ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Staged Tagless Interpreters in Dotty

Semester project, fall 2018

	Benoit Knuchel	(*benoit.knuchel@epfl.ch*)
Supervisor	Nicolas Stucki	(*nicolas.stucki@epfl.ch*)
Supervisor	Aggelos Biboudis	(*aggelos.biboudis@epfl.ch*)
Director	Martin Odersky	(*martin.odersky@epfl.ch*)

January 2019

Contents

Introduction

FIRSTLY what is an interpreter ? Basically it is a program that converts source code from a high-level language into machine code and executes it on the go. It is a way of embedding domain-specific languages into another programming language. In this project we apply the finally tagless approach for embedding libraries in Scala for type-preserving interpretation, compilation and partial evaluation. We use the new research compiler, Dotty, that offers new features for Staging (Partial Evaluation). Conversely, note that the difference with a compiler is that it will translate the program as a whole into machine code and then execute it.

We are interested in adding two different techniques to interpreters :

1. Tagless
2. Staging

Staging allows us to do execution-time optimization while tagless eliminates tagging overhead. Jacques Carette, Oleg Kiselyov and Chung-chieh Shan wrote staged tagless interpreters in OCaml and Haskell [1]. This is one of the followed papers.

The goal of this work is to port the ideas of embedding domain-specific languages using tagless interpreters in Scala. Before, it was not possible to implement these types of interpreters in vanilla Scala since the necessary staging capabilities arrived with Dotty.

The other paper used for this project is about multi-stage programming. It is easily readable and helps familiarize with this paradigm [2].

The code for this project is open source[1].

[1] `https://github.com/benoitknuchel/Staged-Tagless-Interpreters-in-Dotty` .

Part 1

Staged and Tagless

1.1 Staging

Staging uses two operations :

- Quotations, expressed as `'( )` or `'{ }` for expressions and `'[ ]` for types
- Splicing, expressed as ~

Assuming that `e` is an expression then `'(e)` (of type `Expr[T]`) represents the typed abstract syntax tree of `e`. Equivalently, if `T` is a type then `'[T]` (of type `Type[T]`) represents the type structure of `T`. Inversely we have that `~e` evaluates the expression `e`. Respectively for type `T`. These two operations are opposite :

$\sim'(e) = e, \quad \sim'(T) = T$

$'(\sim e) = e, \quad '(\sim T) = T$

To get the result of an expression `e` of type `Expr[T]` we can call the `show` or `run` method. The result can either be transformed into a string with `e.show` or can be executed with `e.run`.

Multi-stage Programming is a meta-programming technique. This means that it is a programming language technology to write programs that generate programs. In this work we consider type-safety and scope-safety as an important property during the development of metaprograms and Multi-stage Programming offers both. For more terminology and information refer to the chapter 4.1 of Smaragdakis et al. [3].

When code is staged, it is prepared to be executed at a later stage. It is really effective in the presence of recursive functions: Assume a recursive program for the calculation of power. The regular version of it, with both parameters available at runtime, will recurse during execution and calculate the result with the normal operational semantics of the language. However, if we know some part of the method call statically, we can *partially evaluate* it.

An example will make that clear :

Listing 1.1: Fast exponentiation algorithm

```scala
def fastExp(x: Double, n: Int): Double = {
    if(n == 0) 1.0
    else if(n == 1) x
    else if(n % 2 == 0) fastExp(x*x, n/2)
    else x*fastExp(x, n-1)
}

def fastExpStaged(x: Expr[Double], n: Int): Expr[Double] = {
    if (n == 0) '(1.0)
    else if (n == 1) x
    else if (n % 2 == 0) '{ val y = ~x * ~x; ~fastExpStaged('(y), n/2) }
    else '{ ~x * ~fastExpStaged(x, n - 1) }
}
```

Listing 1.2: Fast exponentiation algorithm tests

```scala
def main(args: Array[String]): Unit = {
    val e = fastExpStaged('(5), 5)
    println("fastExpStaged(5,5).show : " + e.show)
    println("fastExpStaged(5,5).run : " + e.run)
}

...
```

The `fastExpStaged` method is the staged version of `fastExp`. It takes as argument `x: Expr[Double]` which is a `Double` that is staged and returns an `Expr[Double]`. Assume we have that $n\%2 = 0$, the program will evaluate the value of `x` and stage it back, as `5.0*5.0`, to the next recursion. Which yields as result:

Listing 1.3: Fast exponentiation algorithm results

```
fastExpStaged(5, 5).show :
    5.0.*({
        val y: scala.Double = 5.0.*(5.0)
        val y$2: scala.Double = y.*(y)
        y$2
    })

fastExpStaged(5, 5).run : 3125.0
```

As we can see recursion and calls have been eliminated alongside the runtime overhead they would incur. Meta-programming in Dotty is explained in more details in this paper [4].

1.2 Tagless

Tagless final is a functional pattern that has multiple advantages :

- It helps making sure that the programs we write are correct, especially the fact they are typesafe and predictable, and in our case that the interpreter will not get stuck. Note that it is hard to write incorrect programs since invalid states cannot even be expressed.
- It eases extensibility of the program. This problem is called the "Expression problem" [5] and tagless-interpreters provide a solution to that. In the object oriented world, this was rediscovered under the name "Object Algebras" **oliveira_extensibility_2012**

An example will make that clearer :

Listing 1.4: Two tagless interpreters

```scala
trait Symantics {
  type repr[_]
  def num(x: Int): repr[Int]
  def add(x: repr[Int], y: repr[Int]): repr[Int]
}

object interpreter extends Symantics {
  type repr[A] = A
  def num(x: Int): repr[Int] = x
  def add(x: repr[Int], y: repr[Int]): repr[Int] = x + y
}

object interpreterPrint extends Symantics {
  type repr[A] = String
  def num(x: Int): repr[Int] = x.toString
  def add(x: repr[Int], y: repr[Int]): repr[Int] = x + " + " + y
}

object Main {
  //import interpreter._
  import interpreterPrint._

  def main(args: Array[String]): Unit = {
    val x = add(num(1), add(num(2), add(num(3), num(4))))
    println(x)
  }
}
```

This program represents two tagless interpreters. One will evaluate the expression and the other one will pretty-print it. Note that they both extends the same trait, so nothing has to be changed for the example : it suffices to choose which interpreter should do the work.

The main difference with a tagged interpreter is that it does not use any GADT (Generalized Algebraic Data Type). For more information about GADT, look at figure 2.1 (enum) or at those explanations in Haskell [6].

To add a new functionality to the language we just have to define it in `Symantics` and implement it in `interpreter` and `interpreterPrint`. For example if we want to add a function that multiplies two `num` we just have to add three lines of code (one in `Symantics`, one in `interpreter`, one in `interpreterPrint`).

As we can see it is pretty easy to compose multiple functions of the language to create bigger ones. It will be useful when optimizing expressions with the tagless partial evaluator in section 3.3.

The tag problem is explained in more details in section 1.1 of the paper written by Jacques Carette, Oleg Kiselyov and Chung-chieh Shan [1].

Part 2

Staged Interpreters

We present two tag-full interpreters, one regular and one staged. This chapter presents the first step towards tagless interpreters. We will start with a basic staged interpreter with error handling and then greatly improve it using continuation-passing style. We define what the language will do we use enumerations provided by Dotty as a GADT encoding of our language:

Listing 2.1: Definition of the language used in part 2

```
enum Exp {
  case int(x: Int)
  case Var(s: String)
  case App(s: String, e: Exp)
  case Add(e1: Exp, e2: Exp)
  case Sub(e1: Exp, e2: Exp)
  case Mul(e1: Exp, e2: Exp)
  case Div(e1: Exp, e2: Exp)
  case Ifz(e1: Exp, e2: Exp, e3: Exp)
}

enum Def {
  case Declaration(s1: String, s2: String, e: Exp)
}

enum Prog {
  case Program(list: List[Def], e: Exp)
}
...
```

Consequently we can define factorial(5) as follows :

Listing 2.2: Factorial(5)

```
val factorial = Program(List(Declaration
    ("fact", "x",
     Ifz(Var("x"),
      int(1),
      Mul(Var("x"), (App("fact", Add(Var("x"), int(-1)))))))),
    App("fact", int(5)))
```

2.1 Staged interpreter

The translation from OCaml to Scala is straightforward for this part. There is one thing to note: `int code` in OCaml is `Expr[Int]` in Scala. Listings 2.3 and 2.4 show a part of this interpreter.

Listing 2.3: The eval function in OCaml taken from [2]

```
(* eval2 : exp -> (string -> int code) -> (string -> (int -> int) code)
            -> int code *)

let rec eval2 e env fenv =
match e with
    Int i -> .<i>.
    | Var s -> env s
    | App (s,e2) -> .<.~(fenv s).~(eval2 e2 env fenv)>.
    ...
    | Div (e1,e2)-> .<.~(eval2 e1 env fenv)/ .~(eval2 e2 env fenv)>.
    | Ifz (e1,e2,e3) -> .<if .~(eval2 e1 env fenv)=0
                           then .~(eval2 e2 env fenv)
                           else .~(eval2 e3 env fenv)>.
...
```

Listing 2.4: The eval function in Scala

```
def eval(e: Exp, env: String => Expr[Option[Int]], fenv: String =>
  Expr[Int => Option[Int]]): Expr[Option[Int]] = e match {

  case int(x) => '(Some(~x.toExpr))

  case Var(s) => env(s)

  case App(s, e) => '{
    ~eval(e, env, fenv) match {
      case Some(x) => (~fenv(s))(x) : Option[Int]
      case _ => None : Option[Int]
    }
  }
  ...
  case Div(e1, e2) => '{
    (~eval(e1, env, fenv), ~eval(e2, env, fenv)) match {
      case (Some(x), Some(y)) => if(y != 0) Some(x/y) : Option[Int] else↩
          None : Option[Int]
      case _ => None : Option[Int]
    }
  }
  ...
}
...
```

2.2 Binding-time improvement

Binding-time analysis are techniques that Partial Evaluators used in the past to discover which input is known and which is not. Staging removes that complexity (we annotate the programs with Exprs). Continuation-passing style is a technique long used for these kind of improvements [7] even in staged programs. And again the translation is straightforward.

Listing 2.5: The eval function using CPS in OCaml taken from [2]

```
(* eval6 : exp -> (string -> int code) -> (string -> (int -> int) code)
-> (int code option -> 'b code) -> 'b code *)

let rec eval6 e env fenv k =
match e with
    Int i -> k (Some .<i>.)
    | Var s -> k (Some (env s))
    | App (s,e2) -> eval6 e2 env fenv
                      (fun r -> match r with
                        Some x -> k (Some .<.~(fenv s) .~x>.)
                        | None -> k None)
    | Add (e1,e2) -> eval6 e1 env fenv
                      (fun r ->
                        eval6 e2 env fenv
                          (fun s -> match (r,s) with
                            (Some x, Some y) ->
                              k (Some .<.~x + .~y>.)
                              | _ -> k None))
...
```

Listing 2.6: The eval function using CPS in Scala

```
def eval[B](e: Exp, env: String => Expr[Int], fenv: String => Expr[Int ↩
    => Int], k: Option[Expr[Int]] => Expr[B]): Expr[B] = e match {

    case int(x) => k(Some(x.toExpr))

    case Var(s) => k(Some(env(s)))

    case App(s, e) =>
      eval(e, env, fenv,
        (r: Option[Expr[Int]]) => r match {
          case Some(x) => k(Some('{(~fenv(s))(~x)}))
          case _ => k(None)
        }
      )

    case Add(e1, e2) =>
      eval(e1, env, fenv,
        (r: Option[Expr[Int]]) => {
          eval(e2, env, fenv,
            (s: Option[Expr[Int]]) => (r, s) match {
              case (Some(x), Some(y)) => k(Some('{~x + ~y}))
              case _ => k(None)
            }
          )
        }
      )
    ...
}
```

A little comparison of the generated code by these two interpreters is needed. We will take a simple example: `Div(int(10), int(2))`.

Listing 2.7: The eval function using CPS in Scala

```
scala.Tuple2.apply[scala.Option[scala.Int], scala.Option[scala.Int]](↩
    scala.Some.apply[scala.Int](10), scala.Some.apply[scala.Int](2)) ↩
    match {
  case scala.Tuple2(scala.Some(x), scala.Some(y)) =>
    if (y.!=(0)) (scala.Some.apply[scala.Int](x./(y)): scala.Option[↩
        scala.Int]) else (scala.None: scala.Option[scala.Int])
  case _ =>
    (scala.None: scala.Option[scala.Int])
}
```

It is a lot of code just for a division and it is not easily readable. Let’s take the same example but with the continuation-passing style implemented :

Listing 2.8: The eval function using CPS in Scala

```
if (2.==(0)) throw new java.lang.ArithmeticException() else {
  val z: scala.Int = 10./(2)
  z
}
```

There is not only less code but it’s clear enough to be easily readable!

Part 3

Staged Tagless Interpreters

The next embedding that we will use supports the same constructs, namely, addition, lambda abstraction, application, booleans, integers, ... This time we use a tagless encoding to interpret it.

Listing 3.1: Interface representing the language taken from [1]

```ocaml
module type Symantics = sig
  type ('c, 'sv, 'dv) repr

  val int : int -> ('c, 'int, 'int) repr
  val bool : bool -> ('c, 'bool, 'bool) repr

  val lam : (('c, 'sa, 'da) repr -> ('c, 'sb, 'db) repr as 'x)
            -> ('c, 'x, 'da -> 'db) repr
  val app : (('c, 'x, 'da -> 'db) repr
            -> (('c, 'sa, 'da) repr) -> ('c, 'sb, 'db) repr as 'x)
  val fix : ('x -> 'x) -> (('c, ('c, 'sa, 'da) repr ->
                              ('c, 'sb, 'db) repr,
                              'da -> 'db) repr as 'x)

  val add : ('c, 'int, 'int) repr -> ('c, 'int, 'int) repr
            -> ('c, 'int, 'int) repr
  val mul : ('c, 'int, 'int) repr -> ('c, 'int, 'int) repr
            -> ('c, 'int, 'int) repr
  val leq : ('c, 'int, 'int) repr -> ('c, 'int, 'int) repr
            -> ('c, 'bool, 'bool) repr
  val if_ : ('c, 'bool, 'bool) repr
            -> (unit -> 'x) -> (unit -> 'x)
            -> (('c, 'sa, 'da) repr as 'x)
end
```

In MetaOCaml we have an extra parameter, ’c, that in Scala is not needed (it is called Environment Classifier). The `repr` is an abstract type: we use it to abstract the interpreter. It has to be defined for each interpreter that implements this interface.

The abstract type `repr` defines two types: `'sv`, stands for static, and `'dv`, stands for dynamic. It will be useful with the partial evaluator.

3.1 Tagless interpreter

The translation of most of the methods discussed previously was straightforward. The main challenge was the translation of ‘fix‘, also called fix-point combinator.

Listing 3.2: Fix-point combinator in OCaml taken from [1]

```ocaml
let fix f = let rec self n = f self n in self
```

Listing 3.3: Fix-point combinator in Scala

```scala
override def fix[SA: Type, DA: Type, SB: Type, DB: Type]
  (f: repr[repr[SA, DA] => repr[SB, DB], DA => DB] =>
      repr[repr[SA, DA] => repr[SB, DB], DA => DB]):
      repr[repr[SA, DA] => repr[SB, DB], DA => DB] =
  (x: DA) => f(fix(f))(x)
```

3.2 Staged tagless interpreter

This is just a staged version of the interpreter from last section except for the `fix` method. Here `fix` is tail recursive with as inner function `let rec self n = ... in self`. So in Scala you first define the inner function as `def self(n)= ...` and then you call it with `self(n)`.

Listing 3.4: Fix method staged in MetaOCaml taken from [1]

```ocaml
let fix f = .<let rec self n = .~(f .<self>.) n in self>.
```

Listing 3.5: Fix method staged in Scala

```scala
def fix[SA: Type, DA: Type, SB: Type, DB: Type]
    (f: repr[repr[SA, DA] => repr[SB, DB], DA => DB] =>
        repr[repr[SA, DA] => repr[SB, DB], DA => DB]):
        repr[repr[SA, DA] => repr[SB, DB], DA => DB] =
  '{
    def self(n: DA): DB = {
        ~f('(self)).apply('(n))
    }
    (n: DA) => self(n)
  }
```

3.3 Partial evaluator

A partial evaluator will evaluate the code to do some mathematical optimization such as: $x+0=x$, $x*0=0$, $x*1=1$, $x/1=x$, $0/x=0$.

There were two difficulties with the translation of this code. The first was to define `repr`, as seen in Listing 3.6, in Scala. Here it is defined by a pair, `StatDyn[A, B]`. The first element is the static part, which we do not have all the time so it is an `Option[A]`. The second element is the dynamic part, which is staged.

Listing 3.6: The repr type in OCaml taken from [1]

```
type '(c',sv',dv) repr = {st: 'sv option; dy: '(c',dv) code}
```

Listing 3.7: The repr type in Scala

```
type StatDyn[A, B] = (Option[A], Expr[B])
```

To be able to do the desired optimization, the code needs to be statically known : we cannot do it if we only have the dynamic part so we are using pattern matching on the static part. We will take the `mul` and `div` functions as examples :

Listing 3.8: mul and div optimization

```
override def mul(x: StatDyn[Double, Double],
                 y: StatDyn[Double, Double]):
                 StatDyn[Double, Double] =
  (x._1, y._1) match {
    case (Some(0), Some(a)) => num(0)
    case (Some(a), Some(0)) => num(0)
    case (Some(1), Some(a)) => y
    case (Some(a), Some(1)) => x
    case _ => pdyn(evalStaged.evalStaged.mul(abstr(x), abstr(y)))
  }

override def div(x: StatDyn[Double, Double],
                 y: StatDyn[Double, Double]):
                 StatDyn[Double, Double] =
  (x._1, y._1) match {
    case (Some(0), Some(a)) => num(0)
    case (Some(a), Some(0)) => throw new ArithmeticException
    case (Some(a), Some(1)) => x
    case _ => pdyn(evalStaged.evalStaged.div(abstr(x), abstr(y)))
  }
```

The second difficulty was to translate the `fix` method, especially the `self` function (it is called a partial function). It goes like this in OCaml :

Listing 3.9: Fix method in OCaml taken from [1]

```
let fix f = let fdyn = C.fix (fun x -> abstr (f (pdyn x)))
            in let rec self = function
                              | {st = Some _} as e ->
                                    app (f (lam self)) e
                              | e -> pdyn (C.app fdyn (abstr e))
                in {st = Some self; dy = fdyn}
```

And it can be represented like this in Scala :

Listing 3.10: Fix method in Scala

```scala
override def fix[SA: Type, DA: Type, SB: Type, DB: Type]
    (f: repr[repr[SA, DA] => repr[SB, DB], DA => DB] =>
        repr[repr[SA, DA] => repr[SB, DB], DA => DB]):
        repr[repr[SA, DA] => repr[SB, DB], DA => DB] = {

    def fdyn: Expr[DA => DB] = evalStaged.fix((x: Expr[DA => DB]) =>
    abstr(f(pdyn(x))))

    lazy val self: StatDyn[SA, DA] => StatDyn[SB, DB] = {
      case e @ (_: Some[_], _) => app(f(lam(self)), e)
      case a => pdyn(evalStaged.evalStaged.app(fdyn, (abstr(a))))
    }

    (Some(self), fdyn)
}
```

Let's compare the generated code of the staged interpreter from last section and the partial evaluator.

```scala
//factorial(5)
val ex =
    app(
      fix((fact: repr[repr[Double, Double] => repr[Double, Double], ↩
          Double => Double]) =>
        (lam((n: repr[Double, Double]) => if_(leq(n, num(1)), n, mul(n, ↩
            app(fact, add(n, neg(num(1)))))))
      ), num(5)
    )
```

We have these results when using `.show` on the dynamic part :

```
Staged interpreter:
val x$3: scala.Double = 5.0
  def self(n: scala.Double): scala.Double = {
    val x$2: scala.Double = n
    if (x$2.<=(1.0)) x$2 else x$2.*({
      val x$1: scala.Double = x$2.+(-1.0)
      self(x$1)
    })
  }
  self(x$3)

Partial evaluator:
5.0.*(4.0.*(3.0.*(2.0)))
```

Part 4

Tagless Interpreters and Continuation-passing Style

4.1 Call-by-name

The object language takes the evaluation strategy of the metalanguage, in our case Scala which is call-by-value. Here we want to decouple the evaluation strategy to get a call-by name interpreter.

Again the difficulty here was to define `repr`. The name of this structure in OCaml is called a record which do not have an equivalent as is in Scala.

Listing 4.1: repr in OCaml using a record taken from [1]

```
type ('c, 'sv, 'dv) repr = {ko: 'w. ('sv -> 'w) -> 'w)}
```

In CPS, `k` represents the continuation of the program. The interpretation of an object term is a function mapping `k` to the answer returned by `k`. In the add example, Listing 4.3, the interpretation has type `(int -> 'w)-> 'w`.

After trying out different options we found out that the easiest solution was to define it as an abstract class.

Listing 4.2: repr in Scala as an abstract class

```
abstract class repr[SV, DV] {
    def ko[W](k: SV => W): W
}
```

To see how to use this we will take the `add` method as example :

Listing 4.3: Add method in OCaml taken from [1]

```
let add e1 e2 = {ko = fun k ->
    e1.ko (fun v1 -> e2.ko (fun v2 -> k (v1 + v2)))}
```

Listing 4.4: Add method in Scala

```
def add[DV](x: repr[Double, DV], y: repr[Double, DV]):
              repr[Double, DV] =
    new repr[Double, DV] {
        def ko[W](k: Double => W): W =
            x.ko((v1: Double) => y.ko((v2: Double) => k(v1 + v2)))
    }
```

To show that this is really a CBN interpreter we will take an example that can only terminate under CBN and not CBV :

Listing 4.5: $(\lambda x.1)(fixf.f)2$

```scala
val diverg = app(
    lam((x: repr[Double, Double]) => num(1)),
    app(fix((f: repr[repr[Double, Double] =>
                repr[Double, Double], Double]) => f),
        num(2)))
```

Which gives the desired result : 1.0.

4.2 Call-by-value CPS transformer

Now we want to make a program that takes as input an interpreter and transforms it into a call-by value one.

For this part we make an exception and do a slight change in the trait `Symantics` : we change `type repr[_,_]` to `type repr[_]` and keep only the second argument(we change `repr[A, ↩ B]`, to `repr[B]`).

So here our interpreter will not be represented by an object but by a class that takes one argument :

Listing 4.6: The class representing the CBV transformer in OCaml taken from [1]

```ocaml
module CPST(S: Symantics) = struct
...
```

Listing 4.7: The class representing the CBV transformer in Scala

```scala
class CPSTransformer[Sym <: Symantics2 & Singleton](val S: Sym){
    type W = Unit
    type repr[DV] = S.repr[DV]
    ...
}
```

The rest of the translation is direct except the fact that we have to be more precise with the types, especially with the continuation. Let's take again the `add` example :

Listing 4.8: Add method in OCaml taken from [1]

```ocaml
let add e1 e2 = S.lam (fun k -> S.app e1 (S.lam (fun v1 ->
                                S.app e2 (S.lam (fun v2 ->
                                S.app k (S.add v1 v2))))))
```

Listing 4.9: Add method in Scala

```scala
def add[W: Type](x: repr[(Double => W) => W],
                 y: repr[(Double => W) => W]):
                 repr[(Double => W) => W] =
    S.lam(k =>
      S.app(x, S.lam(v1 =>
        S.app(y, S.lam(v2 =>
          S.app(k, S.add(v1, v2)))))))
```

4.3 State and imperative features

For the last part, the interpreter is extended with imperative features. There are 3 possible operations with a state :

1. !state : returns the current state
2. state ⟵ e : sets the state to e and returns the previous value
3. case e1 of x.e2 : applies e1 in the function e2 as the parameter x, note that we want to first evaluate e1 before e2 even if e2 does not use x

Those 3 operations are represented by `deref`, `set(e)` and `lapp(e2, e1)`. We will take an example to make that clear. This program adds 2 to a given initial value using the new feature :

```scala
We write :
case !state of x.(state <- 2; x + !state)

as
val ex = lapp(deref(), (x: repr[Double, Double]) =>
    lapp(set(num(2)), (_: repr[Double, Double]) => add(x, deref())))
```

To add those features we first have to create a new trait, called `SymSI`, that will extend `Symantics`. It defines a new type and the 3 new operations needed. The interpreter will extend this new interface.

Listing 4.10: New trait with added imperative features

```scala
trait SymSI extends SymanticsTypeNotStaged {

  abstract class repr[SV, DV] {
    def ko[W](k: (SV) => (State_ST => W)): State_ST => W
  }

  type State
  type State_ST

  def lapp[SA, DA, SB, DB]
    (arg: repr[SA, DA],
     f: repr[SA, DA] => repr[SB, DB]): repr[SB, DB]

  def deref(): repr[State_ST, State]

  def set(a: repr[State_ST, State]): repr[State_ST, State]
}
```

Note that we will not use the new type `State_ST` here, we will just set it to be equal to `State`. More information about the usefulness of this type can be found here [1].

The interpreter with the three additional features and a function to get the result looks like this in OCaml :

Listing 4.11: Interpreter with imperative features in OCaml taken from [1]

```
module RCPS(ST: sig
  type state
  type 'c states
  type '(c',sv',dv) repr =
    {ko: 'w. '(sv -> 'c states -> 'w) -> 'c states -> 'w}
end) = struct include ST ...
  let lapp e2 e1 = {ko = fun k ->
    e2.ko (fun v -> (app (lam e1) {ko = fun k -> k v}).ko k)}
  let deref () = {ko = fun k s -> k s s}
  let set e = {ko = fun k -> e.ko (fun v s -> k s v)}
  let get_res x = fun s0 -> x.ko (fun v s -> v) s0
end
```

And can be translated to this in Scala :

Listing 4.12: Interpreter with imperative features in Scala

```
class RCPS() extends SymSI {
  ...
  def lapp[SA, DA, SB, DB]
    (arg: repr[SA, DA], f: repr[SA, DA] => repr[SB, DB]):
      repr[SB, DB] = new repr[SB, DB] {
        def ko[W](k: (SB) => (State_ST => W)): (State_ST => W) = {
          arg.ko((v: SA) => app[SA, DA, SB, DB](lam[SA, DA, SB, DB](f),
            new repr[SA, DA] {
              def ko[W](k: (SA) => (State_ST => W)): (State_ST => W) = {
                k(v)
              }
            }).ko(k))
          }
      }

  def deref(): repr[State_ST, State] =
    new repr[State_ST, State] {
      def ko[W](k: State_ST => (State_ST => W)): (State_ST => W) =
        (s: State_ST) => k(s)(s)
    }

  def set(e: repr[State_ST, State]): repr[State_ST, State] =
    new repr[State_ST, State] {
      def ko[W](k: (State_ST) => (State_ST => W)): (State_ST => W) =
        e.ko(v => s => k(s)(v))
    }

  def getResult(x: repr[State_ST, State], init: State_ST) =
    x.ko(v => s => v)(init)

}
```

More examples, especially the power and factorial functions, using these new features can be found with the code of this project. [1].

[1] https://github.com/benoitknuchel/Staged-Tagless-Interpreters-in-Dotty/blob/master/interpreters/src/main/scala/tagless/continuation_passing_style/stateImperativeFeatures/stateImperativeFeatures.scala

Part 5

Conclusion

This work shows the implementation of staged tagless interpreters in Dotty. This project was based on two papers [1] [2].

The aim of this project was to test Dotty's staging capabilities. These allowed us to implement a new family of interpreters in Scala by translating what has been done before in OCaml and Haskell. Most of the translations are straightforward for people who know about those languages but can be challenging for someone who never used them.

The results of the examples show that it is indeed effective, especially for the partial evaluator.

The next step would be to apply this in practice with a real language. Following LIFT [8] about generating code for high-performance GPU would be a good test.

Bibliography

[1] J. Carette, O. Kiselyov, and C. chieh Shan, “Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages”, 2008.

[2] W. Taha, “A gentle introduction to multi-stage programming”, en, no. 3016, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds., pp. 30–50, 2004.

[3] Y. Smaragdakis, A. Biboudis, and G. Fourtounis, “Structured program generation techniques”, in *Grand Timely Topics in Software Engineering*, J. Cunha, J. P. Fernandes, R. Lämmel, J. Saraiva, and V. Zaytsev, Eds., Cham: Springer International Publishing, 2017, pp. 154–178, ISBN: 978-3-319-60074-1.

[4] N. Stucki, A. Biboudis, and M. Odersky, “A Practical Unification of Multi-stage Programming and Macros”,

[5] P. Wadler, *The Expression Problem*, https://web.archive.org/web/20170322142231/http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, Dec. 1998. (visited on 03/22/2017).

[6] “Haskell/GADT”, [Online]. Available: https://en.wikibooks.org/wiki/Haskell/GADT.

[7] K. Nielsen and M. H. Sørensen, “Call-by-name cps-translation as a binding-time improvement”, in *STATIC ANALYSIS, NUMBER 983 IN LECTURE NOTES IN COMPUTER SCIENCE*, Springer-Verlag, 1995, pp. 296–313.

[8] M. Steuwer, T. Remmelg, and C. Dubach, “Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation”, 2017. [Online]. Available: http://www.lift-project.org/publications/2017/steuwer17LiftIR.pdf.