

A Gentler Introduction to Multi-stage Programming and Metaocaml

Van Bui
Department of Computer Science
University of Houston

May 22, 2004

Abstract

In this paper we provide novice learners of the Metaocaml programming language with an introductory resource to use during the initial learning stages of the language. Metaocaml is a multi-stage extension of the Ocaml programming language. Ocaml is a well-documented language with many resources available in the forms of electronic media (such as on-line sources), published books, and research articles [1–3]. In contrast, Metaocaml has a dearth of available resources for new learners of the language. The available sources for learning Metaocaml are in the form of technical papers on the applications of the language, a few introductory papers on the language, and on-line tutorial slides [5–8]. We provide an additional source here in the form of brief summaries and analysis of benchmark programs written in the language and also provide a brief introduction to using and interpreting the timing functions for the language. The summaries detail each program's operation and staging aspects.

1 Introduction

Widely used programming languages such as C/C++, FORTRAN, Perl, etc have a large amount of resources available for individuals interested in learning how to use those languages. A novice learner might even be overwhelmed with having to choose amongst all the works available for those popular languages. The same thing cannot be said about programming languages that have not yet been widely dispersed to the programming community. Programming languages that are not yet well documented can be a major obstacle for a programmer faced with learning how to use the language. Once the programmer figures out how to use the language, ideally, he/she will write a structured document detailing what was learned about the language adding to the currently available resources.

1.1 Metaocaml and its Constructs

The programming language that we examine in this paper is Metaocaml. Metaocaml is a multi-stage extension of the Ocaml programming language. Ocaml is a statically typed general-purpose language. Statically typed languages ensure its programs are evaluated correctly [4] by performing early error checking. Ocaml is a well-documented programming language with a vast amount of introductory and advanced level literature available on the web and in published format for programmers interested in learning more about the language [1–3].

Metaocaml extends the Ocaml language with the addition of the following new constructs: bracket, escape, and run. The new constructs are added into an Ocaml program in a procedure termed staging. Staging an Ocaml program transforms it into a Metaocaml program as the programmer incorporates the three constructs into the Ocaml program.

Adding the three Metaocaml constructs to an Ocaml program yields relatively faster execution time than un-staged programs. A description of each construct can explain the faster execution times. The brackets construct takes the following form:

```
exp=.<expression>.
```

It is simply brackets placed around an expression and a period surrounding the brackets. The brackets delay the computation of the expression until program run time. Bracketed expressions are combined together using the escape construct. The escape construct may be applied in the following manner:

```
newExp=.< .~exp + .~exp >.
```

A period followed by a tilde represents the escape construct in the previous example. Escaping a delayed expression has the effect of evaluating that code fragment while the outer expression is being constructed rather than when it is being executed. For example, if we had declared the following expression:

```
exp=.<4+2>.
```

earlier in the program, we would have the following expression after the code is generated with the escape construct:

```
newExp=.<(4+2) + (4+2)>.
```

The early computation of the code fragment yields faster program execution times when compared to un-staged programs. The run construct executes the code fragment. In following with the current example, we apply the run construct as follows:

```
.!newExp
```

The exclamation character is representative of the run construct in Metaocaml [6]. As the programmer adds these three constructs to a program, he/she has more control over determining what stage (generation, compilation, or execution) each part of the program is being evaluated. The programmer can then modify their program with the Metaocaml constructs to evaluate terms earlier in the process (during generation), resulting in execution time speed-up.

We summarize two staged Ocaml programs in the present work. The summaries are intended as a supplemental resource for beginners just starting to learn the language and is by no means a comprehensive low-level discussion concerning Metaocaml. We provide small-sized examples and explanations for novice learners of the language to use as a learning tool. We also assume the reader is familiar with coding in Ocaml.

1.2 Related Works

The literature available on Metaocaml is both introductory and comprehensive in form. The introductory sources consist of discussions on both small and large sized programs in Metaocaml [5–8]. These sources usually begin discussing the constructs in the language and follows with a word on practical applications of the language, sample programs, and a brief discussion on the timing functions. The most comprehensive work on Metaocaml is a thesis written by the author of the language [8]. For beginners just starting to learn the language, the first three chapters of the thesis can be a useful reading tool. The first three sections cover language application, the staging constructs, and several sample staged programs. The rest of the thesis goes into a detailed discussion about the theory behind the language and semantics for the interested reader.

2 Program Summaries

In this section, we gather together two benchmark Metaocaml programs, discuss what the programs do, and briefly note on the aspects and effects of staging each program. We began with an explanation on the timing functions.

2.1 Timing Functions

Metaocaml provides functions for gathering performance data on both staged and un-staged programs for comparison. Timing functions located in Metaocaml's Trx module are available for measuring the run-time for un-staged and staged programs.

The first step in collecting performance data is initialization or resetting of internal timing elements. This initialization of time measurements is executed by calling the following function:

```
Trx.init_times ( )
```

The statement is usually placed on the first line of a Metaocaml program, but can also be placed anywhere in the program prior to calling any other timing functions.

We can also collect times for the first stage, compilation, and second stage processes. The first stage consists of the time it takes to generate the code. In the first stage, the terms that are escaped in the program are replaced with its previously assigned code (see introduction for example). The compilation time includes type checking amongst other routine tasks performed at compile time. The second stage includes the time it takes to run the staged code. We can also collect performance data on the time to run the un-staged code. The following shows declarations that compute the compilation and staging times for the power function:

```
Trx.timenew "unstaged running"(fun () -> power 50000 1)
Trx.timenew "stage 1 running" (fun () -> .<fun x-> .~(power' 50000 .<x>.)>.)
Trx.timenew "compiling" (fun () -> .! stage1Running)
Trx.timenew "stage 2 running" (fun () -> (compiling 1))
```

The function in the Trx module for collecting run and compilation times is timenew. Function timenew takes as arguments a string specifying the type of time measurements collected and a function that calls on the appropriate functions to execute the measured process.

Finally, Metaocaml provides a function that outputs the timing results. To view timing outputs, declare the following on the last line of the program:

```
Trx.print_times()
```

The function from the Trx module to output the results is print_times and it does not take any arguments. Additional examples of calling timing functions are included in the implementations of each of the programs analyzed below.

Following are brief summaries for Metaocaml benchmark programs and the implementation details for each program. Both staged and un-staged versions of each program are shown as well as the performance data.

2.2 Dot.ml

Program dot.ml calculates the dot product of two vectors. The function iproc, the main function in dot.ml, sums the numbers resulting from the vectors' dot multiplication. Iproc takes as input n, x and y, where x and y are vectors, of size n. For example, if x=[1;2;3] and y=[4;5;6] then their dot product is [4;10;18] and the result produced by iproc will be 4+10+18=32. See Program 1 below for implementation of Dot.ml.

Program 1. Implementation of Dot.ml

```
(*Author: Walid Taha*)

(*Unstaged*)
let n = 100
let xx = Array.create n 6
let yy = Array.create n 7
let rec iproc n x y =
  if n > 0 then
    x.(n-1) * y.(n-1) + (iproc (n-1) x y)
  else 0

(* Staged-only 2 level *)
let rec iproc' n x y =
  if n > 0
  then .< ( x.(n-1) * ((.~y).(n-1)) + .~(iproc' (n-1) x y)) >.
  else .<0>..

(*Timing calls*)
let unstage =
  Trx.timenew "unstaged running"
    (fun () -> iproc n xx yy)
let st1Run =
  Trx.timenew "stage 1 running"
    (fun () -> .<(fun y -> .~(iproc' n xx .<y>..))>.)
let comp =
  Trx.timenew "compiling"
    (fun () -> (! st1Run) )
let st2Run =
  Trx.timenew "stage 2 running"
    (fun () -> comp yy)
let _ = Trx.print_times()
```

In the staged version, function `iproc` always returns a delayed expression. So the expression returned by `iproc` will not be evaluated until run-time (as a side note, note how the `run` construct is used in the program to directly run stage 1 and indirectly run stage 2). Inside function `iproc`, the recursive call to `iproc` and the variable `y` (one of the vectors) are consistently escaped. The effect is that the code representative of those terms is evaluated during the first stage. Therefore, the overhead from making recursive function calls to `iproc` and evaluating the variable `y` is included in the timing for the first stage, but not for the second stage. Making these slight modifications to the un-staged version of `dot.ml` yields the performance results shown in Figure 1 below.

Figure 1. Dot.ml Performance Numbers

```
-- unstaged running ----- 65536x avg= 1.702917E-02 msec
-- stage 1 running ----- 128x avg= 8.610204E+00 msec
-- compiling ----- 32x avg= 5.475233E+01 msec
-- stage 2 running ----- 131072x avg= 1.284377E-02 msec
```

As expected, running the staged code is faster than running the un-staged code. The time to run the un-staged code is approximately 1.3X slower than the time it takes to execute the staged code. The time difference here is not great and should not be because the purpose here is to show a small and simple staging example.

2.3 Membership.ml

Program Membership.ml simply determines whether a certain term is located inside a list. The recursive function called member takes as arguments a list l and a term y to be searched for in the function. The program returns the appropriate boolean value depending on whether the program succeeded in locating the term inside the list. See Program 2 for the implementation of Membership.ml.

Program 2. Implementation for membership.ml
(*Author: Walid Taha*)

```
(* unstaged *)
let rec member l y =
  match l with
  [] ->
    false
  | x::xs ->
    if x=y then
      true
    else
      member xs y

(* staged *)
let lift x = .<x>.
let rec member' l y =
  match l with
  [] ->
    .<false>.
  | x::xs ->
    .<if .~(lift x) = .~y then
      true
else
```

```

        .~(member' xs y)>. ;;

let _ = .<fun x -> .~(member' [1;2;3] .<x>.)>.;;

Trx.init_times ();;

let r = Trx.timenew
    "run unstaged"
    (fun () -> member [1;2;3] 5);;
let g = Trx.timenew
    "generate"
    (fun () -> .<fun x -> .~(member' [1;2;3] .<x>.)>.);;
let c = Trx.timenew
    "compile"
    (fun () -> .! g);;
let t = Trx.timenew
    "run staged"
    (fun () -> c 5);;
Trx.print_times ();;

```

Staging the membership program is quite similar to staging the dot.ml program. Like function iproc (from program dot.ml), the membership function returns a delayed result, either true or false, by bracketing the return value. Calls made to the membership function are escaped in the program. Escaping the membership function results in expanding the program during stage 1 since the implemented function is essentially in-lined into the program for each recursive call to the membership function. The recursive function call overheads of the membership function are avoided during program run time resulting in a faster execution time for the staged version. See Figure 2 below for actual performance data for Membership.ml.

Figure 2. Performance Data for Membership.ml

```

-- run unstaged ----- 4194304x avg= 2.492352E-04 msec
-- generate ----- 262144x avg= 7.152414E-03 msec
-- compile ----- 2048x avg= 1.233451E+00 msec
-- run staged ----- 16777216x avg= 7.634871E-05 msec

```

Here, the staged program runs much faster than the un-staged version when compared with the performance data from the prior programming example. In this example, the un-staged program runs approximately 3X slower than the staged implementation.

3 Conclusion

We have presented two fully implemented Metaocaml programs and made remarks on staging and gathering performance numbers on both programs. Each program implementation included the un-staged, staged, and timing functions. Explanations of the use of each of the Metaocaml constructs, bracket, escape, and run, were given for each program.

Our goal here was to give sample implementations in their entirety for the novice Metaocaml programmer to model after and analyze for himself or herself. We present the big picture behind staging in Metaocaml by making brief remarks about the staging constructs, staging, and gathering and interpreting performance numbers. The examples were small in order to make the initial learning stage less overwhelming for a new Metaocaml programmer. The next step for the programmer would be to peruse more detailed documentation on the language and stage Ocaml programs comparable in size to the two shown here.

Future work along these lines would involve writing a comprehensive and structured manual on the language. Alternatively, short papers, such as the present one can also be written and can be helpful for giving a brief overview about the language. Ideally, all these papers should be amalgamated into one source that is written coherently and concisely and can offer a structured process and procedure to learning Metaocaml. The manual could include the range of discussions spanning introductory remarks on the uses, the installation process, staging, functions in the Trx module, other applications, sample exercises, etc. . . The typical comprehensive manual on Metaocaml remains to be written.

References

- [1] E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications with Objective Ocaml*. O'Reilly and Associates, 2000.
- [2] R. Jones. *Learning Ocaml for C, C++, Perl, and Java Programmers*. Merjiss Ltd., 2004.
- [3] X. Leroy, D. Doligez, J. Garrigue, D. Remy, and J. Vouillon. *The Objective Caml System Release 3.07*, 2003.
- [4] D. Remy, X. Leroy, and P. Weis. Objective caml-a general-purpose high-level programming language. *ERCIM News*, 36, 1999.
- [5] W. Taha. Multi-stage programming in metaocaml, 2003.
- [6] W. Taha. A gentle introduction to multi-stage programming. DSPG, 2004.

- [7] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. The Symposium on Partial Evaluation and Semantic Based Program Manipulation, ACM Press, 1997.
- [8] W. Taha and T. Sheard. *Multi-stage programming: Its theory and application*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.