Partial Evaluation, Part 1
CS701

Thomas Reps

Abstract

This lecture concerns partial evaluation. Comparisons are made between partial evaluation and compiling. Two principles are given that help determine when to use partial evaluation. We show how a partial evaluator can be used to translate a program from one language L_2 into another language L_1 even if the partial evaluator's capabilities are limited so that its input is an L_1 program and its output is an L_1 program.

Contents

Note: quite a lot (but not all) of the material presented here is taken from the following book [**?**].

1 Introduction

When using a normal evaluator, one must supply all of the parameters. Otherwise, the output is undefined.

$$\begin{array}{rrcl} \text{program:} & P(x,y,z) & \longrightarrow & \boxed{\begin{array}{c}\text{Evaluator}\\ \text{(Interpreter)}\end{array}} \longrightarrow \bot \\ \text{data:} & \langle x,y\rangle & \longrightarrow & \end{array}$$

This outcome may be what we expected, but it can be improved. Using a partial evaluator, we can obtain a modified version of the input program that is optimized for the supplied parameters. Of course we expect the modified version of the input program to behave that same when given the rest of its input as the original version of the program when it is given the same input.

$$\begin{array}{rrcl} \text{program:} & P(x,y,z) & \longrightarrow & \boxed{\begin{array}{c}\text{Partial}\\ \text{Evaluation}\end{array}} \longrightarrow P_{\langle x,y\rangle}(z) = P(x,y,z) \\ \text{data:} & \langle x,y\rangle & \longrightarrow & \end{array}$$

The input program is called the *subject program*[1] and the output program in called the *residual program*.

1.1 Example: Ray Tracing

In ray tracing, there is a tree T that describes the object(s) in the scene. Given this tree, ray tracing code will determine how each ray contributes to the final image.

Algorithm 1: Ray Tracing Code
(1) **foreach** ray $\overrightarrow{r}$
(2) $\quad$ *pixel* $\leftarrow$ Trace(T, $\overrightarrow{r}$)
(3) $\quad$ display *pixel*

In other words, for many different rays, Trace is called with the same value of tree T. Using partial evaluation, we can create a version of Trace that is optimized for T.

$$\begin{array}{rcl} \text{Trace}(T,\overrightarrow{r}) & \longrightarrow & \boxed{\begin{array}{c}\text{Partial}\\ \text{Evaluation}\end{array}} \longrightarrow \text{Trace}_{\langle T\rangle}(\overrightarrow{r}) \\ \langle T\rangle & \longrightarrow & \end{array}$$

The residual program $\text{Trace}_{\langle T\rangle}(\overrightarrow{r})$ would be used as follows:

Algorithm 2: Partially Evaluated Ray Tracing Code
(1) **foreach** ray $\overrightarrow{r}$
(2) $\quad$ *pixel* $\leftarrow$ $\text{Trace}_{\langle T\rangle}(\overrightarrow{r})$
(3) $\quad$ display *pixel*

Other examples include

- Encryption: For many different blocks, `Encrypt(key, block)` is called with the same value of `key`.
- Output: For many strings `s`, `Write(fd, s)` is called with the same value of `fd`.
- String matching: For many substrings `s`, `Match(pat,s)` is called with the same value of `pat`.

[1] In logic and philosophy, this program would be called the *object program*. We use "subject program" because in the field of compilers the term "object program" is already used for something else.

2 Relationship between Partial Evaluation and Compilation

The notation $[\![p]\!]$ denotes the meaning of p, as distinct from uses of p, which indicate that p is being treated as a data object (i.e., the text, abstract syntax tree, or control-flow graph for p).

For a program p and input i, let $[\![p]\!][i]$ denote the result of running p on i.

2.1 Properties of a Partial Evaluator

Let p be a program that takes a pair of inputs $[s, d]$. A partial evaluator pe has the following property: for all programs p and inputs s, it produces

$$[\![pe]\!][p, s] = p_s, \text{ such that} \tag{1}$$
$$\text{for all } d, [\![p_s]\!][d] = [\![p]\!][s, d]. \tag{2}$$

The input variables s and d typically stand for *static* and *dynamic* (or *supplied* and *delayed*).

2.2 The Futamura Projections

Compilation/Translation An interpreter *int* is an example of a program that takes a pair of inputs (in this case, a program p and input-datum i) and has the same behavior as that program on its input:

$$[\![int]\!][p, i] = [\![p]\!][i]. \tag{3}$$

Suppose that we want to run the same interpreted program p on many (dynamic) inputs. This desire leads to the idea of applying a partial evaluator to an interpreter.

$$[\![pe]\!][int, p] = int_p, \text{ such that} \tag{4}$$
$$\text{for all } i, [\![int_p]\!][i] = [\![int]\!][p, i] \quad \text{(from Eqn. (2))}$$
$$= [\![p]\!][i] \quad \text{(from Eqn. (3))}$$

Notice that int_p and p take the same input and produce the same output. The explanation is that int_p is a "compiled" version of p.

Compiler/Translator Suppose that we want to partially evaluate the same interpreter on many (dynamic) programs. This desire leads to the idea of applying a partial evaluator to a second partial evaluator whose input is an interpreter.

$$[\![pe]\!][pe, int] = pe_{int}, \text{ such that} \tag{5}$$
$$\text{for all } p, [\![pe_{int}]\!][p] = [\![pe]\!][int, p] \quad \text{(from Eqn. (2))}$$
$$= int_p \quad \text{(from Eqn. (4))}$$

Note that pe_{int} takes in p and returns int_p, the compiled version of p. Therefore, pe_{int} is a compiler.

Compiler-Compilation/Translator-Generation Suppose that we want to partially evaluate the same partial evaluator on many (dynamic) interpreters. This desire leads to the idea of applying a partial evaluator to second partial evaluator that expects a partial evaluator as input.

$$[\![pe]\!][pe, pe] = pe_{pe}, \text{ such that} \tag{6}$$
$$\text{for all } int, [\![pe_{pe}]\!][int] = [\![pe]\!][pe, int] \quad \text{(from Eqn. (2))}$$
$$= pe_{int} \quad \text{(from Eqn. (5))}$$

Note that pe_{pe} takes in an interpreter and returns a (compiled) compiler. Therefore, pe_{pe} is a compiler generator—also known as a "compiler-compiler"—whose input specification of the compiler to be generated is an interpreter. What we mean by this last remark should become clearer in §3.

Futamura Projections Eqns. (4), (5), and (6) are known as the first, second, and third *Futamura projections*, respectively. Note the pattern of right-shifting in Eqn. (3) and the Futamura projections:

$$\begin{array}{l} [\![int]\!][p,\ \ i] \\ [\![pe]\!]\ [int,p] \\ [\![pe]\!]\ [pe,\ int] \\ [\![pe]\!]\ [pe,\ pe] \end{array}$$

3 Compiling (Translating) via Partial Evaluation

We now look at various possibilities for the programming languages that could be involved. We can write down more detailed versions of the Futamura projections by using subscripts and superscripts on *pe*, *int*, and p to indicate various languages with which these programs are associated.

Notation:

1. $[\![p_L]\!]_L$ means that the program p is an L-program that is interpreted as an L-program. (The outer L is typically redundant because it would not make sense to perform something like $[\![p_{L_1}]\!]_{L_2}$, and thus the outer L can be omitted.)
2. $int_{L_1}^{L_2}$ means that the interpreter *int* is an L_1-program that interprets L_2-programs. Thus, the fully annotated version of Eqn. (3) is
$$[\![int_{L_1}^{L_2}]\!]_{L_1}[p_{L_2}, i] = [\![p_{L_2}]\!]_{L_2}[i]. \tag{7}$$
3. $pe_{L_3}^{L_1}$ means that the partial evaluator *pe* is an L_3-program that partially evaluates an L_1-program (to produce another L_1-program).

Notice that interpreters and partial evaluators are special because they must indicate which language they interpret and partially evaluate respectively.

To be more general, we could have introduced notation like $pe_{L_3}^{L_1 \to L_2}$ to mean that *pe* is an L_3-program that partially evaluates L_1-programs to produce L_2-programs. However, that would mean that *pe* has language translation already "built into it," and as we now show, translation between languages can be achieved *without* the partial evaluator having the capability to perform language translation *per se*. The only feature that we rely on is that *pe* and *int* can be written in one language L, but operate on programs written in a different language L'.

Using the above-defined notation, we can rewrite the first Futamura projection (Eqn. (4)) as follows:

$$\begin{array}{r} [\![pe_{L_3}^{L_1}]\!][int_{L_1}^{L_2}, p_{L_2}] = (int_p)_{L_1}, \text{ such that} \\ \text{for all } d, [\![(int_p)_{L_1}]\!][d] = [\![p_{L_2}]\!][d]. \end{array}$$

For all inputs, the programs int_p and p produce the same output. However, $(int_p)_{L_1}$ is written in L_1, whereas p_{L_2} is written in L_2. Therefore, $(int_p)_{L_1}$ is a *compiled*—or at least *translated*—version of p_{L_2}.

Note that what captured our knowledge of how languages L_1 and L_2 are related is the interpreter $int_{L_1}^{L_2}$. In other words,

> *If we are given a partial evaluator that works on L_1 programs, to be able to translate an arbitrary L_2 program into L_1, all we need do is specify in L_1 how to interpret L_2 programs.*

4 Applications of Partial Evaluation

There are two guiding principles to follow when considering where to use partial evaluation. Partial evaluation is best used where (1) there are multiple parameters and (2) the parameters have different rates of variation.

4.1 Examples

4.1.1 String Matching

Given a two string s and p, we want to know if p is a substring of s. The string p is called the *pattern* and we say there is a *match* is p is a substring of s. We express this problem as Match(p, s). To use partial evaluation, we need to determine which argument changes more frequently. Initially, it seems that s changes more slowly than p.

Naive String Matching The naive algorithm just tries to match p to every possible location of s.

$$\begin{aligned} s =& s_1 s_2 s_3 \cdots s_m s_{m+1} s_{m+2} \cdots s_n \\ p =& p_1 p_2 p_3 \cdots p_m && \text{(first try)} \\ & p_1 p_2 p_3 \cdots p_m && \text{(second try)} \\ & p_1 p_2 p_3 \cdots p_m && \text{(third try)} \\ & \vdots \end{aligned}$$

Notice that p is just compared to the fist $|p|$ spots in s, and then, if there is no match, p is shifted and compared to the second through $|p| + 1$ spots. This continues until a match is found or all the (valid) locations of s have been tested. The code for the naive string-matching algorithm is:

Algorithm 3: Naive String Matching
(1) **for** $i = 1$ **to** $|s| - |p| + 1$
(2) **if** Match$(p, s_i \cdots s_{i+|p|-1})$
(3) **return** true
(4) **return** false

The cost of this approach is $O(n \cdot m)$. Notice that p is fixed throughout the loop, and hence looks like a good candidate for partial evaluation. The second argument of Match is clearly changing.

Knuth-Morris-Pratt (KMP) String-Matching Algorithm KMP takes advantage of redundancy by using memory to make larger shifts (rather than just shifts of size one as in the naive algorithm).

$$\begin{aligned} & 1234567 \\ s =& abbabab \\ p =& abbb && \text{(first try)} \end{aligned}$$

Observe that after this comparison, we know that a single shift will also fail (remembering that the second character was a b while the first character in p is an a. Thus, the second test of KMP will begin at s_4.

The cost of the KMP algorithm is $O(n + m)$. This same effect can be obtained via partial evaluation.[2]

4.1.2 Printf

The printf function takes a format string s and a variable number of arguments va and prints the format string in the context of the variable arguments. The format string can be thought of as a program. If $s = ab\%d$, then the corresponding program for printf(s, va) would be

[2]Consel and Danvy

Algorithm 4: Program of a Format String
(1) print a
(2) print b
(3) printSignedDecimalInteger $va[0]$.

Now in C/C++, the format string must be constant or static. Even in other languages though, programmers are most likely to have a static format string and a dynamic list of variable arguments. Using partial evaluation, we can convert printf(s, va) into printf$_s(va)$. It is also interesting to point out that printf$_s$ will also be partially evaluated to take an exact number of arguments instead of a variable number.

4.1.3 Cryptography

While the creation of a cryptographic scheme maybe quite complex, using a cryptographic scheme is quite easy. There are two functions, Encrypt$(key, plainText)$ and Decrypt$(key, crypterText)$ which work as expected. Normally though, the entire text is not encrypted or decrypted at once. Instead, the text in encrypted and decrypted a block at a time.

Algorithm 5: Encryption Code
(1) **foreach** $block \in plainText$
(2) add(Encrypt$(key, block), cypterText$)

Algorithm 6: Decryption Code
(1) **foreach** $block \in cypterText$
(2) add(Encrypt$(key, block), plainText$)

If the text is divided into many blocks, then this cryptographic scheme would benefit from partially evaluated functions of the form Encrypt$_{key}(block)$ and Decrypt$_{key}(block)$. It does not matter if the encryption and decryption keys are the same.

4.1.4 Cutting Through Abstraction Layers to Reduce Software Bloat

Abstraction is typically a good practice to follow. The benefit is that it reduces complexity because you can think of each layer of a system as an abstract machine whose "instruction set" is the layer's API. However, a system constructed with multiple layers of abstraction can be very slow because each operation first has to work its way down the chain of abstract machines—with each higher-level abstract machine calling down into one or more lower-level abstract machines.

For example, consider the Open Systems Interconnection (OSI) protocols used in networking. There are seven layers of abstraction in this model. For one application to communicate a message to another application requires the message to pass through 14 layers: seven on the way out for the sending application to send the message, and seven on the way in for the receiving application to receive the message. While the OSI model can handle many different communication scenarios, a single application that communicates with other instances of itself across the network will probably only use only one or two of these scenarios. Consequently, it might be possible to make a system more efficient if the implementation of the OSI protocols were partially evaluated with the application as static input. The dynamic input would be the messages sent or received.

4.2 A Drawback

Partial evaluation can decrease the time need to execute a function, but it does not always produce universally better code. Often the savings in time is paid for by an increase in program size. Consider the following piece of code:

Algorithm 7: Draw Back Example
(1) **for** $i = 1$ **to** 9
(2) **for** $j = 1$ **to** 9
(3) print f(i, j)

The function f has multiple parameters (two to be exact) and its first parameter i varies much slower than its second parameter j. Because of this property, it would seem like this piece of code is a good candidate for partially evaluation. The code that is produced by partial evaluation can be seen in algorithm Alg. 8.

Algorithm 8: Partially Evaluated Drawback Example
(1) **for** $j = 1$ **to** 9
(2) print $\mathrm{f}_1(j)$
(3) **for** $j = 1$ **to** 9
(4) print $\mathrm{f}_2(j)$
(5) **for** $j = 1$ **to** 9
(6) print $\mathrm{f}_3(j)$
(7) **for** $j = 1$ **to** 9
(8) print $\mathrm{f}_4(j)$
(9) **for** $j = 1$ **to** 9
(10) print $\mathrm{f}_5(j)$
(11) **for** $j = 1$ **to** 9
(12) print $\mathrm{f}_6(j)$
(13) **for** $j = 1$ **to** 9
(14) print $\mathrm{f}_7(j)$
(15) **for** $j = 1$ **to** 9
(16) print $\mathrm{f}_8(j)$
(17) **for** $j = 1$ **to** 9
(18) print $\mathrm{f}_9(j)$

Not only did the body of the original piece of code get much longer, but there are now nine specialized version of the function f. As one might guess, the increased performance is not always justified by the increase in code size.

Moreover, because of hardware caches, there can be a terrible effect on performance from having a larger piece of code, and thus the partially evaluated code can run slower than the original code.

5 A Simple Flow-Chart Language

We will start by learning about *intra*procedural partial evaluation—i.e., partial evaluation of single-procedure program. After that, we will look at *inter*procedural partial evaluation—i.e., partial evaluation across procedure boundaries. For intraprocedural partial evaluation, we will work with an imperative programming language—in particular, a simple while-loop language that works on

integers and lists. For interprocedural partial evaluation, we will work with a simple first-order functional programming language.

Meta-language	Subject language
Pidgin-Algol tables of cases informal graph diagrams	flow-chart language L, with L programs represented as S-expressions and expressions represented as abstract syntax trees

In language L we have the following constructs:

assignment
if cond then goto label else label'
goto label
read of initial data
print
Syntactic sugar (as needed):
begin ... end
while (...) do ... od
repeat ... until (...)

Data type	Operators
Booleans	&, |, $\sim$
integers	plus, $<$, $>$, $=$, ...
S-expressions	hd, tl, cons, nil, isNil

For convenience, we will use the following algebraic datatype (a.k.a. variant record) for representing expressions of L:

exp	::=	ConstExpr(constant)
	|	IdentExpr(identifier)
	|	Compound(operator exp exp)
constant	::=	true, false, $\ldots, -2, -1, 0, 1, 2, \ldots$
identifier	::=	[a-zA-Z]+
operator	::=	+ | − | * | cons | ...

The partial evaluator uses the following subroutine for simplifying L expressions:

```
simplify(e, store) =        // store is a map from names to values
  cases e of
    ConstExpr(*): e,        // nothing to simplify
    IdentExpr(i): definedIn(i, store) ? ConstExpr(lookup(i, store)) : e,
    Compound(op, e1, e2):
      let v1 = simplify(e1, store) and v2 = simplify(e2, store) in
        cases v1 of
          ConstExpr(c1):
            cases v2 of
              ConstExpr(c2): ConstExpr(funcOf(op)(c1,c2)),       // evaluate
              default: Compound(op, v1, v2)       // residuate simplified expression
          default: Compound(op, v1, v2)       // residuate simplified expression
```