# A Gentle Introduction to Multi-stage Programming*

*** Preliminary Draft ***

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 concisely, 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 tutorial is aimed at the reader interested in learning the basics of MSP practice and research. The tutorial is broken into two parts. The first is a hands-on introduction to MSP in the context of an MSP language called MetaOCaml. The second is an introduction to research in this area, and a number of important current research challenges.

## 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 Basic Problems in Building Program Generators

In a general-purpose language, we typically 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)"`. While constructing and combining fragments represented by strings can be done concisely, deconstructing them is verbose. More seriously, there is no *automatically verifiable* guarantee that programs thusly constructed are syntactically correct. For example, `"f (,y)"` can have the static type `string`, but this string is clearly *not* a syntactically correct program.

* This work was supported by National Science Foundation research grant ITR 0113569 entitled "Putting Multi-stage Annotations to Work".

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). Additionally, constructing data type values that represent trees can be verbose.

MSP languages provide concise syntax similar to that used to build strings. In contrast to strings, 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 type correct. In such languages, any statically well-typed program generator can only generate well-typed programs.

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 discussed some more examples of MSP.

### 1.2 Basic MSP Constructs in MetaOCaml

To illustrate how MSP addresses the above problems we consider a small example in MetaOCaml. MetaOCaml [5] is an MSP extension of the function programming language OCaml [13]. In addition to containing traditional imperative, object-oriented, and functional features, MetaOCaml provides constructs for staging. *Staging* a program is modifying it so that the work it performs is carried out in discrete stages. This is done with the goal of improving overall performance.

MetaOCaml has three staging constructs. Brackets 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 session illustrates the behavior of Brackets in MetaOCaml:

```
# 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.

In addition to delaying the computation, Brackets are also reflected in the type. The type in the last declaration is `.<int>.`, read *"Code of Int"*. The type

of a code fragment reflects the type of the value that such code would produce when it is executed. This 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>.`). Note also that when typing multi-stage languages one must ensure that no attempt is ever made to use a variable before its value becomes available. Mechanisms for addressing this concern are described in the second part of the paper.

Escape 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)`. Escape combines delayed computations efficiently in the sense that the combination of the subcomponents of the new computation is performed while the new computation is being *constructed*, rather than while it is being *executed*. This subtle distinction has a significant effect on the runtime performance of the generated code.

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

```
# let c = .! b;;
val b : 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 assure us that that no runtime errors will occur in these subroutines. Not only can these type systems exclude generation-time errors, but they also ensure that generated programs are both syntactically well-formed and well-typed. Therefore, the dynamic invocation of the compiler cannot fail.

Finally, it is important to note that program generation requires renaming. Consider the following contrived but minimal staged program:

```
# 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 1`, we get 7 is the answer. If we apply the function above (with the staging annotations) to `e .<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>.
```

Note that whereas the source code only had `fun x -> ...` inside brackets, this code fragment was generated three times, and each time it produced `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, the answer of running the stage program would be different. Thus, automatic renaming of bound variables is not so much a feature, rather, it is the absence of renaming that seems like a bug.

### 1.3 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. Program generation can help to reduce the runtime overhead of sophisticated abstraction mechanisms. The main goal of MSP is to make it possible for programmers to write generic programs without having to pay a runtime penalty for this generality.

MSP admits an intuitively appealing method of designing and implementing generative systems:

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 subtle, and can be a critical step toward effective multi-stage programming. Fortunately, it has been thoroughly investigated in the context of partial evaluation where it is known as *binding-time engineering* [12].
3. Staging annotations are introduced to specify explicitly 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*, can be summarized by the slogan:

| A Staged Program = A Conventional Program + Staging Annotations |
|---|

The conciseness of annotations means that program generators derived in this way are often simple variations on conventional programs. We expect that, for a wide range of applications, this approach will be more effective than writing generators in a general-purpose programming language and from scratch. We also expect that power of the static type systems available for MSP languages will reduce testing costs.

### 1.4 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 repeatedly perform the work associated with these inputs. To illustrate this, we consider the following implementation of a simple function in MetaOCaml: [12]

```
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* 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 square (the second power) often, it is convenient to define a special function such as:

```
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 simply the integer `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 just multiplying `x` by itself 2 times. Using only unfolding and the definition of `power`, we know that the answer can be computed more efficiently by:

```
let power2 x = 1*x*x;;
```

We also do not want to write this by hand, as there may be many specific constant powers that we want to use (other than two). So the question is, can we *automatically* build such a program?

In an MSP language such as MetaOCaml, the answer is yes. All we need to do is to *stage* the power function by annotating it as follows:

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

This function still takes two arguments, but now the second argument is no longer an integer, but rather, a *code of type integer*. The result of the function 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` on the second line. By inserting Brackets around the multiplication expression, we now return a code of integer instead just an integer. The Escape around the recursive call to power means that it is performed immediately.

Note that added staging constructs can be viewed as simply "annotations" on the original program, and that they 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 (early) 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>.))>.;;
```

The outermost annotation is Run (`.!`). This will compile (and if necessary, execute) its argument. The argument itself is essentially the same as what we used to define `power2` before, except that we have also added some annotations. The annotations say that we wish to construct the code for a function that takes one argument (`fun x ->`). We do not actually spell out what the function itself should do. Instead, we use the Escape construct (`.~`) to make a call to the staged power function `power`. We pass two arguments. The first one is a regular integer argument, `2`. The second argument is a delayed integer. The delayed integer is the term `.<x>.`. Because the call to the `power` function is Escaped, it *escapes* the delaying effect of the surrounding Brackets, and is performed immediately. This means that the first thing that gets done when we enter (or run) the expression above is this call to `power`. It returns a delayed value containing exactly `.<1*x*x>.`. The Escape then inserts this into the context of the surrounding Brackets, and we have `.< fun x -> 1*x*x >.`. Now evaluating the argument to `.!` is complete. It is then passed to the compiler. The compiler returns a value and binds it to `power2`. This value behaves exactly as if we had defined it "by hand" to be `fun x -> 1*x*x`. The staged `power2` will behave exactly like the unstaged `power2`, but it will run faster.

### 1.5 Implementing DSLs Using MSP

An important application for MSP is the implementation of domain-specific languages (DSLs). Languages can be implemented in a variety of different ways. For example, a language can be implemented by an interpreter or by a compiler. Compilers are often orders of magnitude faster than interpreters, but require significant expertise and take orders of magnitude more time to implement than interpreters. Using the strategy outlined above for using MSP languages, if we start by first writing an interpreter for a language and then stage it, we get *a staged interpreter*. Such staged interpreters are often almost as simple as the interpreter we started with, but can have performance comparable to that of a

compiler. The reason for this is that a staged interpreter becomes effectively a *translator* from the DSL to the host language (in this case MetaOCaml). Then, by using MetaOCaml's Run construct, the composition is in fact a function that takes DSL programs and produces machine code[1].

To illustrate this approach, we consider the implementation of a simple programming language (let's call it `lint`) with support for integer arithmetic, conditionals, and recursive functions. 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:

```
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.

To associate both variable and function names with their values, an interpreter for this language will need a notion of environment. Such an environment can be conveniently implemented as a function from names to values. If we lookup a variable and it is not an 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) if first tests to see if it's 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
let env0 = fun x -> raise Yikes    let fenv0 = env0
let ext env x v = fun y -> if x=y then v else env y
```

Given an environment binding variable names to their runtime values and function names to values (let's call them `env` and `fenv`, respectively), an interpreter for an expression `e` can be defined as follows:

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

```
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:

```
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, for which there is vast literature on how 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.

The question then is how the performance of such an interpreter fairs. 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.

MSP allows us to keep the conciseness and clarity of the implementation given above and also regain 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:

```
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)>.

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.

Note that 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:

A staged interpreter is a translator

With a careful analysis of the staged interpreter, the reader can convince herself that the translator we have just implemented introduces no unnecessary work in its result.

It is important to keep in mind that the above example is quite simplistic, and that there are a lot of interesting technical issues that arise when we try to 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 better idea of the scope of this technology.

Returning to our example language, a generally desirable feature of programming languages is error handling. For example, the original implementation of the interpreter uses the division operation. Because the division operation can raise a divide-by-zero exception, if the interpreter is part of a bigger system, we might 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—. Note also

that such a value would have to be propagated and dealt with everywhere in the interpreter:

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

```
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)>.
```

Unfortunately we find that the performance of code generated by this staged interpreter is 4 times slower than the first staged interpreter (with 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. On closer inspection, however, we see that which branch we take in every `match` is determined.

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.

On closer inspection of the staged program, we find that the source of the problem is the `if` statement that appears in the interpretation of `Div`. In particular, because `y` is bound at the second level (that is, inside code), we cannot perform the test `y=0` early. As a result, we cannot determine if we will return a `None` or a `Some` value.

The problem can be avoided by what is called a *binding-time improvement* in the partial evaluation literature. 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, we simply rewrite the interpreter in continuation-passing style (CPS), which produces the following code:

```
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))
...

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;;

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

In the unstaged setting, the performance of this interpreter is even worse than the last one. In terms of staging, however, we can stage the expression interpreter as follows:

```
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)
```

What we could not do here that we could do before is to Escape the application of the continuation to the branches of the `if` statement that was previously problematic. Maybe surprisingly, the generated code is exactly the same as what we got from the first interpreter. This will always be the case when the source program 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 (10 / 2))>.
```

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:

```
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)

let rec inline n body =
  if n=0 then body else fun x -> body (inline (n-1) body x)

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
           inline 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 runs faster than that produced by the first staged interpreter. It 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:

```
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)

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
             inline 1 body (fun y -> .<f .~y>.) .<x>.)
          in .~(pevalK9 (Program(tl,e)) env
                        (ext fenv s1 (fun y -> .<f .~y>.)) k)>.
```

### 1.6 Measuring Performance

MetaOCaml provides support for collecting performance data, so that we can verify that staging a program actually results in changing the time it takes to compute it. 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 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:[2]

[2] System specifications: MetaOCaml bytecode interpreter running under Cygwin on a Pentium III machine, Mobile CPU, 1133MHz clock, 175 MHz bus, 640 MB RAM.

| 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 | 87% | 85% |
| `eval8` | Inlining, no duplication | 97% | 97% |
| `eval9` | Inlining, CPS | 90% | 85% |

Timings are normalized relative to writing the two example programs that we are interpreting (`Fact10` and `Fib20`).

### 1.7 Recognizing Opportunities for Using MSP

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, in 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. Symbolic computation is a form of staged 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[3].

[3] Not to mention the century-long "stages" that were needed to evolve the theory behind many of these 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".

### 1.8 Meta-programming vs. Multi-stage vs. Multi-level vs. two-level

*Meta-programs* are programs that manipulate other programs. *Object-programs* are programs manipulated by other programs. *Program generators* are meta-programs that produce object-programs as their final result. Meta-programming can be used to overcome limitations of an existing programming language. Such limitations can either be performance or expressivity problems. We distinguish between two ways of improving performance using meta-programming: Translation, and staging. Translation is concerned with the specifics of the language of the resulting program. Staging is independent of such specifics.

Abstract machines, whether realized by software or by hardware, vary in speed and resource usage. It is therefore possible to reduce the cost of executing a program by re-mapping it from one abstract machine to another. As hardware machines can be both faster and more space efficient than software ones, such re-mappings commonly involve producing machine or byte code. We will call this technique *translation* to distinguish it from staging. Translation is an integral part of the practical compilation of programming languages, and is typically performed by the *back-end* of a compiler.

Given that staging is not concerned with the specifics of the language that we generate to, what can it achieve? In some sense, the key observation that makes staging interesting is that non-trivial performance gains can be achieved using only staging, and without need for translation into a "different" language. MetaOCaml provides the software developer with a programming language where the staging aspect of a computation can be expressed in a concise manner, both at the level of syntax and types. This way, the programmer does not need to learn a low-level language, yet continues to enjoy many of the performance improvements previously associated only with such low-level languages. Furthermore, when translation is employed in an *implementation* of MetaOCaml, translation too can be exploited by the MSP programmer through using the Run construct.

The main technical challenge in MSP research has to do with the run construct and not so much the "multi-" aspect of these languages. Taking the Run construct out, we are left with what is called multi-level languages (as opposed to multi-stage). The level of a term is the number of Brackets that surround it (minus the number of Escapes). The stage of a computation is the time at which it occurs, usually determined by the number of Runs applied to the term. In multi-level languages, there is no Run in the language, and so Runs are only applied outside programs. This ends up meaning that levels have a direct one-to-one correspondence with stages. In the multi-stage setting, however, Run can be applied as part of the computation of any MSP program. The simple

connection between levels and stages breaks in this setting. The name multi-stage rather than multi-level is chosen to emphasize this difference. Even if we start with only a two-level language and add Run, the number of stages becomes unbounded (at least in the untyped setting, it is easy to write a program in such a language that involves an arbitrary number of stages).

Another interesting feature of multi-stage languages is cross-stage persistence (CSP). This feature is simply the ability to include any value defined in one stage into code that will be executed in a future stage. It simply means that we do not always have to rewrite all the code that we generate, if it is already defined at the current level. As such, CSP can improve code reuse. Unless we have a reason to distinguish $n$-level programs from just "programs", taking $n$ to be any constant will inevitably show up as a limitation.

### 1.9 Relation to Partial Evaluation

Partial evaluation (PE) is a form of automatic MSP. In fact, a particular class of partial evaluators called "offline" consist of two steps: the first, called *binding-time analysis (BTA)* generates a two-level program. The second step simply executes this two-stage program on a static input.

A natural question to ask is why we might be interested in MSP when PE is available. The main reason is that staging decisions can be too complex to determine automatically. There may be more than one way to stage a program, and the partial evaluator might not always know which one is best. MSP languages can help programmers understand where binding time improvements are needed, and what kinds of binding time improvements can help in generating better programs. In general, we cannot expect the partial evaluator to figure out what binding-time improvements can help yield better generated programs.

Whenever performance is an issue, control of evaluation order is important. BTA optimizes the evaluation order given the time of arrival of inputs, but sometimes it is just easier to say what is wanted, rather than to force a BTA to discover it [11]. Automatic analysis such as BTA are necessarily incomplete, and can only approximate the knowledge of the programmer. By using explicit annotations, the programmer can exploit his full knowledge of the program domain. In a language with manual staging, having explicit annotations can offer the programmer a well designed back-door for dealing with instances when the automatic analysis reaches its limits.

Annotations can alter termination behavior in two ways: 1) specialization of an annotated program can fail to terminate, and 2) the generated program itself might have termination behavior differing from that of the original program [12]. While such termination questions are the subject of active investigation in partial evaluation, programming with explicit annotations gives the user complete control over (and responsibility for) termination behavior in a staged system. For example, any recursive program can be annotated with staging annotations in two fundamentally different ways. Consider the power function. The first way of annotating it is:

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

The second way of annotating it is as follows:

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

Intuitively, all we have done is "factored-out" the Brackets from the branches of the `match`-statement to one Bracket around the whole `match`-statement. This function is perfectly well-typed, but the annotations have just created a non-terminating function out of a function that was always terminating (at least for powers of 0 or more). When applied, this function simply makes repeated calls to itself, constructing bigger and bigger code fragments. In partial evaluation, this problem is known as *infinite unfolding*, and partial evaluation systems must take precautions to avoid it. In MetaOCaml, the fact that there are such anomalous annotations is not a problem, because the programmer specifies explicitly where the annotations go. In particular, whereas with partial evaluation an automatic analysis (BTA) can alter the termination behavior of the program, with multi-stage programming the *programmer* is the one who has both control over, and responsibility for, the correctness of the termination behavior of the annotated program.

Finally, it is important to keep in mind that altering the order of evaluation of statements that have side effects can change the final outcome of a program.

Overall, we view MSP and PE to be complementary techniques, and we hope to see more integration between the two techniques in the future.

## 2 Results and Challenges of MSP Research

Over the last few years, significant efforts have been invested in developing the concept of multi-stage programming. An important driving force has been to develop type systems for MSP languages. This, in turn, has motivated a number of studies in the semantics of such languages. While it may not be obvious at first glance, the notion of type safety is highly dependent on the specifics of the formulation of the semantics of a language.

Research on MSP languages can be roughly divided into the following (interrelated) dimensions:

1. **language design** is concerned with the overall form of the language. It includes issues such as what constructors are introduced into the language, the syntax for each particular constructor, the extent to which typing will be static or dynamic, and support for debugging and profiling tools. Naturally, language design is closely intertwined with other areas of research in MSP.

In some sense, however, it specifies a set of desirable features in a language (for a given class of MSP languages) that provides us with a classification for the research on other aspects such as semantics and type systems. At the same time, the design space of languages is bound by what is semantically and type theoretically possible.

In a rather distinct thread of research, MSP languages have been used as a semantic basis for designing and specifying the semantics of macro-languages. It has been shown that building macro systems on top of an MSP calculus can aid both in designing both sound static type systems [9] and sound educational theories for languages that support macros [17].

2. **semantics** is the mathematical definition of what outcome a program in a given language should yield.[4] There is a wide range of ways in which a semantics can be specified, and often, such specifications can be shown to be equivalent. Early studies on two-stage languages generally used denotational semantics. Eventually, there were two different kinds of two-level languages that were simultaneously under investigation. The ones studied by Jones et al. supported computing with unbound variables (and are therefore closer in spirit to MetaOCaml and much of the work on high-level program generation), whereas the ones studied by Nielson and Nielson supported only closed code fragments. Glueck et al. were first to generalize the idea of two-level languages to the multi-level setting, thus allowing generated programs themselves to contain Brackets. The first semantics for a multi-level language was given by Davies [8]. Rather than using a denotational semantics, however, Davies used an operational semantics. This was quite an important step for multi-stage languages (which are multi-level languages with a Run construct), because it proved quite hard to give a denotational semantics to such languages. In fact, even today, there is a limited number of denotational formulations for the semantics of two- or multi-level languages that support open code.

   The big-step operational semantics for multi-stage languages has a number of useful features, including the fact that it provides a concise and self contained definition where the only non-trivial operation is standard substitution. This may surprise the reader given that earlier we stated that MetaOCaml must rename bound variables inside code. The fact is that standard substitution dictates such a renaming, and is a very standard and concise mechanism for specifying the need for such renaming. It has been shown that, starting from only this big-step semantics one can define a notion of observational equivalence. This notion allows us to formally define a notion of equational theory for the language, and show that it respects the big-step semantics. One of the unique features of this semantics compared to denotational models is that it allows us to equate terms inside Brackets

[4] The reader interested in this area of research might find a number of introductory references to be useful, especially for type systems [7,6,2,10,20] and the lambda calculus [1].)

not only syntactically but also semantically. For example, we can correctly in view `.<1+1>.` as equivalent to `.<2>.`.

3. **static type systems** are a test performed on source programs. If the test succeeds, it usually gives us information about the runtime behavior of a program. Generally, we use type systems when we are interested in excluding certain kinds of runtime events (usually called errors). To specify such events accurately enough that we can prove the soundness of a type system requires that we define the notion of undesirable events on top of a given formal semantics. Operational semantics for multi-stage languages have proven particularly useful for this purpose.

   The first type systems for Jones-style two-level languages only attempted to type the first stage. The second stage was assumed to be untyped, and the primary concerns were to 1) ensure that the first stage computations were free of runtime errors, and 2) that there were no binding-time mistakes, in the sense that any variable that will only be available in the second stage is not used in the first stage.

   Davies presented a type system for multi-level languages that statically assured that all computations in all stages will not generate runtime errors. Furthermore, all variables that will be available at level $n$ are only used at level $n$.

   The first key challenge for MSP language research was to safely incorporate the Run construct into the multi-level setting. A first type system that achieves this goal was presented by Taha, Benaissa and Sheard [16]. A limitation of this type system was that it did not allow nested uses of the Run construct, and so in a sense that the Run construct could only appear at top level. A second approach was proposed by Moggi, Taha, Benaissa, and Sheard [14], whereby a new type constructor (like "code" in MetaOCaml) was introduced to assert explicitly that a particular code fragment is closed with respect to dynamic variables. The most recent proposal by Taha and Nilsen [19] allows the execution of open code, and may be notationally lighter than the previous approach.

   An important current challenge in typing MSP is finding expressive static type systems that support the storage of open code. Current type systems for MSP effects [4] restrict storable values to closed code fragments.

   The minimalist DSL we consider in the first part of the paper has a simple type system (we only have integers and functions over integers). If wish to move to a setting where we have an unbounded number of static types (which is the case either when we want to allow the user to define their own types, or when the language itself has an unbounded number of types), the static type system of mainstream functional languages forces to introduce unnecessary tags, that can have a large runtime cost. This limitation can be overcome if the language supports dependent types. In the current implementation of MetaOCaml, this problem is avoided by a specialized runtime transformation that operates on source code [18]. With dependent types, we may be able to statically guarantee that no tags remain at runtime.

4. **implementation** is concerned with both the design, engineering, and theory involved in realizing MSP languages. The earliest efforts in implementing MSP languages predate the term MSP, and are in the context of LISP and Scheme macro systems, and quasi-quote and comma mechanisms. This setting was the starting point for many efforts from the DIKU TOPPS group and the Tempo group.
   MetaOCaml grew directly from the efforts of Sheard and his group of MetaML (and before that CRML). MetaML was built from scratch as an interpreter for SML/NJ extended with staging constructs. The main design contribution of the MetaOCaml effort is that it is based on the existing compiler for the programming language OCaml. At the theoretical level, a formal argument was given for the correctness of the compilation strategy used in implementing MetaOCaml.
   While the current focus on the MetaOCaml effort is on implementations based on representing delayed computations by parse trees, the next step in this project is to consider implementations that employ direct runtime generation of low-level code. Such strategies have been used in the 'C, Tempo, and Popcorn systems. Ultimately, we hope that MetaOCaml will provide the user with several implementation strategies with a number of different performance tradeoffs.
   An often overlooked aspect of MSP implementation is pretty-printing. If a purely high-level representation is used for code, then the problem reduces to (the substantial but nevertheless) standard problem of pretty-printing source programs. Note, however, that CSP means that we might generate code that refers directly to functions defined in the first stage. Generally speaking, at runtime such functions are already compiled. A similar situation arises if we choose to represent delayed computations with dynamically generated code. In both cases, we need techniques for printing the definition of such compiled programs in a portable, readable manner.
   The issue of profiling, debugging, and provide useful error messages seems to be still virgin territories for research.
5. **applications** from a pure operational (untyped and non-hygienic) point of view, MSP has been around for a long time, and has been used to implement a wide range of applications. In the typed setting of languages like ML and Haskell, MSP brings back a useful language feature that had been (for a long time) cast out by the need to ensure static typing. It is because of the clear semantics and static typing for MSP languages that they have regained the attention of the typed programming languages community. Currently, we are not aware of large-scale examples of MSP programs in MetaML or MetaOCaml. MetaOCaml, however, has not been officially released yet. Once it is released, the experience of users of the language will be an important indicator of the success of its type system.
   We expect that DSL implementation will be a particularly successful class of applications for MSP.

Backing up from the "current issues" described above, there are also a number of higher-level questions we hope that future research will answer:

1. *How well does multi-stage programming work in practice?* While significant questions regarding the theory of multi-stage languages still remain open, two usable implementations are coming close to being completed: MetaOCaml and Template Haskell. A big question that remains to be settled is the extent to which this technology can scale to large, industrial strength applications. Answering this question will require conducting a large number of case studies on both applications that are already implemented in the base language for each of these systems (Ocaml and Haskell) and new applications that are implemented from scratch in the multi-stage extensions. Promising domains include DSLs, graphics, numerical computation, and virtual machine implementations.
2. *What new abstraction mechanisms are possible?* The main potential benefit of MSP is facilitating the use of sophisticated abstraction mechanism without forcing programs to pay a runtime penalty. Given this technology, what new abstraction mechanisms can be introduced to increase the safety, security, reliability, or interoperability of software?
3. *Are there more expressive type systems for imperative multi-stage languages?* Existing proposals for typed multi-stage languages restrict the program to storing only closed values.
4. *To what extent can staging annotations both at the level of types and terms be reduced?*
5. *Can we avoid the need for rewriting programs in CPS before staging?* It may be possible, through the use of an $\eta_v$-like reduction, to produce optimal results without having the user rewrite the program in CPS. Bondorf has studied improving binding times without resorting to explicit CPS conversion [3]. The work of Sabry and Wadler suggests that the use of Moggi's $\lambda_c$ may also be relevant to dealing with this problem [15].
6. *Should Run be parameterized?* By exposing the Run constant to programmers, they will eventually have new demands for controlling the behavior of this new operation, and will therefore pose more interesting questions for the developers of compilers supporting this construct. Run is a way for the user to "appeal to the higher power of the compiler developer and to avoid meddling with the complex and fragile implementation details," and the user will necessarily be picky in specifying what he is asking for: A light-weight compilation phase that produces machine code quickly, or a heavy-weight phase that takes longer but produces more efficient code? Compiling for memory or for speed?
7. *Is there a modular way for supporting a heterogeneous MSP language based on MetaOCaml?* By focusing on the staging aspect of MSP, previous work addressed a number of concrete solutions to fundamental problems in staging. However, another important application for MSP is translation. It would be very useful to extend the MSP paradigm to handle multiple different object-languages at the same time. For example, it seems reasonable to index the code type by the name of a language, for which there is an associated syntax, and special typing rules. Such a scheme is implicit in the meta-language used by Jones (see for example [12]) for describing partial evaluation. An

example of this kind of indexing already exists in the environment classifiers approach to typing multi-stage languages [19].
We believe that such a framework may even be appropriate for dealing with run-time code generation. Run-time code generation is concerned with the efficient generation of specialized machine code at run-time. While a compiled implementation of MetaOCaml can provide a version of Run that generates machine code, it is implicitly assumed to be making calls to a compiler, which is generally not considered fast enough for run-time code generation. If machine code is considered as a typed language, we believe that it can be incorporated into a Heterogeneous MetaOCaml setting. With the ongoing research efforts on typed assembly languages, this possibility is not far-fetched.

8. *Are there practical, theoretically sound approaches for allowing the programmer to specify optimizations on MetaOCaml code?* Recent work by Craig Chamber's group seems very relevant and very interesting.
9. *Can multi-stage languages be used to support extensible systems?* It has been shown that MSP calculi can be used to define a large class of generative macros, that this approach supports macros that define new binding constructs, and that it can be used to define type-safe macro languages [9]. Recent work has also shown that this approach can be used to define macro systems that are amenable to educational reasoning [17]. The natural next question in this direction is whether this approach can be extended to support extensible syntax as well as extensible type systems.
10. *To what extent can multi-stage programming be applied to other programming paradigms?* The combination of MSP and functional programming is appealing, mainly because the benefits of each of the two paradigms do not interfere. An important question to answer is the extent to which MSP can be useful introduced into other settings, including imperative, object oriented, and logic programming.
11. *To what extent can MSP and PE be integrated?* The relationship between multi-stage programming and partial evaluation concepts such as self-application and the Futamura projections [12] has been addressed to some extent in the context of tag elimination, but still remains largely unexplored.

*Acknowledgments:* Members of Dagstuhl meeting, especially Charles, Krzysztof, Shriram, David Wile, Todd, Paul Kelly, Kevin Hammond, Chris Lengauer, Yannis. Stephan Ellner for careful proof-reading of a draft.

## References

1. BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.
2. BARENDREGT, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, Oxford, 1991.

3. BONDORF, A. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California* (1992), pp. 1–10.
4. CALCAGNO, C., MOGGI, E., AND TAHA, W. Closed types as a simple approach to safe imperative multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)* (Geneva, 2000), vol. 1853 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–36.
5. CALCAGNO, C., TAHA, W., HUANG, L., AND LEROY, X. Implementing multi-stage languages using asts, gensym, and reflection. In *Generative Programming and Component Engineering (GPCE)* (2003), K. Czarnecki, F. Pfenning, and Y. Smaragdakis, Eds., Lecture Notes in Computer Science, Springer-Verlag.
6. CARDELLI, L. Typeful programming. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds., IFIP State-of-the-Art Reports. Springer-Verlag, New York, 1991, pp. 431–507.
7. CARDELLI, L. Type systems. In *The Computer Science and Engineering Handbook*, A. B. Tucker, Jr., Ed. CRC Press, 1997.
8. DAVIES, R. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)* (New Brunswick, 1996), IEEE Computer Society Press, pp. 184–195.
9. GANZ, S., SABRY, A., AND TAHA, W. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *the International Conference on Functional Programming (ICFP '01)* (Florence, Italy, September 2001), ACM.
10. HINDLEY, J. R. *Basic Simple Type Theory*, vol. 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1997.
11. JONES, N. D. What not to do when writing an interpreter for specialisation. In *Partial Evaluation* (1996), O. Danvy, R. Glück, and P. Thiemann, Eds., vol. 1110 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–237.
12. JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
13. LEROY, X. Objective Caml, 2000. Available from `http://caml.inria.fr/ocaml/`.
14. MOGGI, E., TAHA, W., BENAISSA, Z. E.-A., AND SHEARD, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–207.
15. SABRY, A., AND WADLER, P. A reflection on call-by-value. In *Proceedings of the International Conference on Functional Programming* (Philadelphia, 1996), pp. 13–24.
16. TAHA, W., BENAISSA, Z.-E.-A., AND SHEARD, T. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming (ICALP)* (Aalborg, 1998), vol. 1443 of *Lecture Notes in Computer Science*, pp. 918–929.
17. TAHA, W., AND JOHANN, P. Staged notational definitions. In *Generative Programming and Component Engineering (GPCE)* (2003), K. Czarnecki, F. Pfenning, and Y. Smaragdakis, Eds., Lecture Notes in Computer Science, Springer-Verlag.
18. TAHA, W., MAKHOLM, H., AND HUGHES, J. Tag elimination and Jones-optimality. In *Programs as Data Objects* (2001), O. Danvy and A. Filinksi, Eds., vol. 2053 of *Lecture Notes in Computer Science*, pp. 257–275.
19. TAHA, W., AND NIELSEN, M. F. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)* (New Orleans, 2003).
20. WRIGHT, A. K., AND FELLEISEN, M. A syntactic approach to type soundness. *Information and Computation 115*, 1 (1994), 38–94.