Using Partial Evaluation to Generate Compilers

Niklas Deworetzki

December 1, 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 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. [2, 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 Instruments of Partial Evaluation

3 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.

4 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.

4.1 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.

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

4.2 The second Futamura projection

Looking at the equations from 4.2 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.

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

4.3 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.

4.4 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.

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