Theoretical and Applied Informatics
Vol. 28 (2016), no. 4, pp. 15–32
DOI: 10.20904/284015

Revisiting the Futamura Projections: A Diagrammatic Approach

Brandon M. Williams[1*] Saverio Perugini[1†]

[1]Department of Computer Science
University of Dayton
300 College Park
Dayton, Ohio 45469–2160 USA

Abstract The advent of language implementation tools such as *PyPy* and *Truffle/Graal* have reinvigorated and broadened interest in topics related to automatic compiler generation and optimization. Given this broader interest, we revisit the *Futamura Projections* using a novel diagram scheme. Through these diagrams we emphasize the recurring patterns in the Futamura Projections while addressing their complexity and abstract nature. We anticipate that this approach will improve the accessibility of the Futamura Projections and help foster analysis of those new tools through the lens of partial evaluation.

Keywords compilation; compiler generation; Futamura Projections; *Graal*; interpretation; partial evaluation; program transformation; *PyPy*; *Truffle*

Received 29 MAR 2017 **Revised** 04 AUG 2017 **Accepted** 13 SEP 2017

1 Introduction

The *Futamura Projections* are a series of program signatures reported by [1] (a reprinting of [2]) designed to create a program that generates compilers. This is accomplished by repeated applications of a *partial evaluator* that iteratively abstract away aspects of the program execution process. A partial evaluator transforms a program given any subset of its input to produce a version of the program that has been specialized to that input. The partial evaluation operation is referred to as *mixed computation* because partial evaluation involves a mixture of interpretation and code generation [3]. In this paper, we will provide an overview of typical program processing and discuss the Futamura Projections. We apply to the Projections a novel diagram scheme which emphasizes the relationships between programs involved in the projections. We also discuss related topics in the context of the Futamura Projections through application of the same diagram scheme.

*E-mail: BrandonWilliamsCS@gmail.com
†E-mail: saverio@udayton.edu

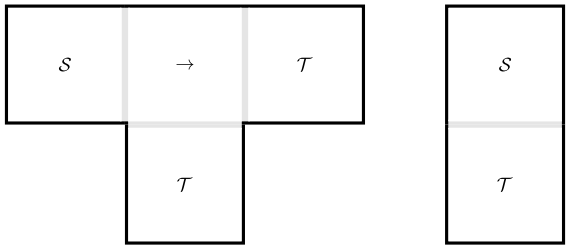


Figure 1 The classical $\mathcal{T}$ and $\mathcal{I}$ diagrams, referring to the shapes of the diagrams [4].

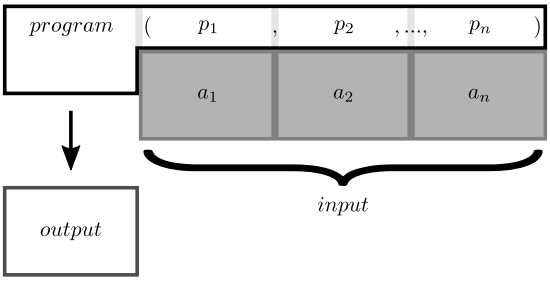
Table 1 Legend of symbols and terms used in § 2 and § 3.

Symbol	Example	Description
program	`pow`	A miscellaneous program.
p_n	`b`	A parameter.
a_n	`3`	An argument.
$compiler_{\mathcal{T}}^{\mathcal{S}\rightarrow\mathcal{T}}$	$\texttt{compiler}_{\texttt{x86}}^{\texttt{C}\rightarrow\texttt{x86}}$	A compiler from language $\mathcal{S}$ to language $\mathcal{T}$, implemented in $\mathcal{T}$.
$program_{\mathcal{L}}$	`pow.c`	A miscellaneous program implemented in language $\mathcal{L}$.
$interpreter_{\mathcal{T}}^{\mathcal{S}}$	$\texttt{interpreter}_{\texttt{x86}}^{\texttt{C}}$	An interpreter for language $\mathcal{S}$ implemented in language $\mathcal{T}$.
$partial\ input_{static}$	$\texttt{partial input}_{\texttt{static}}$	A subset of input for a program being specialized by *mix*.
$program'_{\mathcal{T}}$	$\texttt{square}_{\texttt{x86}}$	A specialized program implemented in language $\mathcal{T}$.
$mix_{\mathcal{T}}$	$\texttt{mix}_{\texttt{x86}}$	A partial evaluator implemented in language $\mathcal{T}$.
compiler	`compiler`	A compiler generator implemented in language $\mathcal{T}$.
$generator_{\mathcal{T}}$	$\texttt{generator}_{\texttt{x86}}$	

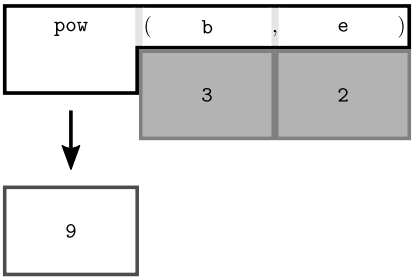
A common syntax for the graphical presentation of program interpretation and transformation involves the classical $\mathcal{I}$ (for interpreter) and $\mathcal{T}$ (for translator, meaning compiler) diagrams, referring to the shapes of the diagrams, respectively [4]. Fig. 1 illustrates the syntax of each diagram. An $\mathcal{I}$ diagram specifies the interpreted language spatially above the implementation language. A $\mathcal{T}$ diagram represents a compiler with an arrow from its source to target languages spatially above its implementation language. These diagrams are simple to draw and recognize, and excel at showing how multiple languages relate during compilation and interpretation. However, although the diagrams can be extended to express a partial evaluator (as described in [5]), they are not especially suited to representing the partial evaluation process due to the secondary input given to the partial evaluator. Additionally, the many languages and programs involved in the Futamura Projections add complexity that is not addressed by this style of diagram. In contrast, we believe our diagrams make clear the relationships between the various languages and programs involved in the Futamura Projections and hope that they will improve the accessibility of the Projections.

2 Programs Processing Other Programs

We use notation in this paper for partial evaluation and associated programming language concepts from [4, 6]. That notation is commonly used in papers on partial evaluation [5, 7]. In particular, we use $[\![-]\!]$ to denote a *semantics function* which represents the evaluation of a pro-



(a) Program execution depicted as a machine.



(b) Program execution instance.

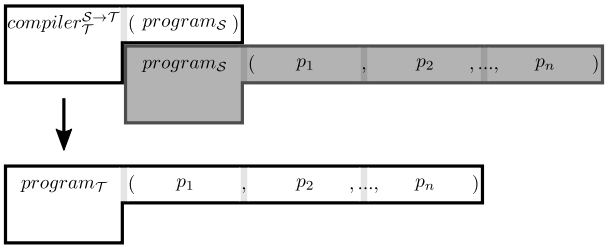
Figure 2 Program execution.

gram by mapping that program's input to its output. For instance, $[\![p]\!][s, d]$ represents the program p applied to the inputs s and d. We use the symbol `mix` to denote the partial evaluation operation [4, 6], which involves a `mix`ture of interpretation and code generation. Table 1 is a legend mapping additional terms and symbols used in this article to their description.

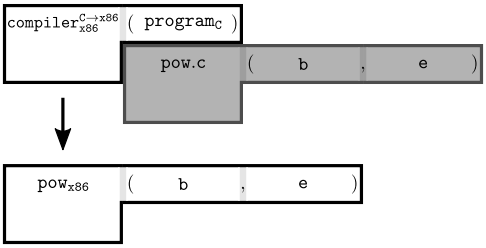
Program execution can be represented equationally as $[\![program]\!][a_1, a_2, ..., a_n] = [output]$ [4]. Alternatively, the diagram in Fig. 2a depicts a program as a machine that takes a collection of input boxes, marked by divided *slots* of the input *bar*, and produces an output box. We use this diagram syntax to aid in the presentation of complex relationships between programs, inputs, and programs treated as inputs (i.e., data). Each input area corresponds to part of a C-function-style signature that names and positions the inputs. The input is presented in gray to distinguish it from the program and its input bar. Fig. 2b shows this pattern applied to a program that takes a base `b` and an exponent `e` and raises the base to the power of the exponent. In this case, `3` raised to the power of `2` produces `9`, or $[\![\texttt{pow}]\!][3, 2] = [9]$.

2.1 Compilation

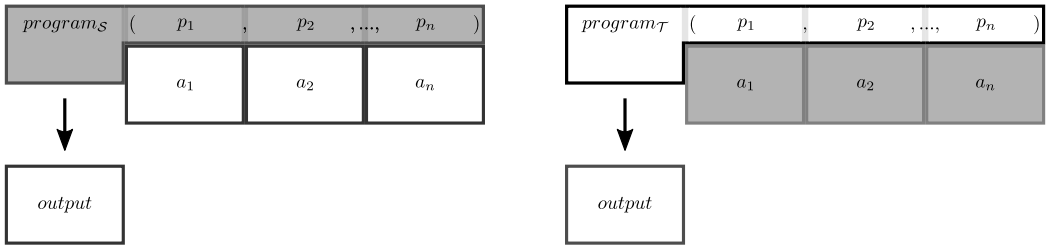
Programs written in higher-level programming languages such as C must be either compiled to a natively runnable language (e.g., the `x86` machine language) or evaluated by an interpreter. A compiler is simply a program that translates a program from its source language to a target language. This process is described equationally as $[\![compiler_{\mathcal{T}}^{\mathcal{S}\to\mathcal{T}}]\!][program_{\mathcal{S}}] = [program_{\mathcal{T}}]$, or diagrammatically in Fig. 3a. For clarity, the implementation language appears as a subscript of any program name. Compilers will also have a superscript with an arrow from the source (input) language to the target (output) language. If language $\mathcal{T}$ is natively executable, both the depicted



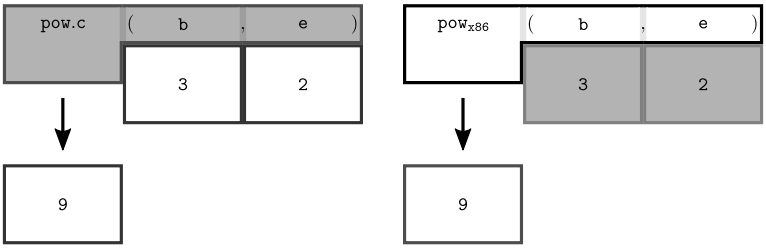
(a) The compilation process.



(b) Compilation example.

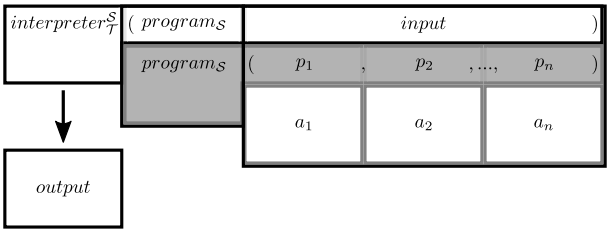


(c) Comparison of compiler input and output.

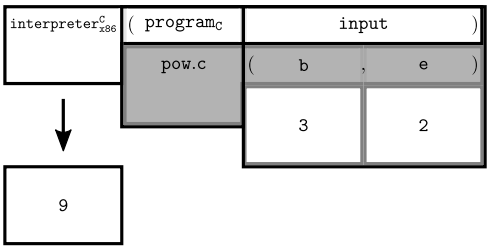


(d) Comparison of compiler example input and output.

Figure 3 Execution through compilation.



(a) The interpretation process.



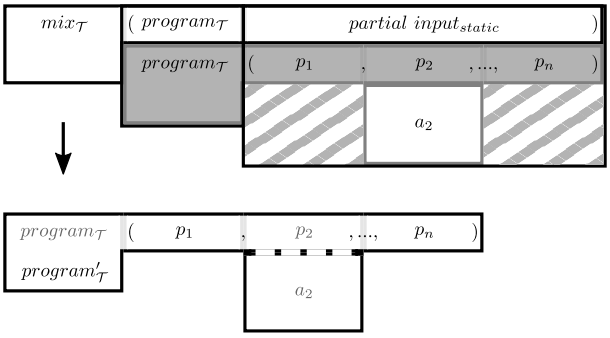
(b) Instance of program interpretation.

Figure 4 Execution by interpretation.

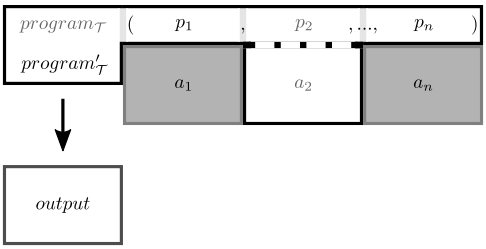
compiler and its output program are natively executable. If `pow` from Fig. 2b is written in C, it can be compiled to the `x86` machine language with a compiler as depicted in Fig. 3b or expressed equationally as $[\![\mathtt{compiler}^{\mathtt{C}\to\mathtt{x86}}_{\mathtt{x86}}]\!][\mathtt{pow.c}] = [\mathtt{pow}_{\mathtt{x86}}]$. Figs. 3c and 3d illustrate that the source and the target programs are semantically equivalent.

2.2 Interpretation

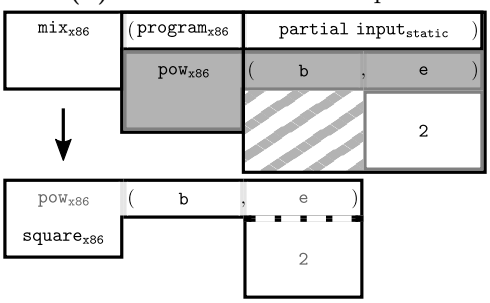
The gap between a high-level source language and a natively executable target language can also be bridged with the use of an interpreter. "The interpreter for a computer language is just another program" implemented in the target language that evaluates the program given the program's input and producing its output [8]. The interpretation pattern, described equationally as $[\![\mathit{interpreter}^{\mathcal{S}}_{\mathcal{T}}]\!][\mathit{program}_{\mathcal{S}}, \mathit{input}] = [\mathit{output}]$, is depicted in Fig. 4a. The interpreter has the previously established implementation language subscript, with a superscript indicating the interpreted language. The input program's input bar extends into the interpreter's next input slot, which serves to indicate which individual input is associated with each of the program's own input slots. However, as the inputs are actually being provided directly to the interpreter, an outline is drawn around each input to the program being executed. By convention, the background shading is alternated to differentiate inputs, while the borders of inputs remain gray. This pattern is applied to the `pow.c` program in Fig. 4b; the C program is being executed by an interpreter implemented in `x86` to produce the output from `pow.c` given its input. This interpretation is represented equationally as $[\![\mathtt{interpreter}^{\mathtt{C}}_{\mathtt{x86}}]\!][\mathtt{pow.c}, 3, 2] = [9]$.



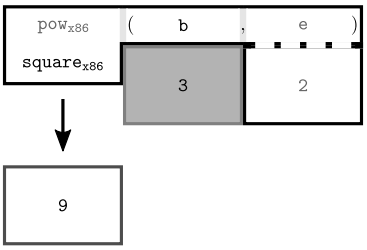
(a) The partial evaluation process.



(b) Partial evaluation output.



(c) Instance of partial evaluation.



(d) Output of partial evaluation instance.

Figure 5 Partial evaluation.

2.3 Partial Evaluation

With typical program evaluation, complete input is provided during program execution. With partial evaluation, on the other hand, partial input—referred to as *static* input—is given in advance of program execution. This partial input is given to `mix` with the program to be evaluated and *specializes* the program to the input. The specialized program then accepts only the remaining input—referred to as the *dynamic* input—and produces the same output as would have been produced by evaluating the original program with complete input. For reasons explained below, the Futamura Projections require that `mix` be implemented in the same language as the program it takes as input; diagrams including `mix` provide a subscript that represents the implementation language of `mix` as well as the that of the input and output programs.

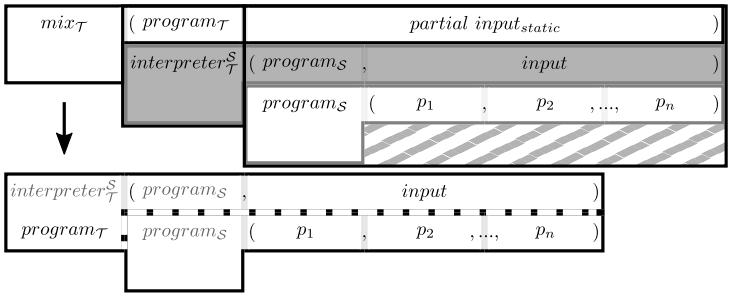
Partial evaluation is described equationally as $[\![mix_{\mathcal{T}}]\!][program_{\mathcal{T}}, partial\ input_{static}] = [program'_{\mathcal{T}}]$ and depicted in Fig. 5a. Here, a program is being passed to `mix` with partial input—specifically, only its second argument. The result is a transformed version of the program specialized to the input; the input has been propagated into the original program to produce a new program. This specialized transformation of the input program is called a *residual program* [3]. Notice how the shape of the residual program matches the shape of the input program combined with the static input. Notice also in Fig. 5b that the shape of the program combined with the remainder of its input matches the shape of the typical program execution shown in Fig. 2a. However, the input has been visually fused to the program, represented by the dotted line. In addition, the labels for the original program, the second input slot, and the static input have been shaded gray; while the residual program is entirely comprised of these two components, its input interface has been modified to exclude them. In other words, while the semantics of the components are still present, they are no longer separate entities. The equational representation of this residual program shows the simplicity of its behavior: $[\![program'_{\mathcal{T}}]\!][a_1, a_3, ..., a_n] = [output]$.

The partial evaluation pattern is applied to the $\texttt{pow}_{\texttt{x86}}$ program in Fig. 5c. If $\texttt{pow}_{\texttt{x86}}$ is partially evaluated with static input $\texttt{e}=2$, the result is a power program that can only raise a base to the power 2. This example is represented equationally as $[\![\texttt{mix}_{\texttt{x86}}]\!][\texttt{pow}_{\texttt{x86}}, \texttt{e}=2] = [\texttt{square}_{\texttt{x86}}]$. The specialization produces a program that takes a single input (as in Fig. 5d and $[\![\texttt{square}_{\texttt{x86}}]\!][3] = [9]$) and squares it. It behaves as a squaring program despite being comprised of a power program and an input; `mix` has propagated the input into the original program to produce a specialized residual program.

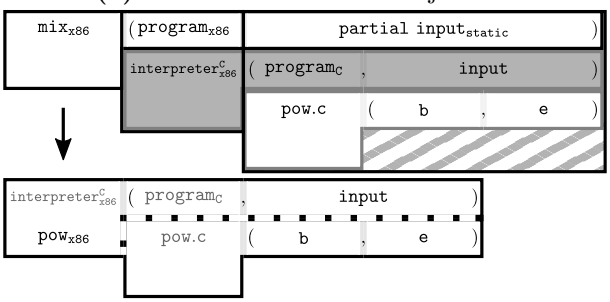
3 The Futamura Projections

3.1 First Futamura Projection: Compilation

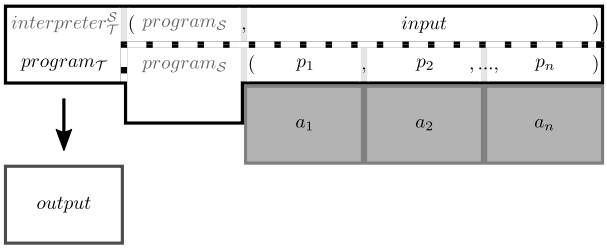
Partial evaluation is beneficial given a program that will be executed repeatedly with some of its input constant, sometimes resulting in a significant speedup. For example, if squaring many values, a specialized squaring program generated from a power program prevents the need for repeated exponent $\texttt{e}=2$ arguments. Program interpretation is another case that benefits from partial evaluation; after all, the interpreter is a program and the source program is a subset of its input. Fig. 6a illustrates that when given $program_{\mathcal{S}}$ and an interpreter for $\mathcal{S}$ implemented in language $\mathcal{T}$, we can partially evaluate the interpreter with the source program as static input



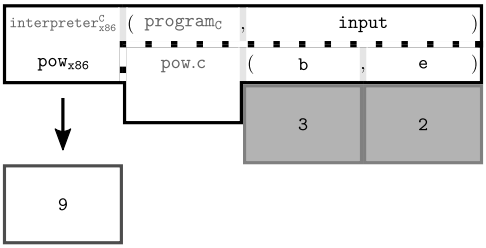
(a) The First Futamura Projection.



(b) An instance of the First Futamura Projection.



(c) The output of the First Futamura Projection.



(d) The output of the First Futamura Projection instance.

Figure 6 The First Futamura Projection and example instance.

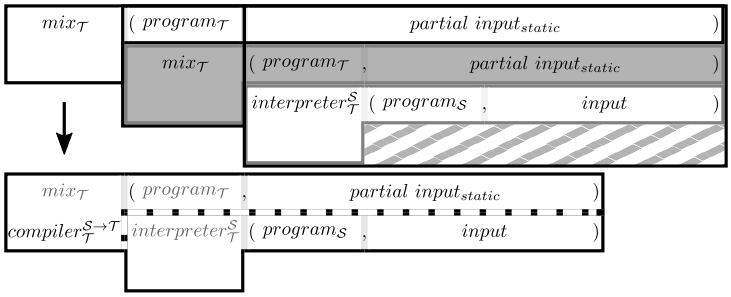
(i.e., $[\![mix_{\mathcal{T}}]\!][interpreter^{\mathcal{S}}_{\mathcal{T}}, program_{\mathcal{S}}] = [program_{\mathcal{T}}]$). This is the *First Futamura Projection.* As with the previous pattern, the partially evaluated program (i.e., the interpreter) has been specialized to the partial input (i.e., the source program), which is indicated visually by the fusion of the source program to the interpreter. Notice that $program_{\mathcal{S}}$ is vertically aligned with the static input slot of the partial evaluator as well as the program input slot of the interpreter. This is because $program_{\mathcal{S}}$ serves both roles. In this case, the dynamic input of the interpreter is the entirety of the input for $program_{\mathcal{S}}$. When that input is provided in Fig. 6c, the specialized program completes the interpretation of $program_{\mathcal{S}}$, producing the output for $program_{\mathcal{S}}$. *In other words, the specialized program behaves exactly the same as* $program_{\mathcal{S}}$*, but is implemented in* $\mathcal{T}$ *rather than* $\mathcal{S}$. The partial evaluator has effectively *compiled* the program from $\mathcal{S}$ to $\mathcal{T}$. Thus, the equational form is identical to that of a compiled program: $[\![program_{\mathcal{T}}]\!][a_1, a_2, ..., a_n] = [output]$. Fig. 6b and equation $[\![\texttt{mix}_{\texttt{x86}}]\!][\texttt{interpreter}^{\texttt{C}}_{\texttt{x86}}, \texttt{pow.c}] = [\texttt{pow}_{\texttt{x86}}]$ express the partial evaluation of a C interpreter when given `pow` as partial input. The residual program, detailed in Fig. 6d, behaves the same as `pow.c`, but is implemented in `x86`. The equational expression for the target program is also identical to the compiled program: $[\![\texttt{pow}_{\texttt{x86}}]\!][3, 2] = [9]$.

> **First Futamura Projection**: A partial evaluator, by specializing an interpreter to a program, can compile from the interpreted language to the implementation language of `mix`.

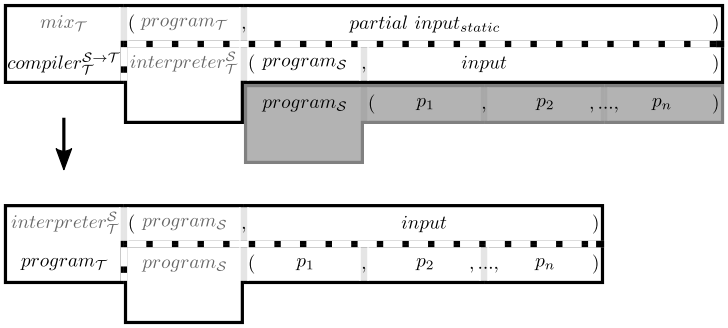
3.2 Second Futamura Projection: Compiler Generation

The First Futamura Projection relies on the nature of interpretation requiring two types of input: a program that may be executed multiple times, and input for that program that may vary between executions. As it turns out, the use of `mix` as a compiler exhibits a similar signature: the interpreter is specialized multiple times with different source programs. This allows us to partially evaluate the process of compiling with a partial evaluator. This is the *Second Futamura Projection*, represented equationally as $[\![mix_{\mathcal{T}}]\!][mix_{\mathcal{T}}, interpreter^{\mathcal{S}}_{\mathcal{T}}] = [compiler^{\mathcal{S}\rightarrow\mathcal{T}}_{\mathcal{T}}]$ and depicted in Fig. 7a. In this partial-partial evaluation pattern, an instance of `mix` is being provided as the program input to another instance of `mix`, to which an interpreter is provided as static input. Just as in earlier partial evaluation patterns, the program input has been specialized to the given static input; in this case, an instance of `mix` is being specialized to the interpreter. The vertical alignment of programs helps clarify the roles of each program present: the interpreter is the partial input given to the executing instance of `mix` as well as the program input given to the specialized instance of `mix`. Additionally, *this specialized residual program as executed in Fig. 7b matches the shape and behavior of the First Futamura Projection shown in Fig. 6a.* This is because the same program is being executed with the same input; the only difference is that the output of the second projection is a single program that has been specialized to the interpreter rather than a separate `mix` instance that requires the interpreter to be provided as input. In the Second Futamura Projection, `mix` has generated the `mix`-based compiler from the first projection. Because the residual program is a compiler, its equational expression is that of a compiler: $[\![compiler^{\mathcal{S}\rightarrow\mathcal{T}}_{\mathcal{T}}]\!][program_{\mathcal{S}}] = [program_{\mathcal{T}}]$.

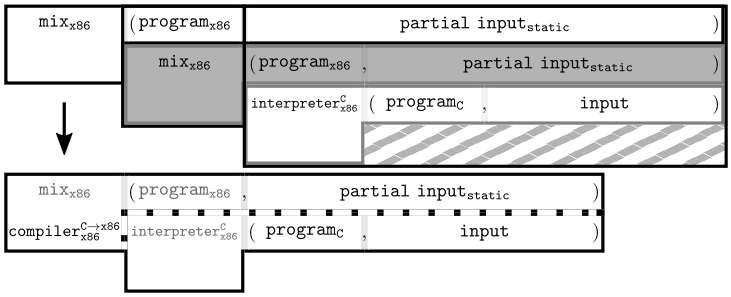
Revisiting the `pow.c` program in Figs. 7c and 7d, `mix` is specialized to a C interpreter to produce a C compiler (i.e., $[\![\texttt{mix}_{\texttt{x86}}]\!][\texttt{mix}_{\texttt{x86}}, \texttt{interpreter}^{\texttt{C}}_{\texttt{x86}}] = [\texttt{compiler}^{\texttt{C}\rightarrow\texttt{x86}}_{\texttt{x86}}]$). When given



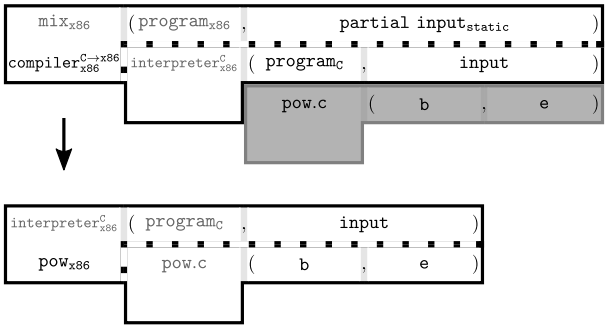
(a) The Second Futamura Projection.



(b) The output of the Second Futamura Projection.

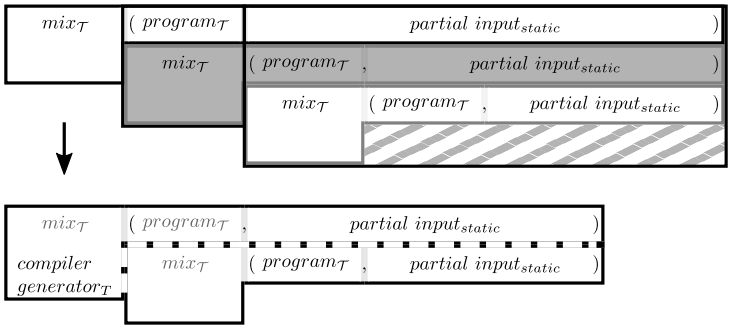


(c) An instance of the Second Futamura Projection.

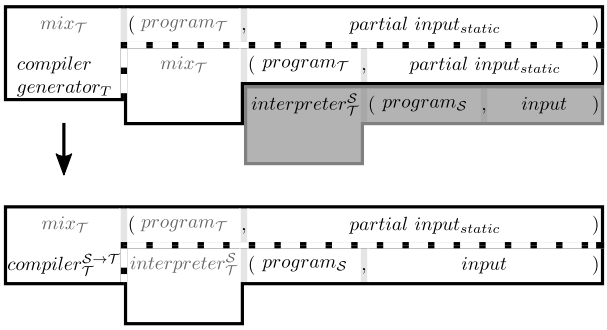


(d) The output of the Second Futamura Projection instance.

Figure 7 The Second Futamura Projection and example instance.



(a) The Third Futamura Projection.



(b) The output of the Third Futamura Projection.

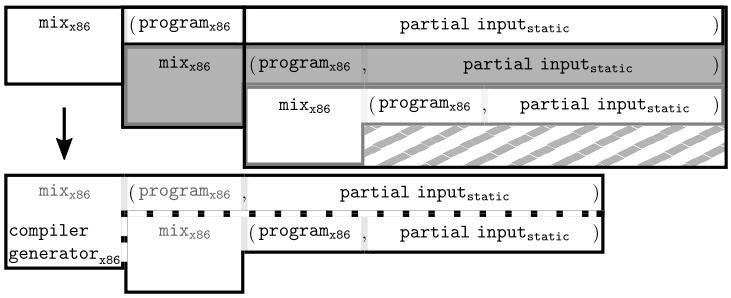
Figure 8 The Third Futamura Projection and output.

`pow` in C, this specialized `mix` program then specializes the interpreter to `pow`, producing an equivalent power program in `x86` ($[\![\texttt{compiler}^{\texttt{C}\to\texttt{x86}}_{\texttt{x86}}]\!][\texttt{pow.c}] = [\texttt{pow}_{\texttt{x86}}]$).

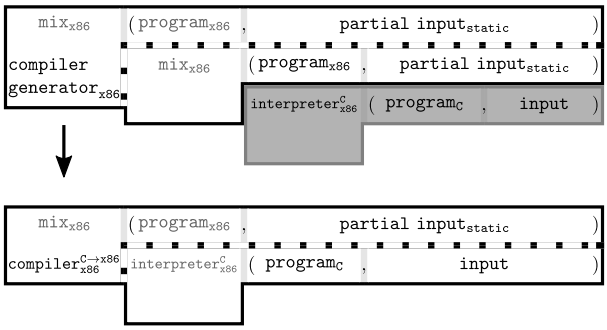
> **Second Futamura Projection**: A partial evaluator, by specializing another instance of itself to an interpreter, can generate a compiler from the interpreted language to the implementation language of `mix`.

3.3 Third Futamura Projection: Generation of Compiler Generators

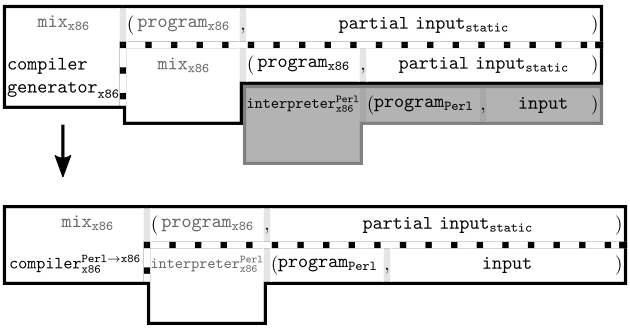
Because `mix` can accept itself as input, we can use one instance of `mix` to partially evaluate a second instance of `mix`, passing a third instance of `mix` as the static input. This is the *Third Futamura Projection*, shown in Fig. 8a and written equationally as $[\![mix_{\mathcal{T}}]\!][mix_{\mathcal{T}}, mix_{\mathcal{T}}] = [compiler\ generator_{\mathcal{T}}]$. The transformation itself is straightforward: partially evaluating a program with some input. The output is still the program in the first input slot specialized to the data in the second input slot; however, this time both the program and the data are instances of `mix`. Again, the positioning of the various instances of `mix` within the diagram serves to clarify how the instances interact. The outermost instance executes with the other two instances as input. The middle instance is the "program" input of the outer instance and is specialized to the inner instance. Finally, the inner instance is being integrated into the middle instance by the outer instance. *Notice that Fig. 8b shows that the execution of the residual*



(a) An instance of the Third Futamura Projection.



(b) The output of the Third Futamura Projection instance.



(c) The compiler generator generating a Perl compiler.

Figure 9 Instance of the Third Futamura Projection and output demonstration.

program matches the shape and behavior of the Second Futamura Projection shown in Fig. 7a when provided with an interpreter as input. The partial evaluator has generated the `mix`-based compiler generator from the second projection. This process is represented equationally as $[\![compiler\ generator_{\mathcal{T}}]\!][interpreter_{\mathcal{T}}^{\mathcal{S}}] = [compiler_{\mathcal{T}}^{\mathcal{S}\to\mathcal{T}}]$.

Interestingly, the only variable part of the Third Futamura Projection is the language associated with `mix`. Previous instance diagrams were specific to the `pow.c` program; for instance, Fig. 7c presents an interpreter for the implementation language of `pow.c`, namely C. However, the diagram in Fig. 9a and the expression $[\![\texttt{mix}_{\texttt{x86}}]\!][\texttt{mix}_{\texttt{x86}}, \texttt{mix}_{\texttt{x86}}] = [\texttt{compiler generator}_{\texttt{x86}}]$ make no reference to `pow.c` or C. This is because the Third Futamura Projection has abstracted the interpretation process to an extent that even the interpreter is considered dynamic input. Fig. 9b shows the `mix`-generated compiler generator accepting a C interpreter and generating a C to `x86` compiler ($[\![\texttt{compiler generator}_{\texttt{x86}}]\!][\texttt{interpreter}_{\texttt{x86}}^{\texttt{C}}] = [\texttt{compiler}_{\texttt{x86}}^{\texttt{C}\to\texttt{x86}}]$), but it will accept *any* interpreter implemented in `x86` regardless of the language interpreted. For example, Fig. 9c shows a compiler for the language Perl being generated by the same compiler generator (i.e., $[\![\texttt{compiler generator}_{\texttt{x86}}]\!][\texttt{interpreter}_{\texttt{x86}}^{\texttt{Perl}}] = [\texttt{compiler}_{\texttt{x86}}^{\texttt{Perl}\to\texttt{x86}}]$). A residual compiler generated through the result of the Third Futamura Projection is called a *generating extension*—a term coined by Ershov [3]—of the input interpreter [7]. In general, the result of The Third Projection creates a generating extension for any program provided to it.

> **Third Futamura Projection**: A partial evaluator, by specializing an additional instance of itself to a third instance, can generate a compiler generator that produces compilers from any language to the implementation language of `mix`.

3.4 Summary: Futamura Projections

The Third Futamura Projection follows the pattern of the previous two projections: the use of `mix` to partially evaluate a prior process (i.e., interpretation, compilation). The first projection compiles by partially evaluating the interpretation process without the input of the source program. The Second Futamura Projection generates a compiler by partially evaluating the compilation process of the first projection without any particular source program. The Third Futamura Projection generates a compiler generator by partially evaluating the compiler generation process of the second projection without an interpreter. Each projection delays completion of the previous process by abstracting away the more variable of two inputs. Just as the first projection interprets a program with various, dynamic inputs and the second projection compiles various programs, the third projection generates compilers for various languages/interpreters. Table 2 juxtaposes the related equations and diagrams from both § 2 and § 3 in each row to make their relationships more explicit. Each row of Table 3 succinctly summarizes each projection by associating each side of its equational representation with the corresponding diagram from § 3.

Table 2 Juxtaposition of related equations and diagrams from § 2 and § 3.

Fig.	Equational Notation	Input Description	Output Description	Similar Fig.	Output Fig.
2a	$[\![program]\!][a_1, a_2, ..., a_n] = [output]$	A list of arguments to the program.	The result of the program's execution.	N/A	N/A
3a	$[\![compiler_{\mathcal{T}}^{\mathcal{S}\to\mathcal{T}}]\!][\![program_{\mathcal{S}}]\!] = [\![program_{\mathcal{T}}]\!]$	A program in language $\mathcal{S}$.	The input program in language $\mathcal{T}$.	N/A	3c
4a	$[\![interpreter_{\mathcal{T}}^{\mathcal{S}}]\!][\![program_{\mathcal{S}}, input]\!] = [\![output]\!]$	A program in language $\mathcal{S}$, along with its input.	The result of the program's execution.	N/A	N/A
5a	$[\![mix_{\mathcal{T}}]\!][\![program_{\mathcal{T}}, partial\ input_{static}]\!] = [\![program'_{\mathcal{T}}]\!]$	A program in language $\mathcal{T}$, along with any subset of its input.	The input program specialized to the partial input.	N/A	5b
5b	$[\![program'_{\mathcal{T}}]\!][a_1, a_3, ..., a_n] = [\![output]\!]$	A list of arguments to the original program, excluding the static input.	The result of the original program's execution.	2a	N/A
6a	$[\![mix_{\mathcal{T}}]\!][\![interpreter_{\mathcal{T}}^{\mathcal{S}}, program_{\mathcal{S}}]\!] = [\![program_{\mathcal{T}}]\!]$	An interpreter for language $\mathcal{S}$ and a program implemented in $\mathcal{S}$.	The program implemented in $\mathcal{T}$.	N/A	6c
6c	$[\![program_{\mathcal{T}}]\!][a_1, a_2, ..., a_n] = [\![output]\!]$	A list of arguments to the original program.	The result of the original program's execution.	4a	N/A
7a	$[\![mix_{\mathcal{T}}]\!][\![mix_{\mathcal{T}}, interpreter_{\mathcal{T}}^{\mathcal{S}}]\!] = [\![compiler_{\mathcal{T}}^{\mathcal{S}\to\mathcal{T}}]\!]$	*mix* and an interpreter for language $\mathcal{S}$, both implemented in $\mathcal{T}$.	A compiler from $\mathcal{S}$ to $\mathcal{T}$	N/A	7b
7b	$[\![compiler_{\mathcal{T}}^{\mathcal{S}\to\mathcal{T}}]\!][\![program_{\mathcal{S}}]\!] = [\![program_{\mathcal{T}}]\!]$	A program in language $\mathcal{S}$.	The input program in language $\mathcal{T}$.	6a	6c
8a	$[\![mix_{\mathcal{T}}]\!][\![mix_{\mathcal{T}}, mix_{\mathcal{T}}]\!] = [\![compiler\ generator_{\mathcal{T}}]\!]$	Two instances of *mix* implemented in $\mathcal{T}$.	A compiler generator implemented in $\mathcal{T}$	N/A	8b
8b	$[\![compiler\ generator_{\mathcal{T}}]\!][\![interpreter_{\mathcal{T}}^{\mathcal{S}}]\!] = [\![compiler_{\mathcal{T}}^{\mathcal{S}\to\mathcal{T}}]\!]$	An interpreter for language $\mathcal{S}$.	A compiler from $\mathcal{S}$ to $\mathcal{T}$.	7a	7b

Table 3 Summary of Futamura Projections.

Projection	Description	Equational Notation	Fig.	Output Fig.
1	`mix` can compile.	$[\![mix_{\mathcal{T}}]\!][interpreter_{\mathcal{T}}^{\mathcal{S}}, program_{\mathcal{S}}] = [program_{\mathcal{T}}]$	6a	6c
2	`mix` can generate a compiler.	$[\![mix_{\mathcal{T}}]\!][mix_{\mathcal{T}}, interpreter_{\mathcal{T}}^{\mathcal{S}}] = [compiler_{\mathcal{T}}^{\mathcal{S}\to\mathcal{T}}]$	7a	7b
3	`mix` can generate a compiler generator.	$[\![mix_{\mathcal{T}}]\!][mix_{\mathcal{T}}, mix_{\mathcal{T}}] = [compiler\ generator_{\mathcal{T}}]$	8a	8b

4 Discussion

4.1 Examples of Partial Evaluators

Partial evaluators have been developed for several practical programming languages including C [9], Scheme [10], and Prolog [11]. These partial evaluators are self-applicable and, thus, can perform all three Futamura Projections [5]. However concerns such as performance make the projections impractical with these programs.

4.2 Beyond the Third Projection

Researchers have studied what lies beyond the third projection and what significance the presence of a fourth projection might have [5]. Two important conclusions have been made in this regard:

- Any self-generating compiler generator, i.e., a compiler generator such that,

$$[\![\textit{compiler generator}]\!][\textit{mix}] = \textit{compiler generator},$$

  can be obtained by repeated self-application of a partial evaluator as in the Third Futamura Projection, and vice versa.

- The compiler generator can be applied to another partial evaluator with different properties to produce a new compiler generator with related properties. For example, applying a compiler generator that accepts C programs to a partial evaluator accepting Python (but written in C) will produce a compiler generator that accepts Python programs.

These two observations are not orthogonal, but rather two sides of the same coin because they both depend on the application of a compiler generator to a partial evaluator. We refer the reader interested in further, formal exploration of these ideas and insights to [5].

4.3 Applications

4.3.1 Truffle and Graal

The *Truffle* project, developed by Oracle Labs, seeks to facilitate the implementation of fast, dynamic languages. It provides a framework of Java classes and annotations that allows language developers to build abstract syntax tree interpreters and indicate ways in which program behavior may be optimized [12]. When such an interpreter is applied to a program through the *Graal* compilation infrastructure, the Graal just-in-time compiler (JIT) compiles program code to a custom intermediate representation by applying a partial evaluator at run-time [13]. We illustrate this JIT compilation using our diagram notation in Fig. 10. This is a special form of the First Futamura Projection where only certain fragments of the source program are compiled based on suggestions communicated through the Truffle Domain Specific Language. Through partial evaluation, Truffle and Graal provide a language implementation option that leverages the established benefits of the host virtual machine such as tool support, language interoperability, and memory management.

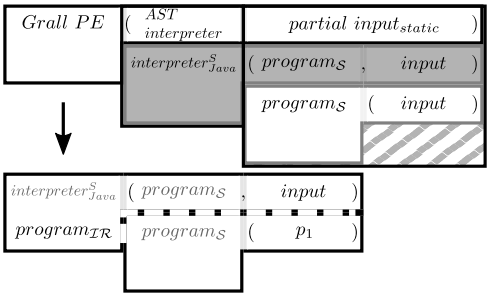
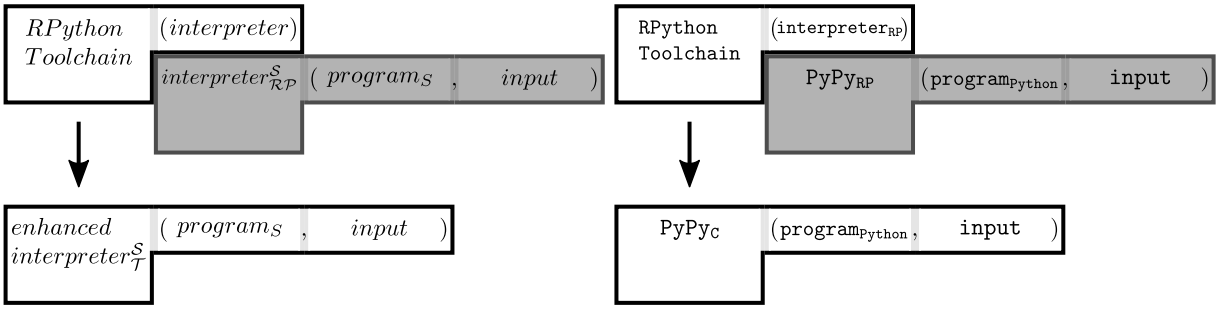


Figure 10 The First Futamura Projection inside the Graal JIT.



(a) The RPython transformation of an interpreter. **(b)** The PyPy self-interpreter, compiled to C.

Figure 11 The PyPy project, represented in our diagram notation.

Because the Graal partial evaluator both specializes and is written in Java, it could in theory be used to reach the second and third projections. However, it was designed specifically to specialize interpreters written with Truffle and makes optimization assumptions about the execution of the program. As a result, the partial evaluator is not practically self-applicable and cannot efficiently be used for the second projection.

4.3.2 PyPy

We would be remiss not to mention the *PyPy* project (`https://pypy.org/`), which approaches the problem of optimizing interpreted programs from a different perspective. While named for its Python self-interpreter, the pertinent part of the project is its RPython translation toolchain. RPython is a framework designed for compiling high-level, dynamic language interpreters from a subset of Python to a selection of low-level languages such as C or bytecode in a way that adds features common to virtual machines such as memory management and JIT compilation. To demonstrate this, the PyPy self-interpreter, once compiled to C, outperforms the CPython interpreter in many performance tests [14]. The translation process is depicted using our diagram scheme in Fig. 11a, with the application to the PyPy interpreter in Fig. 11b.

Although both the Truffle/Graal and PyPy projects make use of JIT compilation for optimization purposes, PyPy's JIT is different from that used by Graal. While both compile the source program indirectly through the interpreter, only Graal performs the First Futamura Pro-

jection by compiling through partial evaluation. PyPy, on the other hand, uses a *Tracing JIT compiler* that traces frequently executed code and caches the compiled version to bypass interpretation [15]. For a more in-depth comparison of the methods used by Truffle/Graal and PyPy, including performance measurements, we refer the reader to [16].

4.4 Conclusion

The Futamura Projections can be used to compile, generate compilers, and generate compiler generators. We are optimistic that this article has demystified their esoteric nature. Increased attention for and broader awareness of this topic may lead to varied perspectives on language implementation and optimization tools like Truffle/Graal and PyPy. Additionally, further analysis may lead to new strides into the development of a practical partial evaluator that can effectively produce the Futamura Projections.

References

[1] Y. Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[2] Y. Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Systems Computers Controls*, 2(5):54–50, 1971.

[3] A.P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977. DOI: 10.1016/0020-0190(77)90078-3.

[4] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[5] R. Glück. Is there a fourth Futamura projection? In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 51–60, New York, NY, USA, 2009. ACM Press. DOI: 10.1145/1480945.1480954.

[6] N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996. DOI: 10.1145/243439.243447.

[7] P.J. Thiemann. Cogen in six lines. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 180–189, New York, NY, 1996. ACM Press. DOI: 10.1145/232627.232647.

[8] D.P. Friedman and M. Wand. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, third edition, 2008.

[9] L.O. Andersen. Partial evaluation of C and automatic compiler generation. In *Proceedings of the Fourth International Conference on Compiler Construction*, pages 251–257, London, UK, 1992. Springer-Verlag.

[10] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variable and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991. DOI: 10.1016/0167-6423(91)90002-F.

[11] T.Æ. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T.P. Clement, editors, *Proceedings of 1992 International Workshop on Logic Program Synthesis and Transformation (LOPSTR)*, pages 214–227. Springer London, London, UK, 1993. DOI: 10.1007/978-1-4471-3560-9_15.

[12] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pages 123–132, New York, NY, 2014. ACM Press. Also appears in *ACM SIGPLAN Notices*, Vol. 50(3):123–132, 2015.

[13] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 187–204, New York, NY, 2013. ACM Press. DOI: 10.1145/2509578.2509581.

[14] PyPy Speed Center. `http://speed.pypy.org`. Last Accessed: March 17, 2017.

[15] C.F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the Fourth Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, pages 18–25, New York, NY, 2009. ACM Press. DOI: /10.1145/1565824.1565827.

[16] S. Marr and S. Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 821–839, New York, NY, 2015. ACM Press. Also appears in *ACM SIGPLAN Notices*, Vol. 50(10):821–839, 2015.