

1 Introduction

The code-base for this project implements an interpreter for Featherweight Java. All of the code in this program can be found in the attached file, `fj.ml`. It consists of five functions to parse and interpret expressions in the Featherweight Java language. This code artifact was taken from the lecture notes [6] of COMP311, taught at Rice University.

In preparing to stage this interpreter, it is obvious that the early value should be the program to interpret, while the late values will be the environments. Through this scheme a staged interpreter can produce code that implements the interpreted program minus the values to be computed upon. Staging will eliminate the work associated with building a run-time representation of the interpreted program: the output of the staged interpreter will be all the code that is needed to produce a value for the given program, once its inputs have been provided. Thus, a significant amount of work will have been done in preparation for the second stage of execution, before any user-specified values have been declared as input to the interpreted program. Recent literature [5] details how implementing interpreters in a host language degrades performance by up to twenty times. Yet staging such interpreters can improve performance back to an acceptable level.

Given a program, and an environment mapping variables to values, the unstaged interpreter will produce the value of the interpreted program. The staged interpreter will produce code that represents the interpreted program. For example, the following piece of code represents the declaration of a class in Featherweight Java. A class declaration is a triple consisting of the name of the superclass that this class inherits from, a list of the fields in the class, and a list of the methods in the class.

```
let thunk = ("myObject",  
            ["val"],  
            [("get", ([], Sel (Var "this", "val")))])
```

The class `thunk` inherits from the root of the class hierarchy, `myObject`. It has one field, `val`, which stores the value that has been thunked. It has one method, `get`, that returns the value which the class stores. In order to create an object out of this class, the `new` constructor must be called. The following code fragment will result in a new object that is a thunk holding the value 0.

```
("thunk",  
("myObject",  
  ["val"],  
  [("get", ([], Sel (Var "this", "val")))]),  
(New ("thunk", "zero")))
```

The previous code fragment is a program. A program is a tuple that consists of a class table and an expression to be evaluated. Here, the class table has one entry: the class `Thunk`. The expression to be evaluated creates a new object based upon the `Thunk` class and initializes it to the value 0. If this program was staged, we could delay the value to be thunked until the second stage of execution. For instance, if we wanted to create a thunk, but the value to be stored was not yet known, we could generate code that would represent a thunk waiting for the value to store. For instance, the code from a staged interpreter given such a program may look like the following:

```
.< New ("thunk", Var "x")>.
```

The interpreter for Featherweight Java uses the `Option` datatype. Options are used within the interpreting function when variables must be looked up in the environment. Either the lookup will be successful and a value will be returned or a value with a `None` tag will be the result of the lookup. This may present a problem because the type of the lookup operation is not known until run-time: will the lookup succeed or fail? The type of the expression depends upon this conditional. This problem can be solved if we write the interpreter in continuation-passing style, or CPS. If we successfully rewrite the interpreter into CPS then the return type of the lookup function will have a definite type - that of the continuation.

Rewriting the interpreter into continuation-passing style may hinder performance. When the interpreter in *A Gentle Introduction to Multi-Stage*

Programming was rewritten into CPS, it suffered from a performance degradation. Although it is not yet clear if the interpreter for Featherweight Java will suffer from the same setback, a remedy to the problem may be to follow the example in the paper and ensure that all applications of the continuation in the interpreter are escaped once the staged version is rewritten.

Previous literature on staging interpreters [2] dictates that it is important to have the formal definition [1] of a language on hand when defining the interpreter for that language. Specialization (in this case, staging) is aided by inspecting the formal definition first, and finding places to stage the definition based upon that definition.

2 Interpreter for Featherweight Java

The interpreter for Featherweight Java contains the function `interp`, which is the top-level interface used to interpret programs. It takes as argument a class table, an environment, and an expression to evaluate. The interpreter is written as a large match-statement in which appropriate actions are taken depending upon the particular type of expression encountered:

```
let rec interp table =
  let interpT env =
    let rec interpTE =
      | Inv (objexp,methodname,arglist) -> . . .
      | Sel (objexp,fieldname) -> . . .
      | New (classname,fieldvalueslist) -> . . .
    in interpTE
  in interpT
```

Field lookup is implemented in the function `lookupField`, which takes as argument a class table, a field name, and the pair consisting of the class that should contain the field and the values of those fields. If `lookupField` does not find the field in the class given as argument, it will recur on the superclass of that class. The function `lookupMeth` works similarly except it takes as arguments a class table, a method name to lookup, and the class name in which the method should reside. Again, if the method is not found

in the base class then the function recurs onto the superclass in order to find the method there.

```
(* class table -> method name -> class name -> methodDef *)

let rec lookupMeth table methodname classname =
  match lookup classname table with
  | None -> raise(EvalError("Error: class not found: "^classname))
  | Some (super,_,meths) ->
      match lookup methodname meths with
      | Some x -> x
      | None -> lookupMeth table methodname super

(* class table -> field name -> (class name * field value list) -> value *)

let rec lookupField table fieldname (classname,fieldvaluelist) =
  match lookup classname table with
  | None -> raise(EvalError("Error: class not found: "^classname))
  | Some (super,fields,_) -> . . .
```

Expressions can be either variables, method invocations, field selections, or object constructors. The only values we have in this language are objects, represented by the `new` tag, the object constructor. A method definition is a tuple of a list of variables and an expression. A class table is a list of tuples consisting of class names and their declarations. A program is a class table and an expression. Finally, a class declaration is a triple consisting of a class superclass, a list of fields belonging to the class, and the method declarations for that class.

```
type methDef    = var list * exp
type classDecl = className * fieldName list * (methName, methDef) map
type classTable = (className, classDecl) map
type program   = classTable * exp
type value     = Obj of className * value list
```

The attached program, `fj.ml`, is executable in its current form and is commented sparsely for the sake of readability. I hypothesize that staging the

interpreter will obfuscate the code, and that staging itself will yield several versions of the interpreter since staging is an iterative process. The final installment of this project will include the interpreter for Featherweight Java in its staged form, along with sample inputs and timing results.

3 Related Work

I have identified four publications for use in this project. They include *Featherweight Java: A Minimal Core Calculus for Java and GJ* by Philip Wadler et al, *A Gentle Introduction to Multi-stage Programming* by Walid Taha, *Tag-less Staged Interpreters for Typed Languages* by Emir Pasalic, Walid Taha, and Tim Sheard, *From Interpreter to Compiler using Staging and Monads* by Tim Sheard and Z. Benaissa, and *What Not to Do When Writing an Interpreter for Specialization* by Neil Jones.

Featherweight Java by Philip Wadler, Benjamin Pierce, and Atsushi Igarashi presents a subset of Java that captures only the functional aspects of Java. The language is side-effect free and implements only class definitions, object creation, field access, method invocation and override, method recursion, subtyping, and casting. The authors of this work said their goal in producing Featherweight Java was to develop a language that would accurately but concisely model the type system of Java. Using this subset of Java, extensions to Java can be modeled in Featherweight Java first. I feel the assistance this paper can give me in this project may be minimal. The code artifact that I have implements a FJ interpreter in OCaml, so I feel as though I have a good starting point, and I don't have to begin from scratch. But I won't count out this paper just yet because I remember reading in *From Interpreter to Compiler using Staging and Monads* that reasoning about the semantics of a language helps in staging an interpreter for a language. I can use the formal semantics in this paper to help me with such a task.

A Gentle Introduction to Multi-stage Programming by Walid Taha presents a concise overview of multi-stage programming. In this paper the author uses an interpreter as a running example throughout the paper. I believe the ideas presented in paper will be very helpful in this project. For instance, the interpreter is rewritten into continuation passing style (CPS) in order to remedy the fact that the return type of the interpreter is unknown after option types

are introduced. Option types are used to avoid errors with division and thus result in an interpreter that returns either `Some x` or `None`. The use of continuations fixes this problem because the interpreter returns nothing once in CPS. Also, I foresee myself using let bindings to avoid code duplication (another technique presented in this paper).

Tagless Staged Interpreters for Typed Languages deals with the issue of superfluous tags [3] that arise when interpreting a typed language within a typed language. I thought this paper would be relevant to my work but after talking to Dr. Taha I realized that it does not. OCaml can not perform tag elimination: I didn't think about it, but it makes perfect sense: if tags are eliminated from expressions, type checking would fail. I felt it was necessary to comment on this paper in this report and why it will not contribute to this work.

From Interpreter to Compiler using Staging and Monads shows the benefits of using a multi-stage language when writing an interpreter [4]. For instance, the authors show how binding time improvements such as optimizing the environment of an interpreter can lead to code that is easier to analyze. If we know that our range of variables in a program will be static and never change, we can build the representation of the environment and leave the mappings to values for the second stage of execution. Furthermore, we can use the properties of monadic programming to encapsulate the effects of the language. If our desired behavior changes later on in development, all we have to do is change the monad. As difficult as this paper was to read the first time, I foresee using some of the techniques described in it. As we saw in class, conducting a monadic transformation on a program we wish to stage can ease staging. Making the monads as general as possible is important so that future modifications can be made easily.

Finally, *What Not to Do When Writing an Interpreter for Specialization* offered some advice on how to write the interpreter so that specialization (staging in this case) would be less troublesome. I will admit that most of the material presented in the paper was difficult to digest, but there were certain sections that echoed advice given in other papers as well. For instance, Jones advises that one should implement the interpreter as closely as possible to the semantics of the language being interpreted. He further goes on to say that it is best if this is done with respect to the small-step semantics of the language. I double checked the Featherweight Java paper, and if I'm reading

the semantics rules correctly, they are given in small-step manner. I will go over this paper again and attempt to digest its offerings in greater detail.

References

- [1] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java. *ACM SIGPLAN Conference on OOP, Systems, Languages & Applications*, 1999.
- [2] Neil D. Jones. What not to do when writing an interpreter for specialization. *The International Seminar on Partial Evaluation*, 1996.
- [3] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. *The International Conference on Functional Programming*, 2002.
- [4] Tim Sheard and Zine el-abidine Benaissa. From interpreter to compiler using staging and monads. 1998.
- [5] Walid Taha. A gentle introduction to multi-stage programming. *Domain Specific Program Generation*, 2003.
- [6] Walid Taha. Lecture 10: Method invocation in object-oriented languages, 2004.