

## 1 Introduction

I propose to stage an interpreter for Featherweight Java (FJ). Featherweight Java [1] is a type safe subset of Java. It is free of side effects and captures only the functional side of the language. This document presents a survey of relevant literature. I have found five publications and one website to aid in this project. Lecture ten [6] from the Comp 300 class at Rice University serves as the source for the central artifact in this work. An interpreter for Featherweight Java written in OCaml is presented in that page along with sample programs and design rationale. The five publications related to this work include work in tagless staged interpreters [3], using staging and monads to go from an interpreter to a compiler [4], an introduction to multi-stage programming (MSP) [5], an introduction to Featherweight Java, and advice on how to write an interpreter for specialization [2].

I expect staging a FJ interpreter to be useful for two reasons: pedagogy and research. Staging this interpreter will give me a better understanding of staging in general, and I feel I will encounter obstacles that will require the use of the techniques described in the papers above. The successful application of such techniques will serve to better prepare me for future projects and could result in publishable material.

## 2 Code artifact

```
type ('a,'b) map = ('a * 'b) list
```

```
let empty = []
```

```
(*=====*)  
(*      LOOKUP      *)  
(*=====*)
```

```

(* symbol -> (symbol * 'a) list -> 'a *)

let rec lookup k' x =
  match x with
  ((k,v)::l) -> if k=k' then Some v else lookup k' l
  | _         -> None

(*=====*)
(*      ZIP                                     *)
(*=====*)

(* ('a list * 'b list) -> ('a * 'b) list *)

let rec zip (a,b) =
  match (a,b) with
  (x::xs, y::ys) -> (x,y) :: zip (xs,ys)
  | _            -> []

(*=====*)
(*      Types for Variable,                   *)
(*      Class Names                           *)
(*      Method Names                           *)
(*      Field Names                            *)
(*      Expressions                            *)
(*=====*)

type var      = string (* including "this" *)
type className = string
type methName = string
type fieldName = string

type exp =
  Var of var

```

```

    | Inv of exp * methName * exp list
    | Sel of exp * fieldName
    | New of className * exp list

(*=====*)
(* Method Definitions: Variable List X Expression *)
(* Class Declarations:
(* Class Name X List of Field Names X List of Method Definitions*)
(* Class Tables: List of Class Names X Class Declarations*)
(* Program: Class Table X Expression *)
(*=====*)

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

(*=====*)
(* Values are Objects *)
(*=====*)

type value = Obj of className * value list

(*=====*)
(* Root of the Class Hierarchy *)
(* class myObject { } *)
(*=====*)

let myObject = ("", [], empty)

(*=====*)
(* Class Thunk to Memoize Values *)
(* class thunk ext myObject *)
(* { myObject val; myObject get () { this.val } } *)
(*=====*)

```

```

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

(*=====*)
(* Boolean Class, True, False Classes *)
(* class bool ext myObject *)
(* { myObject if (thunk x, thunk y) { this.if(x,y) } } *)
(* class true ext bool *)
(* { myObject if (thunk x, thunk y) { x.get() } } *)
(* class myFalse ext bool *)
(* { myObject if (thunk x, thunk y) { y.get() } } *)
(*=====*)

let bool = ("myObject",
  [],
  [("if", (["x";"y"], Inv (Var "this", "if", [Var "x";Var "y"])))])

let myTrue = ("bool",
  [],
  [("if", (["x";"y"], Inv (Var "x", "get", [])))]

let myFalse = ("bool",
  [],
  [("if", (["x";"y"], Inv (Var "y", "get", [])))]

(*=====*)
(* Exception Types *)
(*=====*)

exception EvalError of string

(*=====*)
(* Method Lookup *)
(* *)
(* 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

(*=====*)
(*      Field Lookup                                *)
(*                                                    *)
(* 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,_) ->
      let rec search (a,b) =
          match (a,b) with
          | (name::namelist, value::valuelist) ->
              if fieldname=name then value else search (namelist,valuelist)
          | ([],valuelist) ->
              lookupField table fieldname (super,valuelist)
          | (_,[]) ->
              raise(EvalError("Error: Some fields not defined."))
      in search (fields,fieldvaluelist)

(*=====*)
(*      Interpreter                                *)
(*                                                    *)
(* interp: class table -> env -> exp -> result *)
(*=====*)

```

```

let rec interp table =
  let interpT env =
    let rec interpTE =
      fun a -> match a with
        Var x -> let v' = lookup x env in
          (match v' with
            Some v -> v
            | None ->
              raise(EvalError("Error: variable not found: "^x)))
    in Inv (objexp,methodname,arglist) ->
      let v = (interpTE objexp) in
      let classname = match v with
        (Obj (c,_)) -> c in
      let z = lookupMeth table methodname classname
        and argvlist = (List.map interpTE arglist) in
      let (varlist,evalmeth) = z in
      interp table (("this",v)::(zip (varlist,argvlist))) evalmeth
    | Sel (objexp,fieldname) ->
      let Obj (classname,fieldvaluelist) = (interpTE objexp)
      in (lookupField table fieldname (classname,fieldvaluelist))
    | New (classname,fieldvalueslist) ->
      Obj (classname, (List.map interpTE fieldvalueslist))
      in interpTE
  in interpT

```

### 3 Survey of Literature

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 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. 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.