A Gentle Introduction to Multi-stage Programming, Part II [⋆]

Walid Taha

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

Abstract. As domain-specific languages (DSLs) permeate into mainstream software engineering, there is a need for economic methods for implementing languages. Following up on a paper with a similar title, this paper focuses on dynamically typed languages, covering issues ranging from parsing to defining and staging an interpreter for an interesting subset of Scheme. Preliminary experimental results indicated that the speedups reported in previous work for smaller languages and with smaller benchmarks are maintained.

1 Introduction

A natural question to ask when we consider implementing a new language is whether we can reuse an existing language implementation. If we can, then we have reduced the new problem to one that we have (or someone else has) already solved. In that case, we can achieve reuse by building a translator from the new language to the old. In "A Gentle Introduction to Multi-stage Programming," [2] we introduced the reader to the basics of a semantically inspired approach to building such translators, namely, a staged interpreter. That work, which we will refer to as Part I, introduces the a basic approach that consists of three steps:

1. Write an interpreter and check its correctness.
2. Stage the interpreter by adding staging annotations.
3. Check the performance of the staging implementation.

Based on practical experience, when we get to the third step we should expect that we will need to convert to continuation-passing style to achieve satisfactory staging.

The focus of our previous work was a simple language called Lint, which has only one type, namely integers, and supports only functions from one integer to another integer. The goal of this article is to expand the reader's repertoire of staging expertise to the point where he or she can implement a dynamically typed language as expressive as a substantial subset of the Scheme programming language.

[⋆] Supported by NSF CCR SoD 0439017, CSR/EHS 0720857, and CCF CAREER 0747431.

1.1 Prerequisites

Reading the article mentioned above is not a prerequisite for reading this tutorial. Familiarity with the basics of multi-stage programming and with the basics of staged interpreters is sufficient. However, for a reader not familiar with these topics, that article will provide the necessary background.

To follow the explanation of the reference interpreter, basic familiarity with OCaml, lambda abstractions, and the OCaml List library are needed.

To follow the explanation of staging issues, familiarity with continuation-passing style (CPS) and the CPS transformation are needed.

1.2 Contributions

The key novelties driving this tutorial are the scale and the type discipline of the language being interpreted. We focus on a larger language than has previously been analyzed in the context of multi-stage programming, and in turn we focus on the main sources of difficulty that arise during the process of building a staged interpreter for this language. We also focus on the class of languages that seems easiest to interpret in a modern, statically typed, functional language, which is in fact a dynamically typed language. Writing such an interpreter is facilitated by the fact that we can easily define one data type that would serve as the universal value domain.

The new expository material presented in this paper includes the following:

1. The use of a small, practical parser combinator library as well as the use of universal concrete and abstract syntax to facilitate practical language prototyping.
2. Case-by-case analysis of an OCaml interpreter for a more expressive language than the one we covered in Part I. This language includes higher-order functions with multiple arguments, a dynamic data structure (lists), and mutable structures.
3. Detailed explanation of how to translate a direct style interpreter into CPS.

1.3 Organization

Section 2 introduces a method that we have found useful for quickly building parsers. As that section points out, a practical method for circumventing the issue of building parsers and designing concrete syntax for a programming language is to use a universal syntax, such as that of HTML, XML, or the LISP syntax for representing data (s-expressions). Section 3 presents an interpreter for a basic subset of Scheme that we call Aloe (Another Language of Expressions). This subset includes booleans, integers, strings, mutable variables, mutable lists, and higher-order functions. This section explains how to interpret variable arity operators and functions, syntactic sugar, and how to use side effects to implement recursive definitions. Section 4 explains how to convert the staged interpreter into CPS, giving special attention to the more interesting cases. Section 5 explains how to stage the interpreter that resulted from the previous step. Section 6 presents a new optimization technique for untyped language that could be used to improve the the Aloe staged interpreter.

The complete code for the parsers and the various interpreters described in this tutorial can be found at `http://www.metaocaml.org/examples`.

2 Parsing

Concrete syntax tends to be much more concise than abstract syntax, so it is notationally convenient. Furthermore, if we want to be able to store and load programs in files, it must be done with some concrete syntax. To accommodate this practical concern, this section will provide the reader with a minimal tool for dealing with the issue of parsing.

In the source code accompanying this tutorial, we use Hutton and Meijer's Monadic Parser combinators [1]. We have reimplemented this library in OCaml for convenience, and the mapping is mostly mechanical. The key change is that Haskell is lazy and so allows pure, memoized streams to be implemented concisely. We simply used lambda abstraction to delay computations. Certainly, memoizing implementations are possible [3], but the examples considered here are simple enough that there is no pressing need for this optimization.

To avoid having to define a parser for every new language that we consider, it is helpful to use a single, universal concrete representation for programs. Representing programs concretely (where "concretely" means "as strings of characters") is no different from representing any other form of date. Universal concrete representations include the XML and the LISP data formats (widely known as s-expressions). Because it is syntactically lighter-weight, we will use s-expressions for this tutorial. Specifically, we will use the following grammar for s-expressions:

```
   s-expression :== int | sym | str | ( s-expression^* )
```

where `int`, `sym` and `str` are primitive grammars for integers, symbols, and strings, respectively; and where `^*` means zero or more repetitions of the preceding grammar. We will use the combinator library described above to write a single parser for s-expressions, and when this parser succeeds it will produce a value of the following OCaml type:

```
type sxp =
 | I of int     (* Integer *)  | A of string    (* Atoms  *)
 | S of string  (* String *)   | L of sxp list  (* List   *)
```

Whereas s-expressions are a universal concrete syntax, this data type can be viewed as a universal abstract syntax. Our interpreters for Aloe will always take a value of this type as input.

To illustrate how s-expressions using the traditional concrete syntax would be represented in the OCaml type `sxp`, we will consider a few small examples. The lists `(1 2 3 4)` would be represented by the value

```
L [I 1; I 2; I 3; I 4]
```

The type naturally allows for nesting of lists. For example, `"((1 2) (3 4))"` would be represented as

```
L [L [I 1; I 2]; L [I 3; I 4]]
```

The type also naturally allows us to mix elements of different types, so we can represent `"(lambda (x) (repeat x \"ping \"))"`

```
L [A "lambda"; L [A "x"]; L [A "repeat"; A "x"; S "ping "]]
```

To automate the process of parsing an s-expression, we provide the following utilities:

```
   read_file : string -> string
   parse     : string -> (sxp * string) list
   print     : sxp -> string
```

The first function simply takes the name of a file and reads it into a string. The second function takes a string and tries to parse it. It will return a list of possible ways in which it could have been parsed, as well as the remaining (unparsed) string in each case. For s-expressions, we will always have at most one way in which a string can be parsed.

3 An Interpreter for Aloe

In this section we will present an interpreter for a small programming language that we call Aloe. This language will be the running example for this tutorial, and the interpreter will drive the discussion of CPS conversion and staging.

The Aloe programming language is a subset of Scheme that includes booleans, integers, strings, mutable variables, mutable lists, and higher-order functions. We will begin the design of the interpreter by considering the appropriate definitions for values and environments, and then we will move to defining the interpreter itself. We will conclude this section by describing a timing benchmark, a timing experiment, and the results from this timing experiment. These results will serve as the baseline for assessing the performance of the staged interpreter.

3.1 Denotable Values and Tags

The first question to consider when writing an interpreter for an untyped language is to determine the kinds of values that the language will support. For Aloe, we are interested in the following kinds of values:

```
type dom = Bool of bool | Int of int | Str of string
         | Fun of int * (dom list -> dom) | Undefined
         | Void | Empty | Cons of dom ref * dom ref
```

Thus, our language will support three interesting base types: booleans, integers, and strings. It will also support functions and mutable lists. Each function value will be tagged with an integer that represents the number of arguments it expects. In addition, the set of values will include two special values, `Undefined` and `Void`. The first special value will be used for un-initialized Locations, and the second will be for denoting the absence of a result from a side-effecting computation.

It is worth noting that the type we have defined for values is *computational* and not a purely mathematical type. This allows us to avoid having to specify explicitly where non-termination or exceptions can occur. It also makes it easy for us to represent values that can change during the lifetime of a computation by using the OCaml `ref` type constructor.

3.2 Exceptions and Untagging

To allow for the possibility of error during the computation of an Aloe program, we will introduce one OCaml exception:

```
exception Error of string
```

and we will immediately use this exception to define specific untagging operations that allow us to extract the actually value from a tagged value when we expect a certain tag to be present. For example, for the `Fun` tag we will write:

```
let unFun d =
  match d with
    | Fun (i,f) -> (i,f)
    | _ -> raise (Error "Encountered a non-funcion value")
```

3.3 Environments and Assignable Values

We will represent environments simply as functions, where the empty environment `ext0` produces an error on any lookup. We will define both a simple environment extension function `ext` that introduces one new binding, as well as one that extends the environment with several bindings at a time `lext`:

```
let env0 x =
  raise (Error ("Variable not found in environment "^x))

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

let rec lext env xl vl =
  match xl with
      []    -> env
    | x::xs -> match vl with
                 [] -> raise (Error "Not enough arguments")
       | y::ys -> lext (ext env x y) xs ys
```

An important technical aside for Aloe is that, being a subset of Scheme, some but not all variables can be assigned. To reflect this difference, we will require that all environments map names to values of the following type:

```
type var = Val of dom | Ref of dom ref
```

3.4 Concrete Syntax

As noted earlier, we will use the OCaml data type for s-expressions to represent the abstract syntax for our programs. Nevertheless, it is still useful to state the concrete syntax for the language that we are interested in.

```
   I is the set of integers
   S is the set of strings
   A is the set of atoms

   U :== not | empty? | car | cdr

   B :== + | - | * | < | > | = | cons | set-car! | set-cdr!

   E :== true | false | empty | I | "S" | A | (U E) | (B E E)
       | (cond (E E) (else E)) | (set! X e) | (and E E^*)
       | (or E E^*) | (begin E E^*) | (lambda (A^*) E)
       | (E E^*)

   P :== E | (define A E) P | (define (A A^*) E) P
```

The first three lines indicate that we will assume that we are given three sets, one for integers, one for strings, and one for atoms. Integers are defined as sequences of digits possibly preceded by a negative sign. Strings are sequences of characters with some technical details the definition of which we relegate here to the implementation. Atoms are also sequences of characters, with the most notable restriction being the absence of spaces.

The next line defines the set `U` of unary operator names. This set consists of four terminal symbols. The next set is `B`, which consists of nine terminals, is the names of binary operators. The set of expressions `E` contains terminals to denote booleans and the empty list, and it also embeds integers, strings, and atoms. When we get to atoms, the user should note that there is a possibility for ambiguity here-the expression `true`, for example, can match either the first or the sixth (atom) production. For this reason we will consider our productions order dependent, and the derivation of a string will always use the production that appears left-most in the definition. This does not change the set of strings defined, but it makes the derivation of each string unique. The significance of the order of production will also be important for ensuring the proper behavior from the main case analysis performed in the interpreter.

Note 1 (Practical Consideration: Validating the Reference Interpreter). We caution the reader that even though Aloe is a small language, we spent considerable time removing seemingly trivial bugs from the first version of the interpreter. Staging provides no help with this problem, and in fact any problems present in the original interpreter and that go unnoticed will also be present in the staged interpreter. Thus, we strongly recommend developing an extensive acceptance test that illustrates the correct functionality of each of the language constructs while developing the original interpreter. Such tests will also be useful later when developing the staged interpreter and when assessing the performance impact of staging.

A helpful by-product of using both the syntax and the semantics from Scheme for Aloe was that it was easy to validate the correct behavior of our test examples using standard Scheme implementations. Because the correctness of language implementations is of such great importance, the benefits of devising of a new syntax for your language should be weighed carefully against the benefits of having such a direct method of validating the implementation.

3.5 The Interpreter for Expressions

The Aloe interpreter will consist of two main functions, one interpreting expressions, and the other interpreting programs. To follow the order in which the syntax is presented, we will start by covering the interpreter for expressions, which takes an expression and an environment and returns a value. It is structured primarily as a `match` statement over the input expression. The `match` statement is surrounded by a `try` statement, so that exceptions are caught immediately, some informative debugging information is printed, and the exception is raised again. Thus, the overall structure of the interpreter is as follows:

```
let rec eval e env =
  try (match e with
       ...)
  with
      Error s -> (print_string ("\n"^(print e)^"\n");
                  raise (Error s))
```

where the `...` represents the payload of the interpreter. In the rest of this section, we will discuss the key issues that arise when interpreting the various constructs of Aloe.

Note 2 (Practical Consideration: Reporting Runtime Errors). Our Aloe interpreter provides minimal contextual information to the user when reporting runtime error. More comprehensive reporting about errors as well as debugging support would not only be useful to the users of the language but would also likely be to help the implementor of the language as well. Thus, these issues should be given careful consideration when implementing any language of size comparable to or larger than the Aloe.

Atomic Expressions, Integers, and Strings The semantics of most atomic expressions in Aloe are fairly straightforward:

```
| A "true"   -> Bool true | A "false"  -> Bool false
| A "empty"  -> Empty
| A x ->  (match env x with Val v -> v | Ref r -> !r)
| I i -> Int i | S s -> Str s
```

Booleans, the empty list, integers, and strings are interpreted in a direct manner. Variables that are recognized as an atoms that did not match the first three cases in the `match` statement are interpreted by first looking up the name in the environment. Because an environment lookup might fail, it is possible that an exception may be raised when we apply `env` to the string `x`. If we do get a value back, the interpreter checks to see whether it is assignable or not. If it is a simple value, we return it directly. If it is an assignable value, then we de-reference the assignable value and return the value to which it is pointing.

Unary and Binary Operators The interpretation of unary and binary operators is a bit more involved, but still largely direct. It should be noted, however, that we will have to explicitly define the action of all primitive operations somewhere in our interpretation. For convenience this can be done inline, as we do for each case in our case analysis:

```
| L [A "not"; e1]   -> Bool (not (unBool (eval e1 env)))
| L [A "+"; e1; e2] -> Int (  (unInt (eval e1 env)
                            + (unInt (eval e2 env))))
```

Interpreting logical negation in Aloe is done by evaluating the argument, removing the `Bool` tag, applying OCaml's logical negation to the resulting value, and then tagging the resulting value with `Bool`. The pattern of untagging and re-tagging will repeat in our interpreter, which is a necessity when being explicit about the semantics of a dynamically typed language. Because OCaml is statically typed, this forces us to make the tag manipulation explicit. While this may seem verbose at first, we will see in later discussion that being explicit about tags may be convenient for discussing different strategies for implementing the same dynamically typed computation.

The binary operation of addition follows a similar pattern, as do most of the unary and binary operations in Aloe. An interesting binary operation is the `cons` operation, which creates a new list from a a new element and an old list:

```
| L [A "cons"; e1; e2] ->
   Cons (ref (eval e1 env), ref (eval e2 env))
```

This implementation of the list constructor is sometimes described as being `unchecked`, in that it does not check that the second element is a list. Another interesting case is mutation, namely, of the `set-car!` or `set-cdr!` of a list. Both operations are interpreted similarly. The first is interpreted as follows:

```
    | L [A "set-car!"; e1; e2] ->
       (match (eval e1 env) with
  | Cons (h,t) -> (h:=(eval e2 env);Void)
  | _ -> raise (Error ("Can't assign car of "^(print e1))))
```

Performing this computation first evaluates the first argument, checks that it is a proper list, and if so, evaluates the second expressions and assigns the result to the head of the list. If the first value is not a proper list, an error is detected.

Variable Arity Constructs It not unusual for programming language constructs to allow a varying number of arguments. An example of such a variable arity construct is the equality construct `=` which takes one or more arguments and returns "true" only if all of them are equal. We will interpret this construct as follows:

```
    | L ((A "=") :: e1 :: l) ->
       Bool (let v = unInt (eval e1 env) in
             let l = List.map (fun x -> unInt (eval x env)) l
             in List.for_all (fun y -> y=v) l)
```

First, the first expression is evaluated, and its integer tag is removed. This reflects the fact that this operator is intended to work only on integer values. Then, we map the same operation to the elements of the rest of the list of arguments. Finally, we check that all the elements of that list are equal to the first element.

Logical conjunction and disjunction are implemented in a similar manner.

Conditionals and Syntactic Sugar Conditional expressions are easy to define. However, we limit them to having two arguments and leave the generalization as an exercise to the reader in applying the variable arity technique presented above.

We use conditionals to illustrate how to deal with syntactic sugar. In particular, Aloe includes `if` statements, which can be interpreted by a recursive call to the interpreter applied to the de-sugared version of the expression:

```
    | L [A "if"; c2; e2; e3] ->
       eval (L [A "cond"; L [c2 ; e2]; L [A "else"; e3]]) env
```

The key issue that requires attention using this technique to support syntactic sugar is that the we should make sure that it does not introduce non-termination. While this may seem inconsequential in a setting in which the language being interpreted can itself express diverging computation, the distinction between divergence in the interpretation and divergence in the result of the interpretation will become evident when we stage such interpreters.

Lambda Abstraction If lambda abstractions in Aloe were allowed only to have one argument, then the interpretation of lambda abstractions would be expressible in one line:

```
| L [A "lambda" ; L [S x] ; e] ->
   Fun (1, fun l -> match l with
                    [v] -> eval e (ext env x (Val v)))
```

The pattern matching requires that there is only one argument, that it is a string, and that the value of that string is bound to `x`. The pattern also requires that this be followed precisely by one expression. The interpretation returns a value tagged with the `Fun` tag. The first component of this value represents the number of arguments that this function expects (in this case one). The second argument is an OCaml lambda abstraction that takes a value and pattern matches it to assert that it is a list of one element, which is called `v`. The result of the OCaml lambda abstraction is the result of evaluating the body of the Aloe lambda abstraction, namely, the expression `e`. Evaluation of this expression occurs under the environment `env` extended with a mapping from the name `x` to the value `Val v`. We tag the value `v` with the tag `Val` to reflect the fact that we will not allow the arguments to lambda abstractions to be mutated.

To handle the fact that lambda abstractions in Aloe can handle multiple arguments, the interpretation becomes a bit more verbose, but in reality it is essentially doing little more than what the case for one argument is doing:

```
| L [A "lambda" ; L axs ; e] ->
    let l = List.length axs
    in Fun (l, fun v ->
               eval e (lext env
                       (List.map (function A x -> x) axs)
                       (List.map (fun x -> Val x) v)))
```

Here, we are simply allowing the argument to be a list of names and handling this extra degree of generality by mapping over lists and using the function for extending the environment with a list of mappings (introduced earlier).

Function Application Function application is similarly easier to understand if we first consider only the case of single-argument functions:

```
| L [e1; e2] -> let (1,f) = unFun (eval e1 env) in
                let arg = eval e2 env
                in f [arg]
```

The pattern match assumes that we only have an application of one expression to another. The first `let` statement evaluates the first expression, removes the `Fun` tag, checks that the first component is `1`, and calls the second component `f`. The second `let` statement evaluates the argument expression, and calls the resulting value `arg`. While these two `let` statements can be interchanged, the order is highly significant in a language that allows side effects, as Aloe does. Finally, the result of the interpretation is simply the application of the function `f` to the singleton list `[arg]`. The interpretation for function application in full generality is as follows:

```
    | L (e::es) ->
        let (i,f) = unFun (eval e env) in
        let args = List.map (fun e -> eval e env) es in
        let l = List.length args
        in if l=i
          then f args
          else raise (Error ("Function has "^(string_of_int l)^
               " arguments but called with "^(string_of_int i)))
```

This generalization also evaluates the operator expression first, then it uses a list map to evaluate each of the argument expressions. Once that has been done, the arguments are counted, and we check that the number of arguments we have is consistent with the number of arguments that the function expects. If that is the case, then we simply perform the application. Otherwise, an error is raised.

3.6 The Interpreter for Programs

The interpreter for Aloe programs takes a program and an environment and produces a value. Compared with expressions, Aloe programs are relatively simple, and thus the interpreter for programs can be presented in one-shot, as follows:

```
let rec peval p env =
  match p with
    | [e1] -> eval e1 env
    | (L [A "define"; A x; e1])::p ->
        let r = ref Undefined in
        let env' = ext env x (Ref r) in
        let v = eval e1 env' in
        let _ = (r := v)
        in peval p env'
    | (L [A "define"; L ((A x)::xs); e1])::p ->
        peval (L [A "define"; A x;
                   L [A "lambda" ; L xs ; e1]]::p) env
    | _ -> raise (Error "Program form not recognized")
```

The first case, the case in which a program is simply an expression, is easy to recognize because it is the only case in which a program is a singleton list. In that case, we simply use the interpreter for expressions. The second case is a `define` statement. We wish to interpret all `define` statements as recursive, and there are many ways in which this can be achieved. In this interpreter, we employ a useful trick that can provide an efficient implementation in the presence of side effects-in the interpretation, we create a reference cell initialized to the special value `Undefined`. Then, we create an extension of the current value mapping the value that we are about to define to this reference. Next, we evaluate the body of the reference. Finally, we update the reference with the result of the evaluation of the body and continue the evaluation of the rest of the program in the extended environment. Clearly, this technique only produces a useful value if the definition of the variable that we are about to define is not *strict* in its use of that variable. This is generally the case, for example, if the definition is a lambda abstraction or an operation on lazy streams.

The last case of the interpreter produces an error if any other syntactic form is encountered at the top level of a program.

With this, we have completed our overview of the key features of the reference interpreter for the Aloe language.

3.7 A Benchmark Aloe Program

To collect some representative performance numbers, we will use one Aloe program that defines and uses a collection of functions, including a number of alternative definitions for the factorial, Fibonacci, and tak functions using numbers and lists to perform the core computation, as well as a CPS of the insertion sort. As a sanity check for the correctness of the implementation, the suite includes a function that applies these various functions to different inputs and compares the output. To facilitate the use of the suite for performance evaluation, the main function executes this test 1000 times. For simplicity, all the functions that form the test suite are included in one file called `test.aloe`.

3.8 The Experiment

To study their relative performance, the same experiment will be carried out for the interpreter and for the staged versions of the interpreter. The code for the experiment for the interpreter described above is as follows:

```
let test1 () =
  let f = "test.aloe" in
  let _ = Trx.init_times () in
  let s = Trx.time 10 "read_file" (fun () -> read_file f) in
  let [(L p,r)]
        = Trx.time 10 "parse" (fun () -> parse ("("^s^")")) in
  let a = Trx.time  1 "peval"     (fun () -> peval p env0) in
  let a = Trx.time  1 "readeval"  (fun () -> freadeval f) in
  let _ = Trx.print_times ()
  in  a
```

The functions `Trx.init_times`, `Trx.time`, and `Trx.print_times` are all standard functions that come with MetaOCaml, and they are provided to assist with timing. The experiment times ten different readings of the file into a string, ten parsings of the string into an abstract syntax tree, a single evaluation of the parse program, and a combined run through of all of these steps.

3.9 Benchmarking Environment

All results reported in this paper will be for experiments performed on a machine with the following specifications: MacBook running Mac OS X version 10.4.11, 2 GHz Intel Core 2 Duo, 4 MB L2 cache per processor, 2 GB 667 MHz DDR2 DRAM. All results were collected using MetaOCaml version `3.09.1 alpha 030` using the interactive top-level loop.

3.10 Baseline Results

The results of the first experiment are as follows:

```
# test1 ();;
__ read_file __________________ 10x avg = 1.982000E - 01 ms
__ parse ______________________ 10x avg = 6.398430E + 01 ms
__ peval _______________________ 1x avg = 1.408170E + 04 ms
__ readeval ___________________  1x avg = 1.417668E + 04 ms
```

For this baseline implementation, reading the file is fast, parsing is a little bit slower, but evaluating this test has the dominant cost.

4 Converting into Continuation-Passing Style (CPS)

As noted in Part I, it is generally useful to convert a program into CPS before staging it. To convert our interpreter for the expressions

```
eval : sxp -> (string -> var) -> dom
```

into CPS requires systematically rewriting the code to

1. Taking in an extra "continuation" argument in every function call, and
2. applying this continuation to every value that we would normally simply return in the original code.

This will yield a new function

```
keval : sxp -> (string -> var) -> (dom -> dom) -> dom
```

In the rest of this section, we will describe how this transformation would be carried out for an interpreter similar to the one for Aloe. We conclude the section by reporting the results of running our timing experiments on the CPS-converted interpreter.

Taking in the Continuation Only minor change is needed to the outer-most structure of the interpreter for expressions:

```
let rec keval e env k =
  try
    (match e with
     ...)
  with Error s -> (print_string ("\n"^(print e)^"\n");
                   raise (Error s))
```

In the first line, we have added an extra parameter `k`, through which we will pass the continuation function. This is the function that we will apply in the rest of the interpreter to every value that we simply returned in the original interpreter.

A natural question to ask when we consider the next line is: why not simply add an application of `k` around the `try` statement, and be done with the conversion to CPS? While this would be valid from the point of view of external behavior, to achieve our goal of effective staging, it is important that we push the applications of the continuation `k` as far down as possible to the leaves of our program. In particular, this will mean that we will push the applications down over `try` statements and `match` statements.

A useful observation to make at this point is that pushing this single application of the continuation from around a `match` statement duplicates this application around all the branches. While this duplication is inconsequential in normal evaluation of a `match` statement, it will be significant when evaluating the staged version of a `match` statement.

Cases that Immediately Return a Value When we get to the branches of the `match` statement, the simple cases in which the interpretation returns a value without performing an interesting computation, the CPS version of this code simply applies the continuation `k` to this value as follows:

```
| A "true"   -> k (Bool true)
| A "false"  -> k (Bool false)
| A "empty"  -> k (Empty)
```

The cases for integer and string literals are similar.

Match Statements and Primitive Computation The case for variables allows us to illustrate two points. First, if we encounter another nested `try`, `match`, or `if` statement, we simply push the continuation to the branches:

```
| A x ->
    (match env x with
      | Val v -> k v
      | Ref r -> k (! r))
```

In the second branch, we will also notice that even though the expression `! r` is a computation (and not a value) that consists of applying the de-referencing function `!` to the variable `r`, we leave this expression intact and simply apply the continuation `k` to its result. For general function applications, we will see that this will not be the case. However, for primitive functions such as de-referencing we will simply leave their application intact in the CPS-converted program. The use of primitive functions is seen in several other cases in the interpreter, including logical negation, arithmetic operations, and many others.

Simple, Non-Primitive Function Calls Function calls are converted by replacing the call to the original function with a call to the new function. Because our goal is to push CPS conversion as deeply as possible, we will assume that we have already CPS-converted the function being applied. An easy special case is when the function we are converting is an application of a function inside its definition (we are converting a recursive call inside a recursive definition), we do both things at the same time.

Converting a function call involves providing a continuation at the call site, and this requires some care. Passing the current continuation `k` to the recursive call would mean that we simply want the result of the current call to be used as the rest of the computation for the recursive call. This would only be correct if we were immediately returning result of this recursive call. In our interpreter, this situation arises only in the case of our interpretation of syntactic sugar, such as our interpretation of the `if` statement in terms of the `cond` statement:

```
| L [A "if"; c2; e2; e3] ->
    keval (L [A "cond"; L [c2 ; e2];
                        L [A "else"; e3]]) env k
```

Extending Continuations and Naming Intermediate Results Generally speaking, however, having one case of a recursive function defined directly as a recursive call to the same function (with different arguments) will be the exception, and not the rule. So, the question is, what continuation do we pass when the continuation for the recursive call is not simply the current continuation `k`? The basic rule will be to look at the computation that surrounds the recursive call in the original expression that we are converting. Whatever happens there to the original result is really what the continuation that we pass to the recursive call needs to do *before* the current continuation is applied. This is illustrated clearly by considering the case of logical negation:

```
| L [A "not"; e1] -> keval e1 env (fun r ->
                      k (Bool (not (unBool r))))
```

Compared with the code for logical negation in the reference interpreter (Subsection 3.5), CPS conversion has turned the expression inside-out: the recursive call, which used to be the innermost expression, is now the outermost). In addition, what happens after we return the result of the computation, which was previously implicit in the outermost surrounding context for the code, is now explicitly represented as `k` and is deeply nested in the expression.

This example also illustrates two patterns that will frequently arise in CPS conversion. First, we create new continuations by extending existing ones, as created in the expression `fun r -> k (Bool (not (unBool r)))`. Second, when we create new continuations in this manner we also end up introducing a new name `r` for what was previously simply an unnamed, intermediate computation: the application of `eval` to `e1` and `env`. In other words, a natural side-effect of CPS conversion is to produce code in which all intermediate computations are named and each step of the computation cannot be further reduced into smaller steps.

Multiple, Non-Primitive Functions Calls The "inside-out" metaphor about CPS conversion should help us to see how to convert functions where several non-primitive functions are applied. The case of addition (Subsection 3.5) is an example of such a situation, where we make two recursive calls

```
      | L [A "+"; e1; e2] -> keval e1 env (fun r1 ->
                             keval e2 env (fun r2 ->
                             k (Int ((unInt r1) + (unInt r2)))))
```

The converted code is indented to suggest a particular, convenient way of reading the code. In particular, while the expression `(fun r1 -> ...` only ends at the very end of the statement, and even though this whole expression is technically being passed to the first application of `keval` as a argument, we know how this argument is going to be used: it will be applied to the result of the `keval e1 env` computation. This means that we can read the code line by line as follows:

1. Apply `keval` to `e1` and `env`, and "call" the result `r1`. The name `r1` will be used in the the following lines to refer to this value,
2. Apply `keval` to `e2` and `env`, and "call" the result `r2`, and finally
3. "Return" or "continue" with the value `Int ((unInt r1) + (unInt r2))`.

As we gain more familiarity with CPS-converted code, we will find that this reading is both accurate and intuitive. The reader would be justified in thinking that CPS conversion seems to add a somewhat imperative, step-by-step feel to the code.

Passing Converted Functions to Higher-Order Functions The equality construct (Subsection 3.5) allows us to illustrate two important issues that arise with CPS conversion. The first issue concerns what conversion should do when we are passing a converted function as an argument to another (higher-order) function. For example, the reference interpreter creates functions that internally make calls to the interpreter `eval` and passes them to `map` so that they can be applied to a list of arguments. What continuation do we pass to these calls? Clearly, passing the current continuation to each of these elements would not be appropriate: it would have the effect of running the rest of the computation on each of the elements of the list as a possible, alternate, result. In fact, generally speaking, the result of CPS should only apply the current continuation exactly once. Only in situations where we are essentially backtracking do we consider sequentially applying the same continuation more than once. A practical alternative for what to pass to eval in this situation would be to pass the identity function as the continuation. This is not unreasonable, but passing the identity function as the continuation essentially means that we are locally switching back to direct style rather than CPS. The most natural way to convert the expression that maps the interpretation function to the list elements is to change the map function itself to accommodate functions that are themselves in CPS, as follows:

```
let rec kmap f l k =  match l with
                       | [] -> k []
                       | x::xs -> kmap f xs (fun r1 ->
                                  f x (fun r2 ->
                                  k (r2::r1)))
```

We will need to not only change the list but also to really push the CPS conversion process through. In addition we will need to replace the use of `List.for_all` by a function that follows CPS. Fortunately, there is no to rewrite the whole `List.for_all` function; instead, we can rewrite it using the `List.fold_left` function. The resulting code is as follows:

```
| L ((A "=") :: e1 :: l) ->
   keval e1 env (fun r1 ->
   let v = unInt r1
   in kmap (fun x k -> keval x env (fun r -> k (unInt r))) l
           (fun r -> k (Bool (List.fold_left
                                (fun bc nc -> (bc && nc=v))
                                true r))))
```

Before proceeding further, we recommend that the reader work out the derivation of this code from the original, direct-style code. While converting an expression into CPS it is useful to keep in mind that pushing the conversion as deep into the code as possible will generally improve the opportunities for staging.

Needing to Convert Libraries The second issue that the equality construct allows us to illustrate is an undesirable side-effect of CPS conversion: converting our code may require converting libraries used by the code as well. In our experience, this has only been a limited problem. In particular, interpreters tend to require only a few simple library routines for the interpretation itself rather than for the operations performed by or on the values interpreted. Again, this distinction will become clear when we consider staging the interpreter.

Lambda Abstraction Except for one interesting point, the case for lambda abstraction is straightforward:

```
| L [A "lambda" ; L axs ; e] ->
    let l = List.length axs
    in k (Fun (l, fun v ->
             keval e (lext env
                        (List.map (function A x -> x) axs)
                        (List.map (fun x -> Val x) v))
                        (fun r -> r)))
```

The interesting point is that we pass in the identity function as the continuation to `keval`. This is the correct choice here because it is the case that what we need to return (or pass to the continuation `k`) is a `Fun`-tagged function that takes in an argument and returns the interpretation of the expression `e` in the context of that argument. We simply do not have the continuation for the result of the interpretation at this point because that continuation will only become available (if and) when this function is applied.

Note that it is possible to change the type of the `Fun` tag to allow it to carry functions that are themselves in CPS. In that case, we would construct a function that takes a continuation along with the argument, and we can pass this function to `eval`. This choice, however, is not necessitated by the decision to CPS-convert the interpreter itself, and, we will see in the next section, it is possible to get the basic benefits of staging without making this change to our value domain.

The rest of the cases in the interpreter are relatively straightforward.

Note 3 (Practical Consideration: The time needed to convert to CPS). In our experience, converting the interpreter into CPS takes almost as much time as writing the interpreter itself in the first place. While this process can be automated, we find that doing the conversion by hand helps us better understand the code and leaves us better prepared for staging this code.

4.1 Experiment 2

The results of running our experiment on the CPS-converted interpreter are as follows:

```
# test2 ();;
__ read_file __________________ 10x avg = 1.526000E - 01 ms
__ parse ______________________ 10x avg = 6.843940E + 01 ms
__ kpeval ______________________ 1x avg = 2.082415E + 04 ms
__ readeval ____________________1x avg = 2.092970E + 04 ms
```

First, while there was no change to the implementations of `read_file` and `parsing`, there is some fluctuation in that reading. In our experience, it seems that there is more fluctuation with smaller values. Repeating the experiment generally produced `kpeval` and `readeval` timings within 1-2% of each other.

Contrasting these numbers to the baseline readings, we notice that CPS conversion has slowed down the implementation by about 45%.

5 Staging the CPS-Converted Interpreter

In this section, we explain the issues that arise when CPS-converting the interpreter introduced above.

5.1 Types for the Staged Interpreter

The reader will recall from Part I that MetaOCaml provides a `code` type constructor that can distinguish between regular values and delayed (or staged) values. For conciseness, we elide the so-called environment classifier parameter from types. For example, we will simply write `int code` rather than the full `('a, int) code` used in the MetaOCaml implementation.

Adding staging annotations, we will transform our interpreter from

```
keval : sxp -> (string -> var) -> (dom -> dom) -> dom
```

into

```
seval : sxp -> (string -> svar) -> (dom code -> dom code)
       -> dom code
```

where `svar` is a modified version of the `var` type defined as follows:

```
type svar = Val of dom code | Ref of (dom ref) code
```

Things That Do Not Change The overall structure of the interpreter, and literally all the code down to the main `match` statement do not change.

When we begin considering the different cases in the `match` statement of the Aloe interpreter, we realize that there is only one case that we can describe as trivial, or that it can be effectively staged by simply adding staging annotations. That is the case for `if` statements. We can expect this to be generally the case for the interpretation of all syntactic sugar, because those cases are by definition interpreted directly into calls to the interpreter with modified (source abstract syntax tree) inputs.

Cases That Require Only Staging Annotations For most of the cases in the interpreter, staging is achieved by simply adding staging annotations. Examples include booleans, list construction, integers, strings, variables, and unary, binary, and variable-arity operators. We include here some simple examples for illustration:

```
      | A "true" -> k .<Bool true>.
      | I i -> k .<Int i>.
      | A x ->
          (match env x with
            | Val v -> k v
            | Ref r -> k .< ! .~r >.)
      | L [A "not"; e1] ->
          seval e1 env (fun r ->
          k .<Bool (not (unBool .~r))>.)
```

In all cases, the application of the continuation occurs outside brackets and is therefore always performed statically.

Lambda Abstraction With one exception, the case for lambda abstraction is staged simply by adding brackets and escapes. In particular, because we know the number of arguments that the lambda abstraction takes statically, we would like to generate the code for extracting the individual parameters from the parameter list statically. This can be achieved by essentially eta-expanding the list of arguments by taking advantage of the fact that we know the number of arguments. A function that performs this operation would have the type:

```
eta_list : int -> 'a list code -> 'a code list
```

The CPS version of such a function is expressed as follows:

```
let keta_list l v k =
  let rec el_acc l v a k =
    if l<=0 then k []
    else .<match .~v with
          | x::xs -> .~(el_acc (l-1) .<xs>. (a+1) (fun r ->
                     k (.<x>. :: r)))
          | _ -> raise (Error "Expecting more arguments")>.
  in el_acc l v 0 k
```

The staged version of the lambda abstraction is simply:

```
      | L [A "lambda" ; L axs ; e] ->
          let l = List.length axs
          in k .<Fun (l, fun v ->
                   .~(keta_list l .<v>. (fun r->
                     seval e (lext env
                              (List.map (function A x -> x) axs)
                              (List.map (fun x -> Val x) r))
                           (fun r -> r))))>.
```

Note that it would have been difficult to perform `lext` statically without performing something similar to eta-expansion.

Function Application Similarly, function application requires only one main change. Because the abstract syntax tree provides us with a static list of expressions that we map to a static list of interpretations of these expressions, we need a function to convert the second type of list into a corresponding code fragment for inclusion in the generated code. In other words, we need a function with the following type:

```
lift_list : 'a code list -> 'a list code
```

This function can be expressed as follows:

```
let rec lift_list l =
  match l with | [] -> .<[]>.
               | x::xs -> .< .~x :: .~(lift_list xs)>
```

Using this function, we stage the application case as follows:

```
      | L (e::es) ->
          seval e env (fun r1 ->
            .<let (i,f) = unFun .~r1
              in .~(kmap (fun e -> seval e env) es (fun r2 ->
                   let args = r2 in
                   let l = List.length args
                   in .<if l= i
                   then let r = f .~(lift_list args)
                        in .~(k .<r>.)
                   else raise
                    (Error ("Function has "^(string_of_int l)^
                            " arguments but called with "^
                            (string_of_int i)))>.))>.)
```

Note 4 (Practical Consideration: What to Expect When Staging). To help give the reader a clear picture of the process of staging, we describe the author's experience in developing the interpreter presented in this section. The first pass of putting staging annotations without running the type-checker was relatively quick and revealed only a small number of questions. The main question was about what to do with the semantics of lambda, and then the need for introducing the two-level eta-expansion. Once this pass was completed, we ran the compiler on this staged code. There was a syntax error every 20-50 lines. Once those were fixed, the compiler began reporting typing errors. There were approximately twice as many typing errors as there were syntax errors. Many of these typing errors also revealed interesting issues that required more care while staging than originally anticipated during the first pass.

5.2 Experiment 3

The results of running the experiment on the staged interpreter are as follows:

```
# test3 ();;
__ read_file __________________ 10x avg = 1.707000E - 01 ms
__ parse ______________________ 10x avg = 6.788850E + 01 ms
__ speval ______________________ 1x avg = 1.018300E + 01 ms
__ compile ___________________   1x avg = 2.407380E + 02 ms
__ run _________________________1x avg = 6.653610E + 02 ms
__ readeval ____________________1x avg = 9.668440E + 02 ms
```

Looking at the overall time from file to result, staging provided us with a speedup of about 14 times.

6 Programs as Partially Static Data Structures

In almost any imaginable language that we may consider, there will be many programs that contain computation that can be performed *before* the inputs to the program are available. In other words, even when we ignore the possibility of having one of the inputs early, programs themselves are a source of partially static data. If we look closely at the way we have staged programs in the previous section, we will notice that we made no attempt to search for or take advantage of such information. A standard source of partially static information is closed expressions, meaning expressions that contain no variables, and therefore, contain no unknown information. Some care must be taken with this notion, because some closed programs can diverge. Another, possibly more interesting and more profitable type of partially static information that can be found in programs in *untyped* languages is *partial information about types.* This information can be captured by the data type tags that allow the runtime of an untyped language to uniformly manipulate values of different types. Because the introduction and the elimination of such tags at runtime can be expensive, reducing such *tag churn* can be an effective optimization technique.

6.1 A Partially Static Type for Denotable Values

For Aloe, we can further refine our types for the staged interpreter to facilitate taking advantage of partially static type information in a given untyped program. In particular, instead of having our staged interpreter produce only values of type `dom code`, we will allow it to produce values of a staged `dom` type, which we will call `sdom` defined as follows:

```
type sdom =
    | SBool of bool code | SInt of int code | SStr of string code
    | SFun of int * (sdom list -> sdom) | SUndefined | SVoid
    | SEmpty | SCons of dom ref code * dom ref code
    | SAny of dom code.
```

In essence, this type allows us to push tags out of the code constructor when we know their value statically. The last constructor allows us to also express the case when there is no additional static information about the tags (which was what we assumed all the time in the previous interpreter).

An important special case above is the case of `Cons`. Because each of the components of a cons cell is mutable, we cannot really use same techniques that we will consider here to push information about tags out of the `ref` constructor.

A side effect of this type is that case analysis can become somewhat redundant, especially in cases in which we expect only a particular kind of data. To minimize and localize changes to our interpreter when we make this change, we introduce a matching function for each tag along the following lines:

```
let matchBool r k =
   let m = "Expecting boolean value" in
   match r with
   | SBool b -> k b
   | SAny c -> .<match .~c with Bool b -> .~(k .<b>.)
                             | _ -> raise (Error m)>.
   | _ -> k .<raise (Error m)>.
```

It is crucial in the last case that we do not raise an error immediately. We will come back to this point in the context of lazy language constructs.

We will also change the type for values stored in the environment as follows:

```
type svar = Val of sdom | Ref of dom ref code
```

Again, we cannot really expect to pull tag information over the `ref` constructor, as side-effects change the value stored in a reference cell.

6.2 Refining the Staged Interpreter

To take advantage of partially static information present in a typical program and that can be captured by the data type presented above, we make another pass over the staged interpreter that we have just developed.

The Easy Cases The basic idea of where we get useful information is easy to see from the simple cases in our interpreter:

```
      | A "true"   -> k (SBool .<true>.)
      | A "empty"  -> k SEmpty
      | I i -> k (SInt .<i>.)
      | S s -> k (SStr .<s>.)
```

In all of these cases, it is easy to see that changing the return type from `dom code` to `sdom` allows us to push the tags out of the brackets. Naturally, the first case has more static information than we preserve in the value we return, but we will focus in this tutorial on issues relating to tag information.

Marking the Absence of Information The first case in which we encounter an absence of static tag information is environment lookup:

```
      | A x -> match env x with | Val v -> k v
                                | Ref r -> k (SAny .<(! .~r)>.)
```

Static tag information is absent in `r` because it has `dom code` type, rather than `sdom`. We accommodate this situation by using the `SAny` tag. The intuition here is that, because the de-referencing has to occur at runtime, there is no easy way to statically know the tag on that value. Similar absence of information about the resulting tag will also occur in the interpretation of `car` and `cdr` because they also involve de-referencing.

Reintroducing Information Ground values are not the only source of static information about tags. Generally speaking, the tags on the result of most primitive computations will be known statically. For example, we can refine the case for logical negation as follows:

```
    | L [A "not"; e1] ->
        xeval e1 env (fun r ->
        matchBool r (fun x ->
        k (SBool .<not .~x>.)))
```

Knowing that the tag always has to be `SBool` in the rest of the computation allows us to make sure that this tag will not occur in the generated code when this code fragment occurs in a context that expects a boolean.

Similarly, tag information is reintroduced by all other primitive operations in Aloe.

Strictness and Laziness Care is needed when refining the cases of lazy language. In particular, unlike a static type system, dynamic languages allow lazy operations to succeed even when an unneeded argument has a type that would lead to failure if it was needed. The multi-argument logical operators of Aloe, `and` and `or` are examples of such lazy language constructs. It is also interesting to note that this issue does not arise with conditionals, because even though they are lazy they are indifferent about the type tags on the arguments on which they are lazy.

The refined interpretation for `and` is as follows:

```
    | L ((A ‘‘and’’) :: es) ->
        let rec all l k =
          (match l with
            | [] -> k .<true>.
            | x::xs ->
                xeval x env (fun r ->
                matchBool r (fun x ->
                  .<if .~x
                    then .~(all xs k)
                    else .~(k .<false>.)>.)))
       in all es (fun r -> k (SBool r))
```

This code is essentially what we would expect from the last two examples. What is interesting here is that using `matchBool` instead of the dynamic `if unBool` operation would be incorrect if we were not careful about the last case in the definition of `matchBool`. In particular, if that case immediately raised an exception, then an expression such as `(and true 7)` would fail. In that case, we would be evaluating the types and checking them too strictly. By making sure that we return a code fragment that would raise an error only if we evaluate it, we ensure that the correct semantics is preserved.

Loss of Information at Mutable Locations Intuitively, the boundaries at which we will have to lose static information about tags will be places where the connection between the source and the target of this information computation must be postponed to the second stage. Cases that involve assignment and the construction of new reference cells will be examples of this situation. The places where there is a clear need for losing the static tag information in the following two cases are marked by the use of the `lift` function:

```
      | L [A "set!"; A x; e2] ->
          (match env x with
            | Val v ->
              raise (Error "Only mutable variables can be set!")
            | Ref v ->
              xeval e2 env (fun r ->
              .<let _ = (.~v:= .~(lift r)) in .~(k SVoid)>.))
      | L [A "cons"; e1; e2] ->
          xeval e1 env (fun r1 ->
          xeval e2 env (fun r2 ->
          k (SCons (.<ref .~(lift r1)>., .<ref .~(lift r2)>.))))
```

The `lift` function has type `sdom -> dom code` and is defined as follows:

```
let rec lift x =
match x with
 | SBool b     -> .<Bool .~b>.
 | SInt  i     -> .<Int  .~i>.
 | SStr  s     -> .<Str  .~s>.
 | SFun (n,f)  -> .<Fun (n,fun v ->
                    .~(keta_list n .<v>. (fun args ->
                    (lift (f (List.map (fun x -> SAny x)
                                        args))))))>.
 | SUndefined  -> .<Undefined>.
 | SVoid       -> .<Void>.
 | SEmpty      -> .<Empty>.
 | SCons (h,t) -> .<Cons (.~h, .~t)>.
 | SAny c      -> c
```

Most cases are self evident. The case functions is probably the most interesting, and there we unfold the statically known function into a dynamic function that explicitly unpacks its argument and also lifts the result of this computation. Lifting the result of the function call is easier than this code makes it seem, and the function does not necessarily need to be recursive. In particular, we will see that our interpreter only constructs functions that return values tagged with `SAny`.

Loss of Information at Function Boundaries Pushing static tag information across function boundaries can be difficult. This can be seen by analyzing what happens both in lambda abstractions and in function applications. In a lambda abstraction, we create a function with a body that computes by an interpretation. What continuation should this interpretation use? Previously, we used the identity function, but now the type of the continuation is different. It expects a `sdom` value, and yet it must still return a `code` value. The natural choice seems to be to use `lift` as the continuation and to mark this loss of information in the result by using `SAny`:

```
| L [A "lambda" ; L axs ; e] ->
    let l = List.length axs
    in k (SFun (l, fun r ->
          SAny (xeval e (lext env
                         (List.map (function A x -> x) axs)
                         (List.map (fun x -> Val x) r))
                      lift)))
```

In the case of application, because we generally do not know what function will ultimately result from the expression in the function position, we cannot propagate information across this boundary. If we introduce additional machinery, we can find useful situations in which information can be propagated across this boundary. The simplest solution, however, is as follows:

```
| L (e::es) ->
    xeval e env (fun r1 ->
      .<let (i,f) = unFun .~(lift r1)
        in .~(kmap (fun e -> xeval e env) es (fun r2 ->
             let args = (List.map lift r2) in
             let l = List.length args
             in .<if l= i
             then let r = f .~(lift_list args)
                  in .~(k (SAny .<r>.))
             else raise
              (Error ("Function has "^(string_of_int l)^
                      " arguments but called with "^
                     (string_of_int i)))>.))>.)
| _ -> raise (Error "Expression form not recognized")
```

Using `lift` on the result in the function position and on the arguments means that we are blocking this information from being propagated further. The one use of `SAny` reflects the fact that, without performing the application, we do not know anything statically about what applying the function returns.

6.3 Experiment 4

For a rough quantitative assessment of the impact of this refinement of our staged interpreter, we collect the results of running our experiment with this interpreter:

```
# test4 ();;
__ read_file __________________ 10x avg = 1.679000E - 01 ms
__ parse ______________________ 10x avg = 6.519040E + 01 ms
__ xpeval ______________________ 1x avg = 1.045800E + 01 ms
__ compile ____________________ 1x avg = 2.199970E + 02 ms
__ run ________________________ 1x avg = 4.569750E + 02 ms
__ readeval ___________________ 1x avg = 7.614950E + 02 ms
```

It appears that for overall runtime we tend to get a speedup of around 30% when we take advantage of some of the partially static tag information in our test program. When we look only at the run times for the generated computations, the speedup from this optimization could be as high as 45%.

It is interesting to note that our generation time did not go up when we introduced this optimization, and in fact, compilation time (for the generated code) went down. The shorter compilation times are possibly due to the smaller programs that are generated with this optimization (they contain fewer tagging and untagging operations).

Comparing the overall runtime to the original interpreter, we have a speedup of about 18 times. Comparing just the run times for the generated computation to the original interpreter, we have a speedup of about 30 times.

7 Conclusions

In this paper we have explained how to apply the basic techniques introduced in Part I to a large language with higher-order functions and multiple types. We also introduced an optimization technique that can be incorporated into staged interpreters and that is of particular utility to dynamic languages. We find that reasonable speedups are attainable through the use of staging.

To keep this tutorial focused on the most novel and most interesting ideas, we have not attempted to produce the most effective staged interpreter. For example, we do not consider inlining, which was outlined in Part I. And while we do consider partially static information relating to the typing tags that values carry, we ignore partially static information relating to the values themselves. In fact, we also ignored the propagation of this information across function boundaries, which is a topic we expect to be able to address in future work. Similarly, we have used the simplest possible implementation of multiple-argument functions, and one can imagine that alternative strategies (possibly using arrays) might yield better results. These are only a few examples of additional optimizations that are available to the multi-stage programmer and that can be applied to improve the performance of a staged interpreter.

Acknowledgments: I would very much like to thank the organizers and the participants of the Generative and Transformational Techniques in Software Engineering (GTTSE 2007) and University of Oregon Programming Languages Summer School (2007) for organizing these events and for all the excellent input that they provided. I would like in particular to thank Ralf Lammel for his constant support, and Dan Grossman, and Ron Garcia for their technical input on the material of the lectures. We thank Angela Zhu for proof reading and commenting on a version of this paper.

References

1. Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.
2. Walid Taha. A gentle introduction to multi-stage programming. In Don Batory, Charles Consel, Christian Lengauer, and Martin Odersky, editors, *Domain-specific Program Generation*, LNCS. 2004.
3. Philip Wadler, Walid Taha, and David B. MacQueen. How to add laziness to a strict language withouth even being odd. In *Proceedings of the 1998 ACM Workshop on ML*, pages 24–30, Baltimore, 1998.