Using Partial Evaluation to Generate Compilers

Niklas Deworetzki

January 3, 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

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

3.1 Offline and Online Partial Evaluation

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

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

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

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

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

3.2 Instruments of Partial Evaluation

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

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

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

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

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

```
    }
  }
```

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

```
2 << n
```

4 The Futamura-Projections

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

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

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

4.1 Notation of Multistage Computations

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

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

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

Partial Evaluator works similarly, introducing two nested brackets.

4.2 The First Futamura Projection

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

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

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

4.3 The Second Futamura Projection

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

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

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

4.4 The Third Futamura Projection

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

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

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

4.5 Is there a Fourth Futamura Projection?

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

4.6 Going Further

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

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

5 Critical Assessment

PE 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.