_0, [\![\hat{p}_1]\!] \ x_1, \ldots, [\![\hat{p}_n]\!] \ x_n) \ \{e\} \vdash \ [\![\hat{s}'']\!])\end{array}}$$

$$\frac{[\![\hat{s}]\!] \equiv (D, e)}{[\![\texttt{while} \ \hat{e} \ \texttt{do} \ \hat{s} \ \texttt{done}]\!] \equiv (D \vdash \ \mathsf{while} \ ([\![\hat{e}]\!]) \ \{e\})}$$

$$\frac{[\![\hat{s}]\!] \equiv (D, e)}{\begin{array}{c}[\![\texttt{for} \ \hat{x} = \hat{e}' \ \texttt{to} \ \hat{e}'' \ \texttt{do} \ \hat{s} \ \texttt{done}]\!] \equiv \\ (D; \ \mathsf{int} \ x \vdash \ \mathsf{for} \ (x = [\![\hat{e}']\!]; x <= [\![\hat{e}'']\!]; x + +) \ \{e\})\end{array}}$$

$$\frac{[\![\hat{s}]\!] \equiv (D, e)}{\begin{array}{c}[\![\texttt{for}\ \hat{x} = \hat{e}'\ \texttt{downto}\ \hat{e}''\ \texttt{do}\ \hat{s}\ \texttt{done}]\!] \equiv \\ (D; \mathsf{int}\ x \vdash\ \mathsf{for}\ (x = [\![\hat{e}']\!]; x >= [\![\hat{e}'']\!]; x - -)\ \{e\})\end{array}}$$

$$\frac{[\![\hat{s}]\!] \equiv (D, e) \quad [\![\hat{s}_i]\!] \equiv (D_i, e_i)}{\begin{array}{c}[\![\texttt{match}\ \hat{e}\ \texttt{with}\ (\hat{c}_1 \leftarrow \hat{s}_1 \mid , \ldots, \hat{c}_i \leftarrow \hat{s}_i \mid \_ \leftarrow \hat{s})]\!] \equiv \\ (\mathsf{switch}([\![\hat{e}]\!])\ \{D_i; D \vdash \mathsf{case}\ c_1\colon e_1\ \mathsf{break};, \ldots, \mathsf{case}\ c_i\colon e_i\ \mathsf{break};\ \mathsf{default} : e\})\end{array}}$$

$$\frac{}{[\![\texttt{assert}\ \hat{e}]\!] \equiv \mathsf{assert}([\![\hat{e}]\!])}$$

Variables used by several statements must be globally declared. For example, in the first let rule, the statement $\hat{s}$ is dependent on the variable $\hat{x}$. Therefore, the rule says that the declaration of the translated $\hat{x}$, x has to be present at the top of our C program, that is, the list of global variables. This allows the use of nesting extensively in OCaml constructs (the needed variables are made available).