A Bytecode-Compiled, Type-safe, Multi-Stage Language

Cristiano Calcagno
Queen Mary, London
and
DISI, Genova
ccris@dcs.qmul.ac.uk

Walid Taha, Liwen Huang
Yale University
New Haven, CT.
taha@cs.yale.edu,
liwen.huang@cs.yale.edu

Xavier Leroy
INRIA, Roquencourt
Xavier.Leroy@inria.fr

ABSTRACT

Inspired by the successes of program generation, partial evaluation, and runtime code generation, multi-stage languages were developed as a uniform, high-level, and principled view of staging. Our current goal is to demonstrate the utility of these languages in a practical implementation. As a first step this paper presents MetaOCaml, a type-safe, multi-stage language, built as an extension to OCaml's bytecode compiler. Future-stage computations are represented as source programs. This makes it possible to ensure type-safety, produce better dynamically compiled code, and apply crucial runtime source-to-source transformations such as tag elimination. We have used MetaOCaml to measure performance for a set of small staged programs. The gains are consistent with those of partial evaluation and runtime code generation, and support the claim that multi-stage languages are well-suited for building staged interpreters, even when the runtime compilation times are taken into account.

1. INTRODUCTION

Program generation, partial evaluation (PE) [19], dynamic compilation, and runtime code generation (RTCG) [23] are all techniques to facilitate writing generic programs without unnecessary runtime overheads. These systems have proven effective even for implementing high-performance operating systems [38, 8, 7]. Multi-stage languages have been developed as a uniform, principled, and high-level view of these diverse techniques. A key idea in these languages is the use of three simple *annotations* to allow the programmer to distribute a computation into distinct *stages* [50, 43, 52]. To illustrate how these annotations work, we consider a small example in MetaOCaml, the language presented in this paper. The `power` function [19] can be defined as follows:

```
let power (n,x) = (* :  int×int -> int *)
  if n = 0 then 1
     else if even n then square (power (n/2,x))
             else x * power (n-1,x);;
```

where `square` is defined by `let square x = x * x`.

By staging such a function we get another function that generates specialized code at runtime. This is useful, for example, if we want to compute a certain exponent (say 72) for a number of different values. The three annotations that we will use in this paper are the same as those of MetaML [50, 43, 52], and are Brackets `.<•>.`, Escape `.~•`, and Run `.! •`. The Brackets `.<•>.` construct a future-stage computation, Escape `.~•` interrupts the construction of a future-stage computation to perform an immediate computation. Thus, an Escape must always occur inside a Bracket, and the result of the "escaped computation" must itself be a future-stage computation. Run `.! •` executes a future-stage computation. To stage `power` so that it performs some of its computations before its second argument is known, we annotate it as follows:

```
let power' (n,x) = (* :  int×.<int>.  -> .<int>.  *)
  if n = 0 then .<1>.
     else if even n then .<square .~(power' (n/2,x))>.
             else .< .~x *  .~(power' (n-1,x))>.;;
```

The type has changes to reflect the fact that `x` is "late": the type `.<int>.` (read *"code of int"*) means this is a value of type `int` "but it cannot be used yet". Any sound type system for a multi-stage language (c.f. [47, 32]) ensures that this condition is enforced. For example, because we assumed the input `x` is not yet know (or "late"), the return type of the function (which depends on `x`) can only be late.

To be consistent about the return type we annotate the `1` with Brackets. Annotating the next line causes the construction of a delayed application of the `square` function. The argument in this application is the result of the escaped call to `power'`. Because of the Escape, this call is performed immediately and not in a future stage. This call returns a delayed computation that becomes the argument to `square`. Exactly the same is done on the next line, except that the delayed computation is a multiplication `*` rather that an application of `square`. The `x` is escaped because it is bound to a delayed computation that we want to incorporate into the context of a bigger delayed computation.

To use `power'` to generate a specialized version for x^{72}, the slightly non-intuitive step is how to create a "late value". Such a value is just a variable bound by a dynamic lambda: (system response in italics):

```
let power72' = .<fun x ->  .~(power' (72,.<x>.))>.;;
val power72' = .<->.  :  .<int -> int>.
```

The variable `x` is bound by the `fun` construct (MetaOCaml's version of lambda) inside brackets. So, when we are computing `(power' (72,.<x>.))` the variable `x` is not yet bound.

Nevertheless, the application to `power'` is performed, because `power'` returns its result without ever trying to *use* its argument other than by escaping it into the context of another late fragment.

While MetaOCaml does not print back future-stage computations (to give freedom in how future-stage computations are represented), in MetaOCaml the result of the above computation is essentially the following:

```
.<fun x -> %square (%square (%square
        (x %* %square (%square (%square
        (x %* 1))))))>.  :  .<int -> int>.
```

The percentage `%` is added in front of `square` and `*` to indicate that these names in fact have no significance (except to the human reader) in this representation. Rather, it is the *value* that was bound to those variables at the time this fragment was constructed that is referred to here. This is called *Cross-stage Persistence* (CSP) [50], and is the way constants from outside Brackets are incorporated directly into a delayed computation. Without it, multi-stage programming would be quite impractical, as it would require writing functions such as `print` and `length` inside Brackets before they can be used inside Brackets, and then again inside Bracketed Brackets, and so on. This construct can also complicate the semantics [43, 45], and its implementation can be subtle.[1]

Finally, to re-integrate this new specialized function generated at runtime into the currently executing program we use the Run `.!` construct:

```
let power72 = .! power72' (* :  int -> int *);;
```

This new function can now be used just like any regular function anywhere we need to compute x^{72} efficiently.

Brackets, Escape, and Run are the only staging annotations in MetaOCaml. It is our thesis that they are sufficient for achieving performance benefits in a wide range of applications.

1.1 Background

Staging computations into distinct phases seems to have been first advocated by Jørring and Scherlis [21]. Studying staged computation from the programming language point of view goes back at least to the seminal work of Nielson and Nielson on two-level languages [33, 34]. The notion of multi-stage *languages*, however, originated in work of Jones on a different kind of two-level language developed primarily to model the internal workings of PE systems [20, 15], which were in turn inspired by quasi-quotes in LISP and to Quine's corners $\ulcorner\bullet\urcorner$ [43]. Glück and Jørgensen [12, 13] were the first to generalize the techniques of two-level languages to a multi-level setting. Over the last six years, significant efforts have been invested in the development of a theory of multi-stage computation, starting from the work of Davies on linear temporal logic [9] and Moggi on functor-category and topos-theoretic semantics [30, 31], then continuing as various studies on the semantics [47, 3, 43] and type systems for MetaML [32, 5, 4], and the use of monads for modeling meta-computations [17, 18]. Key contributions include the identification of a wide range of linguistic subtleties that arise from the introduction of staging constructs into a programming language and pointing out practical safety problems (bugs in type systems). Insights from studying these languages have also lead to solving some long-standing problems in PE, such as Jones-optimality [19, Section 6.4] using a new transformation called tag elimination [48, 49].

[1] The `%` construct itself is not available to the programmer, mainly for safety, but also because what it carries is system dependent.

1.2 Problem

The theoretical strengths of multi-stage languages are expected to translate to benefits such as clarity, soundness, and type-safety, without interfering with the key goal of staging, which is to improve performance. The only publicly available implementation of a multi-stage language is MetaML [50, 29]. The presence of MetaML has been invaluable for the conceptual development of multi-stage languages and programming and as a research and educational tool. But MetaML can still be improved in a number of significant ways [40]. For example, MetaML is interpreted, and so is not a convincing basis for demonstrating the practical potential of multi-stage languages in a compiled setting. When earlier studies tried to show how multi-stage languages can be used to implement domain-specific languages, this could only be done by inspecting the generated code, and not by measuring its performance [50, 43, 41, 52]. Printing the result of a staged interpreter and then running it separately using a compiler is problematic for a number of reasons

1. Parsers of mainstream compilers can be particularly bad at parsing automatically generated code.

2. The pervasive use of cross-stage persistence means that there must be a mechanism for de-compiling values that have been interpreted or compiled back into source code.

3. It does not help in estimating how much time it takes to generate this code with a more efficient implementation of MetaML.

In addition, MetaML was built from scratch. While this is an advantage from the point of view of portability (and MetaML runs on the many platforms that SML/NJ supports), being built from scratch makes it quite hard to be fully compatible with an existing implementation of ML, and to make sure that it has good error messages (which is particularly hard in typed functional languages).

Our goal is to show that the theoretical machinery behind multi-stage programming can indeed be put to work in a systematic and practical manner, and in ways that do not require radical changes to existing compilers. As a first step, this paper presents MetaOCaml, a new, full-blown multi-stage programming language, together with a bytecode compiled implementation. This compiler is now available online [28]. Preliminary experiments with the implementation indicate that performance gains are consistent with previous studies on PE and RTCG.

1.3 Organization of the rest of the paper

Section 2 describes the design of MetaOCaml and explains the basic design decisions and their significance. Section 3 describes the implementation of MetaOCaml, the key issues, and how these issues are addressed. Section 4 reports basic performance measurements. We have used some simple programs to collect empirical data about MetaOCaml. The examples include ones used in previous publications on MetaML, in addition to some new examples. Section 6 is a conclusion and a discussion of future work.

2. THE DESIGN OF METAOCAML

MetaOCaml is designed as an implementation of the basic multi-stage programming ideas in MetaML. So MetaOCaml's syntax for staging annotations is a minor variation of MetaML's syntax. The change is made to minimize impact on the system as an OCaml compiler.[2] So, MetaOCaml inherits MetaML's explicit **annotation language**. Also like MetaML, MetaOCaml uses a high-level code **representation of delayed computations**. MetaOCaml departs slightly from MetaML in the treatment of typing, in that type checking is always performed before executing a code fragment. This is necessary for ensuring **type safety**. The key design goal of MetaOCaml, however, is to capitalize on an implementation of an existing functional language in order to achieve a higher-level of *usability*.

In the rest of this section we discuss each of these issues in more detail, and in the context of various related systems that form important points in the design space of staging systems.

2.1 Annotation Language and Uniformity

Every staging framework provides the user with a mechanism for specifying what should be staged. We call any such mechanism the annotation language. The two extreme approaches to annotation can be described as the Binding-Time Analysis (BTA) style and the explicit style. BTA style is more declarative, and only requires the programmer to specify the order in which parameters to a function will become available. For example, DyC (read *"dicey"*) [16] annotations can be considered to be BTA style. In DyC, the programmer can also specify the aggressiveness of the optimizations to be applied based on this information. With the explicit style, the user has to specify when each fragment of the program should be executed.

BTA style can be viewed as an approach to automating explicit style. But automation in this case can come at a severe cost to both controlability and predictability, much in the same way that automation can affect the task of theorem proving. Similarly, unless supported by a clear internal language (which would probably be a multi-stage language), this kind of automation can have the down-side of making staging system harder to predict and debug.

MetaOCaml annotations are explicit style, and its three simple constructs give the programmer full control over what is generated. Furthermore, the type system statically prevents the programmer from writing unsafe programs. By interacting with the type inference/checking system in MetaOCaml's top-level loop, a programmer can learn very quickly how programs can and cannot be correctly staged. Here is a simple example (system response in italics):

```
.<fun x -> .~(x+1)>.;;
Wrong level: variable bound at level 1 and
                        used at level 0
```

Note that the system also underlines the offending variable.

An important aspect of the annotation language is its uniformity. The design of MetaOCaml is extremely uniform: as long as the phase distinction is respected, staging constructs can be introduced anywhere in an expression (this is inherited directly from MetaML). This is true even if the expression already contains staging annotations. Thus, the basic advantage of multi-level languages with respect to two level-languages is that programs of the former are closed under staging. The ability to stage staged programs is essentially the main reason why MetaOCaml supports "multi"-stage annotations. The design allows us to inherit various key features of OCaml: expressivity of type system, higher-orderness, ML-style effects (references, state, exceptions, etc), and a module system.[3] It is useful in this context to note that 'C (read *"Tick C"*) [11] does not allow the construction of future-stage computations that contain a goto, continue, case, or break statement, and only one function can be generated at a time. MetaOCaml has non of these restrictions, and so, in a sense, it is a more expressive imperative staging framework that 'C. Finally, when explicit staging annotations are combined with higher-order features, they become an excellent tool for explaining many of the basic concepts of staged computation. For example, the introduction presented the full code of the staged power example that any programmer can enter into MetaOCaml and start using runtime code generation and compilation.

2.2 Code Representation and Cost Model

The primary goal of staging is enhancing the performance of programs. The extent and the manner to which a system makes this possible is the cost model for the system. A key determinant for the cost model for staging systems is the choice of representation for future-stage computations. The two extremes of representation are as high-level (HL) source code, represented at runtime by an abstract syntax tree, or as low-level (LL) machine code. HL has been often used by PE such as Tempo [35] and by meta-programming languages such as MetaML [50]. LL has been used by dynamic compilation systems such as Dynamo [2], DyC, and by runtime code generation systems such as Fabius [25, 24, 26], 'C, the categorical RTCG machine [54], Thiemann's higher-order code splicing [53], and binary-level components [22].

The basic trade-off is that HL requires substantially more compilation at runtime, but can produce fast programs than LL. Fabius and 'C in particular developed extremely fast instruction generation (as little as six instructions might be executed at runtime to generate one instruction). Slightly slower generation times (on the order of a thousand instructions) can lead to substantially improved code. Dyc employs dynamic versions of many standard optimizations, and applies a wide range of ideas from the PE literature (which has been largely in the HL setting) to the LL setting. In addition, it uses interesting techniques like caching dynamically generated/compiled code.

MetaOCaml uses the HL model. There were a number of reasons for this choice. First, it seemed easier to implement, and easier to implement in a way that could scale from the bytecode compiler to the native code compiler. Tempo introduced a clever technique for using existing compilers to do LL RTCG, but it still requires an part which is dependent on the machine language, and is somewhat fragile in that an optimizing compiler can produce incorrect results [35]. Second, it is necessary for ensuring type safety, as ensuring type safety currently requires dynamic type checking. If LL is chosen, it would be necessary to tag all values with

[2]We choose not to implement MetaML's `lift` [50] as a separate construct. Instead, cross-stage persistence automatically performs lifting for values that can be lifted.

[3]The current implementation allows staging inside modules. Staging whole modules and structures is still work in progress.

typing information, which would mean a substantial runtime overhead. Fourth, the HL representation allows more optimizations that can be hard with LL, such as strength reduction, global scheduling, or register allocation. One of the most interesting features of Tempo is that it combines both LL and HL approaches, and those provide a rare insight into the relative effectiveness of the two approaches. In that work, it seemed that the code generated with LL is about 70-90% as fast and 1.5-3 times as big as the code generated by HL. The quality of the generated code by the LL strategy can never be better than the HL strategy: if there is a better compiler for the LL, it can be used to construct a better compiler for the HL. But more importantly, we are particularly interested in the using MetaOCaml for building staged interpreters, where a runtime source-to-source transformation called tag-elimination [48, 49] (see next subsection) can yield on the order of two or even three fold speedups (see [49] and performance section in this paper). It is not at all clear how such a transformation can be implemented in an LL context.

Practical experience reports with the performance of various representations in general is still sparse, and have often compared compilers that do a lot of optimizations with optimized RTCG engines. Being a bytecode compiler is an interesting feature of the current implementation of MetaOCaml, because the byte code interpreter is significantly faster (5-10 times) the native code compiler (which itself is considered to be a fast optimizing compiler for a functional).

An interesting possibility is to have MetaOCaml use bytecode as the representation for future-stage code. This would make the cost model to closer to LL, but it is likely make dynamic type checking, tag elimination, and various kinds of type-based optimizations much harder. It is also more work than the system that we have built, because relative jumps, for example, may need to be re-computed. Once a fully static type system is implemented, a more interesting medium will be to represent future-stage code by OCaml's `lambda` intermediate language, which is more or less a tree-like representation of bytecode.

2.3 Interpreters and Tag Elimination

A key application for multi-stage programming languages is rapidly developing compiled implementations of domain-specific languages. The feasibility of developing implementations of domain-specific languages has been demonstrated (See for example [41]). The basic idea is to develop the implementation in two steps:

Develop an interpreter Interpreters are generally the simplest way of implementing a domain-specific language. Unfortunately, they tend to suffer from a performance cost called the "interpretive overhead" [19].

Add multi-stage annotations Staging interpreters often yields a clear separation between the cost of interpreting a program and the cost of running. By separating the two, it becomes possible to completely isolate the first part of the cost. Thus, the interpretive overhead is paid only in a phase right after parsing, and is *not* repeatedly incurred during the execution of the program.

The annotated programs are generally only marginally different from interpreters, yet they can have performance comparable to that of hand-written compilers. In essence, a staged interpreter translates a user's representation of a lambda term to the compiler's representation. Thus instead of being interpreted, the term can now be compiled and executed by the existing compiler. In the PE literature, this is known as "removing an entire level of interpretive overhead". Pragmatically, a staged interpreter gives the user direct access to the compiler at runtime, and providing a simple mechanism for dynamically loading new components.

The staged lambda interpreter that will be discussed in the performance section does not achieve optimal speedups. In fact, it could be made two or even three times faster [48]. Assume the syntax for the interpreted language is represented by a datatype `e` for expressions:

```
type e = D of int | P of e * e | V of string
       | A of e * e | L of string * e;;
```

Then we define a datatype `v` for values of the language:

```
type v = I of int | F of v -> v;;
```

For convenience, we can also define two helper functions:

```
let unI (I i) = i;; let unF (F v) = v;;
```

The staged interpreter `interp'` is simply an interpreter with some staging annotations, namely brackets `.<•>.` and escapes `.~•`, added in the appropriate places. When we apply the staged interpreter to the representation of the term $(\lambda x.4)\ 5$ as follows:

```
let code1 = interp' (A (L (''a'',D 4),D 5)) env0;;
val code1 = .<unF (F (fun a -> I 4)) (I 5)>.  :  .<v>.
```

Note that the result is a host-language version of the object-language program which, when compiled, would execute at native speed. In MetaOCaml we compile and run this code by simply writing .! *code*1. The key problem, though, is that there are many extra tagging and untagging operations that are still around in `code1`. Some of them are even obviously superfluous, such as `unF (F ...)` which can be immediately reduced. But the superfluous tagging and untagging operations can be less obvious (consider the term resulting from $(\lambda x.(x\ 7))\ (\lambda y.y)$). This problem of tags is a fundamental one that arises when both the object and the meta-language are statically typed (and there is more than one type). All instances of tags in the term above are handled by a new transformation called tag-elimination [48, 49] [4] which we intend to implement in future distributions of MetaOCaml.

2.4 Type Safety in MetaOCaml

Like OCaml, MetaOCaml supports polymorphism, higher-order functions, ML-style effects (pointers, state, exceptions, and so on). The language is type-safe. This means, for example, that programs that pass the type checker never attempt to add an integer to a string or a pointer at runtime. For the most part, MetaOCaml is statically typed. But because of subtleties that can arise from the use of Run or effects (such as state, exceptions, or continuations) in combination with open code, certain staging errors can escape static typing. While there are currently type systems that can allow us to statically avoid these kinds of errors [47, 32, 5], their expressivity is not yet fully understood. Thus, in

[4] For the reader familiar with boxing/unboxing optimizations [37], tag elimination is a generalization.

designing MetaOCaml, the choice has been made to type-check any dynamically generated future-stage code before executing it. This scheme can be viewed as a hybrid between the standard multi-stage type systems, and the staged type inference system developed by Shields, Sheard, and Peyton-Jones [42].

The only reason runtime type checking can fail is "late" variables. In particular, it has been shown that having the Run construct or side-effects can cause a confusion of the level of a variable, and that this can lead to runtime errors.[5] By always type checking code before executing it we avoid this problem. If runtime type checking fails it raises an exception `Trx.TypeCheckingError`, which can be caught like an regular OCaml exception. To illustrate, consider the following example:

```
try .<fun x -> .~.! .<x>.>.
with Trx.TypeCheckingError -> .<fun y -> y>.;;
```

The `try-with` construct is OCaml's way for running a computation while providing a handler for an exception that this computation might raise. The first part of the clause contains a classic example of a simple program that would cause a runtime error during the evaluation that is initiated by the inner `.!` call [47]. The computation leads to a runtime type-checking error. When this error is encountered, the `Trx.TypeCheckingError` is raised. The last line of the example above catches this errors, and returns the computation `.<fun y -> y>.` instead.

If Run and side-effects are not used, type safety is completely guaranteed statically, and the runtime type checks are all redundant.

2.5 Usability

A system that implements new ideas but is hard to use and easy to break might not be usable in practice. Usability is a function of many factors, such as robustness, testing, portability, and user-base, among others. Robustness is in part a function of engineering, but also a function of the basic design and the analytic understanding of the new idea. Thus we view robustness as including verifiability and the ability to provide guarantees (at least) about the basic design. The verifiability of the design and type safety in a staging framework are unique features of the design of MetaOCaml.

Robustness is also affected by the extent to which the system can be tested. Many of the early staging systems do not seem to have undergone significant testing. While we have so far only written a few small multi-stage programs in MetaOCaml, the system compiles the whole MetaOCaml compiler (37K lines of executable OCaml code). This gives potential users the assurance that the new feature will at least not break old programs. This puts MetaOCaml in par with systems such as 'C, DyC, and Tempo, which do not interfere with basic functionality of the system they extend.

Portability is a function of both the language design and the implementation. This includes the nature of the annotations themselves (their syntax, type system, and semantics), whether we allow the observation of code. We don't have full information on all the systems in this regard, but DyC, for example, is part of a proprietary DEC alpha backend. 'C is available online, but only for a SPARC and a DEC microprocessor. Tempo is available online and has been used independently by people other than its developers.[6] It is available online for the 386 architecture and SPARC Solaris. By carefully extending OCaml, we are able to maintain the portability of MetaOCaml, which includes Unix, PC, and the Macintosh platforms.

Finally, a key dimension of usability is the quality of error messages, error handling, ease of downloading and availability of documentation.

Various features of OCaml contributed to the overall usability of MetaOCaml:

- Availability of a runtime bytecode generator: Like most Lisp and ML implementations, OCaml supports interactive use via a top-level loop where program fragments are compiled and executed on the fly. It was easy to re-package this top-level loop as a runtime code generator suitable for use within multi-stage programs. Compilation is done not to native machine code, but to an efficient bytecode for a virtual machine, which strikes an interesting compromise between fast compilation times and fast execution of the generated code.

- Reasonable performance: The speed of the code generated by the bytecode compiler is, on average, one-fourth of that of the code generated by the native code compiler, which itself is one of the most efficient compilers for functional languages. (The bytecode-to-native speed ratio varies between 2 and 15 depending on applications.) Also, the two compilers differ mostly in the details of the back-end. These observations give us more confidence that the gains from staging reported here should also carry over to the native code setting.

- Good error reporting: The front end of the OCaml compiler tracks the line source code location associated with each construct. When an error is encountered, this information is used to simply underline the source of the error in the source text. We have found this way of reporting errors to be very informative.

- Good runtime error handling: The bytecode compiler and the runtime system interact well, even when the compiler is being used at runtime. This is partly because the compiler is written in OCaml and can be loaded as a regular library that uses the same exception mechanism, and also because of the overall robustness and effectiveness of the exception mechanism (implemented by the runtime system).

- Wide use: OCaml is one of the most widely used functional languages, and is ported to many architectures and operating systems.

- Large code base: Not only is this essential for further investigation of the effectiveness of multi-stage features in improving the performance of real programs, but it is also important for new languages to have existing libraries that they can build on.

[5] The problematic example shown here is a variant of an example by Rowan Davies, personal communication 1997. A more detailed analysis of this example can be found elsewhere [47].

[6] Julia Lawall. Personal communication, 2000.

2.6 Summary and Interdependencies

Some of the key features of the systems we discussed and MetaOCaml can be summarized as follows:

System	L	R	A	I	P
Fabius	ML subset	LL	E	C	1
'C	C	LL(2)	E	C	2
DyC	C	LL	I	C	[1]
Tempo	C	HL & LL	I	C	2
MetaML	SML	HL	E	I	+
MetaOCaml	OCaml	HL	E	C	+

Where the columns are: – **L** Language chosen as basis; – **A** Annotation language: **E**xplicit or **I**mplicit; – **R** Representation: High-level (**HL**) or low-level (**LL**). In the case of 'C, two different LLs were studied; – **I** Implementation: **C**ompiled or **I**nterpreted; – **P** Portability: Number of available ports. In the case of DyC, this port is not publicly available.

The various dimensions of the design space that we have described above are not unrelated. Two important points to make are:

1. The combination of HL and a native code compiler can yield the biggest swings (positive or negative) in overall runtime cost. This may make it more naturally suited to the explicit style of annotation, where the programmer has direct control over all the staging decisions.

2. The LL approach may be more naturally suited for the BTA-style annotations or fully-automated optimizations in general, because of the smaller swings. [7]

3. IMPLEMENTATION

The MetaOCaml implementation is a modified OCaml bytecode compiler. Minor changes are made to the parser and type-inference/check to accommodate the three staging annotation and the one type constructor. Because these changes are very localized and are done by direct analogy with other constructs in the language, error reporting continues to be reliable and accurate (as mentioned earlier, good error reporting is particularly hard in typed functional languages).

The main change is the addition of a source-to-source compilation phase right after type-inference/checking. This one-pass phase translates the staging constructs into operations that build and manipulate abstract syntax trees, invoke the compiler at runtime, and execute the result. The parse trees are exactly the same ones used internally by the compiler.

3.1 The Core Translation Function

Because we use a functional language to implement the translation, the implementation is similar in appearance to the formal translation presented in this section. The essence of the compilation function can be illustrated as follows. Assume the OCaml compiler takes a language with the syntax:

$$e \in O ::= x \mid e\ e \mid \lambda x.e \mid e, ..., e \mid C\ e$$

[7]This also seems to be part of the appeal of the JIT approach in Java systems [1, 6]

Where x is a variable, $e_1\ e_2$ is an application, $\lambda x.e$ is a lambda abstraction, and $e_1, ..., e_n$ is a tuple of length n, and $C\ e$ is an application of a value constructor (or attachment of a union type tag) C to the result of the expression e. The idea will be to represent the staging manipulations as operations on parse trees, while will be constructed using value constructors coming from a datatype declaration such as the following:

```
type exp = Var of string          | App of exp * exp
         | Lam of string * exp    | Brk of exp
         | Esc of exp             | Run of exp
         | Csp of val
```

where each variant (respectively) represents a production in the syntax taken by the MetaOCaml compiler we want to build:

$$e \in M ::= x \mid e\ e \mid \lambda x.e \mid .\ <e>\ . \mid\ .\tilde{}e \mid .!\ e \mid \%e$$

where the first three new constructs are the standard ones, and $\%e$ is a placeholder for CSP constants. We will use the following standard short-hand:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \triangleq (\lambda x.e_2)\ e_1$$

The translation is a function $[\![\bullet]\!]^n_\rho : M \to O$, were n is an integer called the *level* of the term, and ρ is an environment mapping free variables to levels.[8] Intuitively, the level of the term will just be the number of surrounding brackets less the number of surrounding escapes. At level 0, the translation $[\![\bullet]\!]^0_\rho$ does essentially nothing but traverse the term. When the translation finds a Bracket, the level is raised by one, and the translation shifts to $[\![\bullet]\!]^{n+1}_\rho$. At levels $n+1$, for the most part, the translation takes a term and generates a new term that represents that first one. That is, if the translation encounters the term $(\lambda x.x)\ (\lambda y.y)$ at level 0 it generates the term $\texttt{App}(\texttt{Lam}(x, \texttt{Var}\,x), \texttt{Lam}(y, \texttt{Var}\,y))$.[9]

In implementation terms, at level $n+1$ the source-to-source translation roughly generates the abstract syntax tree for an expression which will construct at runtime the abstract syntax tree for the specialized program that will be passed to the runtime compiler and evaluator `metaocaml-run`.

The translation is presented in Figure 3.1. Note that the result of the translation consists mostly of either unchanged programs, operations to build trees (using C constructors), or the use of the two functions `gensym` and `metaocaml-run`. Taking the interesting cases from top to bottom (all of them at a level greater than 0), first, we encounter variables. There are two cases at level $n+1$, depending on whether the variable was bound at level 0 or any other level. If the variable was bound at level zero, then we construct a marker around it as a CSP, otherwise it is treated as expected. The next interesting case is that of a lambda abstraction, where the essential trick is to use a fresh-name function to generate a new name so as to avoid accidental name capture. Such a function is used in the translation by Gomard and Jones [15] and in the macro systems by Kent Dybvig [10]. For Brackets and Escapes, we always adjust the level parameter. But note that no `Brk` is generated at level 0, or `Esc` at level 1. Note also that there is no case for `Esc` at level 0 (such terms are rejected by the type system). At level 0, a Run

[8][With more space, we will present a few lines from the implementation of the translation here.]

[9][With more space, we will add more examples here.]

$$
\begin{array}{lcll}
[\![x]\!]^0_\rho & = & x & \\
[\![x]\!]^{n+1}_\rho & = & \texttt{Var}\ x & \rho(x) = m+1 \\
[\![x]\!]^{n+1}_\rho & = & \texttt{Csp}\ x & \rho(x) = 0 \\
[\![e_1\ e_2]\!]^0_\rho & = & [\![e_1]\!]^0_\rho\ [\![e_2]\!]^0_\rho & \\
[\![e_1\ e_2]\!]^{n+1}_\rho & = & \texttt{App}\ ([\![e_1]\!]^{n+1}_\rho, [\![e_2]\!]^{n+1}_\rho) & \\
[\![\lambda x.e]\!]^0_\rho & = & \lambda x.[\![e]\!]^0_\rho & \\
[\![\lambda x.e]\!]^{n+1}_\rho & = & \left\{\begin{array}{l}\texttt{let}\ x = \texttt{gensym}\ () \\ \texttt{in Lam}\ (x, [\![e]\!]^{n+1}_{\rho;x\mapsto(n+1)})\end{array}\right. & \\
[\![.\ <e>\ .]\!]^0_\rho & = & [\![e]\!]^1_\rho & \\
[\![.\ <e>\ .]\!]^{n+1}_\rho & = & \texttt{Brk}\ [\![e]\!]^{n+2}_\rho & \\
[\![\ .\tilde{}e]\!]^1_\rho & = & [\![e]\!]^0_\rho & \\
[\![\ .\tilde{}e]\!]^{n+2}_\rho & = & \texttt{Esc}\ [\![e]\!]^{n+1}_\rho & \\
[\![.!\ e]\!]^0_\rho & = & \texttt{metaocaml-run}\ [\![e]\!]^0_\rho & \\
[\![.!\ e]\!]^{n+1}_\rho & = & \texttt{Run}\ [\![e]\!]^{n+1}_\rho & \\
[\![\%e]\!]^0_\rho & = & e & \\
[\![\%e]\!]^{n+1}_\rho & = & \texttt{Csp}\ e &
\end{array}
$$

Figure 1: The Core Translation Function

is replaced by an application of the constant `metaocaml-run`, where, assuming that `native-run` is the existing compilation routine, `metaocaml-run` is defined as:

$$\texttt{metaocaml-run p} = \texttt{native-run env0}\ ([\![\texttt{p}]\!]^0_\emptyset)$$

CSP constants are essentially left unchanged.

3.2 Formal Verification

Based on our previous experience with the formal semantics of MetaML, we expect that formally establishing the operational correctness of this translation will be straightforward. The definitions, theorems, and proofs needed for establishing correctness will be presented elsewhere.[10]

3.3 Implementation Issues

Naturally, the implementation itself is more complicated than the core translation function presented above. In addition to the fact that abstract syntax trees for the rest of the language are bigger, OCaml propagates source code file information in each construct, for very accurate error messages. Furthermore, the type of the abstract syntax tree changes during type-checking (although its structure does not change very much), raising the need to work with two different representations.

In extending the treatment to the full language, a number of interesting subtleties arise. Special treatment has to be given to binding constructs, CSP, datatype declarations, and exception handling. Fortunately, the treatment of all three is not complex, and we expect that extending the soundness proof to include these refinements should not be problematic.

3.3.1 Care is Still Needed with Binding Constructs

In principle, once we have addressed lambda at levels higher than 1 correctly, we have captured the essence of how all binding constructs should be handled. In practice, some care is still needed when dealing with syntactic sugar for other syntax that uses binders. It is easy to get it wrong. For example, the following translation for let is wrong:

$$[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]^{n+1}_\rho = \left\{\begin{array}{l}\texttt{let}\ x = \texttt{gensym}\ () \\ \texttt{in Let}\ (x, [\![e_1]\!]^{n+1}_\rho, [\![e_2]\!]^{n+1}_{\rho;x\mapsto(n+1)})\end{array}\right.$$

In particular, whereas an x in e_1 would have been bound somewhere completely outside this expression, in the new term, we are replacing it everytime by a completely new fresh name that is not bound in the scope of e_2. Fortunately, once attention is drawn to this kind of problem, it is easy to fix as follows[11]:

$$[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!]^{n+1}_\rho = \left\{\begin{array}{l}\texttt{let}\ \ z = [\![e_1]\!]^{n+1}_\rho \\ \qquad x = \texttt{gensym}\ () \\ \texttt{in Let}\ (x, z, [\![e_2]\!]^{n+1}_{\rho;x\mapsto(n+1)})\end{array}\right.$$

3.3.2 Implementing Cross-Stage Persistence (CSP)

We were able to implement CSP almost exactly as suggested in the first work on MetaML [50]: CSP values are pointers to values that have already been compiled. This idea is very easy to implement in an interpreter, but less obvious to implement in a compiled setting. However, it turns out that CSP is essentially identical to the "quote" operator of Lisp and Scheme, which also allows embedding any value resulting from an arbitrary computation in the abstract syntax tree. We were able to add this "quote" construct to the OCaml compiler with little effort.

The translation presented above does not reflect this subtlety because it abstracts away from the distinction between source code and compiled code. We opted to do so because CSP is the only place where it is useful to make this distinction, and it simplifies the presentation substantially.

During the development of MetaOCaml, we had experimented with a less system-dependent way of implementing CSP, providing a reasonable alternative when a "quote"-like construct is not natively supported by the compiler. The alternate approach hinges on the idea of using a global registry for all CSP variables. This way, all the translation needs to do is to construct a piece of code that represented the "number" of the constant at hand. This means the translation was implemented as follows:

$$
\begin{array}{lcll}
[\![x]\!]^{n+1}_\rho & = & \texttt{Csp(store}\ x) & \rho(x) = 0 \\
[\![\%l]\!]^0_\rho & = & \texttt{lookup}\ l & \\
[\![\%l]\!]^{n+1}_\rho & = & \texttt{Csp}\ l &
\end{array}
$$

where `store` is a function that stored its argument in some global repository, and returned a new number l corresponding to this new constant. Then, when this CSP constant is encountered during a future compilation phase, it is simply replaced by a code fragment to `lookup` this constant in the global repository by its number. There are also variants of this strategy that ensure that the new constant is garbage-collected when it is not needed, and are also equally system-independent. All of these techniques, however, incur more runtime overheads than the "quote"-like mechanism that we have chosen.

For literals, a simple optimization on the basic scheme is quite desirable, namely, simply producing a parse tree that would later just compile to that literal. Doing this

[10] [We've been busy hacking, so, as of yet, this proof has not been carried out. We do not view it as essential to this paper, but if the result is achieved before publication, it will be reported in this paper and will be made available in a technical report.]

[11] Note that we do not need a substitution function (which is costly) or a complex environment to get this right.

has two advantages: First, a pointer de-reference is eliminated, making such constants a bit more efficient. Second, it enables further optimizations such as constant propagation and strength reduction, which would otherwise be blocked.

CSP values are similar to ASM sections in a C program. The two differences are that CSP values need no work during compilation, and they are not a construct available directly to the user (mainly to ensure safety).

3.3.3 *Implementing Datatypes and Exceptions*

In OCaml, datatypes constructors and case-statements are partially interpreted during the type-checking phase. The reason is that runtime values of datatypes do not contain constructor names, but instead integer tags[12] identifying the constructors, and determined during type-checking by examination of the datatype declaration to which the constructors belong. Thus it would be incorrect to build a code fragment that uses just constructor names. If this is done, and the code is type-checked and executed in the context of another declaration for another datatype that happens to use the same constructor name, the representation of that constructor would be determined in the wrong typing environment (dynamic scoping), and type safety would also be violated. This issue would arise with the following code sequence:

```
type t = C of int;;
let v = .<C 7>.;;
type t = C of string;; v;;
```

The correct implementation would allow us to execute the last statement safely. To deal with this problem, we interpret all constructors and case-statements when we type-check the source program for the first time. In addition, the parse tree is extended for these two cases with an extra field that tells us if these constructs have been interpreted before or not. The first time they are type-checked, their interpretation is also recorded in the same field. If they are type-checked again, they are not re-interpreted, but rather, their original interpretation is used directly. This treatment was inspired by the implementation of CSP.

We had expected exceptions to be very similar to datatypes. But because there is only one global namespace for exceptions, and they are represented at runtime by their names, the treatment of exceptions is in fact simpler than for datatypes: all that needs to be done is to construct a code fragment that carries the exact same name of the exception.

4. PERFORMANCE

As we mentioned earlier, one of the key goals behind developing MetaOCaml was to have a means for collecting concrete data about multi-stage programming. At this point, we have only collected preliminary data on a number of small examples. Larger and more extensive studies are under way. In this section, we will present our findings on these small examples. For the most part they are encouraging and are consistent with the experience with PE and RTCG. The data provides concrete evidence to support the utility of multi-stage languages for building simple yet efficient DSLs, and also point out some pecularities of the bytecode compiler setting.

[12]Note that the use of "tag" here and in "tag elimination" are different. Here a tag is an integer, there a tag is a constructor.

4.1 What is Measured

Figure 2 summarizes the timings for a number of examples programs, and two measures based on the raw data. The first column, **Run**, contains a pair (t/n) where t is the total time in seconds for n runs of an unstaged version of the program at hand. The number of n is chosen so as to 1) be at least 10, and 2) so that the total time is at least 0.5 seconds. [13] The rest of the timings related to a staged version of the program, but are all normalized with respect to the time of the unstaged program. **Generate** is the time needed to perform the first stage of the staged program, which involves the generation of a code fragment. **Compile** is the time needed to compile that fragment using the `.!` construct. **Run (2)** is the time needed to run the code that results from compiling that fragment. The next two columns are computed from the first four. **Speedup** is the first run time divided by the second run time. This is an indicator of the improvement if the generation and compilation times can be amortized. **Break-even** is the number of times that we would need to run the program before the total time of running the staged version (including a one-time generation and compilation cost) would be greater than the time of running the unstaged version. This is an alternative measure of the effectiveness of staging.

4.2 The programs

Examples in Figure 2 are chosen from a variety of fields: matrix computation, scientific computation, interpreter and some classic examples for multi-stage programming.

- `power`: is a classic example of multi-stage programming as we described in Section 1. The staged version runs about 5.4 times as fast as the unstaged program. The gain of performance comes from moving the recursion and function calls from runtime to code generation time.
- `dot`: Dot product turns out to be a poor example for staging, not because of the small performance gains, but the very large compilation times. This situation does not improve significantly when we move to RTCG [35].

Nevertheless this example pointed us to the issue of how functional languages deal with arrays, because our gains were slightly lower than those reported in Tempo [35]. In the setting of Tempo, when the array and the index are both known at RTCG time, the computation of the address of the array element can be done entirely at RTCG time, resulting in a constant address being put in the generated code. For garbage collected languages such as OCaml, arrays are dynamically managed by the garbage collector, which might even relocate them, so it is not possible to treat the array pointer as a constant. It is still possible to exploit the knowledge of the array index and the array size to 1- simplify the address computation, and 2- eliminate the runtime bound check (like Java, OCaml enforces array bounds checking). The OCaml native-code compiler implements some of these optimizations, but the bytecode compiler that MetaOCaml uses does not. Because of the overhead involved in interpreting bytecode, in our setting, the specialization of array

[13]All the test results were collected from a PIII 900M machine with 128MB of main memory running Linux redhat 6.1. Times were measured using the OCaml timing function. The exact and full benchmarks instrumented with the measuring functions are part of the distribution in directly `mex/benchmark1/`. The `MetaOCaml_302_alpha_002` distribution was the one used, and is available online [28].

Name	Run	Generate	Compile	Run (2)	Speed-up	Break-even
power	$(7.44s/1e^6)$=1x	2.65x	336x	0.18x	5.43	416
dot	$(2.14s/1e^5)$=1x	39.4x	3490x	0.80x	1.25	17500
dotU	$(1.68s/1e^5)$=1x	49.1x	4360x	0.72x	1.40	15400
evalLambda	(29.9s/10)=1x	$5.55 * 10^{-6}$x	$4.35 * 10^{-4}$x	0.14x	7.29	1
evalIntFun	(1.92s/10)=1x	$7.76 * 10^{-5}$x	$2.97 * 10^{-3}$x	0.03x	29.3	1
evalIntFunT	(2.14s/10)=1x	$7.57 * 10^{-5}$x	$4.30 * 10^{-3}$x	0.10x	10.3	1
rewrite	$(6.61s/1e^6)$=1x	13.2x	1200x	0.90x	1.11	12100
rewriteCPS	$(7.42s/1e^5)$=1x	5.69x	229x	0.08x	13.1	255
chebyshev	$(1.37s/1e^4)$=1x	8.03x	1010x	0.32x	3.08	1510
chebyshevU	$(1.37s/1e^4)$=1x	8.39x	1050x	0.32x	3.12	1550

Figure 2: Performance Measurements

accesses does not improve performance significantly. Switching off bounds checking in our experiment does not improve runtimes significantly.

• evalLambda: The evalLambda example is an interpreter for a lambda calculus with recursion. It is a particularly good example of staging. This can be seen both by the relatively high speedup, but more importantly, by the optimal break-even point. If tag elimination [49] is performed after generation, the speed up can be substantially higher. We return to this point in the next example and in the discussion section.

• evalIntFun: This is an interpreter for a micro-language which only has an integer type and provides if-statement and function calls as control constructs. It does not support high order functions as evalLambda does. Because this language only has one type, it eliminates the need of introducing a global datatype for values. Thus we avoid extra tagging and untagging operations on values which are necessary in evalLambda. To show how important a role the tag elimination plays in the overall performance, we did a comparison version in which we define a datatype v for values.

```
datatype v = I of int
```

Except for the tagging and untagging operation on value, evalIntFunT is exactly the same as evalIntFun. From Figure 2, we can see that the staged program of evalIntFun runs about 3 times as fast as evalIntFunT. This is even better than what had been anticipated in the work on tag elimination [49].

• rewrite: This example appeared in the first paper on MetaML [52], and is addressed in more detailed in Taha's thesis [43]. It is a matching function that takes a left-hand side of a rule and a term and returns either the same term or the corresponding right-hand side. Staging this program directly produced mediocre results, but staging it after it converted to Continuation-Passing Style (CPS) rewriteCPS produces siginificantly better results.

• chebyshef: Chebyshev polynomial is an example from a study on specializing scientific programs by Glück *el al.* [14]. Although we achieve some speedup, our results are weaker on this example than those achieved by others [14, 35]. We suspect that the reason is again the issue with array lookups in OCaml. chebyshefU is without bounds checks.

4.3 Bytecode vs. Native Code Compilers

In interpreting the performance figures given in this paper, it is useful to keep in mind that bytecode interpreters do not have the same timing characteristics as real processors. Bytecode interpreters cannot execute several instructions in parallel, and incur an interpretation overhead on each instruction (fetching, decoding, and branching to the appropriate piece of code to perform the instruction) that accounts for approximately half of the total execution time. Thus, in a bytecode interpretation setting, most of the benefits of runtime code specialization comes from the reduction in the number of instructions executed, e.g. when eliminating conditional branches, or removing the loop overhead by total unrolling.

In contrast, as we mentioned in the case of arrays, specializing the arguments to a bytecode instruction gains little or nothing. A typical example is that of an integer multiplication by 5. A native code compiler might replace the multiplication instruction (typical latency: 5 cycles) by a shift and an add (typical latency: 2 cycles). In a bytecode setting, this strength reduction is not beneficial: the overhead of interpreting two instructions instead of one largely offsets the 3 cycles saved in the actual computations.

For these reasons, we believe that using a native code compiler instead of a bytecode compiler and interpreter would result in higher speedups (of staged code w.r.t. unstaged code), because this would take advantage of instruction removal and of argument specialization. On the other hand, runtime compilation times could be significantly higher, especially if the native code compiler performs non-trivial optimizations, resulting in higher break-even points. The effect of moving to the native code compiler on the break-even point, however, is much less predictable. In particular, the native code compiler can be ten times slower, and code generated by the native code compile can be ten times fast. If we assume that the speedup will be constant in the native code setting (which is a pessimistic approximation), then the estimated break-even point in the native code compiler shoots up by a factor of 40 times for many of the cases above. The notable exceptions, however, are the interpreter examples, where the break-even point remains unchanged at 1. This gives us confidence that no matter how things will be for generic programs in the native compiler setting, MetaOCaml will remain particularly well-suited for building staged compilers.

5. CONCLUSIONS AND FUTURE WORK

Multi-stage programming languages have reached a point where the presence of usable and practical implementations is essential for a healthy progress in their development. Towards addressing this need, this paper presented a simple yet highly usable staging implementation of a multi-stage language based on a thoroughly investigated theoretical foundation and an industrial-strength compiler. MetaOCaml has allowed us to run a number of small experiments to support previous claims about the utility of multi-stage for building staged interpreters, and to show that multi-stage language yield can achieve performance gains consistent with those of PE and RTCG. The current implementation is a bytecode compiler, and the next step is extending the work to the native code compiler. We expect that the performance gains will carry over to that setting.

Starting with an existing system has been invaluable for allowing us to run experiments in a realistic setting, in addition to providing significant assurances about robustness and usability. MetaOCaml has a notably simple and transparent but nevertheless practical design that qualifies it as an important point in the design space of staging systems.

An annoying problem that we have run into is the difficulty in implementing the translation function for the full language. In particular, constructing parse trees that represent other parse trees is not only tedious, but the type system of languages like ML don't help in this regard (one needs at least a dependently typed system). This has hindered our work (we cannot stage modules yet.) This also seems to be another instance of the parsing/unparsing problem noted by Ramsey [39]. In the future, we intend to explore the use of the Camlp4 pre-processor [27] to produce at least some parts of our implementation.

We hope that MetaOCaml will facilitate the accumulation of a real code base of multi-stage programs that can be used to assess the practical effectiveness of these languages. As we gain more experience and more confidence with the current design, we expect to be able to extend our work to the native code compiler. In this effort some implementation machine-dependence may be needed, but that is not certain yet. The current challenge is to make the compiler available in the same memory space at runtime, and to be able to link the code that it generates into the same memory space. The data reported here is encouraging, and suggests that even when we move to the native code compiler setting (which has substantially higher compilation times), it is reasonable to expect that multi-stage languages will continue to be very well-suited for building staged interpreters. The data also suggests that performance gains from tag elimination may be even higher than was previously anticipated.

6. REFERENCES

[1] Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V. M., and Stichnoth, J. M. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (1998), pp. 280–290.

[2] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A transparent dynamic optimization system. In *Proceedings of the Conference on Programming Language Design and Implementation* (1999), pp. 1–12.

[3] Benaissa, Z. E.-A., Moggi, E., Taha, W., and Sheard, T. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)* (1999).

[4] Calcagno, C., and Moggi, E. Multi-stage imperative languages: A conservative extension result. In *[44]* (2000), pp. 92–107.

[5] 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.

[6] Cierniak, M., Lueh, G.-Y., and Stichnoth, J. M. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)* (N.Y., June 18–21 2000), vol. 35.5 of *ACM Sigplan Notices*, ACM Press, pp. 13–26.

[7] Consel, C., and Noël, F. A general approach for run-time specialization and its application to C. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (St. Petersburg Beach, 1996), pp. 145–156.

[8] Consel, C., Pu, C., and Walpole, J. Incremental specialization: The key to high performance, modularity, and portability in operating systems. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (New York, 1993), ACM Press, pp. 44–46.

[9] 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.

[10] Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4 (Dec. 1992), 295–326.

[11] Engler, D. R., Hsieh, W. C., and Kaashoek, M. F. 'C: A language for high-level, efficient, and machine-independent dynaic code generation. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (St. Petersburg Beach, 1996), pp. 131–144.

[12] Glück, R., and Jørgensen, J. Efficient multi-level generating extensions for program specialization. In *Programming Languages: Implementations, Logics and Programs (PLILP'95)* (1995), S. D. Swierstra and M. Hermenegildo, Eds., vol. 982 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 259–278.

[13] Glück, R., and Jørgensen, J. Fast binding-time analysis for multi-level specialization. In *Perspectives of System Informatics* (1996), D. Bjørner, M. Broy, and I. V. Pottosin, Eds., vol. 1181 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 261–272.

[14] Glück, R., Nakashige, R., and Zöchling, R. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization* (1995), J. Doležal and J. Fidler, Eds., Chapman & Hall, pp. 137–146.

[15] Gomard, C. K., and Jones, N. D. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming 1*, 1 (1991), 21–69.

[16] Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S. J. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the Conference on Programming Language Design and Implementation* (1999), pp. 293–304.

[17] Harrison, W. L. *Modular Compilers and their Correctness Proofs.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2001. Available from author (wlhcse.ogi.edu).

[18] Harrison, W. L., and Kamin, S. N. Metacomputation-based compiler architecture. In *Mathematics of Program Construction* (2000), pp. 213–229.

[19] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[20] Jones, N. D., Sestoft, P., and Sondergraard, H. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, J.-P. Jouannaud, Ed., vol. 202 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985, pp. 124–140.

[21] Jørring, U., and Scherlis, W. L. Compilers and staging transformations. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (St. Petersburg, Florida, 1986), ACM Press, pp. 86–96.

[22] Kamin, S., Callahan, M., and Clausen, L. Lightweight and generative components II: Binary-level components. In *[44]* (2000), pp. 28–50.

[23] Keppel, D., Eggers, S. J., and Henry, R. R. A case for runtime code generation. Tech. Rep. 91-11-04, University of Washington, 1991.

[24] Lee, P., and Leone, M. Optimizing ML with run-time code generation. In *Proceedingsof the Conference on Programming Language Design and Implemantation* (New York, 1996), ACM Press, pp. 137–148.

[25] Leone, M., and Lee, P. Lightweight run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1994), Technical Report 94/9, Department of Computer Science, University of Melbourne, pp. 97–106.

[26] Leone, M., and Lee, P. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)* (1996).

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

[28] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://cs-www.cs.yale.edu/homes/taha/MetaOCaml/`, 2001.

[29] The MetaML Home Page, 2000. Provides source code and documentation online at `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.

[30] Moggi, E. A categorical account of two-level languages. In *Mathematics Foundations of Program Semantics* (1997), Elsevier Science.

[31] Moggi, E. Functor categories and two-level languages. In *Foundations of Software Science and Computation Structures (FoSSaCS)* (1998), vol. 1378 of *Lecture Notes in Computer Science*, Springer Verlag.

[32] 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.

[33] Nielson, F. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems 7*, 3 (1985), 359–379.

[34] Nielson, F., and Nielson, H. R. Two-level semantics and code generation. *Theoretical Computer Science 56*, 1 (1988), 59–133.

[35] Noël, F., Hornof, L., Consel, C., and Lawall, J. L. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages* (1998), IEEE Computer Society Press, pp. 132–142.

[36] 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`. Last viewed August 1999.

[37] Peyton Jones, S. L., and Launchbury, J. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming and Computer Architecture* (1991).

[38] Pu, C., and Walpole, J. A study of dynamic optimization techniques: Lessons and directions in kernel design. Tech. Rep. CSE-93-007, Oregon Graduate Institute, 1993. Available from [36].

[39] Ramsey, N. Pragmatic aspects of reusable software generators. In *[44]* (2000), pp. 149–171.

[40] Sheard, T. Accomplishments and research challenges in meta-programming(invited talk). In *[46]* (2000), pp. 2–44.

[41] Sheard, T., Benaissa, Z. E.-A., and Pašalić, E. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)* (Austin, Texas, 1999), USEUNIX.

[42] Shields, M., Sheard, T., and Peyton Jones, S. L. Dynamic typing through staged type inference. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (1998), pp. 289–302.

[43] Taha, W. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [36].

[44] Taha, W., Ed. *Semantics, Applications, and Implementation of Program Generation* (Montréal, 2000), vol. 1924 of *Lecture Notes in Computer Science*, Springer-Verlag.

[45] Taha, W. 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 Manipulation (PEPM)* (Boston, 2000), ACM Press.

[46] Taha, W., Ed. *Semantics, Applications, and Implementation of Program Generation* (Firenze, 2001), vol. 2196 of *Lecture Notes in Computer Science*, Springer-Verlag.

[47] 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.

[48] Taha, W., and Makholm, H. Tag elimination – or – type specialisation is a type-indexed effect. In Subtyping and Dependent Types in Programming, APPSEM Workshop. INRIA technical report, 2000.

[49] 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.

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

[51] Taha, W., and Sheard, T. MetaML and multi-stage programming with explicit annotations. Tech. Rep. CSE-99-007, Department of Computer Science, Oregon Graduate Institute, 1999. Extended version of [50]. Available from [36].

[52] Taha, W., and Sheard, T. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science 248*, 1-2 (2000). Revision of [51].

[53] Thiemann, P. Higher-order code splicing. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag.

[54] Wickline, P., Lee, P., and Pfenning, F. Run-time code generation and Modal-ML. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)* (Montreal, 1998), pp. 224–235.