Practical Second Futamura Projection[*]

Partial Evaluation for High-Performance Language Interpreters

Florian Latifi
Johannes Kepler University
Linz, Austria
florian.latifi@jku.at

Abstract

Partial evaluation, based on the first Futamura projection, allows compiling language interpreters with given user programs to efficient target programs. *GraalVM* is an example system that implements this mechanism. It combines partial evaluation with profiling information and dynamic compilation, to transform interpreters into high-performance machine code at run time. However, partial evaluation is compile-time intensive, as it requires the abstract interpretation of interpreter implementations. Thus, optimizing partial evaluation is still subject to research to this day. We present an approach to speed up partial evaluation, by generating source code ahead of time, which performs partial evaluation specific to interpreter implementations. Generated code, when executed for a given user program at run time, directly emits partially evaluated interpreter instructions for language constructs it knows and sees in the program. This yields the target program faster than performing the first Futamura projection. The generated source code behaves similarly to a specialized partial evaluator deduced by performing the second Futamura projection, although no self-applying partial evaluator is involved during code generation.

CCS Concepts • **Software and its engineering** → **Interpreters**; **Dynamic compilers**; **Translator writing systems and compiler generators**; **Source code generation**.

Keywords partial evaluation, Futamura projection, code generation, interpretation, dynamic compilation

ACM Reference Format:
Florian Latifi. 2019. Practical Second Futamura Projection: Partial Evaluation for High-Performance Language Interpreters. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '19), October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3359061.3361077

[*]This research project is partially funded by Oracle Labs.

1 Motivation

In 1971, Futamura [5] described partial evaluation, i.e., a process to specialize a given interpreter for a source program to be interpreted. By assuming that the input program is constant, optimizations such as constant folding, inlining, and loop unrolling allow removing the dispatching overhead of the interpreter. This effectively compiles the source program to an efficient target program of the language in which the interpreter is implemented. The process is known as the first Futamura projection. Since then, partial evaluation has been used in different ways to specialize programs [2, 10, 11]. In 2017, Würthinger et al. [12] described *Truffle*, a practical realization of the first Futamura projection, to derive efficient target programs from interpreter code automatically. Truffle allows implementing languages in the form of abstract syntax tree (AST) interpreters, which are partially evaluated for given source programs. The resulting code is further optimized and compiled by the *Graal* compiler [3] to high-performance machine code. Truffle language implementations compete in peak performance with existing production systems that already have been heavily optimized for the one single language they support.

During interpretation, profiling information is collected, and the AST adapts to the observed input, i.e., types and values. As soon as the AST stabilizes, this information is assumed to be constant. Partial evaluation can then generate efficient code that speculates [3] on assumptions derived from the collected information and deoptimizes [6] if speculation fails, which invalidates the code and transfers program execution back to the interpreter. This allows updating profiling information and re-executing partial evaluation. For this to work, Truffle provides a few primitives (e.g., annotations, functions), which programmers have to incorporate in their AST interpreter implementations.

Partial evaluation in Truffle starts at the root node of a given AST representing the user method to be compiled. The instructions for interpreting this node are parsed into an intermediate representation (IR) graph, which is partially evaluated according to the incorporated Truffle primitives. For example, annotated fields of AST nodes are treated as constants and their values are folded, which enables other

IR simplifications. Invocations of child nodes are inlined, and loops over multiple children are unrolled. Finally, the resulting graph is handed over to the Graal compiler. These steps make partial evaluation a compile-time intensive task, which is worth optimizing to reduce overall compile time.

2 Problem

Experiments with real-world programs have shown that partial evaluation in Truffle can make half of the overall compile time. The rest is spent by Graal for optimizations and machine code generation. This means, even if we could make the compiler infinitely fast, we would only get a 2x speedup in overall compile time, with partial evaluation remaining the bottleneck. So far, much effort has been put into optimizing the compiler itself using novel techniques [4, 7]. However, we think that it is also necessary and worth optimizing partial evaluation to further reduce overall compile time.

Similar to AST interpretation, partial evaluation has a certain overhead. It requires abstract interpretation of interpreter implementations to specialize them for a fixed input program. Self-application is one idea to remove this overhead, and is known as the second Futamura projection [5]. In this case, the partial evaluator specializes itself for a given interpreter. With this, it can directly emit target programs for given source programs from partially evaluated interpreter instructions. This should yield the target program much faster than a general partial evaluator. However, self-application approaches often don't support all language features, may produce code that only gives modest speedups, or impose other problems which are avoided by a hand-written code generator that creates specialized partial evaluators [1, 8]. For self-application to work in Truffle, partial evaluation itself would need to be re-implemented in form of an AST interpreter, i.e., a dedicated Truffle language with properly incorporated Truffle primitives. This language would also have to support the Java language features that are used in Truffle's original partial evaluation mechanism. Therefore, we propose an approach that uses a hand-written code generator instead of self-application, because we think that it has the potential to give similar results in terms of language support and speedup, and is also easier to realize.

3 Approach

In this paper, we propose an approach to speed up partial evaluation, by using source code generation instead of self-application. We first present a simplified version of Truffle's current partial evaluation to illustrate the approach. Listing 1 shows the doPE method, taking an AST node as the starting point. It first parses the node's exec method into an IR graph. The exec method denotes the logic for the interpretation of this node. Then, doPE iterates the IR and performs partial evaluation accordingly. If it sees, e.g., the invocation of another exec method, and the receiver represents a child node, the receiver is assumed to be constant. This allows parsing the exec method of the given child and replacing the invocation, by inlining the child graph into the current graph. Method parsing and dispatching on given IR is what makes partial evaluation slow for large ASTs.

```
IRGraph doPE(ASTNode node) {
 IRGraph graph = node.parse();
 for (IRNode inst : graph.instructions) {
  if (inst instanceof IRInvoke) {
   IRInvoke invoke = (IRInvoke) inst;
   IRNode receiver = invoke.args[0];
   if (isExec(invoke.method) && isChild(receiver)) {
    ASTNode child = (ASTNode) receiver.asConstant();
    IRGraph childGraph = child.parse();
    graph.inline(childGraph, invoke);
   } else { /* handle invoke ... */ }
  } else { /* handle other ... */ }
 }
 return graph;
}
```

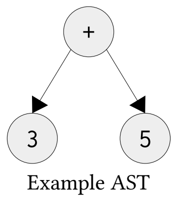
Listing 1. Truffle's Partial Evaluation.

In order to speed up partial evaluation, we propose to generate specialized versions of doPE for specific node types of an AST interpreter ahead of time, removing the method parsing and instruction dispatching overhead. If partial evaluation is then requested for such an AST node, a partially evaluated IR graph is emitted directly. We illustrate this for a very simple AST interpreter which only adds long constants. Listing 2 shows the node classes that implement this logic. AddNode has two fields, left and right. Both are annotated with Truffle's @Child primitive, which the isChild method in doPE checks to enable parsing and inlining of these nodes. The exec method calls both children and returns the sum of the two long values. ConstNode has a final long field that is returned by the exec method.

```
class AddNode extends ASTNode {
 @Child ASTNode left;
 @Child ASTNode right;
 long exec() { return left.exec() + right.exec(); }
}
class ConstNode extends ASTNode {
 final long value;
 long exec() { return value; }
}
```

Listing 2. AST Interpreter Node Classes.

Source code for the specialized versions of doPE is generated ahead of time by parsing the exec methods of these node classes and traversing the IR graph in reverse post-order. IR nodes describing values already known, e.g., literals, are constant-folded through the graph. IR Nodes describing values annotated with Truffle primitives, e.g., loads of fields annotated with @Child, are assumed to be known during partial evaluation. In this case, source code is generated that computes these values without emitting any IR. Nodes describing other values are assumed to be unknown during partial evaluation. In this case, source code is generated that emits IR for the target program to compute the values at run time when the program is executed.



doPE	1) Parse AddNode	2) Parse left & inline	3) Parse right & inline	4) Optimize
	`IRAdd{ IRInvoke{left, exec}, IRInvoke{right, exec}}`	`IRAdd{ IRConst{3}, IRInvoke{right, exec}}`	`IRAdd{ IRConst{3}, IRConst{5}}`	`IRConst{8}`
doPE2	1) Emit IR	2) Optimize		
	`IRAdd{ IRConst{3}, IRConst{5}}`	`IRConst{8}`		

Figure 1. Comparison of doPE and doPE2.

Listing 3 shows the generated partial evaluation methods for the AST node classes, denoted as doPE2. The first method dispatches between the other two, based on the given node type. The other methods take concrete AddNode and ConstNode objects. Since the fields of AddNode are annotated with @Child, these field loads are assumed to be known during partial evaluation. This allows replacing the original exec invocations in the interpreter, by calling doPE2 for both fields and inlining the resulting graphs, to emit an IRAdd node with the return values as inputs. For the ConstNode, its value is stored in a final field, i.e., the value is also assumed to be known during partial evaluation. This allows emitting an IRConst node by reading the value field.

```
IRGraph doPE2(ASTNode node) {
 if (node instanceof AddNode) {
  return doPE2((AddNode) node);
 } else if (node instanceof ConstNode) {
  return doPE2((ConstNode) node);
 } else { /* handle other ... */ }
}
IRGraph doPE2(AddNode node) {
 IRGraph leftVal = doPE2(node.left);
 IRGraph rightVal = doPE2(node.right);
 return new IRAdd(leftVal, rightVal);
}
IRGraph doPE2(ConstNode node) {
 return new IRConst(node.value);
}
```

Listing 3. Specialized Partial Evaluation.

Finally, Figure 1 shows a comparison of doPE and doPE2 for an example AST which adds two constants. doPE parses the AddNode and sees both child invocations, so it parses and inlines the IR of both exec methods, respectively. Finally, the compiler optimizes the addition in the IR to a constant. In comparison, doPE2 directly emits partially evaluated IR, i.e., the addition of both constants, which the compiler then optimizes. Compared to doPE, doPE2 requires two steps less to produce the same IR.

4 Evaluation Methodology

We are currently implementing our approach in Truffle, to test our hypothesis that the generation of source code for language interpreters ahead-of-time significantly speeds up partial evaluation and reduces overall compile time. We will continuously evaluate our approach and compare it with Truffle's existing partial evaluation mechanism, by measuring partial evaluation and overall compile time, as well as any impacts on code size due to code generation. For first experiments, we will use *PELang*[1], a language we developed for penetrating specific corner cases of partial evaluation. It allows programmatically generating ASTs with nested loops to unroll, invocations to inline, and other complex control flow structures. We will also evaluate our approach with industry-standard language benchmarks for important Truffle interpreter implementations, such as *Graal.js* [12] for JavaScript and *Sulong* [9] for LLVM-Bitcode.

Generated source code may explode due to the complexity of real language implementations. Thus, we plan to find trade-offs between partial evaluation time and code size to avoid code explosion. We will experiment with generating source code only for frequently used parts of interpreters and using Truffle's existing partial evaluation mechanism as fallback. Here, we have to identify how selecting interpreter parts for code generation affects partial evaluation time and code size.

References

[1] L. Birkedal and M. Welinder. 1994. Hand-writing program generator generators. In *PLILP*.

[2] C. F. Bolz, M. Leuschel, and A. Rigo. 2010. Towards Just-In-Time Partial Evaluation of Prolog. In *LOPSTR*.

[3] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*.

[4] J. Eisl, M. Grimmer, D. Simon, T. Würthinger, and H. Mössenböck. 2016. Trace-based Register Allocation in a JIT Compiler. In *PPPJ*.

[5] Y. Futamura. 1971. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls* (1971).

[6] U. Hölzle, C. Chambers, and D. Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *PLDI*.

[7] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. 2018. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *CGO*.

[8] L. Michael, J. Jesper, V. Wim, and B. Maurice. 2002. Offline Specialisation in Prolog Using a Hand-Written Compiler Generator. *CoRR* (2002).

[9] M. Rigger, M. Grimmer, C. Wimmer, T. Würthinger, and H. Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *VMIL*.

[10] U. P. Schultz, J. L. Lawall, and C. Consel. 2003. Automatic Program Specialization for Java. *TOPLAS* (2003).

[11] A. Shali and W. R. Cook. 2011. Hybrid Partial Evaluation. In *OOPSLA*.

[12] T. Würthinger, C. Wimmer, C. Humer, A. Wöss, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *PLDI*.

[1] https://github.com/flortsch/graal/tree/pe-bench