TECHNISCHE HOCHSCHULE MITTELHESSEN

CAMPUS GIESSEN

MNI Mathematik, Naturwissenschaften und Informatik

MS5001 – Masters Seminar

Subject:

Using Partial Evaluation to Generate Compilers

Submitted by:	Niklas Deworetzki niklas.deworetzki@mni.thm.de
Matriculation number:	1234567
Tutor:	Prof. Dr. Uwe Meyer
Date of Submission:	January 12, 2021

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 [CN]. 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 to partial evaluation lies in the $\mathrm{S}_{\mathrm{n}}^{\mathrm{m}}$ theorem [CN]. 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, 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 on this topic is the partial evaluator itself, the program performing partial evaluation. Since the act of partial evaluation is also referred to as the specialization of programs, a partial evaluator is also called 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 tangability. 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 in Chapter 2 different execution strategies are presented and a classification of partial evaluation is given in respect to them. The second topic presented in Chapter 3 deals with inner workings of partial evaluation. Here are different strategies presented, that are partial evaluator can use to specialize programs. Additionally the difference 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 Chapters 5 and **??** focus on a critical assessment of the entire topic.

2 Execution of Programs

Programs are implementations of algorithms, which are used to solve computational problems. In contrast to algorithms, programs handle details as acquiring inputs, producing outputs, structuring memory and other tasks, that are required in order to be executed by a machine.

Figure 1 shows a schematic representation of how a program is used to solve a computational problem. The program accepts inputs and produces an output, both of which are represented by oval shapes, with arrows describing their relationship with the program. As it can be seen, a single program may accept many inputs, allowing it to handle as many cases of the computational problem and producing an output in each of these cases. 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 occurs as a single computational stage, regardless of the required resources.

In reality, however, it is often not possible to directly execute a program, as the language it is written is can not be executed by a computer. Computers are limited in that they can only execute instructions, that are described in a low-level machine language. Programs on the other hand are usually written in a higher-level language, since the abstractions provided by this language allows a better description of the required instructions. In order to execute such a program, written in a higher language, a translation has to be performed. The tedious task of translating programs can be performed manually. But it turns out, that translating a program is also a computational problem, that can automatically be solved by a program. These programs, that accept other programs as an input in order to solve computational problems, are called meta-programs. In this paper, three 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. Starting with interpreters and compilers, a fundamental understanding on the execution of programs is provided, before partial evaluators are introduced.

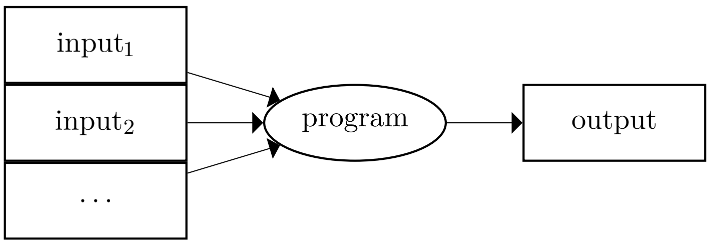

Figure 1: Direct execution of a program

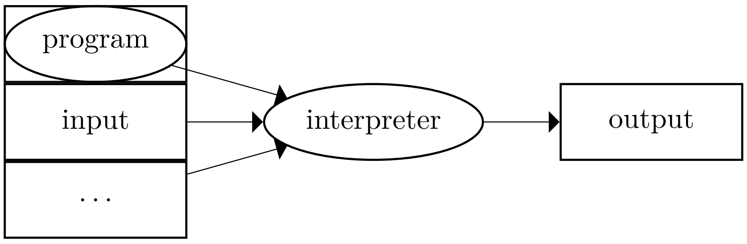

Figure 2: An interpreter executing a program

2.1 Interpreters

Interpreters are meta programs that execute other programs. The program to be executed is passed as an input to the interpreter, which is then analyzed and usually transformed into an intermediate representation. This tree-like structure mirrors the structure and contents of the program. By assigning a meaning to every node of this tree, a program becomes executable by traversing the tree and performing actions according to the assigned meaning. Now, given that the interpreter can be executed, programs written in a language, the interpreter can accept as an input, can be executed too. As Figure 2 shows, the interpreted execution of a program looks very similar as the direct, native execution. The only difference is, that the program to be executed is now itself along with its inputs passed as an input to the interpreter. 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.

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 it into another language. As a result, a compiler produces a single output, which is the so-called target program. This program holds the meaning of the input program translated into another language. As

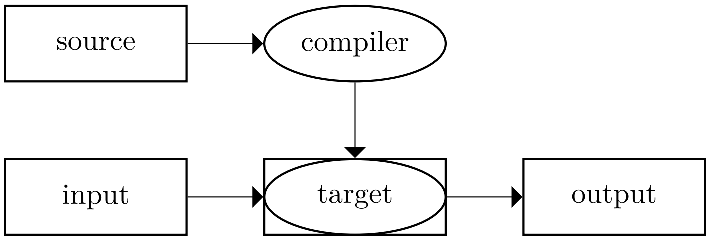

Figure 3: A compiler generating an executable target program

seen in Figure 3, this way an additional stage of computation is introduced, since the target program has to be executed, to accept inputs and produce an output like the original program.

As already apparent visually, the translation process of a compiler is more complex than the process of an interpreter. Since the input program has to be only analyzed once and the target program is executed as a standalone program, compilation usually provides a 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 is has been created once.

2.3 Partial Evaluators

Partial evaluators, as the third meta program introduced and focus of the following paper, generalize the concept of splitting computational stages that has been introduced by compilers. 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 of the source that are available under the given inputs. As not all inputs for the source program are present, some computations cannot be performed. These computations form a so-called residual program, the output of a partial evaluator.

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

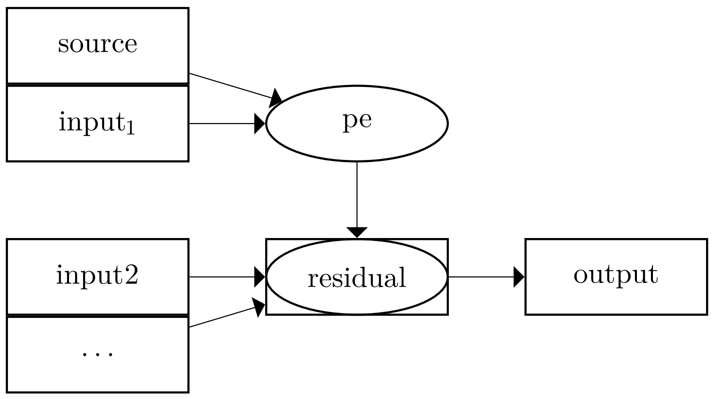

Figure 4: A partial evaluator generating a residual program

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`.

```
def f(x: Int, y: Int): Int =
  y / x + 3 * x * y

def f1(y: Int): Int =
  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.

```
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.

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.

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

It might not become obvious at first. But looking at equation (3) one may realize, that $[\![\texttt{pe}]\!] \; (\texttt{interpreter}, \texttt{source})$ 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.

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

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.

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

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.

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

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 $[\![\texttt{pe}]\!]\ (\texttt{pe}, \texttt{interpreter})$ 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
$[\![\texttt{interpreter}]\!]\ (\texttt{source}, \texttt{input})$	$[\![[\![\texttt{compiler}]\!]\ (\texttt{source})]\!]\ (\texttt{input})$
$[\![\texttt{parser}]\!]\ (\texttt{grammer}, \texttt{text})$	$[\![[\![\texttt{parser-gen}]\!]\ (\texttt{grammar})]\!]\ (\texttt{text})$
$[\![\texttt{pe}]\!]\ (\texttt{interpreter}, \texttt{source})$	$[\![[\![\texttt{compiler-gen}]\!]\ (\texttt{interpreter})]\!]\ (\texttt{source})$

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, that are dependend on a configuration file or the specialization of a renderer to a fixed scene [CN].

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 [CN]. 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, has no problems with termination, but generally yields less optimized residual programs. The gains of optimization are with both cases hard to predict and largely dependent on the concrete input [CN].

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.