A Gentle Introduction to Multi-stage Programming[⋆]

Walid Taha

Department of Computer Science, Rice University, Houston, TX, USA
`taha@rice.edu`

Abstract. Multi-stage programming (MSP) is a paradigm for developing generic software that does not pay a runtime penalty for this generality. This is achieved through concise, carefully-designed language extensions that support runtime code generation and program execution. Additionally, type systems for MSP languages are designed to statically ensure that dynamically generated programs are type-safe, and therefore require no type checking after they are generated.
This hands-on tutorial is aimed at the reader interested in learning the basics of MSP practice. The tutorial uses a freely available MSP extension of OCaml called MetaOCaml, and presents a detailed analysis of the issues that arise in staging an interpreter for a small programming language. The tutorial concludes with pointers to various resources that can be used to probe further into related topics.

1 Introduction

Although program generation has been shown to improve code reuse, product reliability and maintainability, performance and resource utilization, and developer productivity, there is little support for *writing* generators in mainstream languages such as C or Java. Yet a host of basic problems inherent in program generation can be addressed effectively by a programming language designed specifically to support writing generators.

1.1 Problems in Building Program Generators

One of the simplest approaches to writing program generators is to represent the program fragments we want to generate as either strings or data types ("abstract syntax trees"). Unfortunately, both representations have disadvantages. With the string encoding, we represent the code fragment `f (x,y)` simply as `"f (x,y)"`. Constructing and combining fragments represented by strings can be done concisely. But there is no *automatically verifiable* guarantee that programs constructed in this manner are syntactically correct. For example, `"f (,y)"` can have the static type `string`, but this string is clearly *not* a syntactically correct program.

[⋆] Supported by NSF ITR-0113569 and NSF CCR-0205542.

With the data type encoding the situation is improved, but the best we can do is ensure that any generated program is syntactically correct. We cannot use data types to ensure that generated programs are well-typed. The reason is that data types can represent context-free sets accurately, but usually not context sensitive sets. Type systems generally define context sensitive sets (of programs). Constructing data type values that represent trees can be a bit more verbose, but a quasi-quotation mechanism [1] can alleviate this problem and make the notation as concise as that of strings.

In contrast to the strings encoding, MSP languages statically ensure that any generator only produces syntactically well-formed programs. Additionally, statically typed MSP languages statically ensure that any generated program is also well-typed.

Finally, with both string and data type representations, ensuring that there are no name clashes or inadvertent variable captures *in the generated program* is the responsibility of the programmer. This is essentially the same problem that one encounters with the C macro system. MSP languages ensure that such inadvertent capture is not possible. We will return to this issue when we have seen one example of MSP.

1.2 The Three Basic MSP Constructs

We can illustrate how MSP addresses the above problems using MetaOCaml [2], an MSP extension of OCaml [9]. In addition to providing traditional imperative, object-oriented, and functional constructs, MetaOCaml provides three constructs for staging. The constructs are called Brackets, Escape, and Run. Using these constructs, the programmer can change the order of evaluation of terms. This capability can be used to reduce the overall cost of a computation.

Brackets (written `.<...>.`) can be inserted around any expression to delay its execution. MetaOCaml implements delayed expressions by dynamically generating source code at runtime. While using the source code representation is not the only way of implementing MSP languages, it is the simplest. The following short interactive MetaOCaml session illustrates the behavior of Brackets[1]:

```
# let a = 1+2;;
val a : int = 3
# let a = .<1+2>.;;
val a : int code = .<1+2>.
```

Lines that start with `#` are what is entered by the user, and the following line(s) are what is printed back by the system. Without the Brackets around `1+2`, the addition is performed right away. With the Brackets, the result is a piece of code representing the program `1+2`. This code fragment can either be used as part of another, bigger program, or it can be compiled and executed.

[1] Some versions of MetaOCaml developed after December 2003 support environment classifiers [21]. For these systems, the type `int code` is printed as `('a,int) code`. To follow the examples in this tutorial, the extra parameter `'a` can be ignored.

In addition to delaying the computation, Brackets are also reflected in the type. The type in the last declaration is `int code`. The type of a code fragment reflects the type of the value that such code should produce when it is executed. Statically determining the type of the generated code allows us to avoid writing generators that produce code that cannot be typed. The code type constructor distinguishes delayed values from other values and prevents the user from accidentally attempting unsafe operations (such as `1 + .<5>.`).

Escape (written `.~`...) allows the combination of smaller delayed values to construct larger ones. This combination is achieved by "splicing-in" the argument of the Escape in the context of the surrounding Brackets:

```
# let b = .<.~a * .~a >. ;;
val b : int code = .<(1 + 2) * (1 + 2)>.
```

This declaration binds `b` to a new delayed computation `(1+2)*(1+2)`.

Run (written `.!`...) allows us to compile and execute the dynamically generated code without going outside the language:

```
# let c = .! b;;
val c : int = 9
```

Having these three constructs as part of the programming language makes it possible to use runtime code generation and compilation as part of any library subroutine. In addition to not having to worry about generating temporary files, static type systems for MSP languages can assure us that no runtime errors will occur in these subroutines (c.f. [17]). Not only can these type systems exclude generation-time errors, but they can also ensure that generated programs are both syntactically well-formed and well-typed. Thus, the ability to statically type-check the safety of a computation is not lost by staging.

1.3 Basic Notions of Equivalence

As an aside, there are two basic equivalences that hold for the three MSP constructs [18]:

$$\texttt{.\textasciitilde}\ \texttt{.<}e\texttt{>.} = e$$
$$\texttt{.!}\ \texttt{.<}e\texttt{>.} = e$$

Here, a value v can include usual values such as integers, booleans, and lambdas, as well as Bracketed terms of the form `.<`e`>.`. In the presentation above, we use e for an expression where all Escapes are enclosed by a matching set of Brackets. The rules for Escape and Run are identical. The distinction between the two constructs is in the notion of values: the expression in the value `.<`e`>.` cannot contain Escapes that are not locally surrounded by their own Brackets. An expression e is unconstrained as to where Run can occur.

Avoiding accidental name capture. Consider the following staged function:

```
# let rec h n z = if n=0 then z
                  else .<(fun x -> .~(h (n-1) .<x+ .~z>.)) n>.;;
val h : int -> int code -> int code = <fun>
```

If we erase the annotations (to get the "unstaged" version of this function) and apply it to `3` and `1`, we get `7` as the answer. If we apply the staged function above to `3` and `.<1>.`, we get the following term:

```
.<(fun x_1 -> (fun x_2 -> (fun x_3 -> x_3 + (x_2 + (x_1 + 1))) 1) 2) 3>.
```

Whereas the source code only had `fun x -> ...` inside Brackets, this code fragment was generated three times, and each time it produced a different `fun x_i -> ...` where `i` is a different number each time. If we run the generated code above, we get `7` as the answer. We view it as a highly desirable property that the results generated by staged programs are related to the results generated by the unstaged program. The reader can verify for herself that if the `x`s were not renamed and we allowed variable capture, the answer of running the staged program would be different from `7`. Thus, automatic renaming of bound variables is not so much a feature; rather, it is the absence of renaming that seems like a bug.

1.4 Organization of This Paper

The goal of this tutorial is to familiarize the reader with the basics of MSP. To this end, we present a detailed example of how MSP can be used to build staged interpreters. The practical appeal of staged interpreters lies in that they can be almost as simple as interpreter-based language implementations and at the same time be as efficient as compiler-based ones. To this end, Section 2 briefly describes an idealized method for developing staged programs in a programming language that provides MSP constructs. The method captures a process that a programmer iteratively applies while developing a staged program. This method will be used repeatedly in examples in the rest of the paper. Section 3 constitutes the technical payload of the paper, and presents a series of more sophisticated interpreters and staged interpreters for a toy programming language. This section serves both to introduce key issues that arise in the development of a staged interpreter (such as error handling and binding-time improvements), and to present a realistic example of a series of iterative refinements that arise in developing a satisfactory staged interpreter. Section 4 concludes with a brief overview of additional resources for learning more about MSP.

2 How Do We Write MSP Programs?

Good abstraction mechanisms can help the programmer write more concise and maintainable programs. But if these abstraction mechanisms degrade performance, they will not be used. The primary goal of MSP is to help the programmer reduce the runtime overhead of sophisticated abstraction mechanisms. Our hope is that having such support will allow programmers to write higher-level and more reusable programs.

2.1 A Basic Method for Building MSP Programs

An idealized method for building MSP programs can be described as follows:

1. A single-stage program is developed, implemented, and tested.
2. The organization and data-structures of the program are studied to ensure that they can be used in a staged manner. This analysis may indicate a need for "factoring" some parts of the program and its data structures. This step can be critical step toward effective MSP. Fortunately, it has been thoroughly investigated in the context of partial evaluation where it is known as *binding-time engineering* [7].
3. Staging annotations are introduced to explicitly specify the evaluation order of the various computations of the program. The staged program may then be tested to ensure that it achieves the desired performance.

The method described above, called *multi-stage programming with explicit annotations [22]*, can be summarized by the slogan:

$$\boxed{\text{A Staged Program} = \text{A Conventional Program} + \text{Staging Annotations}}$$

We consider the method above to be idealized because, in practice, it is useful to iterate through steps 1-3 in order to determine the program that is most suitable for staging. This iterative process will be illustrated in the rest of this paper. But first, we consider a simpler example where the method can be applied directly.

2.2 A Classic Example

A common source of performance overhead in generic programs is the presence of parameters that do not change very often, but nevertheless cause our programs to repeatedly perform the work associated with these inputs. To illustrate, we consider the following classic function in MetaOCaml: [7]

```
let rec power (n, x) =
   match n with
     0 -> 1 | n -> x * (power (n-1, x));;
```

This function is generic in that it can be used to compute `x` raised to *any* nonnegative exponent `n`. While it is convenient to use generic functions like `power`, we often find that we have to pay a price for their generality. Developing a good understanding of the source of this performance penalty is important, because it is exactly what MSP will help us eliminate. For example, if we need to compute the second power often, it is convenient to define a special function:

```
let power2 (x) = power (2,x);;
```

In a functional language, we can also write it as follows:

```
let power2 = fun x -> power (2,x);;
```

Here, we have taken away the formal parameter `x` from the left-hand side of the equality and replaced it by the equivalent "`fun x ->`" on the right-hand side. To use the function `power2`, all we have to do is to apply it as follows:

```
let answer = power2 (3);;
```

The result of this computation is `9`. But notice that every time we apply `power2` to some value `x` it calls the `power` function with parameters `(2,x)`. And even though the first argument will always be `2`, evaluating `power (2,x)` will always involve calling the function recursively two times. This is an undesirable overhead, because we *know* that the result can be more efficiently computed by multiplying `x` by itself. Using only unfolding and the definition of `power`, we know that the answer can be computed more efficiently by:

```
let power2 = fun x -> 1*x*x;;
```

We also do not want to write this by hand, as there may be many other specialized power functions that we wish to use. So, can we *automatically* build such a program?

In an MSP language such as MetaOCaml, all we need to do is to *stage* the power function by annotating it:

```
let rec power (n, x) =
   match n with
     0 -> .<1>. | n -> .<.~x * .~(power (n-1, x))>.;;
```

This function still takes two arguments. The second argument is no longer an integer, but rather, a *code of type integer*. The return type is also changed. Instead of returning an integer, this function will return a code of type integer. To match this return type, we insert Brackets around `1` in the first branch on the third line. By inserting Brackets around the multiplication expression, we now return a code of integer instead of just an integer. The Escape around the recursive call to `power` means that it is performed immediately.

The staging constructs can be viewed as "annotations" on the original program, and are fairly unobtrusive. Also, we are able to type check the code both outside and inside the Brackets in essentially the same way that we did before. If we were using strings instead of Brackets, we would have to sacrifice static type checking of delayed computations.

After annotating `power`, we have to annotate the uses of `power`. The declaration of `power2` is annotated as follows:

```
let power2 = .! .<fun x -> .~(power (2,.<x>.))>.;;
```

Evaluating the application of the Run construct will compile and if execute its argument. Notice that this declaration is essentially the same as what we used to define `power2` before, except for the staging annotations. The annotations say that we wish to construct the code for a function that takes one argument (`fun x ->`). We also do not spell out what the function itself should do; rather , we use the Escape construct (`.~`) to make a call to the staged `power`. The result of evaluating this declaration behaves exactly as if we had defined it "by hand" to be `fun x -> 1*x*x`. Thus, it will behave exactly like the first declaration of `power2`, but it will run as though we had written the specialized function by hand. The staging constructs allowed us to eliminate the runtime overhead of using the generic power function.

3 Implementing DSLs Using Staged Interpreters

An important application for MSP is the implementation of domain-specific languages (DSLs) [4]. Languages can be implemented in a variety of ways. For example, a language can be implemented by an interpreter or by a compiler. Compiled implementations are often orders of magnitude faster than interpreted ones, but compilers have traditionally required significant expertise and took orders of magnitude more time to implement than interpreters. Using the method outlined in Section 2, we can start by first writing an interpreter for a language, and then stage it to get *a staged interpreter*. Such staged interpreters can be as simple as the interpreter we started with and at the same time have performance comparable to that of a compiler. This is possible because the staged interpreter becomes effectively a *translator* from the DSL to the host language (in this case OCaml). Then, by using MetaOCaml's Run construct, the total effect is in fact a function that takes DSL programs and produces machine code[2].

To illustrate this approach, this section investigates the implementation of a toy programming language that we call `lint`. This language supports integer arithmetic, conditionals, and recursive functions. A series of increasingly more sophisticated interpreters and staged interpreters are used to give the reader a hands-on introduction to MSP. The complete code for the implementations described in this section is available online [10].

3.1 Syntax

The syntax of expressions in this language can be represented in OCaml using the following data type:

```
type exp = Int of int | Var of string | App of string * exp
         | Add of exp * exp | Sub of exp * exp
         | Mul of exp * exp | Div of exp * exp | Ifz of exp * exp * exp

type def = Declaration of string * string * exp
type prog = Program of def list * exp
```

Using the above data types, a small program that defines the factorial function and then applies it to 10 can be concisely represented as follows:

[2] If we are using the MetaOCaml native code compiler. If we are using the bytecode compiler, the composition produces bytecode.

```
Program ([Declaration
          ("fact","x", Ifz(Var "x",
                     Int 1,
                     Mul(Var"x",
                         (App ("fact", Sub(Var "x",Int 1))))))
         ],
         App ("fact", Int 10))
```

OCaml lex and yacc can be used to build parsers that take textual representations of such programs and produce abstract syntax trees such as the above. In the rest of this section, we focus on what happens after such an abstract syntax tree has been generated.

3.2 Environments

To associate variable and function names with their values, an interpreter for this language will need a notion of an environment. Such an environment can be conveniently implemented as a function from names to values. If we look up a variable and it is not in the environment, we will raise an exception (let's call it `Yikes`). If we want to extend the environment (which is just a function) with an association from the name `x` to a value `v`, we simply return a new environment (a function) which first tests to see if its argument is the same as `x`. If so, it returns `v`. Otherwise, it looks up its argument in the original environment. All we need to implement such environments is the following:

```
exception Yikes

(* env0, fenv : 'a -> 'b *)

let env0 = fun x -> raise Yikes         let fenv0 = env0

(* ext : ('a -> 'b) -> 'a -> 'b -> 'a -> 'b *)

let ext env x v = fun y -> if x=y then v else env y
```

The types of all three functions are polymorphic. Type variables such as `'a` and `'b` are implicitly universally quantified. This means that they can be later instantiated to more specific types. Polymorphism allows us, for example, to define the initial function environment `fenv0` as being exactly the same as the initial variable environment `env0`. It will also allow us to use the same function `ext` to extend both kinds of environments, even when their types are instantiated to the more specific types such as:

```
 env0 : string -> int                    fenv0 : string -> (int -> int)
```

3.3 A Simple Interpreter

Given an environment binding variable names to their runtime values and function names to values (these will be the formal parameters `env` and `fenv`, respectively), an interpreter for an expression `e` can be defined as follows:

```
(* eval1 : exp -> (string -> int) -> (string -> int -> int) -> int *)

let rec eval1 e env fenv =
match e with
  Int i -> i
| Var s -> env s
| App (s,e2) -> (fenv s)(eval1 e2 env fenv)
| Add (e1,e2) -> (eval1 e1 env fenv)+(eval1 e2 env fenv)
| Sub (e1,e2) -> (eval1 e1 env fenv)-(eval1 e2 env fenv)
| Mul (e1,e2) -> (eval1 e1 env fenv)*(eval1 e2 env fenv)
| Div (e1,e2) -> (eval1 e1 env fenv)/(eval1 e2 env fenv)
| Ifz (e1,e2,e3) -> if (eval1 e1 env fenv)=0
                    then (eval1 e2 env fenv)
                    else (eval1 e3 env fenv)
```

This interpreter can now be used to define the interpreter for declarations and programs as follows:

```
(* peval1 : prog -> (string -> int) -> (string -> int -> int) -> int *)

let rec peval1 p env fenv=
    match p with
     Program ([],e) -> eval1 e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        let rec f x = eval1 e1 (ext env s2 x) (ext fenv s1 f)
        in peval1 (Program(tl,e)) env (ext fenv s1 f)
```

In this function, when the list of declarations is empty (`[]`), we simply use `eval1` to evaluate the body of the program. Otherwise, we recursively interpret the list of declarations. Note that we also use `eval1` to interpret the body of function declarations. It is also instructive to note the three places where we use the environment extension function `ext` on both variable and function environments.

The above interpreter is a complete and concise specification of what programs in this language should produce when they are executed. Additionally, this style of writing interpreters follows quite closely what is called the denotational style of specifying semantics, which can be used to specify a wide range of programming languages. It is reasonable to expect that a software engineer can develop such implementations in a short amount of time.

Problem: The cost of abstraction. If we evaluate the factorial example given above, we will find that it runs about 20 times slower than if we had written this example directly in OCaml. The main reason for this is that the interpreter repeatedly traverses the abstract syntax tree during evaluation. Additionally, environment lookups in our implementation are not constant-time.

3.4 The Simple Interpreter Staged

MSP allows us to keep the conciseness and clarity of the implementation given above and also eliminate the performance overhead that traditionally we would have had to pay for using such an implementation. The overhead is avoided by staging the above function as follows:

```
(* eval2 : exp -> (string -> int code) -> (string -> (int -> int) code)
              -> int code *)

let rec eval2 e env fenv =
match e with
  Int i -> .<i>.
| Var s -> env s
| App (s,e2) -> .<.~(fenv s).~(eval2 e2 env fenv)>.
...
| Div (e1,e2)-> .<.~(eval2 e1 env fenv)/ .~(eval2 e2 env fenv)>.
| Ifz (e1,e2,e3) -> .<if .~(eval2 e1 env fenv)=0
                       then .~(eval2 e2 env fenv)
                       else .~(eval2 e3 env fenv)>.

(* peval2 : prog -> (string -> int code) -> (string -> (int -> int) code)
                -> int code *)

let rec peval2 p env fenv=
    match p with
     Program ([],e) -> eval2 e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
     .<let rec f x = .~(eval2 e1 (ext env s2 .<x>.)
                                 (ext fenv s1 .<f>.))
       in .~(peval2 (Program(tl,e)) env (ext fenv s1 .<f>.))>.
```

If we apply `peval2` to the abstract syntax tree of the factorial example (given above) and the empty environments `env0` and `fenv0`, we get back the following code fragment:

```
.<let rec f = fun x -> if x = 0 then 1 else x * (f (x - 1)) in (f 10)>.
```

This is exactly the same code that we would have written by hand for that specific program. Running this program has exactly the same performance as if we had written the program directly in OCaml.

The staged interpreter is a function that takes abstract syntax trees and produces MetaOCaml programs. This gives rise to a simple but often overlooked fact [19,20]:

$\boxed{\text{A staged interpreter is a translator.}}$

It is important to keep in mind that the above example is quite simplistic, and that further research is need to show that we can apply MSP to realistic programming languages. Characterizing what MSP cannot do is difficult, because of the need for technical expressivity arguments. We take the more practical approach of explaining what MSP can do, and hope that this gives the reader a working understanding of the scope of this technology.

3.5 Error Handling

Returning to our example language, we will now show how direct staging of an interpreter does not always yield satisfactory results. Then, we will give an example of the wide range of techniques that can be used in such situations.

A generally desirable feature of programming languages is error handling. The original implementation of the interpreter uses the division operation, which can raise a divide-by-zero exception. If the interpreter is part of a bigger system, we probably want to have finer control over error handling. In this case, we would modify our original interpreter to perform a check before a division, and return a special value `None` if the division could not be performed. Regular values that used to be simply `v` will now be represented by `Some v`. To do this, we will use the following standard datatype:

```
type 'a option = None | Just of 'a;;
```

Now, such values will have to be propagated and dealt with everywhere in the interpreter:

```
(* eval3 : exp -> (string -> int) -> (string -> int -> int option)
               -> int option *)

let rec eval3 e env fenv =
match e with
  Int i -> Some i
| Var s -> Some (env s)
| App (s,e2) -> (match (eval3 e2 env fenv) with
                   Some x -> (fenv s) x
                 | None   -> None)
| Add (e1,e2) -> (match (eval3 e1 env fenv, eval3 e2 env fenv)
                   with (Some x, Some y) -> Some (x+y)
                       | _ -> None) ...
| Div (e1,e2) -> (match (eval3 e1 env fenv, eval3 e2 env fenv)
                   with (Some x, Some y) ->
                          if y=0 then None
                                 else Some (x/y)
                      | _ -> None)
| Ifz (e1,e2,e3) -> (match (eval3 e1 env fenv) with
                      Some x -> if x=0 then (eval3 e2 env fenv)
                                       else (eval3 e3 env fenv)
                    | None   -> None)
```

Compared to the unstaged interpreter, the performance overhead of adding such checks is marginal. But what we really care about is the staged setting. Staging `eval3` yields:

```
(* eval4 : exp -> (string -> int code)
               -> (string -> (int -> int option) code)
               -> (int option) code *)

let rec eval4 e env fenv =
match e with
```

```
  Int i -> .<Some i>.
| Var s -> .<Some .~(env s)>.
| App (s,e2) -> .<(match .~(eval4 e2 env fenv) with
                     Some x -> .~(fenv s) x
                   | None   -> None)>.
| Add (e1,e2) -> .<(match (.~(eval4 e1 env fenv),
                           .~(eval4 e2 env fenv)) with
                     (Some x, Some y) -> Some (x+y)
                    | _ -> None)>. ...
| Div (e1,e2) -> .<(match (.~(eval4 e1 env fenv),
                           .~(eval4 e2 env fenv)) with
                     (Some x, Some y) ->
                       if y=0 then None
                              else Some (x/y)
                    | _ -> None)>.
| Ifz (e1,e2,e3) -> .<(match .~(eval4 e1 env fenv) with
                        Some x -> if x=0 then
                                    .~(eval4 e2 env fenv)
                                  else
                                     .~(eval4 e3 env fenv)
                      | None   -> None)>.
```

Problem: The cost of error handling. Unfortunately, the performance of code generated by this staged interpreter is typically 4 times slower than the first staged interpreter that had no error handling. The source of the runtime cost becomes apparent when we look at the generated code:

```
.<let rec f =
   fun x ->
    (match (Some (x)) with
       Some (x) ->
        if (x = 0) then (Some (1))
        else
         (match
            ((Some (x)),
             (match
                (match ((Some (x)), (Some (1))) with
                   (Some (x), Some (y)) ->
                    (Some ((x - y)))
                 | _ -> (None)) with
                Some (x) -> (f x)
              | None -> (None))) with
            (Some (x), Some (y)) ->
             (Some ((x * y)))
          | _ -> (None))
     | None -> (None)) in
  (match (Some (10)) with
     Some (x) -> (f x)
   | None -> (None))>.
```

The generated code is doing much more work than before, because at every operation we are checking to see if the values we are operating with are proper values or not. Which branch we take in every `match` is determined by the explicit form of the value being matched.

Half-solutions. One solution to such a problem is certainly adding a pre-processing analysis. But if we can avoid generating such inefficient code in the first place it would save the time wasted both in generating these unnecessary checks and in performing the analysis. More importantly, with an analysis, we may never be certain that all unnecessary computation is eliminated from the generated code.

3.6 Binding-Time Improvements

The source of the problem is the `if` statement that appears in the interpretation of `Div`. In particular, because `y` is bound inside Brackets, we cannot perform the test `y=0` while we are building for these Brackets. As a result, we cannot immediately determine if the function should return a `None` or a `Some` value. This affects the type of the whole staged interpreter, and effects the way we interpret all programs even if they do not contain a use of the `Div` construct.

The problem can be avoided by what is called a *binding-time improvement* in the partial evaluation literature [7]. It is essentially a transformation of the program that we are staging. The goal of this transformation is to allow better staging. In the case of the above example, one effective binding time improvement is to rewrite the interpreter in continuation-passing style (CPS) [5], which produces the following code:

```
(* eval5 : exp -> (string -> int) -> (string -> int -> int)
               -> (int option -> 'a) -> 'a *)

let rec eval5 e env fenv k =
match e with
  Int i -> k (Some i)
| Var s -> k (Some (env s))
| App (s,e2) -> eval5 e2 env fenv
                  (fun r -> match r with
                    Some x -> k (Some ((fenv s) x))
                  | None   -> k None)
| Add (e1,e2) -> eval5 e1 env fenv
                   (fun r ->
                     eval5 e2 env fenv
                       (fun s -> match (r,s) with
                         (Some x, Some y) -> k (Some (x+y))
                        | _ -> k None)) ...
| Div (e1,e2) -> eval5 e1 env fenv
                   (fun r ->
                     eval5 e2 env fenv
```

```
                        (fun s -> match (r,s) with
                          (Some x, Some y) ->
                            if y=0 then k None
                                     else k (Some (x/y))
                         | _ -> k None)) ...

(* pevalK5 : prog -> (string -> int) -> (string -> int -> int)
                  -> (int option -> int) -> int *)

let rec pevalK5 p env fenv k =
    match p with
     Program ([],e) -> eval5 e env fenv k
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        let rec f x = eval5 e1 (ext env s2 x) (ext fenv s1 f) k
          in pevalK5 (Program(tl,e)) env (ext fenv s1 f) k

exception Div_by_zero;;

(* peval5 : prog -> (string -> int) -> (string -> int -> int) -> int *)

let peval5 p env fenv =
     pevalK5 p env fenv (function Some x -> x
                                | None -> raise Div_by_zero)
```

In the unstaged setting, we can use the CPS implementation to get the same functionality as the direct-style implementation. But note that the two algorithms are *not the same*. For example, performance of the CPS interpreter is in fact worse than the previous one. But when we try to stage the new interpreter, we find that we can do something that we could not do in the direct-style interpreter. In particular, the CPS interpreter can be staged as follows:

```
(* eval6 : exp -> (string -> int code) -> (string -> (int -> int) code)
               -> (int code option -> 'b code) -> 'b code *)

let rec eval6 e env fenv k =
match e with
  Int i -> k (Some .<i>.)
| Var s -> k (Some (env s))
| App (s,e2) -> eval6 e2 env fenv
                   (fun r -> match r with
                     Some x -> k (Some .<.~(fenv s) .~x>.)
                   | None   -> k None)
| Add (e1,e2) -> eval6 e1 env fenv
                    (fun r ->
                      eval6 e2 env fenv
                        (fun s -> match (r,s) with
                          (Some x, Some y) ->
                            k (Some .<.~x + .~y>.)
                         | _ -> k None)) ...
| Div (e1,e2) -> eval6 e1 env fenv
                    (fun r ->
                      eval6 e2 env fenv
                        (fun s -> match (r,s) with
                          (Some x, Some y) ->
                            .<if .~y=0 then .~(k None)
                              else .~(k (Some .<.~x / .~y>.))>.
                         | _ -> k None))
| Ifz (e1,e2,e3) -> eval6 e1 env fenv
                     (fun r -> match r with
                         Some x -> .<if .~x=0 then
                                        .~(eval6 e2 env fenv k)
                                     else
                                        .~(eval6 e3 env fenv k)>.
                        | None   -> k None)

(* peval6 : prog -> (string -> int code) -> (string -> (int -> int) code)
                -> int code *)

let peval6 p env fenv =
     pevalK6 p env fenv (function Some x -> x
                                | None -> .<raise Div_by_zero>.)
```

The improvement can be seen at the level of the type of `eval6`: the `option` type occurs outside the `code` type, which suggests that it can be eliminated in the first stage. What we could not do before is to Escape the application of the continuation to the branches of the `if` statement in the `Div` case. The extra advantage that we have when staging a CPS program is that we are applying the continuation multiple times, which is essential for performing the computation in the branches of an `if` statement. In the unstaged CPS interpreter, the continuation is always applied exactly once. Note that this is the case even in the `if` statement used to interpret `Div`: The continuation does *occur* twice (once in each branch), but only one branch is ever taken when we evaluate this statement. But in the the staged interpreter, the continuation is indeed duplicated and applied multiple times[3].

The staged CPS interpreter generates code that is exactly the same as what we got from the first interpreter as long as the program we are interpreting does not use the division operation. When we use division operations (say, if we replace the code in the body of the fact example with `fact (20/2)`) we get the following code:

```
.<let rec f =
   fun x -> if (x = 0) then 1 else (x * (f (x - 1)))
  in if (2 = 0) then (raise (Div_by_zero)) else (f (20 / 2))>.
```

[3] This means that converting a program into CPS can have disadvantages (such as a computationally expensive first stage, and possibly code duplication). Thus, CPS conversion must be used judiciously [8].

3.7 Controlled Inlining

So far we have focused on eliminating unnecessary work from generated programs. Can we do better? One way in which MSP can help us do better is by allowing us to unfold function declarations for a fixed number of times. This is easy to incorporate into the first staged interpreter as follows:

```
(* eval7 : exp -> (string -> int code)
               -> (string -> int code -> int code) -> int code *)

let rec eval7 e env fenv =
match e with
... most cases the same as eval2, except
| App (s,e2) -> fenv s (eval7 e2 env fenv)

(* repeat : int -> ('a -> 'a) -> 'a -> 'a *)

let rec repeat n f =
  if n=0 then f else fun x -> f (repeat (n-1) f x)

(* peval7 : prog -> (string -> int code)
                 -> (string -> int code -> int code) -> int code *)

let rec peval7 p env fenv=
    match p with
     Program ([],e) -> eval7 e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        .<let rec f x =
            .~(let body cf x =
                    eval7 e1 (ext env s2 x) (ext fenv s1 cf) in
                 repeat 1 body (fun y -> .<f .~y>.) .<x>.)
          in .~(peval7 (Program(tl,e)) env
                      (ext fenv s1 (fun y -> .<f .~y>.)))>.
```

The code generated for the factorial example is as follows:

```
.<let rec f =
   fun x ->
    if (x = 0) then 1
    else
     (x*(if ((x-1)=0) then 1 else (x-1)*(f ((x-1)-1))))
  in (f 10)>.
```

This code can be expected to be faster than that produced by the first staged interpreter, because only one function call is needed for every two iterations.

Avoiding Code Duplication. The last interpreter also points out an important issue that the multi-stage programmer must pay attention to: code duplication. Notice that the term `x-1` occurs three times in the generated code. In the result of the first staged interpreter, the subtraction only occurred once.

The duplication of this term is a result of the inlining that we perform on the body of the function. If the argument to a recursive call was even bigger, then code duplication would have a more dramatic effect both on the time needed to compile the program and the time needed to run it.

A simple solution to this problem comes from the partial evaluation community: we can generate `let` statements that replace the expression about to be duplicated by a simple variable. This is only a small change to the staged interpreter presented above:

```
let rec eval8 e env fenv =
match e with
... same as eval7 except for
| App (s,e2) -> .<let x= .~(eval8 e2 env fenv)
                  in .~(fenv s .<x>.)>. ...
```

Unfortunately, in the current implementation of MetaOCaml this change does not lead to a performance improvement. The most likely reason is that the bytecode compiler does not seem to perform `let`-floating. In the native code compiler for MetaOCaml (currently under development) we expect this change to be an improvement.

Finally, both error handling and inlining can be combined into the same implementation:

```
(* eval9: exp -> (string -> int code) -> (string -> int code -> int code)
              -> (int code option -> 'b code) -> 'b code *)

let rec eval9 e env fenv k =
match e with
... same as eval6, except
| App (s,e2) -> eval9 e2 env fenv
                  (fun r -> match r with
                    Some x -> k (Some ((fenv s) x))
                  | None   -> k None)

(* pevalK9 : prog -> (string -> int code)
                  -> (string -> int code -> int code)
                   -> (int code option -> int code) -> int code  *)

let rec pevalK9 p env fenv k =
    match p with
     Program ([],e) -> eval9 e env fenv k
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        .<let rec f x =
          .~(let body cf x =
                  eval9 e1 (ext env s2 x) (ext fenv s1 cf) k in
             repeat 1 body (fun y -> .<f .~y>.) .<x>.)
          in .~(pevalK9 (Program(tl,e)) env
                        (ext fenv s1 (fun y -> .<f .~y>.)) k)>.
```

3.8 Measuring Performance in MetaOCaml

MetaOCaml provides support for collecting performance data, so that we can empirically verify that staging does yield the expected performance improvements. This is done using three simple functions. The first function must be called before we start collecting timings. It is called as follows:

```
Trx.init_times ();;
```

The second function is invoked when we want to gather timings. Here we call it twice on both `power2 (3)` and `power2'(3)` where `power2'` is the staged version:

```
Trx.timenew "Normal" (fun () ->(power2  (3)));;
Trx.timenew "Staged" (fun () ->(power2' (3)));;
```

Each call to `Trx.timenew` causes the argument passed last to be run as many times as this system needs to gather a reliable timing. The quoted strings simply provide hints that will be printed when we decide to print the summary of the timings. The third function prints a summary of timings:

```
Trx.print_times ();;
```

The following table summarizes timings for the various functions considered in the previous section[4]:

Program	**Description of Interpreter**	`Fact10`	`Fib20`
(none)	OCaml implementations	100%	100%
`eval1`	Simple	1,570%	1,736%
`eval2`	Simple staged	100%	100%
`eval3`	Error handling (EH)	1,903%	2,138%
`eval4`	EH staged	417%	482%
`eval5`	CPS, EH	2,470%	2,814%
`eval6`	CPS, EH, staged	100%	100%
`eval7`	Inlining, staged	87%	85%
`eval8`	Inlining, no duplication, staged	97%	97%
`eval9`	Inlining, CPS, EH, staged	90%	85%

Timings are normalized relative to handwritten OCaml versions of the DSL programs that we are interpreting (`Fact10` and `Fib20`). Lower percentages mean faster execution. This kind of normalization does not seem to be commonly used in the literature, but we believe that it has an important benefit. In particular, it serves to emphasize the fact that the performance of the resulting specialized programs should really be compared to specialized hand-written programs. The fact that `eval9` is more than 20 times faster than `eval3` is often irrelevant to a programmer who rightly considers `eval3` to be an impractically inefficient implementation. As we mentioned earlier in this paper, the goal of MSP is to remove unnecessary runtime costs associated with abstractions, and identifying the right reference point is essential for knowing when we have succeeded.

[4] System specifications: MetaOCaml bytecode interpreter for OCaml 3.07+2 running under Cygwin on a Pentium III machine, Mobile CPU, 1133MHz clock, 175 MHz bus, 640 MB RAM. Numbers vary when Linux is used, and when the native code compiler is used.

4 To Probe Further

The sequence of interpreters presented here is intended to illustrate the iterative nature of the process of exploring how to stage an interpreter for a DSL so that it can be turned into a satisfactory translator, which when composed with the Run construct of MetaOCaml, gives a compiler for the DSL.

A simplifying feature of the language studied here is that it is a first-order language with only one type, `int`. For richer languages we must define a datatype for values, and interpreters will return values of this datatype. Using such a datatype has an associated runtime overhead. Eliminating the overhead for such datatypes can be achieved using either a post-processing transformation called tag elimination [20] or MSP languages with richer static type systems [14]. MetaOCaml supports an experimental implementation of tag elimination [2].

The extent to which MSP is effective is often dictated by the surrounding environment in which an algorithm, program, or system is to be used. There are three important examples of situations where staging can be beneficial:

– We want to minimize total cost of all stages *for most inputs*. This model applies, for example, to implementations of programming languages. The cost of a simple compilation followed by execution is *usually* lower than the cost of interpretation. For example, the program being executed usually contains loops, which typically incur large overhead in an interpreted implementation.
– We want to minimize *a weighted average* of the cost of all stages. The weights reflect the relative frequency at which the result of a stage can be reused. This situation is relevant in many applications of symbolic computation. Often, solving a problem symbolically, and then graphing the solution at a thousand points can be cheaper than numerically solving the problem a thousand times. This cost model can make a symbolic approach worthwhile even when it is 100 times more expensive than a direct numerical one. (By symbolic computation we simply mean computation where free variables are values that will only become available at a later stage.)
– We want to minimize the cost of the *last stage*. Consider an embedded system where the *sin* function may be implemented as a large look-up table. The cost of constructing the table is not relevant. Only the cost of computing the function at run-time is. The same applies to optimizing compilers, which may spend an unusual amount of time to generate a high-performance computational library. The cost of optimization is often not relevant to the users of such libraries.

The last model seems to be the most commonly referenced one in the literature, and is often described as “there is ample time between the arrival of different inputs”, “there is a significant difference between the frequency at which the various inputs to a program change”, and “the performance of the program matters only after the arrival of its last input”.

Detailed examples of MSP can be found in the literature, including term-rewriting systems [17] and graph algorithms [12]. The most relevant example to domain-specific program generation is on implementing a small imperative language [16]. The present tutorial should serve as good preparation for approaching the latter example.

An implicit goal of this tutorial is to prepare the reader to approach introductions to partial evaluation [3] and partial evaluation of interpreters in particular [6]. While these two works can be read without reading this tutorial, developing a full appreciation of the ideas they discuss requires either an understanding of the internals of partial evaluators or a basic grasp of multi-stage computation. This tutorial tries to provide the latter.

Multi-stage languages are both a special case of meta-programming languages and a generalization of multi-level and two-level languages. Taha [17] gives definitions and a basic classification for these notions. Sheard [15] gives a recent account of accomplishments and challenges in meta-programming. While multi-stage languages are "only" a special case of meta-programming languages, their specialized nature has its advantages. For example, it is possible to formally prove that they admit strong algebraic properties even in the untyped setting, but this is not the case for more general notions of meta-programming [18]. Additionally, significant advances have been made over the last six years in static typing for these languages (c.f. [21]). It will be interesting to see if such results can be attained for more general notions of meta-programming.

Finally, the MetaOCaml web site will be continually updated with new research results and academic materials relating to MSP [11].

Acknowledgments

Stephan Jorg Ellner, John Garvin, Christoph Hermann, Samah Mahmeed, and Kedar Swadi read and gave valuable comments on this tutorial. Thanks are also due to the reviewers, whose detailed comments have greatly improved this work, as well as to the four editors of this volume, who have done an extraordinary job organizing the Dagstuhl meeting and organizing the preparation of this volume.

References

1. Alan Bawden. Quasiquotation in LISP. In O. Danvy, editor, *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–99, San Antonio, 1999. University of Aarhus, Dept. of Computer Science. Invited talk.
2. 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.
3. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.
4. Krzysztof Czarnecki1, John O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In this volume.

5. O. Danvy. Semantics-directed compilation of non-linear patterns. Technical Report 303, Indiana University, Bloomington, Indiana, USA, 1990.
6. Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.
7. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
8. 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.
9. Xavier Leroy. Objective Caml, 2000. Available from `http://caml.inria.fr/ocaml/`.
10. Complete source code for `lint`. Available online from `http://www.metaocaml.org/examples/lint.ml`, 2003.
11. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2003.
12. The MetaML Home Page, 2000. Provides source code and documentation online at `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.
13. 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`.
14. Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.
15. Tim Sheard. Accomplishments and research challenges in meta-programming. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineer SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 2–44. ACM, Springer, October 2002.
16. Tim Sheard, Zine El-Abidine Benaissa, and Emir Pašalić. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, 1999. USENIX.
17. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [13].
18. 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.
19. Walid Taha and Henning Makholm. Tag elimination – or – type specialisation is a type-indexed effect. In Subtyping and Dependent Types in Programming, APPSEM Workshop. INRIA technical report, 2000.
20. Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinksi, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 257–275, 2001.
21. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
22. 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.