CS150 APL: Partial Evaluation

Guannan Wei
guannan.wei@tufts.edu

Oct 21, 2025

Tufts University

Logistics

- Project kick-off
- Midterm course evaluation

Optimizing programs by transformation

Transform programs to improve performance while preserving semantics.

Performance:

- Time (faster execution)
- Space (smaller code size, less memory usage, etc.)

Optimizations:

- Constant propagation/folding
- Dead code elimination
- Function inlining
- Common subexpression elimination
- ...

Partial evaluation

- Partial evaluation:
  - "The fox knows many things, but the hedgehog knows one big thing."
  - Generate specialized programs from general programs
  - Semantics preserving

Partial evaluation

- Partial evaluation:
  - "The fox knows many things, but the hedgehog knows one big thing."
  - Generate specialized programs from general programs
  - Semantics preserving
- Binding-time distinctions: the time when the value of a variable is known.
  - Static: known at compile time
    - `let x = 5 in x + 3`
  - Dynamic: known at run time
    - `let x = readInt() in x + 3`

Partial evaluation: example

- "Hello world": computing the x to the power of n:

```
def pow(x: Int, n: Int): Int =
  if (n == 0) 1
  else x * pow(x, n - 1)
```

Partial evaluation: example

- "Hello world": computing the x to the power of n:

```
def pow(x: Int, n: Int): Int =
  if (n == 0) 1
  else x * pow(x, n - 1)
```

- We know n = 3 at compile time, but x is only known at run time:

```
   pow(x, 3)
=> if (3 == 0) 1 else x * pow(x, 2)
=> x * pow(x, 2)
=> x * (if (2 == 0) 1 else x * pow(x, 1))
=> x * (x * (if (1 == 0) 1 else x * pow(x, 0)))
=> x * (x * (x * 1))
```

- Result: all computation on static input (n) has been performed.

Partial evaluation: Regular expression matcher

- Compiling regular expressions

Example: Python's `re` library

`re.compile(`*`pattern, flags=0`*`)`

Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods, described below.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the flags variables, combined using bitwise OR (the `|` operator).

The sequence

```
prog = re.compile(pattern)
result = prog.match(string)
```

is equivalent to

```
result = re.match(pattern, string)
```

but using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

Partial evaluation: Regular expression matcher

- Defining the "AST" of regular expressions:

```
enum Regex:
  case Alter(p1: Regex, p2: Regex)   // p1 | p2
  case Concat(p1: Regex, p2: Regex)  // p1 p2
  case Star(p: Regex)                // p*
  case Char(c: Char)                 // single character
  case Empty
  case Fail

def regexMatch(pattern: Regex, str: String): Boolean = ...
```

Partial evaluation: Regular expression matcher

```
def regexMatch(pattern: Regex, str: String): Boolean =
  pattern match {
    case Alter(p1, p2) => regexMatch(p1, str) || regexMatch(p2, str)
    case Concat(p1, p2) => ...
    ...
  }

regexMatch(Concat(Char('a'), Char('b')), x)
```

Partial evaluation: Regular expression matcher

`regexMatch(Concat(Char('a'), Char('b')), x)` specializes/compiles to

```
def apply(x0: String): Boolean = {
  val x1 = x0.length
  val x5 = if (0 < x1) 'a' == x0(0) else false
  if (x5) {
    if (1 < x1) 'b' == x0(1) else false
  } else false
}
```

- Result: all pattern matching and recursion on static input (pattern) has been performed.
- How can we automate this process mechanically?

From interpreter to partial evaluator

Let's build a partial evaluator!

Core idea:

- Start from an interpreter of the target language.
- Allow undefined variables during evaluation.
- Preserve syntactical structures if they cannot be evaluated completely.

Tutorial: https://www.cs.utexas.edu/~wcook/tutorial/PEnotes.pdf

Language

- A simple untyped first-order functional language

```
enum Val:
  case Num(n: Int)
  case Bool(b: Boolean)
enum Expr:
  case Lit(v: Val)
  case Var(x: String)
  case Prim(op: String, e1: Expr, e2: Expr)
  case Cond(e1: Expr, e2: Expr, e3: Expr)
  case App(f: String, args: List[Expr])
case class FunDef(name: String, params: List[String], body: Expr)
```

Language

Example:

```
def pow(x: Int, n: Int): Int =
  if (n == 0) 1
  else x * pow(x, n - 1)
```

AST:

```
FunDef("pow", List("x", "n"),
  Cond(Prim("==", Var("n"), Lit(Num(0))),
    Lit(Num(1)),
    Prim("*", Var("x"),
      App("pow", List(Var("x"), Prim("-", Var("n"), Lit(Num(1))))))
)))
```

AST Interpreter

```
type Env = Map[String, Val]
def eval(e: Expr, env: Env, fenv: Map[String, FunDef]): Val =
  e match
    case Lit(v) => ...
    case Var(x) => ...
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) => ...
    case App(f, args) => ...
```

AST Interpreter

```
def eval(e: Expr, env: Env, fenv: Map[String, FunDef]): Val =
  e match
    case Lit(v) => v
    case Var(x) => env(x)
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) => ...
    case App(f, args) => ...
```

AST Interpreter

```
def eval(e: Expr, env: Env, fenv: Map[String, FunDef]): Val =
  e match
    case Lit(v) => v
    case Var(x) => env(x)
    case Prim(op, e1, e2) =>
      val v1 = eval(e1, env, fenv)
      val v2 = eval(e2, env, fenv)
      primEval(op, v1, v2)
    case Cond(e1, e2, e3) => ...
    case App(f, args) => ...
```

AST Interpreter

```
def primEval(op: String, v1: Val, v2: Val): Val =
  (op, v1, v2) match
    case ("+", Num(n1), Num(n2)) => Num(n1 + n2)
    case ("-", Num(n1), Num(n2)) => Num(n1 - n2)
    case ("*", Num(n1), Num(n2)) => Num(n1 * n2)
    case ("==", Num(n1), Num(n2)) => Bool(n1 == n2)
    case ("<", Num(n1), Num(n2)) => Bool(n1 < n2)
    ...
```

AST Interpreter

```
def eval(e: Expr, env: Env, fenv: Map[String, FunDef]): Val =
  e match
    case Lit(v) => v
    case Var(x) => env(x)
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) =>
      val v1 = eval(e1, env, fenv)
      v1 match
        case Bool(true) => eval(e2, env, fenv)
        case Bool(false) => eval(e3, env, fenv)
        case _ =>
          throw new Exception(s"Condition must be a boolean: $v1")
    case App(f, args) => ...
```

AST Interpreter

```
def eval(e: Expr, env: Env, fenv: Map[String, FunDef]): Val =
  e match
    case Lit(v) => v
    case Var(x) => env(x)
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) => ...
    case App(f, args) =>
      val FunDef(_, params, body) = fenv(f)
      val newEnv = params.zip(args.map(eval(_, env, fenv))).toMap
      eval(body, newEnv, fenv)
```

AST Interpreter

Example:

```
// fun max(a, b) = if (a < b) then b else a
val fenv = Map(
  "max" -> FunDef("max", List("a", "b"),
    Cond(Prim("<", Var("a"), Var("b")), Var("b"), Var("a"))
  )
)

// max(5, 2)
val expr = App("max", List(Lit(Num(5)), Lit(Num(2))))
val result = eval(expr, Map(), fenv) // result: Num(5)
```

Partial evaluator

- Same structure as interpreter, but returns residual AST instead of values.

```
type Env = Map[String, Expr] // Map from variables to ASTs

def pe(e: Expr, env: Env, fenv: Map[String, FunDef]): Expr =
  e match
    case Lit(v) => ...
    case Var(x) => ...
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) => ...
    case App(f, args) => ...
```

Partial evaluator

```
type Env = Map[String, Expr] // Map from variables to ASTs

def pe(e: Expr, env: Env, fenv: Map[String, FunDef]): Expr =
  e match
    case Lit(v) => Lit(v)
    case Var(x) => env.getOrElse(x, Var(x))
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) => ...
    case App(f, args) => ...
```

Partial evaluator

```
type Env = Map[String, Expr] // Map from variables to ASTs
def pe(e: Expr, env: Env, fenv: Map[String, FunDef]): Expr =
  e match
    case Lit(v) => Lit(v)
    case Var(x) => env.getOrElse(x, Var(x))
    case Prim(op, e1, e2) =>
      val pe1 = pe(e1, env, fenv)
      val pe2 = pe(e2, env, fenv)
      (pe1, pe2) match
        case (Lit(v1), Lit(v2)) => Lit(primEval(op, v1, v2))
        case _ => Prim(op, pe1, pe2)
    case Cond(e1, e2, e3) => ...
    case App(f, args) => ...
```

Partial evaluator

```
type Env = Map[String, Expr]

def pe(e: Expr, env: Env, fenv: Map[String, FunDef]): Expr =
  e match
    case Lit(v) => Lit(v)
    case Var(x) => env.getOrElse(x, Var(x))
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) =>
      val pe1 = pe(e1, env, fenv)
      pe1 match
        case Lit(Bool(true)) => pe(e2, env, fenv)
        case Lit(Bool(false)) => pe(e3, env, fenv)
        case _ => Cond(pe1, pe(e2, env, fenv), pe(e3, env, fenv))
    case App(f, args) => ...
```

Partial evaluator

```
type Env = Map[String, Expr]

def pe(e: Expr, env: Env, fenv: Map[String, FunDef]): Expr =
  e match
    case Lit(v) => Lit(v)
    case Var(x) => env.getOrElse(x, Var(x))
    case Prim(op, e1, e2) => ...
    case Cond(e1, e2, e3) => ...
    case App(f, args) =>
      val FunDef(_, params, body) = fenv(f)
      val newEnv = params.zip(args.map(pe(_, env, fenv))).toMap
      pe(body, newEnv, fenv)
```

Partial evaluator: example

```
// fun pow(x, n) = if (n == 0) then 1 else x * pow(x, n - 1)
val fenv = Map(
  "pow" -> FunDef("pow", List("x", "n"),
    Cond(Prim("==", Var("n"), Lit(Num(0))),
      Lit(Num(1)),
      Prim("*", Var("x"),
        App("pow", List(Var("x"), Prim("-", Var("n"), Lit(Num(1)))))))
    )))
// pow(x, 3)
val expr = App("pow", List(Var("x"), Lit(Num(3))))
val result = pe(expr, Map(), fenv)
//: Prim(*,Var(x),Prim(*,Var(x),Prim(*,Var(x),Lit(Num(1)))))
```

Partial evaluation

What it does:

- Constant propagation
- Constant folding (for primitive operations and conditionals)
- Function inlining

Partial evaluation

What we didn't cover:

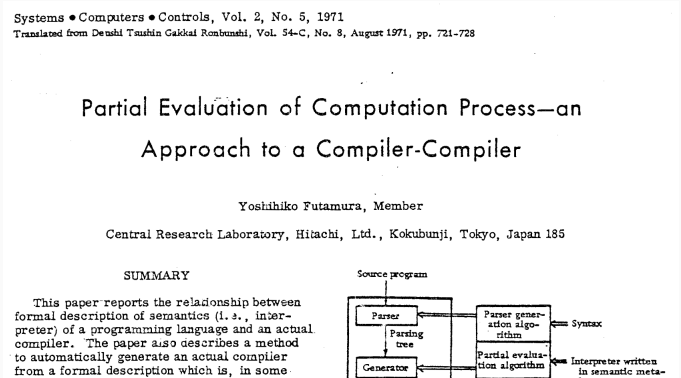
- Generating function definitions (in the Tutorial paper)
- Side effects
- Higher-order functions
- Termination of partial evaluation
- ...

Why care about partial evaluation?

- Writing an optimizing compiler is hard, but writing an interpreter is easier. Can we generate a compiler from an interpreter?

Why care about partial evaluation?

- Writing an optimizing compiler is hard, but writing an interpreter is easier. Can we generate a compiler from an interpreter?
- Idea: specializing self-applicable partial evaluator
- Futamura projections (Yoshihiko Futamura, 1971)



Systems • Computers • Controls, Vol. 2, No. 5, 1971
Translated from Denshi Tsushin Gakkai Ronbunshi, Vol. 54-C, No. 8, August 1971, pp. 721-728

Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler

Yoshihiko Futamura, Member

Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo, Japan 185

SUMMARY

This paper reports the relationship between formal description of semantics (i. e., interpreter) of a programming language and an actual compiler. The paper also describes a method to automatically generate an actual compiler from a formal description which is, in some

Futamura projections

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$

Futamura projections

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$
- Second projection (compiler generation): $[\![pe]\!](pe, eval) = compiler$
  - such that $[\![compiler]\!](srcPrg) = [\![pe]\!](eval, srcPrg) = tgtPrg$

Futamura projections

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$
- Second projection (compiler generation): $[\![pe]\!](pe, eval) = compiler$
  - such that $[\![compiler]\!](srcPrg) = [\![pe]\!](eval, srcPrg) = tgtPrg$
- Third projection (compiler-compiler): $[\![pe]\!](pe, pe) = cogen$
  - such that $[\![cogen]\!](eval) = [\![pe]\!](pe, eval) = compiler$

Futamura projections

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$
- Second projection (compiler generation): $[\![pe]\!](pe, eval) = compiler$
  - such that $[\![compiler]\!](srcPrg) = [\![pe]\!](eval, srcPrg) = tgtPrg$
- Third projection (compiler-compiler): $[\![pe]\!](pe, pe) = cogen$
  - such that $[\![cogen]\!](eval) = [\![pe]\!](pe, eval) = compiler$
- Fourth projection (self-generation): $[\![cogen]\!](pe) = cogen$

More applications

- Parser generator: given a grammar specification, generate a parser
- Matrix multiplication: one matrix is known at compile time
- Ray tracing: specialized to the scene
- Database query compiler: SQL query known at compile time
- GraalVM/Truffle (Oracle): partial evaluation framework for building compilers from interpreters
- ...

Various principled approaches to partial evaluation

- Offline partial evaluation:
  - conduct binding-time analysis before specialization
- Online partial evaluation (Tutorial):
  - conduct binding-time analysis during specialization

Various principled approaches to partial evaluation

- Offline partial evaluation:
  - conduct binding-time analysis before specialization
- Online partial evaluation (Tutorial):
  - conduct binding-time analysis during specialization
- Multi-stage programming:
  - Programmer fully controls binding-time annotations
  - E.g. LMS framework in Scala, MetaOCaml, Scala 3 staging, Template Haskell

Multi-stage programming

- Intuition:
  - Split program into multiple stages
  - The language provides constructs to build and manipulate code fragments
  - Programmers use annotations to indicate what to compute at compile time vs. run time
- Why multi-stage programming (MSP)?
  - PE is often a black box; automatic binding-time analysis is undecidable in general, not always precise
  - Runtime code generation improves reusability
  - Build (embedded) domain-specific languages
  - Type safety

Multi-stage programming

- Example: Scala 3
- `'{ e }` constructs a code fragment representing expression `e`
- `${ e }` splices a code fragment from computing `e` into another code fragment
- Type `Expr[T]` represents a code fragment that computes a value of type `T`

Staged power function:

```
def pow(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  if n == 0 then '{ 1.0 }
  else '{ $x * ${ pow(x, n-1) } }
```

Multi-stage programming

- Example: Scala 3

```
def pow(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  if n == 0 then '{ 1.0 }
  else '{ $x * ${ pow(x, n-1) } }
// runtime code generation
def specPow(n: Int): Double => Double =
  run {
    val stagedPow: Expr[Double => Double] =
      '{ (x: Double) => ${pow('x, n)} }
    stagedPow
  }
```

Multi-stage programming

A staged interpreter is a compiler (the first Futamura projection)!

```
def eval(t: Term, env: Expr[Env])(using Quotes): Expr[Value] = t match
  case Lit(i) => '{ IntV(${Expr(i)}) }
  case Var(x) => '{ ${env}(${Expr(x)}) }
  case Prim("+", e1, e2) => '{
    (${eval(e1, env)}, ${eval(e2, env)}) match
      case (IntV(v1), IntV(v2)) => IntV(v1 + v2)
    }
  case App(t1, t2) => '{
    (${eval(t1, env)}, ${eval(t2, env)}) match
      case (FunV(f), v) => f(v)
    }
  case Lam(x, e) => ...
```

Conclusion

- How to build an optimizer:
  - Constructing a partial evaluator from an interpreter
- Futamura projections: connecting compilation and interpretation
  - Compilation = partially evaluating an interpreter
  - Compiler generation = partially evaluating a partial evaluator with an interpreter
  - Compiler-compiler = partially evaluating a partial evaluator with itself
- As a language feature and/or programming paradigm
  - Multi-stage programming for manual partial evaluation