Using Partial Evaluation to Generate Compilers

Niklas Deworetzki

December 31, 2020

1 Introduction

Partial evaluation is a technique to specialize programs or functions to their given input. This specialization appears to be similar to currying or partial application, that is known in programming. A partial evaluator however, does not simply fix one input parameter of a function, but propagates this fixing through the whole program text.

1.1 Interpreter

An interpreter can be used to evaluate computations from a source program. When executing an interpreter, the source program is given as an input to the interpreter. The interpreter will then analyze the source program and compute its output, given an input for the source program.

To fully evaluate the computations from the source programs, all of its inputs have to be known by the interpreter.

1.2 Compiler

Similarly to an interpreter, a compiler can be used to evaluate computations from a source program. To execute a compiler, the source program has to be given as an input to the compiler. The compiler will then analyze the source program and generate a target program which contains equivalent computations to the original source program.

The target program generated is usually in some kind of low-level language, that can be executed directly by a machine. In this way the target program can be executed, whereby its input has to be specified. The compiler itself does not need to know about the source or target programs inputs.

1.3 Partial Evaluator

A partial evaluator can be thought of as an hybrid between compiler and interpreter. Similarly to an interpreter, a partial evaluator will evaluate the given program text

Figure 1: Direct execution of a program

and will perform computations as they become available by fixing input parameters, just like an interpreter would do. On the other hand, it is not possible to perform all computations inside the evaluator, since the input is only partially fixed and some of it remains unknown. In these cases, a residual program is generated, representing the deferred computations and the partial evaluator acts more like a compiler.

From this mental image of an partial evaluator as some kind of mix between compiler and interpreter, two variants of partial evaluators emerged.

- **Offline** partial evaluation is guided by an analysis of the source code, where computations are either annotated as *static* or *dynamic*. During specialization the computations annotated as static are then performed, while the residual program generated holds all remaining computations marked as dynamic. [CN]
- **Online** partial evaluation on the other hand perform specialization on the fly. The partial evaluator steps through the source program and found static values are propagated through the program, while computations relying on dynamic input are generated as the residual program. [3, 1]

Contrary to the early theoretical roots of partial evaluation [CN], new concerns arose when implementing partial evaluators and using them for program specialization. While it is possible to construct a specialized program, that is slower than the original unspecialized one, it is of little relevance in the real world. Rather it is more attractive to use the additional information from specialization to speed up deferred computations. Used this way, partial evaluators can be used to optimize programs in a way that is similar to optimization techniques implemented in an optimizing compiler.

As we will see in later sections, a partial evaluator can not only be used to optimize programs like a compiler would do, but can even be used to generate a complete compiler.

2 Execution of Programs

Computational Problems can be solved with Programs. Programs are written in programming language. Can be executed to perform computations. Programs accept input and produce output; solution to problem Variable inputs creates complex program, allows multiple outputs, automatically solving.

1 shows schematic representation of execution of program. Notation is used during first chapter to separate data, programs and execution.

Computer must be able to execute language. Abstract programs, or algorithms don't have this problem, since human acts as computer. On real hardware, execution is limited. Machine instructions are efficient and executable. Are not easy to write, not portable, etc. Higher languages provided easier use, but cannot execute directly. It is possible to transform a program in language, to execute it.

Figure 2: An interpreter executing a program

Figure 3: A compiler generating an executable target program

Transformation can be done by a human, tedious as well. Execution or transformation of program is computational problem => Program can be used to automate.

Programs that accept programs are meta-programs. Technique called meta programming. Different kinds of these programs:

2.1 Interpreters

Interpreter can execute program, usually higher language. Accept program as input, analyze and produce intermediate representation: AST representing structure and content of program. AST has nodes, expressions, assignments, etc. Interpreter assigns meaning to these nodes Traversal of nodes, by executing meaning, equals execution of program.

Given interpreter can be executed, program can be executed too. Interpreter accepts source program and input to program as data. produces output. Interpreter executes program directly, like native program: single stage of computation

Advantage of interpreter: easy to write. Less performance, overhead of interpretation, shares resources.

2.2 Compilers

Compilers can execute programs too. Accept program as input, analyze and produce intermediate representation. Assigned meaning to nodes is not executed, is transformed into code.

Compiler accepts source program and produces target program. Target program can be executed directly, accepting input and producing output. As seen in 3 computation is split into two stages.

Visually seems like disadvantage, since both languages need execute (target and compiler) Praxis not a problem usually. Advantage is removal of overhead. Execution fast, especially if executed multiple times. Are harder to write, since programmer needs to think more (two languages, two execution stages).

2.3 Partial Evaluators

Partial evaluator generalizes concept of splitting computational stages. Accepts program as input as well as inputs for program itself. Performs computations that are available under partial input and deferres other computations.

On a high level working is similar to compiler and interpreter. Program as input, analyzed and intermediate representation. Instead of executing all, only parts are executed while the residue is generated. The generated program is called residue program.

Figure 4: A partial evaluator generating a residual program

From this scheme, partial evaluator is mix between interpreter and compiler. Sometimes called mix in literature.

While 4 is similar to 3 describing compiler, concept is generalized. Compiler splits overhead of analyzing and computation in different stages. Partial evaluator can split calculations depending on any input in different stages. A general program, accepting multiple inputs can be specialized to a fixed input. Allows higher performance, since decisions depending on input are removed.

Coming from initial description of programs. General problem solver can be fixed on problems. Other Examples are: Ray tracing, web servers, config files in general.

3 Partial Evaluation

Overview of inner workings of partial evaluators. Already covered: PE accepts a source program as input and additionally inputs for the source program itself. Source program is analyzed and structured, to allow for later transformation Section explains different variants of PEs and how they decide their "targets" (computable stuff) Afterwards different methods/tools/intruments of evaluations are explained.

3.1 Offline and Online Partial Evaluation

Two main variants on partial evaluators. Both act on the structured source program as input data. Both require some known input data for the source. Difference is in how computations are decided.

To decide what to compute: Static vs. Dynamic. Static means (similar to compiler), that it depends on fixed values[CN] Since PE knows input for program, it can find static computations All static computations can be performed by PE. Dynamic computations are generated for runtime.

The decision of static vs dynamic is called division. Every part of program has to recognized either static or dynamic. How this division is made is different in the two main variants [CN].

Offline makes decision before specialization in a preprocessing phase called binding-time analysis(BTA). Conservative as everything is treated dynamic until proven static. Annotates program [CN] without knowing concrete values of static input. Then transforms input.

Online makes decision during specialization with the value of a static input. Is more complex, since binding time analysis is not factored out as separate preprocessing step. Harder to predict speedup, difficult to self application or guarantee termination. Traverses source code deciding on the fly, if static or dynamic. Can use the actual values of input to perform decision, but may encounter same code multiple times.

3.2 Instruments of Partial Evaluation

Overview what techniques are available for PE to compute static. Mainly PE is based on the propagation of static values. Since it allows to unlock more computations.

Constant propagation (sparse constant propagation). constants (input) are known and computed. Expressions depending on constants can be computed too. Also conditions -> control flow is decided statically. Allows to cut off unused paths in code, perform many calculations.

```
x := 2, n := 8
def power(int n, int x) {
  int result = 1;
  while (n > 1) {
    result = result * x;
  }
}
```

Unfolding (of function calls) Allows to propagate constants into called procedures. Or unroll loops into a sequence of computations.

```
def power(int n, int x) {
  if (n == 0) {
    return 1;
  } else {
    return x * power(n - 1, x);
  }
}
```

symbolic computation Uses statically known structure of expressions to deduce simplifications. E.g mathematic simplicitations based on structure or mathematic identities.

```
2 << n
```

4 The Futamura-Projections

Until now, PE was only used as a way to specialize programs. As seen, specialized program does not have new properties. Only reduction of general programs, simplifying their structure, optimizing runtime by removing expressions.

Futamura proposed three projections, showing that PE can seemingly create new functionality. Programs with different properties appear through specialization. Clever use of interpreters and the way PEs work.

Firstly a new notation is used to simplify his findings. Notation shows evaluation as equations.

4.1 Notation of Multistage Computations

As seen before, computations can be split into multiple stages.

Program of language S can be executed directly, accepting input and generating output if S is executable. Interpreter can also be used to perform computations.

Another example: Compiler splitting two stages of computation. Two nested brackets in equation.

Partial Evaluator works similarly, introducing two nested brackets.

4.2 The First Futamura Projection

The first projection is based on a fact, visible before but not directly emphasized. During multistage computation and intro on execution we see, interpreters accept two inputs. First input: The executed program. Second input: The input to the executed program.

Usually two are passed in at once, PE can be used to create multiple stages of computation. Interpreter is specialized to program, input remains variable.

Resulting residual program accepts the input of the program and produces output of program corresponding to input. Residual program is compiled program. PE and interpreter acted like compiler.

4.3 The Second Futamura Projection

Looking at previous projection, it can be seen that a program is applied to two inputs at once. PE accepts interpreter and source program. The second projection deals with this case and shows what happens if input is separated.

PE is used to specialize PE on interpreter. Source program remains variable.

Residual program accepts a source program to create target program. Residual program acts as compiler. PE and interpreter acted like compiler compiler.

4.4 The Third Futamura Projection

Again, looking at previous projection, program is applied to two inputs at once. PE accepts PE and interpreter. The third projections deals with this case and shows what happens if input is separated.

PE is used to specialize PE on PE. Interpreter remains variable.

Residual program accepts an interpreter, to create a program. This program then accepts source to generate target. Residual program acts as compiler compiler. PE and PE acted like compiler compiler compiler, requiring only PEs. Interpreter determines how compiler acts.

4.5 Is there a Fourth Futamura Projection?

Equation now only consists of PEs. While additional is possible, it does not seem to change equation itself.

4.6 Going Further

No new functionality seems to be yield. Its still desirable to further specialize.

Multiple ways to create program are equal on a theoretical level, shown in equations. Practical differences in generated real code. As shown, performance gains.

5 Critical Assessment

PE seems like great technology.

Software development, specialization can yield good results. General programs, “do all” -> performant programs for specific use case.

Not widespread, bad tooling/complicated use. Surely could improve existing software. Even has some applications.

Use as compiler is nice in theory, not really in practice. Wont replace classical compiler construction. Compilers are not created “from nothing”, interpreters must exist and PEs too. Compiler constructors would be needed in those fields, developing PEs.

Even existing PE and interpreter wont completely replace CC. For optimizing compilers, as industry requires, clever people are needed. PEs can’t invent new data structures, can’t invent mathematics, simplifications or dirty tricks. PE only restructure input using known rules. People are needed to implement or invent those rules, people are needed to keep up to date, since technology is evolving.

6 Instruments of Partial Evaluation

In this section we will look into the inner workings of a partial evaluator. More concrete, this section shows the different methods that can be used by a partial evaluator to specialize programs and what effect they have on real-world examples.

The main goal of an partial evaluator is to evaluate all computations that can be evaluated statically given parts of the programs input. But how can the partial evaluator decide, which computations can be evaluated statically and for which a residual program has to be generated? Naturally some kind of analysis is performed on the source program, deciding what computations can be performed. In [2, S.3] three main techniques are presented, that can be used to specialize a program:

1. Symbolic computation
2. Unfolding of function calls and
3. program point specialization.

In the following, these three techniques are explained individually and examples are given for each one.

6.1 Unfolding of function calls

6.2 Symbolic Computation

The term “symbolic computation” is a broad term but can be described by arithmetic or algebraic simplifications in regards to partial evaluation.
[CN]

6.3 Program point specialization

7 Multistage Computations

When solving computational problems, the required calculations can be performed either in a single computational stage, or can be separated into multiple stages. This is especially obvious when working with compiled programming languages. Here the compiler itself provides an additional computing stage, performing compile-time computations and transforming the source language to an (executable) target language. Since multistage computations will be a recurring theme in the following sections, we now introduce a notation to reason about it in a formal way. This notation was adopted from .

The simplest form of computation is a single-stage computation. Here we have a source program written in a language S, that defines meaning and behavior of this program. Applying that program to an input, yields an output in one computation step. This is written as follows:

$$\texttt{output} = [\![\texttt{source}]\!]_{\texttt{S}}\ \texttt{input}$$

Since a programming language S usually cannot be executed directly by the underlying hardware, an interpreter can be used to execute the program. This interpreter then accepts the source program as well as the original input to produce an output in one computational stage.

$$\texttt{output} = [\![\texttt{interpreter}]\!]_{\texttt{L}}\ [\texttt{source}, \texttt{input}]$$

Next we reconsider the case described above, where a compiler is used to split the computation into two stages. Here the compiler accepts only a single input (the source program) and generates another program, which then accepts some input to produce the desired output.

$$\begin{aligned} \texttt{output} &= [\![[\![\texttt{compiler}]\!]_{\texttt{L}}\ \texttt{source}]\!]_{\texttt{T}}\ \texttt{input} \\ &= [\![\texttt{target}]\!]_{\texttt{T}}\ \texttt{input} \end{aligned}$$

In this equation, the nested brackets indicate that multiple computational stages are present, since the result of one computation is used to perform another.

8 Futamura Projections

As seen before in **??** a computation can be performed in different ways and multiple computational stages, for example by using an interpreter or compiler. Also visible, altough not specifically emphasized, was the fact that the interpreter used to execute the source program does accept two kinds of input, namely the source program and the problem input for the source program. One might wonder now what would happen if an partial evaluator is used to separate those two inputs.

In ?? Futamura ... presented three projections.

8.1 The first Futamura projection

The first of Futamura's projections deals with the case mentioned above, where an partial evaluator is used to specialize an interpreter to a source program. In this case the partial evaluator is used, to specialize an interpreter to a fixed source program. The resulting residual program now accepts an input to produce the desired output, as if the original source program would be executed.

$$\begin{aligned} \texttt{output} &= [\![\texttt{source}]\!]_{\texttt{S}}\ \texttt{input} \\ &= [\![\texttt{interpreter}]\!]_{\texttt{L}}\ [\texttt{source}, \texttt{input}] \\ &= [\![[\![\texttt{mix}]\!]\ [\texttt{interpreter}, \texttt{source}]]\!]\ \texttt{input} \\ &= [\![\texttt{target}]\!]\ \texttt{input} \end{aligned}$$

8.2 The second Futamura projection

Looking at the equations from 7.2 it is noticeable, that again some program is applied to two different inputs. This time it is the partial evaluator itself, that is applied to an interpreter and the source program. The second Futamura projection shows the effects of using a partial evaluator to specialize a partial evaluator to an interpreter. The resulting residual program from specialization of the partial evaluator to an interpreter is a program, that accepts a source program as an input and creates a target program as an output. Using a partial evaluator and an interpreter, it is possible to create a residual program acting like a compiler.

$$\begin{aligned}\texttt{target} &= [\![\texttt{mix}]\!]\ [\texttt{interpreter}, \texttt{source}] \\ &= [\![[\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{interpreter}]]\!]\ \texttt{source} \\ &= [\![\texttt{compiler}]\!]\ \texttt{source}\end{aligned}$$

8.3 The third Futamura projection

The second projection shows, how a partial evaluator and an interpreter can be combined to create an compiler. But again it is noticeable in the equations, that some program is applied to two different inputs. In this case, the partial evaluator is applied to itself and an interpreter to create the compiler. The third Futamura projection shows what happens, if a partial evaluator is used to specialize this application.

Since previously the partial evaluator accepted itself and an interpreter as input, we now use the partial evaluator to specialize itself to itself. The resulting program accepts an interpreter to create a compiler. Using nothing more than a partial evaluator and an interpreter, we created a compiler generator.

$$\begin{aligned}\texttt{compiler} &= [\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{interpreter}] \\ &= [\![[\![\texttt{mix}]\!]\ [\texttt{mix}, \texttt{mix}]]\!]\ \texttt{interpreter} \\ &= [\![\texttt{compiler-gen}]\!]\ \texttt{interpreter}\end{aligned}$$

Even though the equations still show, that some program is applied to two inputs, it becomes clear why further application of the previous scheme will not yield any new functionality. Using an partial evaluator to specialize the partial evaluator to one of its inputs, will yield the same equation as before, only introducing an additional stage of computation.

8.4 Advantages of Self-Application

While not yielding any new functionality it may still be desirable to further specialize the partial evaluator. As shown in [CN] it is possible to gain significant performance gains, when a specialized program is used for specialization instead of the general partial evaluator.

$$\begin{aligned}
\texttt{target} &= [\![\texttt{mix}]\!]\,[\texttt{interpreter}, \texttt{source}] = [\![\texttt{compiler}]\!]\,\texttt{source}\\
\texttt{compiler} &= [\![\texttt{mix}]\!]\,[\texttt{mix}, \texttt{interpreter}] = [\![\texttt{compiler-gen}]\!]\,\texttt{interpreter}\\
\texttt{compiler-gen} &= [\![\texttt{mix}]\!]\,[\texttt{mix}, \texttt{mix}] = [\![\texttt{compiler-gen}]\!]\,\texttt{mix}
\end{aligned}$$