MS5001 – Masters Seminar

Subject:
Using Partial Evaluation to Generate Compilers

Submitted by:	Niklas Deworetzki niklas.deworetzki@mni.thm.de
Matriculation number:	5185551
Tutor:	Prof. Dr. Uwe Meyer
Date of Submission:	January 26, 2021

Abstract

Partial evaluation provides a unifying paradigm for program generation and program analysis. The key feature of partial evaluation is the specialization of programs; an optimization technique and execution strategy, that uses fixed inputs to a program to pre-compute parts of it and thereby increasing the program's performance.

This paper introduces the basic concepts of partial evaluation and classifies them in contrast to classical approaches of program execution and optimization. Core concepts of partial evaluation such as the difference between online and offline partial evaluation as well as basic specialization strategies are explained. As a key insight, the Futamura projections are presented and explained, providing a direct link between partial evaluation and compiler construction. Finally, a critical assessment of partial evaluation is provided, highlighting different use cases from compiler construction to software engineering, while also showing its boundaries.

1 Introduction

Partial evaluation provides a unifying paradigm for many forms of program generation and analysis. It offers solutions and insights to many areas of computer science, from program analysis, interpretation and compilation to the automated generation of programs.

The classical use case of partial evaluators is the partial evaluation (or specialization) of programs. Specialization of programs is similar to currying, as it is known from logic or functional programming, or the creation of projections in mathematics [5]. The idea behind these concepts is to reduce a function accepting many arguments to a function accepting only a single argument. Especially in partial evaluation, this can be achieved by fixing parameters. However, instead of functions, partial evaluation deals with programs, that process an arbitrary number of inputs.

The theoretical basis for partial evaluation lies in the S_n^m theorem [6]. In this theorem, Kleene showed that given a program accepting $m + n$ inputs and m values for those inputs, it is possible to construct a specialized program accepting only the n remaining inputs, where the first m have been fixed. While this work is important in the area of computability, the speed and efficiency of generated programs were of little importance to him. When aiming for practical specializations, the information about fixed inputs can be used to speed up computations, highlighting the connection with the optimization of programs.

1.1 Terminology

Before proceeding further into different areas of partial evaluation, it is important to establish some commonly used terminology. The arguably most important part of this topic is the partial evaluator itself, the program performing a partial evaluation. Since the act of partial evaluation is also referred to as the specialization of programs, a partial evaluator is also called a specializer. Another occasionally used name is Mix since it mixes different computations (see Chapter 2).

In the following chapters, the terminology used to describe partial evaluation may sometimes be imprecise, since strictly speaking partial evaluation deals with the specialization of programs and not single functions. As it is possible to convert a function into a program executing only this single function, this imprecision is accepted in the remainder of this work to promote readability and brevity.

In a similar sense, some details exist in the application and implementation of partial evaluators that are not discussed here, as they do not necessarily aid tangibility. Partial evaluators are limited to their implementation, source and target language, similar to compilers, which can not be used for any program and programming language. Additionally, in this work it is assumed, that partial evaluators and presented example programs always halt, as it aids simplicity. In a real-world application, this assumption may not always be correct and must be considered appropriately.

1.2 Structure of this paper

This work is separated into four main chapters, highlighting different topics on partial evaluation. First of all, in Chapter 2 different execution strategies are presented and partial evaluation is classified with respect to them. The second topic presented in Chapter 3 deals with the inner workings of partial evaluation. Here are different strategies presented, that are partial evaluator can use to specialize programs. Additionally, the differences between online and offline partial evaluation are explained. Chapter 4 presents the Futamura Projections, which are a central scheme of partial evaluation, showing relations to compiler construction and interpretation. Finally, the last Chapter 5 focuses on a critical assessment of the entire topic.

2 Execution of Programs

Usually, we think of the execution of programs as one single step. Either a program is executed, producing an output to its given inputs, or a program sits on a hard drive without being executed. This way, we don't have to worry about the exact details of how a program is executed and can focus on the computational problem a program tries to solve. However, in the following chapters, we will look into many ways to execute a program, all of which lead to the same output but are drastically different in their use cases, advantages and behaviors.

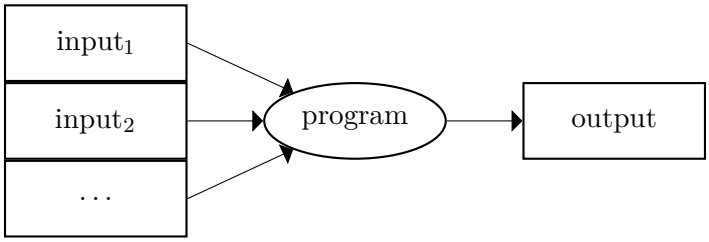

Figure 1: Direct execution of a program

Figure 1 schematically presents the way, we usually think of the execution of a program. A program (described by an oval shape) accepts some data (described by boxes) as an input and transforms this data into an output. While this diagram conceals details of how execution actually takes place, it gives an appropriate intuition about the general flow of data during execution. The conventional execution of a program takes place as a single computational step. Even if a program gets more complex, accepting many different inputs and producing output for each of these cases, it is only the run-time that grows, not the computational stages.

The figure, even if useful as an introduction, does not correctly represent reality in most cases. Usually, a program written by a programmer is not executable directly on a computer and the help of other programs is required to transform the raw source code into an executable format. These programs are so-called meta-programs, as the datasets

they work on are programs as well. In the context of this work, we will come into contact with three different meta-programs that are used for the execution of other programs, namely interpreters, compilers and partial evaluators. While interpreters and compilers are usually familiar, the concept of a partial evaluator often seems foreign. To put this concept in a better perspective, we will start with a brief description of interpreters and compilers before partial evaluators are finally introduced.

2.1 Interpreters

Interpreters, as already mentioned are meta-programs that are used to execute other programs. An interpreter accepts the program, which is to be executed, as an input and analyzes it, whereby it is usually transformed into an intermediate representation. This tree-like structure mirrors the structure and contents of the source program. By assigning meaning to every node of this tree, a program becomes executable, as the tree can be traversed with the interpreter performing actions corresponding to the meaning of each node.

Now, given that the interpreter is executable, all programs written in a language the interpreter can "understand" are executable too. As Figure 2 shows, the interpreted execution of a program looks very similar to the direct, native execution. The only difference is, that the program to be executed is now itself passed as an input to the interpreter along with its inputs. Execution this way is still a single computational stage.

Interpreters provide a simple solution to execute programs in another language and are relatively easy to implement. On the downside, execution via an interpreter provides less performance, as interpreter and interpreted program share the same resources and some computational overhead is required for analysis of said program.

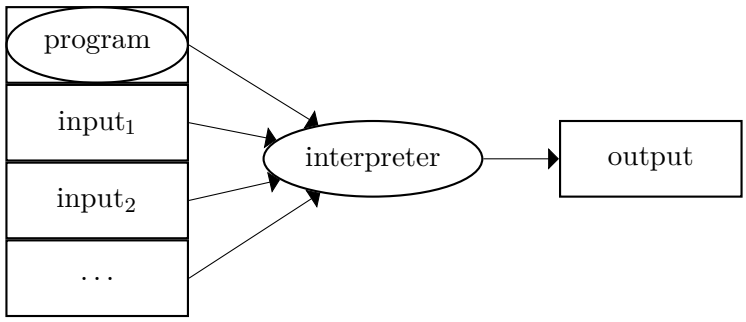

Figure 2: An interpreter executing a program

2.2 Compilers

Compilers, similar to interpreters, are programs that are used to transform other programs. While an interpreter uses the intermediate representation of a program to directly executed the assigned meaning to it, a compiler translates the meaning into another language. As a result, a compiler produces a single output, which is the so-called target

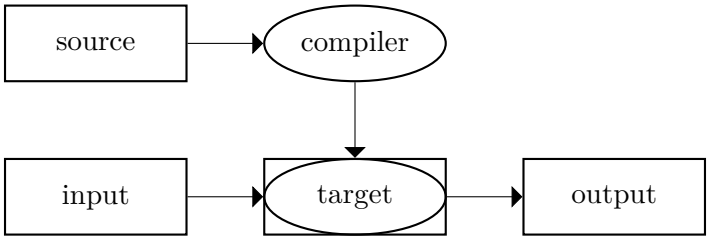

Figure 3: A compiler generating an executable target program

program. This program holds the meaning of the compiler's input – the source program – translated into another language. As seen in Figure 3, an additional stage of computation is introduced this way. The target program has to be executed for it to accept inputs and produce an output like the original source program.

As already apparent visually, the translation process of a compiler is more complex than the process of an interpreter. But since the input program has to be only analyzed once and the target program is executed as a standalone program, compilation usually provides better performance in contrast to interpretation. This is especially true if the same program is executed multiple times since no computational overhead is present during the execution of the target program after it has been created once.

2.3 Partial Evaluators

Partial evaluators, as the third meta-program introduced and focus of this paper, generalize the previously mentioned concept of creating additional computational stages. The inputs of a partial evaluator are a source program as well as *some* inputs for this program. Given these inputs, a partial evaluator will perform all computations in the source program that are available under the given inputs. As not all inputs for the source program are present, some computations cannot be performed. These remaining computations form a so-called residual program, the output of a partial evaluator.

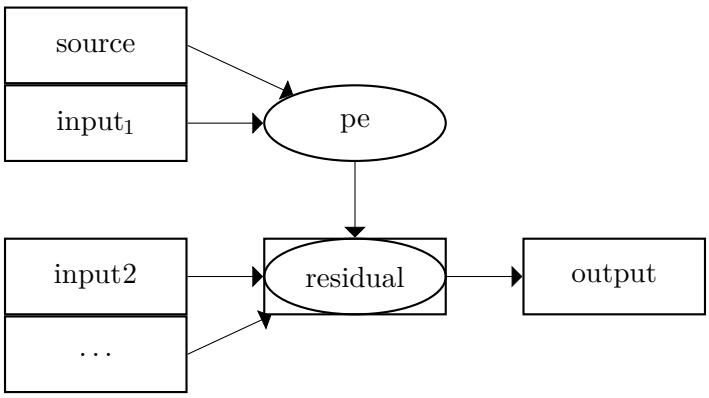

Figure 4: A partial evaluator generating a residual program

While Figure 4, describing a partial evaluator, is structurally similar to Figure 3 describing a compiler, the concept is generalized. A compiler only splits the overhead of analysis and the actual computation into different computational stages. A partial evaluator can split calculations depending on different inputs into different computational stages. This way, a general program accepting multiple inputs can be specialized to a fixed input, which usually leads to higher performance, since the overhead of recognizing inputs is removed.

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.

During specialization a partial evaluator will analyze the components of a program, that represent computations. While the specific components are dependent on the programming language used for the input program, a partial evaluator will usually work on expressions and statements, as well as the definitions accompanying them. Consequently a partial evaluator manages information about known names (such as the fixed inputs) and transforms the present components during specialization. This way, redundant computations are removed or the structure of statements or expressions is simplified using the known information.

This chapter will give an overview of different methods, 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. Afterwards it is explained, how a partial evaluator can decide, which computations can be performed statically and which have to be present in the residual program.

3.1 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. [5, Chap. 1] 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 a 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 performed computations as well as constant folding, constant propagation and their subfields (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 a specialized function `f1` as part of a residual program. The original function contains two multiplications, one addition as well as one division and one subtraction, while the specialized function contains only a single multiplication operation. This optimization is achieved by applying the aforementioned patterns:

1. Replacing the variable `x` with the fixed value of `1` yields the following expression as an intermediate result: `y / (1 - 2) + 3 * 1 * y`.
2. The constant subexpressions `3 * 1` and `1 - 2` 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 `2 * y`.

```
1  def f(x: Int, y: Int): Int =
2    y / (x - 2) + 3 * x * y
3
4  def f1(y: Int): Int =
5    2 * 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 function's parameters.

Listing 2 shows an example of a function that is specialized and optimized by unfolding its recursive calls. Unfolding introduces the whole expression inside the function's body into the initial expression, where the parameter is consistently renamed with `(n - 1)`. This introduces another recursive call with parameter `(n - 2)`, which unfolded introduces another recursive call, etc.

If applied without care, this scheme can result in infinite loops. While in this example it is possible to terminate the recursive unfolding if the condition within the recursive expression is evaluated, more complex functions are not guaranteed to terminate. This

example also shows, that optimization schemes can be combined to yield even more optimized results, as the resulting expression could be reduced even further using constant folding.

```
def factorial(n: Int): Int =
  if (n == 0) 1
  else n * factorial(n - 1)

def factorial3(): Int =
  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.

```
def ack(n: Int, m: Int): Int =
  if (m == 0) n + 1
  else if (n == 0) a(m - 1, 1)
  else a(m-1, a(m, n-1))

def ack2(n: Int): Int =
  if (n == 0) 3
  else a1(a2(n - 1))

def ack1(n: Int): Int =
  if (n == 0) 2
  else a1(n - 1)
```

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

3.2 Offline and Online Partial Evaluation

As it was shown, a partial evaluator has to differentiate between static and dynamic parts of a program. If some expression or statement only depends on static data, the computation represented by these elements is static too and can be performed during evaluation. On the other hand it is possible, that some parts may depend on input data that is only known at runtime. In this case, it specialization is not always possible and the computations have to be included in the residual program.

To properly perform the specialization, a partial evaluator must divide all parts of the input program into either static or dynamic computations. The way this division is made is the key difference between the two main variants of partial evaluators. This variants differentiate between *online* and *offline* partial evaluation, each of which has its own advantages and disadvantages.

Offline Partial Evaluation depends on annotations in the source code in order to decide, whether a computation can be performed during specialization or has to be 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, the program is analyzed and annotations are propagated. This way of dividing a source program is usually rather conservative. Annotations are only based on the knowledge of what input variables are fixed and without knowledge of the inputs values [5, Chap. 7]. After the division is complete, all computations annotated as static are performed, 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 between static and dynamic is made during specialization when expressions or statements 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 [1]. 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, variables and expressions and thus can often perform more optimizations compared to an offline evaluator. On the other hand, a (recursive) definition could be encountered many times leading to many or potentially infinite versions of it known to the evaluator. Not only does this run the risk of generating a huge residual program, but it can also lead to the partial evaluator not terminating if these cases are not accounted for.

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 [2] Futamura showed, that it is not only possible to use partial evaluation for the 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.

Before elaborating on the projections, a new notation is introduced first. 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 a visual introduction, it lacks the brevity for more advanced explanations. The following equation serves as an example to introduce the new notation:

$$\texttt{output} = [\![\texttt{p}]\!]\ (\texttt{input}_1, \texttt{input}_2, \texttt{input}_3) \tag{1}$$

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:

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

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 compiler's output. 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.

$$\begin{aligned}
\texttt{output} &= [\![\texttt{source}]\!]\ (\texttt{input}) && (5)\\
&= [\![\texttt{interpreter}]\!]\ (\texttt{source}, \texttt{input}) && (6)\\
&= [\![[\![\texttt{pe}]\!]\ (\texttt{interpreter}, \texttt{source})]\!]\ (\texttt{input}) && (7)\\
&= [\![\texttt{target}]\!]\ (\texttt{input}) && (8)
\end{aligned}$$

It might not become obvious at first. But looking at equation (7) 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 with respect to an interpreter.

$$\texttt{target} = [\![\texttt{pe}]\!]\ (\texttt{interpreter}, \texttt{source}) \tag{9}$$

$$= [\![[\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{interpreter})]\!]\ (\texttt{source}) \tag{10}$$

$$= [\![\texttt{compiler}]\!]\ (\texttt{source}) \tag{11}$$

In this case, the residual program created in equation (10) 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 with 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 with respect to a partial evaluator.

$$\texttt{compiler} = [\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{interpreter}) \tag{12}$$

$$= [\![[\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{pe})]\!]\ (\texttt{interpreter}) \tag{13}$$

$$= [\![\texttt{compiler-gen}]\!]\ (\texttt{interpreter}) \tag{14}$$

The residual program created in the third Futamura Projection in equation (13) is a program that can generate a compiler. This program is a compiler generator, that accepts the description of a language's 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. The following equations show further applications of this scheme, while the partial evaluator introduced in the third projections is highlighted to aid readability.

$$\texttt{compiler-gen} = [\![\underline{\texttt{pe}}]\!]\ (\texttt{pe}, \texttt{pe}) \tag{15}$$

$$= [\![[\![\texttt{pe}]\!]\ (\underline{\texttt{pe}}, \texttt{pe})]\!]\ (\texttt{pe}) \tag{16}$$

$$= [\![[\![[\![\texttt{pe}]\!]\ (\texttt{pe}, \underline{\texttt{pe}})]\!]\ (\texttt{pe})]\!]\ (\texttt{pe}) \tag{17}$$

$$= \ldots$$

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 [3]. 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 with respect to i. A good example for a generating extension can be seen in the second Futamura Projection: The compiler $[\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{interpreter})$ is a generating extension of the interpreter since it accepts a source program and specializes the interpreter with 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
$[\![\texttt{interpreter}]\!]\ (\texttt{source}, \texttt{input})$	$[\![[\![\texttt{compiler}]\!]\ (\texttt{source})]\!]\ (\texttt{input})$
$[\![\texttt{parser}]\!]\ (\texttt{grammer}, \texttt{text})$	$[\![[\![\texttt{parser-gen}]\!]\ (\texttt{grammar})]\!]\ (\texttt{text})$
$[\![\texttt{pe}]\!]\ (\texttt{interpreter}, \texttt{source})$	$[\![[\![\texttt{compiler-gen}]\!]\ (\texttt{interpreter})]\!]\ (\texttt{source})$

Table 1: Programs and their generating extensions

5 Critical Assessment

As demonstrated in the previous chapters, partial evaluation as a technology has applications in many fields of programming and computer science. The use of partial evaluation in software development would allow developers to create a single general program, that could then be specialized to a fixed problem domain. This way, development costs could be reduced as it is only necessary to develop and maintain a single application, or existing programs depending on a fixed configuration could be optimized. Often cited examples for the second case are web servers, which are dependent on a configuration file or the specialization of a renderer to a fixed scene [5].

Nevertheless, partial evaluation is not widespread and its use is often limited to research areas. The focus of existing specializers is often the exploration of new strategies and possibilities rather than user experience [1]. Additionally, partial evaluation does not provide a one-size-fits-all solution, as the two common approaches online and offline partial evaluation both come along with advantages and disadvantages. Online partial evaluation generally has the capability to produce more optimized residual programs. It is, however, not always applicable, as the specialization will not always halt if its inputs are not restricted. Offline partial evaluation, on the other hand, makes it easier to ensure termination, as the division is computed during the separate binding time analysis and not "on the fly". At the same time it generally yields less optimized residual programs. The gains of optimization are with both cases hard to predict and largely dependent on the concrete input [7].

It is unlikely, that the use of partial evaluators and interpreters as an alternative to traditional compiler construction will be widespread. While the creation of a compiler from a partial evaluator and an interpreter looks easy in theory, in practice the effort of constructing a partial evaluator and an interpreter is not substantially smaller than the effort of constructing a single compiler. Additionally, the right combination of implementation, input and output languages is required to make this way of compilation useful, as normally compilers are used to generate low-level machine code.

Even if this way of constructing compilers would become widespread, classical compiler construction would still be relevant. All optimizations a partial evaluator and a compiler created from it can use, have to be implemented at first. Efficient data structures, mathematical simplifications and clever use of available hardware are not created "from nothing". A partial evaluator will only use existing structures within itself, an interpreter or the source program and thus, human ingenuity is required to create truly efficient programs.

References

[1] W. Cook and R. Lämmel. "Tutorial on Online Partial Evaluation". In: *Electronic Proceedings in Theoretical Computer Science* 66 (Sept. 2011), pp. 168–180.

[2] Y. Futamura. "Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler". In: *Higher-Order and Symbolic Computation* 12.4 (1999), pp. 381–391.

[3] R. Glück. "Is There a Fourth Futamura Projection?" In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. Savannah, GA, USA, 2009, pp. 51–60.

[4] N. Jones. "An Introduction to Partial Evaluation". In: *ACM Computing Surveys* 28.3 (1996), pp. 480–503.

[5] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Upper Saddle River, NJ, USA: Prentice Hall, 1993. ISBN: 0130202495.

[6] S. Kleene. "General Recursive Functions of Natural Numbers". In: *Mathematische Annalen* 112.1 (Dec. 1936), pp. 727–742. DOI: 10.1007/bf01565439.

[7] E. Sumii and N. Kobayashi. "A Hybrid Approach to Online and Offline Partial Evaluation". In: *Higher-Order and Symbolic Computation* 14.2 (2001), pp. 101–142.