Using Partial Evaluation to Generate Compilers

Niklas Deworetzki

January 6, 2021

1 Introduction

PE captures many forms of automated generation and analysis of programs. optimization, interpretation and automatic generation, as compilers or more general program generators.

Base use case is to specialize programs. Specialization similar to currying (Logic, FP) or projection (mathematics). Idea behind this concepts is to reduce a many argument function to single argument function, by fixing parameters. Specialization yields same result for programs, that accept many inputs. Some parts of input are known statically, specialization generates program requiring remaining inputs.

THeoretical base is s-n-m theorem[CN]. Important work computability, no attention to efficiency/speed. In computation/praxis, intuitively specialized programs can be faster. Aim for practical specialization.

1.1 Terminology

Program performing PE is called PE or specializer. Sometimes called Mix, since mixes computation (see chapter 2). Also mixes languages, implementation language, source language, target language, as seen later. Not explicitly mentioned, language is not relevant. Useful to keep in mind, that boundaries exist. Sometimes not clear with terminology, since PE specializes programs and not just simple functions. Since single functions can be turned into programs, terminology might change with context of explanation. Another point: Programs and computability are intertwined, we assume programs always halt for simplicity, unless noted. Must be considered, when implementing or real world application.

Figure 1: Direct execution of a program

1.2 Structure of this paper

2 Execution of Programs

In computer science, we use algorithms to describe, how computational problems can be solved. Building on decades of research, we can often determine which computational problems are solvable and can even categorize them depending on different strengths of the problem[CN]. While this is a great success for the field of computer science, the effect of algorithms on the real world is rather limited. Since algorithms describe abstract rules on how to structure and solve a program, a rather abstract computational framework is required to execute them. Consequently the "computer" executing an algorithm in its pure form, is usually a human being.

In order to use the capabilities of modern computers, which can perform billions of calculations per second and are therefore much better suited to automatically processing large data sets, algorithms need to be transformed in a more practical notation. The description of an algorithm in a concrete programming language is called program. Additionally to the description of an algorithm, a program written in a programming language also describes mechanisms to read an input (parameters to the computational problem) and produce an output (the solution to the problem). This way solving a computational is now a matter of executing the program.

Figure 1 shows a schematic representation of a program and its execution. As can be seen, a program may accept many inputs, allowing it to handle as many cases of the computational problem and producing (hopefully) correct solutions as an output. Naturally with the capability of reading, distinguishing and validating different inputs, the complexity of a program grows. But as figure 1 also shows, the execution of the program represents a single computational step regardless of the programs complexity or the required resources.

In order to execute a program, a computer must be able to understand the language it is written in. Machine instructions are the only language, a modern computer is truly able to execute. While these instructions are efficient and executable by the computer, they are not made for humans. Thus a program written directly in machine instructions is usually not easy to write, easy to maintain or easy to understand. Higher programming languages are available, which provide easier usage and a simpler description of programs. These higher languages however are not executable by the hardware directly and programs written in a higher language have to be transformed into machine instructions, to be executed.

Naturally this transformation can be done by a human being, although this task is very tedious. The transformation of programs is however a computational problem itself. Consequently programs can be used to automate this task. These programs, that accept programs as their input, are so-called meta programs. In this paper, three

Figure 2: An interpreter executing a program

different variants of meta-programs are relevant as they show different ways of how this transformation can be done and how the execution of a program can be achieved.

2.1 Interpreters

Interpreters are meta programs, that execute other programs. The input of an interpreter is a program, that is then analyzed and usually transformed into a intermediate representation. This intermediate representation—the abstract syntax tree—represents the structure and contents of the program as a tree of nodes. The nodes represent different expressions, statements and other language constructs present in the source program and their relation to each other forms a tree structure. By assigning a meaning to these nodes, a program becomes executable. The traversal of nodes, while performing actions according to the meaning of the traversed nodes, is equivalent to the execution of the program.

Given interpreter can be executed, program can be executed too. The way an interpreter accepts input and produces output is very similar to the native execution of an program as seen in the previous section. Additionally to the programs inputs, an interpreter accepts the program itself as input and executes it. The output produced by an interpreter is the output the program itself would produce, given the same input. Figure 2 shows that the execution via an interpreter is similar to the native execution of a program, in the chosen way to represent execution.

Atomic execution step, just accepting one more input.

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

2.2 Compilers

Compilers, similar to interpreters, are programs that are used to transform other programs. The way of working even is similar, as compilers also accept a program as input, analyze this input and usually transform it into an intermediate representation. The difference between a compiler and an interpreter is, that the meaning of the input program is not directly executed. Rather, a compiler translates the meaning of its input into another language and produces an output program with the same meaning as the input program, just written in another language.

This so-called target program is the single output of a compiler. A compiler does not accept any of the input intended for the source program. A compiler also does not produce any output, the source program would produce. As seen in figure 3, a compiler splits the execution into two computational stages this way, since the target program has to be executed, accepting input and producing output.

Visually in this figure, this seems like a disadvantage, since now multiple languages need to be executable in order to execute the source program. In practice this is usually

Figure 3: A compiler generating an executable target program

Figure 4: A partial evaluator generating a residual program

not a problem. More importantly, an advantage arises since the overhead of analyzing the input program is removed from the generated target program. If the target program can now be executed directly, the execution is usually efficient. This is especially true, if the target program is going to be executed multiple times, since the overhead of analyzing the source program is only done once during compilation, while an interpreter would need to analyze the program every time it is executed. On the other hand, compilers are harder to develop, since multiple languages and execution stages have to be taken into account.

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.

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

In this chapter, we dive deeper into the inner workings and implementation details of partial evaluators. The previous chapters already covered the operational mode of partial evaluators on a larger scale. Thus, we already know that a partial evaluator accepts a source program as well as some fixed inputs to the source program as its own input and produces a residual output consisting of computations that cannot be evaluated under the given static input. This chapter will explain, how a partial evaluator can decide, which computations can be performed statically and which have to be present in the residual program. Afterwards an overview of different methods are given, that

can be used to perform computations given some static input. These methods act as instruments for a partial evaluator, to optimize a program during specialization.

3.1 Offline and Online Partial Evaluation

Partial evaluators come in two main variants: *Offline* and *Online*. On a large scale, both variants act entirely the same. Both variants accept a source program as input and both require some known input data for this input program. The difference between those variants lies in how it is decided, which calculations will be performed by the partial evaluator and which have to be included in the residual program.

To decide whether an expression of the source program can be computed during specialization or has to be part of the residual, the partial evaluator has to decide whether said part relies only on static data or not. If all required values are given either by the source code itself or as part of the fixed input, the resulting expression is static. Static expressions can be computed by the evaluator. On the other hand, some parts of an expression may depend on input data, that is only known at runtime. These expressions are called dynamic, as their value may change and thus is not known during specialization. Such expressions have to be included in the residual program.

During specialization, must divide all parts of the input program either as static or dynamic. The way this division is made, is the key difference between both variants of partial evaluators [CN].

Offline Partial Evaluation depends on annotations in the source code to decide, whether an expression should be evaluated or generated as part of the residual program. These annotations can either be provided by the programmer or are inserted automatically during a preprocessing phase of specialization. During so-called binding time analysis, expressions are analyzed and annotations propagated. This way of dividing a source program is usually rather conservative, as annotations are made only knowing what inputs are fixed and not the values of fixed input [CN]. After the division is complete, all expressions annotated as static are evaluated, while those annotated as dynamic are generated as part of the residual program.

Online Partial Evaluation is more of an "on the fly"-approach to partial evaluation. The division of static and dynamic is made during specialization, when expressions are encountered in an online partial evaluator. This way, partial evaluation is more complex, since the overhead of creating a division is not factored out as a separate preprocessing step. Consequently it is harder to predict the speedup, online partial evaluation can achieve. During traversal of the source program, an online partial evaluator has information about the values of different inputs and expressions and thus can often perform better optimizations as an offline evaluator. On the other hand, a (recursive) expression could be encountered many times leading to many or infinite versions of an expression known to the evaluator. Not only does this run the risk of generating a large residual program, if parts of the expression are dynamic, but it can also lead to the partial

evaluator not terminating, if these cases are not accounted for.

3.2 Instruments of Partial Evaluation

In the following we will see different available techniques, that allow partial evaluators to compute static values or expressions containing (at least some) static values. Partial evaluation is mainly based on the propagation and removal of statically known data. Since initially only parts of the input of a source program are known as static data, this is the only way a partial evaluator is able to analyze and optimize an entire program. [CN] mentions three main techniques, which are discussed further next: symbolic computation, unfolding of function calls and program point specialization.

Symbolic Computation

Symbolic computation can be used as an optimization technique in an partial evaluator. The essence of symbolic computation as an optimization technique is to use the structure of an expression, to simplify and thus optimize it. Under this aspect we also consider actual performed computations as well as constant folding, constant propagation and their sub fields (e.g. sparse constant propagation, which is based on conditionals). Symbolic computation represents a powerful technique, since it is not only possible to evaluate completely constant expressions, but also simplify expressions that are based on constants.

As an example consider listing 1. If the input `x` of function `f` was fixed as `x = 1`, a partial evaluator could generate function `f1` as part of a residual program. The original function contains two multiplication operations and one additional addition as well as an division, while the specialized function contains only a single multiplication operation. All of this optimization is achieved by applying the aforementioned patterns:

1. Constantly replacing the variable `x` with the fixed value `1` yields the expression `y / 1 + 3 * 1 * y` as an intermediate result.
2. The constant subexpression `3 * 1` can be evaluated using constant folding, resulting in the new expression `y / 1 + 3 * y`.
3. The division `y / 1` can be simplified to `y` since `1` represents the neutral element of division, resulting in the expression `y + 3 * y`.
4. Finally symbolic computation allows us to simplify this expression into `4 * y`.

```
1  def f(x: Int, y: Int): Int =
2    y / x + 3 * x * y
3
4  def f1(y: Int): Int =
5    4 * y
```

Listing 1: Definition of a simple function and its specialization.

Unfolding Function Calls

The unfolding of function calls (also known as *inlining*) is a comparatively simple technique, that allows for optimizations to be applied beyond limits of a single function. When unfolding a function call, the body of the called function is simply inserted at call position renaming occurring variables according to the functions parameters.

Listing 2 shows an example of a function that is specialized and optimized using unfolding of its recursive calls. Fixing the input `n` with `n = 3`, it is possible to decide which of the branches is taken. In the resulting expression `3 * factorial(3 - 1)`, the recursive call can be unfolded. This results in the rather complicated expression of `3 * (if ((3 - 1) == 0) 1 else (3 - 1) * factorial((3 - 1) - 1))`, which contains another recursive call, that can be unfolded as well. Unfolding all recursive calls (and pruning the conditional) yields the specialized function `factorial3`, as it is shown in listing 2. Of course it is possible to further reduce the expression shown in the specialized function, since it only consists of static subexpressions.

```
1  def factorial(n: Int): Int =
2    if (n == 0) 1
3    else n * factorial(n - 1)
4
5  def factorial3(): Int =
6    3 * (3 - 1) * (3 - 2) * 1
```

Listing 2: Definition of the `factorial` function and its specialization.

Program Point Specialization

Program point specialization describes a technique to specialize functions as part of a larger program. Especially in larger programs, a function may be called multiple times with the same fixed partial inputs. Instead of generating a new function definition for each occurrence or unfolding every call, it may be preferred to extract a shared function definition to minimize code size. Listing 3 shows the specialization of Ackermann's function with the fixed input `n = 2` as an example, while also applying the previously mentioned optimization strategies.

```
1  def ack(n: Int, m: Int): Int =
2    if (m == 0) n + 1
3    else if (n == 0) a(m - 1, 1)
4    else a(m-1, a(m, n-1))
5
6  def ack2(n: Int): Int =
7    if (n == 0) 3
8    else a1(a2(n - 1))
9
10 def ack1(n: Int): Int =
11   if (n == 0) 2
12   else a1(n - 1)
```

Listing 3: Definition of the `ackermann` function and its specialization.

4 The Futamura-Projections

Until now, partial evaluation was only presented as a tool to specialize and at the same time optimize programs. The specialized programs shown did not have any new or interesting properties, except that they were faster than their general counterparts.

In [CN] Futamura showed, that it is not only possible to use partial evaluation for specialization of programs or to divide a computation into multiple stages. He proposed in total three projections, now known as the Three Futamura Projections, in which partial evaluators create programs with interesting and seemingly new properties. In the meantime, these projections have been confirmed multiple times by existing partial evaluators. This does not seem surprising, considering that only a partial evaluator and an interpreter is required for the three projections.

Next, before elaborating on the projections, a new notation is introduced. This notation is used to describe the behavior of programs and computations in multiple stages.

4.1 Notation of Multistage Computations

As seen before, computations can be split into multiple stages, performed by multiple programs, that accept multiple inputs (which themself can be other programs). While the notation from Chapter 2 might be useful as an visual introduction, it lacks the brevity for more advanced explanations. The following equation serves as an example to introduce the new notation:

$$\texttt{output} \;=\; [\![\texttt{p}]\!]\;(\texttt{input}_1,\, \texttt{input}_2,\, \texttt{input}_3)$$

It shows a simple program called `p`, that accepts three inputs named $\texttt{input}_1$ to $\texttt{input}_3$. The output of this program is aptly named `output` and (as well as the three inputs) may be data or an executable program. The actual execution of `p` is shown by the square brackets, that simultaneously separate different computational stages. This property becomes clear, when the operation of a compiler is described as an equation this way:

$$\begin{aligned}\texttt{output} &= [\![\texttt{target}]\!]_{\texttt{T}}\;(\texttt{input}) \\ &= [\![\texttt{source}]\!]_{\texttt{S}}\;(\texttt{input}) \\ &= [\![[\![\texttt{compiler}]\!]_{\texttt{I}}\;(\texttt{source})]\!]_{\texttt{T}}\;(\texttt{input})\end{aligned}$$

The output produced by executing a target program is the same as if a source program were to be executed directly. Also the same output is produced, by compiling the source program and executing the compilers output. Two things are important to note here:

Firstly, the brackets to describe evaluation can be annotated by a subscript, describing the language that is used to determine the meaning of a program. The compiler is executed according to the meaning of its implementation language `I`, while source is a program written in a source language `S`, that is translated into a target language `T` by the compiler. Additionally it is important to note, what equality means in this context. While the output produced by the right-hand sides of this equation is the same, the details of how this output is produced in each case can vary widely.

4.2 The First Futamura Projection

The first of Futamura's projections is based on a fact, that was already shown in this paper but was not directly emphasized. As we already know, an interpreter accepts two inputs: A program to execute and the input for this program itself. Usually these inputs are passed in at once, resulting in one computational stage. But a partial evaluator (`pe`) could be used to separate these inputs, as shown in the following equation.

$$
\begin{aligned}
\texttt{output} &= [\![\texttt{source}]\!]\ (\texttt{input}) && (1)\\
&= [\![\texttt{interpreter}]\!]\ (\texttt{source}, \texttt{input}) && (2)\\
&= [\![[\![\texttt{pe}]\!]\ (\texttt{interpreter}, \texttt{source})]\!]\ (\texttt{input}) && (3)\\
&= [\![\texttt{target}]\!]\ (\texttt{input}) && (4)
\end{aligned}
$$

It might not become obvious at first. But looking at equation (3) one may realize, that $[\![\texttt{pe}]\!]$ (`interpreter`, `source`) creates a program with the same meaning as `source` itself. The difference, however, is that while `source` is a program written in the source language, the residual program is written in the output language of the partial evaluator.

Notably the partial evaluator acted like a compiler and the act of specialization produced a compiled program.

4.3 The Second Futamura Projection

The second of Futamura's projections is based on the first. Looking at the second projection, it becomes clear that the same scheme can be applied again. The partial evaluator itself is a program that accepts two inputs: an interpreter and a source program.

So for the second projection, a partial evaluator is used, to create another stage of computation, abstract away the source as an input parameter. The following equation describes a partial evaluator that is used to specialize a partial evaluator in respect to an interpreter.

$$
\begin{aligned}
\texttt{target} &= [\![\texttt{pe}]\!]\ (\texttt{interpreter}, \texttt{source}) && (5)\\
&= [\![[\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{interpreter})]\!]\ (\texttt{source}) && (6)\\
&= [\![\texttt{compiler}]\!]\ (\texttt{source}) && (7)
\end{aligned}
$$

In this case, the residual program created in equation (6) is a program, that can transform a source program into a target program. Consequently, using the partial evaluator and an interpreter, it is possible to create a compiler, that can translate arbitrary other programs. Furthermore the partial evaluator acted like a compiler generator, creating a compiler from nothing more than a partial evaluator and an interpreter.

4.4 The Third Futamura Projection

The third of Futamura's projections uses the same scheme as the previous two. This time, it is the partial evaluator, specializing the partial evaluator in respect to an interpreter, that is the program accepting two inputs. Again it is possible to separate these two inputs by introducing another computational stage. The following equation describes a partial evaluator that is used to specialize a partial evaluator in respect to an partial evaluator.

$$\begin{aligned} \texttt{compiler} &= [\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{interpreter}) && (8)\\ &= [\![[\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{pe})]\!]\ (\texttt{interpreter}) && (9)\\ &= [\![\texttt{compiler-gen}]\!]\ (\texttt{interpreter}) && (10) \end{aligned}$$

The residual program created in the third Futamura Projection in equation (9) is a program that can generate a compiler. This program is a compiler generator, that accepts the description of a languages semantic to generate a standalone compiler. The description is passed as an interpreter, which decides the behavior of the generated compiler.

4.5 Is there a Fourth Futamura Projection?

The equations emerging from the third Futamura Projection still have the same structure as the previous equations, which would allow further to apply the previous abstraction scheme. It is notable, however, that further applications do not change the resulting equations. While this property may be called the Fourth Futamura projection, the equations themselves do not bear any new insights.

$$\begin{aligned} \texttt{compiler-gen} &= [\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{pe}) && (11)\\ &= [\![[\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{pe})]\!]\ (\texttt{pe}) && (12)\\ &= \ldots \end{aligned}$$

4.6 Going Further

While no new functionality or properties arise from further application of the above scheme, it may still be desirable to further apply partial evaluation. The key insight is, that there are multiple ways to create a program with some specific behavior. But while

these programs behave equally on a theoretical level as it is shown in equations, there are differences in practice. As it was shown with existing implementations, it is possible to gain significant speed-ups of a program, if a "specialized" compiler or compiler-generator was used.

Another interesting property, that becomes clear through Futamura's projections is the relationship between programs and their generating extension. A generating extension of a program p is a program $\texttt{pe}_p$ that accepts an input i to produce a version of p that is specialized in respect to i [CN]. A good example for a generating extension can be seen in the second Futamura Projection: The compiler ⟦pe⟧ (pe, interpreter) is a generating extension of the interpreter, since it accepts a source program and specializes the interpreter in respect to this source program. It turns out that compiler generators represent the generating extension of partial evaluators, which indicates a strong connection between them.

program	generating extension
⟦interpreter⟧ (source, input)	⟦⟦compiler⟧ (source)⟧ (input)
⟦parser⟧ (grammer, text)	⟦⟦parser-gen⟧ (grammar)⟧ (text)
⟦pe⟧ (interpreter, source)	⟦⟦compiler-gen⟧ (interpreter)⟧ (source)

Table 1: Programs and their generating extensions

5 Critical Assessment

PE is great technology to solve many problems.

Advantageous in Software Development. General programs can be fast, speed-up. Why not widespread, could tooling/compilcated use.

Not ripe in some areas. Online is fast, but has termination problems. Offline has less problems, but less speed up. Generally speed-up is hard to predict/dependent on input.

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. Target language is language from PE, for some tricks limited in language. 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.