Principles of Staged Static+Dynamic Partial Analysis

Aditya Anand[ORCID] and Manas Thakur[✉][ORCID]

Indian Institute of Technology Mandi, Kamand, India
ud21002@students.iitmandi.ac.in, manas@iitmandi.ac.in

Abstract. In spite of decades of static-analysis research behind developing precise whole-program analyses, languages that use just-in-time (JIT) compilers suffer from the imprecision of resource-bound analyses local to the scope of compilation. Recent promising approaches bridge this gap by splitting program analysis into two phases: a static phase that identifies interprocedural dependencies across program elements, and a dynamic phase that resolves those dependencies to generate final analysis results. Though this approach is capable of generating precise analysis results without incurring analysis cost in JIT compilers, such "staged analyses" lack a theoretical backing. In particular, it is unclear if one could transform a general whole-program analysis (that resolves dependencies across all program elements) to a staged one that involves evaluation of statically generated partial results later. Similarly, it would be interesting if one could generate such "partial-result evaluators" in a way that can also be used to argue about their correctness. In this paper, we propose a novel model of static+dynamic partial analysis that addresses all these points, based on the classic theory of partial evaluation.

We begin by shedding light on the enigmatic idea of partial evaluation as well as the associated notion of Futamura projections to generate specialized program interpreters. We then describe *partial analysis* as the process of evaluating dependencies across program elements with respect to the statically available parts of a program, resulting into *partial results*. Next, we devise a strategy (by deriving a novel notion of *AM projections* from Futamura projections) to statically generate specialized evaluators that can process partial results using dynamic dependencies, at runtime. Later, we use our proposed model to straightforwardly establish the correctness and precision properties of the idea of staging, independent of the program analysis under consideration. We demonstrate the applicability of our model by showcasing examples from non-trivial Java program analyses, implementing the pipeline for one of them, and also discussing future possibilities to extend the same. We believe that our contributions in formulating this theory of partial analysis will significantly extend the usage of existing partial analyzers, as well as promote the design of new ones, for and even beyond Java.

Keywords: Staged analysis · Partial evaluation · Partial analysis

1   Introduction

A lion's share of research in the programming language community is focused on devising novel compilation technologies for performance. In order to generate a performant binary, compilers for various programming languages perform a series of program analyses and optimizations on the program being compiled. The quality of the optimizations performed depends on the precision of the underlying program analyses. The holy grail in the space of precise program analyses is the ability to analyze the whole program. However, in case of languages such as Java and C#, where the complete program is available only during run-time (e.g., in Java Virtual Machines), performing whole-program analysis during just-in-time (JIT) compilation is prohibitively expensive. On the other hand, owing to the separate compilation assumption [2], it is possible to "partially analyze" various parts of the program statically.

Partial analysis is a program analysis technique used in compilation systems where the whole program is not available for analysis [8,9,18,28]. Traditionally, this implies generating analysis results without the ability to incorporate the effects of the unavailable parts of the program, which again loses precision. However, for languages like Java and C# where program translation is spread across static and JIT compilation, it is a promising idea to "stage" the program analysis itself across the static and the dynamic phases of compilation. Recent approaches such as the PYE framework [28] use this idea to statically generate "dependencies" from various elements of the known program to the unknown parts of the program, which are then resolved during JIT compilation. As an example, consider the Java code snippet shown in Fig. 1; say the object(s) allocated at line l are represented using the abstract object O_l. Here, what happens to the object O_3 depends on what happens to the respective first parameters in methods `A.bar` and `B.bar`. Assuming all the code of class `A` is available statically whereas that of class `B` is available only during run-time, we can resolve the dependencies related to `A.bar` statically, but for `B.bar` only during run-time. The idea behind staging is to generate such dependencies, resolve them as much as possible statically, and then complete the analysis results by resolving the residual dependencies during run-time. A point worth noting though is that this promising approach has stark similarities with the idea of *partial evaluation*.

Partial evaluation [14] is a well-known program optimization technique that specializes a given program with respect to its statically available inputs. The resultant partially evaluated program can later be executed with the dynamic inputs, to generate the final output. The advantage of performing partial evaluation is that the specialized program often executes faster compared to executing the original program provided both static and dynamic inputs together. Apart from specializing a program with its static inputs, partial evaluation has also been used to specialize interpreters and their generators, based on the notion of Futamura projections [12]. In this paper, drawing inspirations from the theory of partial evaluation, we devise a model to stage the process of obtaining the results of a whole-program analysis, by staging the same into static and dynamic components, independent of the analysis being performed.

```
1 class A {
2    void foo(A a1) {
3       A a2 = new A(); // Object O₃
4       a1.bar(a2);
5    }
6    void bar(A p) {...} }

7 class B extends A {
8    void bar(A q) {...}
9 }
```

Fig. 1. A Java code snippet to demonstrate generation of dependencies.

Observe that staging a whole-program analysis based on prior evaluation of static dependencies and residual evaluation of dynamic dependencies, as noted above and as illustrated in Fig. 2, would require a special component (say a "partial-result evaluator"), which is capable of consuming dynamic inputs and completing the analysis results. An important question that begs an answer here is whether and how could one generate such special evaluators that can "process partial results". Further, in order to hold the efficiency advantages of staging, it is important that the generation of such evaluators is itself efficient and if possible, offloaded to the static compiler. The next question thence is, can we design a "generator" that efficiently generates partial-result evaluators, given the standard evaluator for a particular whole-program analysis. Finally, assuming these components exist, can we assert that the staged analysis would generate the same result as the corresponding whole-program analysis. In this paper, we answer all these questions with a strong affirmation by modeling the staging scheme based on the classic theory of partial evaluation.

We begin by formulating whole-program analysis as the process of computing and resolving dependencies of various elements on different parts of the program. Followed by this, we define partial analysis as the process of partially evaluating those dependencies with respect to the statically available parts of a program, thus generating partial results for the analysis being performed. We then devise a strategy (called *first AM projection*) to "generate" an evaluator that can process these partial results by performing residual resolution using the evaluated values of dynamically available dependencies. Later, to improve the efficiency of generating such partial-result evaluators for different analyses, we propose a series of specializations (called *second and third AM projections*). Finally, illuminating the similarities between our model of partial analysis and the theory of partial evaluation, we prove that the results generated by an analysis staged using our scheme would be the same as the ones generated by its whole-program version.

In order to validate the concepts presented in our manuscript, we implemented prototypes of the different components of our staging scheme. Specifically, we first designed a simple evaluator that could resolve a set of dependencies and generate the analysis result for a given program element. This evaluator is written in a way that in case of staged analysis (where only static dependencies can be resolved initially), it generates a partial result. We next implemented a
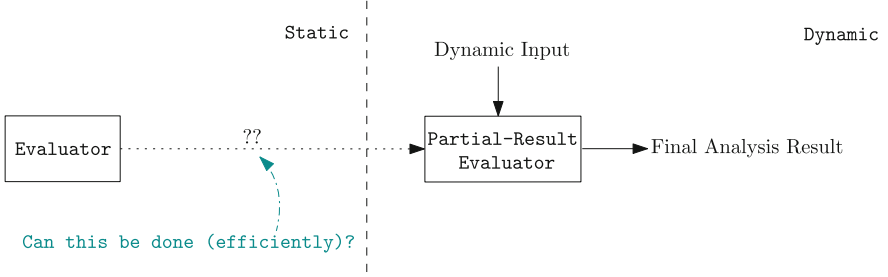
Fig. 2. Generation of partial-result evaluators.

specializer that takes the above evaluator and specializes it with the given partial result, to generate the partial-result evaluator. This partial-result evaluator can take evaluated values of dynamic dependencies as input and generate the final analysis result(s). Notably, such partial-result evaluators are independent of the way dependencies are generated for a particular analysis, agnostic of the tiered nature of modern managed runtimes, and can be invoked as soon as the values of dynamic dependencies are available. Our prototype demonstrates this by using dependencies generated for escape analysis [7,27] of Java programs, generating partial-result evaluators for abstract objects therein, and invoking them for dependencies on methods from the Java class library. Further, though the example analysis is based on method summaries, the idea of staging can be applied to other models of program analysis as well (as long as the dependencies can be categorized into static and dynamic components).

Having described partial-result evaluators and their generators in the said form (which describes standard staging), we observe that certain ways modern tiered runtimes (such as JVMs) operate raise few interesting questions. For example: "Can residual dependencies be always resolved?" "What if at certain execution points all the dynamic inputs are not available?" "In case resolution cannot proceed, can we generate a more precise value than falling back to the most conservative solution?" We discuss possible directions to address these questions, along with drawing connections with few other ways of performing program analysis, as interesting future extensions to the foundational model of staged static+dynamic partial analysis proposed in this paper.

Contributions:

- We explain partial evaluation and Futamura projections in context of speeding up the execution of programs and of the generation of program translators, in a lucid and easy-to-understand manner.
- We formalize the definition of partial analysis and devise a scheme to stage whole-program analyses into static and dynamic components, along with a novel notion of AM projections.
- We establish the correctness and precision properties of our staging scheme, based on results from the theory of partial evaluation.

– We validate the presented concepts by implementing a prototype that generates partial-result evaluators independent of the analysis under consideration.

The rest of the paper is organized as follows. In Sect. 2, we give an overview of relevant concepts from an existing staging framework required for further reading of the paper. In Sect. 3, we describe partial evaluation and Futamura projections in a readily comprehensible manner. We then present our staging scheme for whole-program analysis, along with the AM projections for generating partial-result evaluators, in Sect. 4. We describe the details of our prototype implementation to validate the presented concepts in Sect. 5. In Sect. 6, we highlight few of the challenges posed by contemporary programming-language runtimes, possible ways to deal with the same, as well as connections of our staging scheme with few other ways of performing program analysis. Finally, we discuss related work in Sect. 7, and conclude the paper in Sect. 8.

2   Background: The PYE Framework

To address the problem of imprecision of program analysis in JIT compilers, Thakur and Nandivada [28] propose a two-step solution called the PYE (Precise Yet Efficient) framework. PYE uses the concept of partial analysis [9] to generate partial results for all the statically analyzable parts of a program, and uses those results during run-time to generate the final result. In order to account for the unavailability of libraries while analyzing applications (and vice-versa) without losing precision, PYE generates dependencies across the elements of a program as *conditional values* statically, and evaluates them during run-time. We next describe the generation and evaluation of such conditional values, along with a representation that fits in with the notations that we use throughout this paper.

2.1   Conditional Values

Given a method m in a program P, a traditional whole-program analysis ψ generates a summary f_m mapping each program element $x \in m$ in the domain $\mathcal{D}$ of the analysis to one of the values in the set of dataflow values $\mathtt{Val}$ for that analysis. Thus, $f_m(x)$ denotes the *analysis result* for the element x present in method m. As an instance, for escape analysis [7], the set $\mathcal{D}$ could consist of all the abstract objects allocated in the method m and the set $\mathtt{Val}$ could be $\{D, E\}$, denoting *DoesNotEscape* and *Escapes*, respectively.

On the other hand, let $g_m(x)$ represent the set of conditional values for a program element x present in method m. A conditional value denotes dependence on another program element, and is defined in PYE as a 3-length tuple $\langle \Theta, v, v' \rangle$, where $\Theta = \langle u, y \rangle$ represents the dependee element y in method u, and v and v' are values from the lattice $\mathtt{Val}$ of the traditional analysis. A conditional value $\langle \langle n, y \rangle, v, v' \rangle$ can be evaluated to obtain v' if the analysis result $f_n(y)$ equals v.

For example, consider the code shown in Fig. 3. If the analysis ψ being performed is escape analysis, then the set $g_{\mathtt{A.foo}}(O_4)$ of conditional values that determines the escape status of the abstract object O_4 allocated at line 4 is:

```
1  class A {
2     void foo(B b) {
3        A a1 = new A(); // Object O₃          1  class L {
4        A a2 = new A(); // Object O₄          2     // A library class
5        a1.bar(a2);                           3     void lib(A r1) {
6        L l1 = new L(); // Object O₆          4        // A library method
7        l1.lib(a2); }                         5        ...
8     void  bar(A p1) {                        6     } }
9        // no assignment to p1
10    } }
```

Fig. 3. A Java code snippet to demonstrate static+JIT analysis. Class L is a library class not available during the analysis of the application class A.

$$g_{\text{A.foo}}(O_4) = \{\langle\langle\text{A.bar}, \text{p1}\rangle, D, D\rangle, \langle\langle\text{L.lib}, \text{r1}\rangle, D, D\rangle, \\ \langle\langle\text{A.bar}, \text{p1}\rangle, E, E\rangle, \langle\langle\text{L.lib}, \text{r1}\rangle, E, E\rangle\} \tag{1}$$

Here, the set of conditional values indicates that the escape status of O_4 depends on the escape statuses of the first parameters of the methods A.bar and L.lib. The conditional values denoting dependence on class A can be resolved statically, whereas those depending on the library class L are resolved at run-time. Note that we are directly using the names of parameters (that is, p1 and r1) in the conditional values for brevity; in practice they would be placeholders representing the first parameter of the corresponding method.

2.2   Evaluation of Conditional Values

Given a set $g_m(x)$ of conditional values generated for an analysis ψ, PYE evaluates them at run-time using an *evaluator*; we denote the same as CEval_ψ (see Fig. 4). CEval_ψ takes $g_m(x)$ along with the analysis results for all the dependencies contained therein ($\text{IN}_{g_m(x)}$ in Fig. 4), and generates $f_m(x)$ as follows:

$$f_m(x) = \sqcap_{\forall \mathcal{T} \in g_m(x)} [\![\mathcal{T}]\!] \tag{2}$$

where $\mathcal{T} = \langle\langle\text{n}, \text{y}\rangle, v, v'\rangle$ is resolved as:

$$[\![\langle\langle\text{n}, \text{y}\rangle, v, v'\rangle]\!] = (f_{\text{n}}(\text{y}) == v) \ ? \ v' \ : \ \bot \tag{3}$$

$\bot$ being the most precise element in the lattice of the analysis ψ.

Using Eqs. 2 and 3, CEval_ψ can evaluate the conditional values for each program element and generate final results for the analysis ψ. Note that when a dependence in $g_m(x)$ cannot be evaluated statically, PYE takes *meet* simply as a union of the conditional values present therein.

For example, evaluating $\mathcal{T} = \langle\langle\text{A.bar}, \text{p1}\rangle, D, D\rangle$ in $g_{\text{A.foo}}(O_4)$ amounts to analyzing the method A.bar (to obtain $f_{\text{A.bar}}$), then looking up the escape status
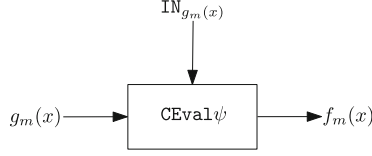
Fig. 4. Whole-program analysis.

of the abstract object pointed-to by `p1` (i.e. the escape status $f_{\text{A.bar}}(O_{p1})$, if `p` points to O_{p_1}), and then checking if it equals D; if yes, then $\mathcal{T}$ gets evaluated to D, else to the most precise element of the analysis lattice (which also happens to be D for escape analysis). Finally, after evaluating each conditional value in $g_{\text{A.foo}}(O_4)$, we can use Eq. 2 to compute the meet of the individual evaluated values, in order to obtain the final analysis result $f_{\text{A.foo}}(O_4)$.

The approach of offloading complex analyses to static time and finishing the results during run-time can be used to perform various kinds of program analyses, and has been used in the past for escape analysis to elide synchronization and points-to analysis to elide null-checks [28], and even to perform dependence analysis for parallelizing loops [25]. In general, this approach can be used for any analysis with a finite lattice (to generate a finite number of conditional values), as discussed in detail in prior work [28].

In this paper, we develop a model of staged static+dynamic partial analyses, which allows us to straightforwardly prove the correctness and precision of the idea of staging as discussed above. We start with shedding light on the enigmatic [23] theory of partial evaluation and Futamura projections in a novel way (Sect. 3), and then use it to describe our model for partial analysis (Sect. 4).

3   Partial Evaluation and Futamura Projections

Partial evaluation [14] is a program evaluation technique that specializes a program with respect to its available inputs. The specialized program can take the remaining input[1] and generate the same output as the original program. The program that specializes other programs in this manner is called a *partial evaluator* (traditionally referred to as `Mix`). This way of specializing a program `P` with respect to a statically available input in_1 to generate the specialized program P_{in_1} that can take the remaining input in_2, often speeds up the overall execution as well. Thus, if the time taken by `Mix` to specialize `P` is $\text{T}_{\text{Mix}}(\text{P}, \text{in}_1)$, the time taken by P_{in_1} to generate the final output is $\text{T}_{\text{P}_{\text{in}_1}}(\text{in}_2)$, and the time taken by the original program `P` to generate the output in a single run is $\text{T}_{\text{P}}(\text{in}_1, \text{in}_2)$, then partial evaluation is often advantageous, as:

$$\text{T}_{\text{Mix}}(\text{P}, \text{in}_1) + \text{T}_{\text{P}_{\text{in}_1}}(\text{in}_2) < \text{T}_{\text{P}}(\text{in}_1, \text{in}_2)$$

---

[1] Note that at the machine level, there is an interpreter that actually executes the program along with its input; we are simply avoiding verbosity here.
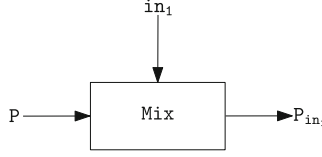
Fig. 5. Partial evaluation of program P using in_1.

In context of just-in-time (JIT) compilers, the time spent in performing program analysis gets added to the execution time of the program, thus making whole-program analysis during JIT compilation practically infeasible. Consequently, JIT compilers resort to very imprecise (e.g., intraprocedural) analyses. Thus, motivated by the possible efficiency advantages of partial evaluation, a promising way to obtain whole-program analysis results efficiently during JIT compilation is to perform partial analysis of the statically available program, and then complete the partial results during JIT compilation. In order to establish that this way of staging whole-program analysis across static and JIT compilation is correct, in this paper, we formalize a theory of partial analysis based on the prior theory of partial evaluation.

We now present an intuitive formulation of partial evaluation, along with the projections proposed by Futamura [12] to describe the generation of various partial evaluators; we extend this formulation to partial analysis in Sect. 4.

3.1   Partial Evaluation

Consider the partial evaluation scheme shown in Fig. 5. For a given program P and its available input in_1, the partial evaluator Mix generates the residual program P_{in_1}. This partially evaluated program, when given the remaining input in_2, yields the same result as running the original program on all of the inputs:

$$\llbracket P_{in_1} \rrbracket(in_2) = \llbracket P \rrbracket(in_1, in_2)$$

where $\llbracket P_{in_1} \rrbracket(in_2)$ denotes the evaluation of P_{in_1} with in_2 as input, and $\llbracket P \rrbracket(in_1, in_2)$ denotes the evaluation of P with in_1 and in_2 as inputs. The idea behind partial evaluation is that if the input in_2 changes more frequently than the input in_1, then evaluating the partially evaluated program P_{in_1} on in_2 will be faster than evaluating P on the complete input.

Partial evaluation can also be used to generate specialized versions of higher levels of abstraction in the program translation ecosystem. For example, an interpreter is a program that takes other programs along with their inputs and generates the output for those programs. "What if we use the idea of partial evaluation to specialize an interpreter with respect to a given input program? We get a faster interpreter for that program!" This specialization was described by Futamura as the first Futamura projection (FP), as discussed next.
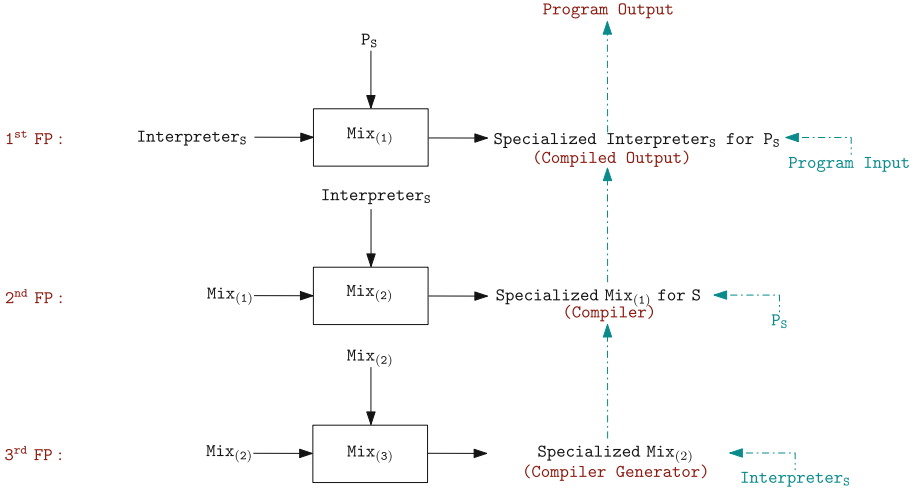
Fig. 6. Futamura projections in partial evaluation. Note that the three `Mix`es are the same specializers; we have added subscripted numbers for brevity in referencing them.

3.2 First Futamura Projection

The first Futamura projection describes how to specialize an interpreter for a given source program; see Fig. 6 (1st FP). Here, the partial evaluator ($\text{Mix}_{(1)}$) essentially applies the interpreter for a language S to a given source program P_S and generates a specialized interpreter, as illustrated by the equation below:

$$[\![\text{Mix}_{(1)}]\!](\text{Interpreter}_S)(P_S) = \text{Specialized Interpreter}_S \text{ for } P_S$$

The generated specialized interpreter can directly take the inputs of the program for which it was specialized and produce the final output. Observe that the behavior of the specialized interpreter is similar to how the binary produced by a compiler (i.e. `Compiled Output`, see Fig. 6) takes the input of the source program and generates final output. "Can we use the idea of partial evaluation to generate a higher order program, which when given a source program as input, generates its compiled version faster?" This is achieved using the second Futamura projection, as discussed next.

3.3 Second Futamura Projection

The second Futamura projection describes how to specialize the specializer ($\text{Mix}_{(1)}$) used in first Futamura projection with the interpreter for a given programming language S; see Fig. 6 (2nd FP). Here, the specializer ($\text{Mix}_{(2)}$) takes the specializer itself as one of the inputs along with the interpreter, and generates a specialized $\text{Mix}_{(1)}$ for language S as the output, as shown below:

$$[\![\text{Mix}_{(2)}]\!](\text{Mix}_{(1)})(\text{Interpreter}_S) = \text{Specialized Mix}_{(1)} \text{ for } S$$

The generated specialized $\text{Mix}_{(1)}$ can directly take a program P_S in the language S as input and generate a specialized interpreter for P_S. Observe that the behavior of the specialized $\text{Mix}_{(1)}$ is similar to how a source program P_S written in a language S is compiled. Hence the output of the second Futamura projection can also be called a `Compiler` (see Fig. 6). "Can we again use the idea of partial evaluation to generate another higher order program, which when given an interpreter for programs written in S, efficiently generates a compiler for programs in S?" This is achieved using the third Futamura projection, as discussed next.

3.4   Third Futamura Projection

The third Futamura projection describes how to specialize the specializer ($\text{Mix}_{(2)}$) used in second Futamura projection with itself; see Fig. 6 (3$^\text{rd}$ FP). Here, the specializer ($\text{Mix}_{(3)}$) takes the specializer itself as both the inputs, and generates a specialized $\text{Mix}_{(2)}$ as the output, as illustrated by the equation below:

$$\llbracket \text{Mix}_{(3)} \rrbracket (\text{Mix}_{(2)})(\text{Mix}_{(2)}) = \text{Specialized Mix}_{(2)}$$

The generated specialized $\text{Mix}_{(2)}$ can directly take an interpreter for a language S as input and generate a compiler for programs written in S as the output. Hence the specialized $\text{Mix}_{(2)}$ can also be called a `Compiler Generator` for programs written in the language S. Note that it is possible to extend this idea further and describe a fourth Futamura projection to generate a compiler-generator generator, and so on.

In a nutshell, the idea of partial evaluation can be used to automatically generate specialized tools in the program translation ecosystem. Though we could not find a standard implementation of the specializer `Mix`, Jones [14] describes it as a two-phase process: first a *division prepass* classifies program inputs into `static` and `dynamic`, followed by which the division and the static inputs are used to *compress* the program, to the extent possible, statically. Further, note that though higher levels of Futamura projections do make sense, the literature finds practical use primarily of the first projection [16], and sometimes the second projection [5]. We next highlight how even staged partial analysis is similar to partial evaluation, and then come up with novel projections that allow one to stage a whole-program analysis into static and dynamic components.

4   Staged Partial Analysis

As described in Sect. 2, a promising way to avoid incurring the cost of performing precise (whole-program) analysis during JIT compilation is to first analyze the available program statically, and then complete the results when the statically unavailable (or *dynamic*) dependencies are available (could be done either during program execution in a VM, or possibly for each version of the unavailable program, i.e., libraries, ahead of time). In one way, this implies that the evaluation of conditional values for a given analysis ψ, as performed by the conditional-value evaluator CEval_ψ, has to be split across static and dynamic components.

Notation	Description
$g_m(x)$	Set of conditional values for element x in method m
$\text{IN}_{g_m(x)}$	Set of all dependencies for evaluating $g_m(x)$
CEval_ψ	A conditional-value evaluator for analysis ψ
$f_m(x)$	Final analysis result for element x in method m
S_x	Set of statically available dependencies of x
D_x	Set of dynamically available dependencies of x
$[g_m(x)]_{S_x}$	Set of conditional values specialized with S_x
Mix	Specializer program used in partial evaluation

Fig. 7. A reference to the notations used in rest of the sections.

The consequence of this splitting (or *staging*) is that the static analysis can only compute *partial results*. Such a static analysis, which works on part of the whole program, is called *partial analysis*, and the corresponding module to perform partial analysis can be called a *partial analyzer*. Subsequently, the partial results generated by the partial analyzer need to be completed by resolving the dynamically available dependencies. This in turn requires a *special evaluator* that can take partial results along with the evaluated values of dynamic dependencies, to generate final analysis results.

As one of the key contributions of this paper, we now present a novel description of the process of performing partial analysis using statically resolved conditional values, followed by a series of specializations to efficiently generate the dynamic component of the conditional-value evaluator.

4.1   Partial Analysis

Recall (from Fig. 4) that computing the final analysis result (of analysis ψ) for a program element x in a method m requires supplying the evaluated values of all the dependencies of x to the conditional-value evaluator CEval_ψ. Whereas for languages like Java, several of these dependencies might not be available statically. We now define partial analysis in context of evaluating the set of dependencies available statically; see Fig. 7 for a list of the notations used throughout this section.

Definition 1 *(Partial analysis). For a program element x in method m with a set $g_m(x)$ of conditional values, let the set S_x denote the statically available dependencies of x. Here, while trying to evaluate $g_m(x)$, if we supply S_x to the partial evaluator Mix (see Sect. 3), we get the result of partially evaluating $g_m(x)$ with respect to the dependencies present in S_x. This process can also be seen as "specializing" the set of conditional values for the statically available inputs. We formally term this specialization as "partial analysis", illustrated in Fig. 8.*

When we perform partial analysis of the statically available program, using the schema shown in Fig. 8, we obtain a specialized set of conditional val-
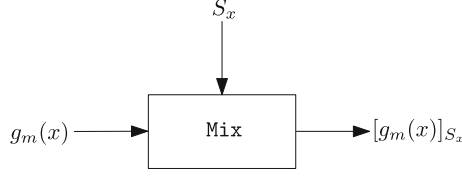
Fig. 8. Partial analysis: specializing $g_m(x)$ using the set S_x of static dependencies to obtain partial result.

ues ($[g_m(x)]_{S_x}$), which can be termed as the[2] *partial result* for the given element x. For example, in the code shown in Fig. 3, as class $\texttt{L}$ is a library class not available for partial analysis, the dependencies in $g_{\texttt{A.foo}}(O_4)$ related to $\texttt{L.lib}$ are not available statically (whereas those related to $\texttt{A.bar}$ are available, forming the set S_{O_4}). Thus, the partial result $[g_{\texttt{A.foo}}(O_4)]_{S_{O_4}}$ generated after resolving the statically available dependencies S_{O_4} in Eq. 1, can be computed as:

$$= \ \sqcap \ \{D, D, \langle\langle\texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, D, \langle\langle\texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$
$$(\because f_n(y) \neq v, so \ \mathcal{T} = \bot \ i.e. \ D)$$
$$= \ \sqcap \ \{D, \langle\langle\texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle\texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$
$$(\because D \sqcap D = D)$$
$$= \ \sqcap \ \{\langle\langle\texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle\texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$
$$(\because D \sqcap X = X) \qquad (4)$$

Given such a partial result, we need a special evaluator that can consume the runtime (dynamic) inputs to generate final analysis results for the element x. We now present a novel notion of *AM projections* that generate these special evaluators that can be used to accomplish the same.

4.2 First AM Projection

As discussed in Sect. 4.1, the output of performing partial analysis for a given program element x is a partial result (comprising of specialized conditional values $[g_m(x)]_{S_x}$). However, in order to be able to perform any optimization or transformation involving x, we need the final analysis result $f_m(x)$. Thus, we require a new evaluator that can take the partial result $[g_m(x)]_{S_x}$ as input, resolve the residual dependencies based on the evaluated values of dynamically available dependencies (say D_x), and generate $f_m(x)$. We now describe how can we generate such a "partial-result evaluator" for any program element x.

Recall the conditional-value evaluator ($\texttt{CEval}_\psi$) from Fig. 4, which, when given the set $g_m(x)$ of conditional values for an element x and the set $\texttt{IN}_{g_m(x)}$ of all dependencies of x, generates the final analysis result $f_m(x)$ for the analysis ψ. The first AM projection (see $1^{\texttt{st}}$ $\texttt{AMP}$ in Fig. 9) specializes $\texttt{CEval}_\psi$ with

---

[2] We obtain a result later (Corollary 2) which implies that this is the only possible partial result for a given set S_x of statically available dependencies.
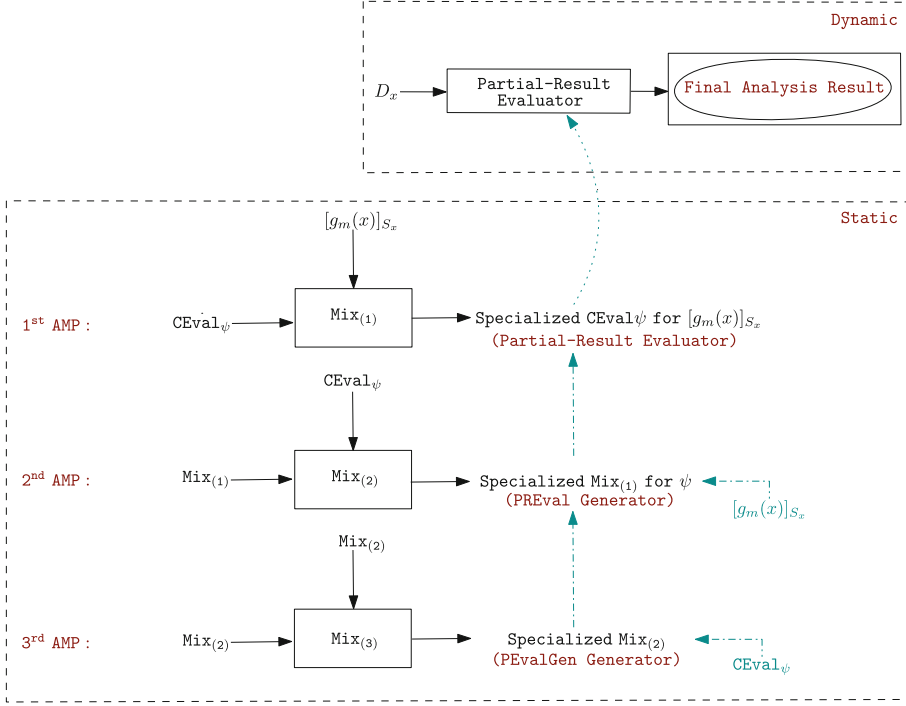
Fig. 9. AM projections in partial analysis. Note that $\mathtt{Mix}_{(1)}$, $\mathtt{Mix}_{(2)}$ and $\mathtt{Mix}_{(3)}$ are the same specializers as used in partial evaluation; also, we have added subscripted numbers for brevity in referencing them.

respect to the partial result $[g_m(x)]_{S_x}$. The output is a specialized conditional-value evaluator that can take the set D_x of dynamically available dependencies of x to generate the final analysis result $f_m(x)$. This process of specializing the conditional-value evaluator can be summarized as follows:

$$[\![\mathtt{Mix}_{(1)}]\!](\mathtt{CEval}_\psi)([g_m(x)]_{S_x}) = \mathtt{Specialized\ CEval}_\psi \text{ for } [g_m(x)]_{S_x}$$

Note that the specialized conditional-value evaluator obtained above can be used (see the $\mathtt{Dynamic}$ module in Fig. 9) to complete the staging of the whole-program analysis ψ for the program element x. As this evaluator eventually evaluates a given partial result, we name the output of the first AM projection as $\mathtt{Partial\text{-}Result\ Evaluator}$.

As an example, for the object O_4 of method $\mathtt{A.foo}$ from Fig. 3, consider the partial result obtained in Eq. 4. Once the analysis result for the library class $\mathtt{L}$ is available, the $\mathtt{Partial\text{-}Result\ Evaluator}$ takes D_{O_4} (which is formed by the analysis result available for $\mathtt{L.lib(r1)}$) and generates the final analysis result for O_4, as follows:

$$f_{\texttt{A.foo}}(O_4) = \sqcap (D, D) \quad \text{(assuming } f_{\texttt{L.lib}}(\texttt{r1}) = D, \text{ and } \because \bot = D)$$
$$= D$$

Thus, the consequence of the first AM projection is not only the fact that it is possible to derive a partial-result evaluator, but also that it can be generated statically. Further, the first AM projection also tells that for a given whole-program analysis, there also exists a component that can serve as the generator for such partial-result evaluators (which is the specializer Mix from the theory of partial evaluation). Note that the first AMP thus parallels the first Futamura projection; we next show that it is sensible to also extend the latter Futamura projections in the world of partial analysis.

4.3   Second AM Projection

Observe that the partial-result evaluator generated by the first AM projection is obtained by specializing the conditional-value evaluator for a single element in the domain of the analysis being staged. However, program analyses often generate results for multiple elements in a given program, implying that one may need to perform this specialization multiple times. To improve the efficiency of generating such specialized evaluators, we next propose a higher level of specialization, as the second AM projection.

The second AM projection (see 2nd AMP in Fig. 9) specializes the specializer $\texttt{Mix}_{(1)}$ itself with respect to the conditional-value evaluator $\texttt{CEval}_\psi$. The output is a specialized mix for analysis ψ that takes the specialized set of conditional values $[g_m(x)]_{S_x}$ for each element x and generates the specialized evaluator $\texttt{CEval}_\psi$ for that x. This process of specializing $\texttt{Mix}_{(1)}$ with $\texttt{CEval}_\psi$ can be summarized as follows:

$$[\![\texttt{Mix}_{(2)}]\!](\texttt{Mix}_{(1)})(\texttt{CEval}_\psi) = \texttt{Specialized Mix}_{(1)} \texttt{ for } \psi$$

As the output of the second AM projection can directly be used to generate the specialized $\texttt{CEval}_\psi$ for each program element x, the second AM projection is a faster way of generating the partial-result evaluator compared to the first. Hence we name the output of the second AM projection as PREval Generator. We hypothesize that though this generator would give the same partial-result evaluator as the first AM projection, one could adopt the second AM projection in case of time constraints during static compilation.

4.4   Third AM Projection

Observe that the PREval Generator obtained by second AM projection can generate the partial-result evaluator for any partial result (of the form $[g_m(x)]_{S_x}$), for a particular analysis ψ. However, typical compilers perform many different program analyses. Consequently, in order to perform multiple analyses in a staged manner, one may need to perform the specialization of $\texttt{Mix}_{(1)}$ each time with different $\texttt{CEval}_\psi$, for varying ψ. To improve the efficiency of generating

such specialized generators, we next propose a higher level of specialization, as the third AM projection.

The third AM projection (see 3rd AMP in Fig. 9) specializes the specializer $\texttt{Mix}_{(2)}$ with itself. The output is a specialized mix that can take as input the $\texttt{CEval}_\psi$ for any analysis ψ, and generate the specialized mix for that analysis ψ. This process of specializing $\texttt{Mix}_{(2)}$ with itself can be summarized as follows:

$$\llbracket\texttt{Mix}_{(3)}\rrbracket(\texttt{Mix}_{(2)})(\texttt{Mix}_{(2)}) = \texttt{Specialized Mix}_{(2)}$$

A noteworthy point is that just by providing a conditional-value evaluator for any analysis ψ, the specialized mix can directly be used to generate the PREval Generator for that analysis ψ. Hence we name the output of the third AM projection as PREvalGen Generator.

In a nutshell, the three AM projections describe ways to efficiently generate the dynamic components required for staged static+dynamic whole-program analyses. We can summarize the specializations proposed in the three projections as follows:

1. Partial-Result Evaluator
   $= \llbracket\texttt{Mix}_{(1)}\rrbracket(\texttt{CEval}_\psi, \texttt{Partial Result}) = \llbracket\texttt{PREval Generator}\rrbracket(\texttt{Partial Result})$

2. PREval Generator $= \llbracket\texttt{Mix}_{(2)}\rrbracket(\texttt{Mix}_{(1)}, \texttt{CEval}_\psi) = \llbracket\texttt{PREvalGen Generator}\rrbracket(\texttt{CEval}_\psi)$

3. PREvalGen Generator $= \llbracket\texttt{Mix}_{(3)}\rrbracket(\texttt{Mix}_{(2)}, \texttt{Mix}_{(2)})$

Thus, provided the conditional-value evaluator $\texttt{CEval}_\psi$ for a whole-program analysis ψ, a PREvalGen Generator can be used to generate a PREval Generator, which, when given a statically obtained partial result, can generate the corresponding Partial Result Evaluator, which can further be used to obtain the final analysis result given the evaluated values of dynamic dependencies. In Sect. 5, we discuss our implementation of these components using the first AM projection; the higher-order AM projections can be used to generate partial-result evaluators and their generators statically for multiple analyses.

4.5   Correctness, Precision, and Efficiency of Staging

In the previous subsections, we have seen how can we stage a whole-program analysis into static and dynamic components, based on ideas taken from the theory of partial evaluation. We now state and prove (by construction) few important properties of such a staging scheme, with respect to the correctness of staging and the precision of the results obtained, and comment on the overall efficiency of the process.

Lemma 1. *If the set of statically available dependencies is empty, then the specialization performed by the first AM projection for a conditional-value evaluator can be seen in same light as the specialization performed by the first Futamura projection for a program interpreter.*
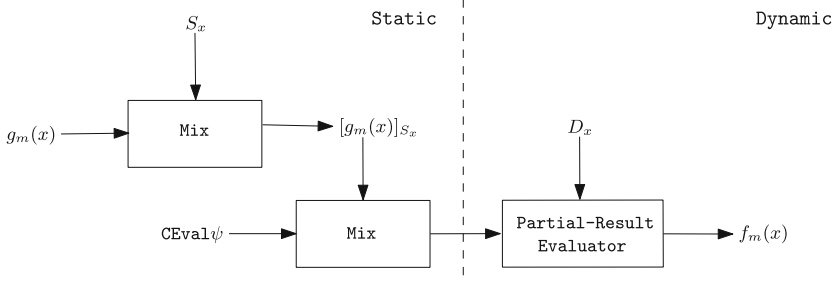
Fig. 10. A complete staging of whole-program analysis.

Proof Sketch. The first AM projection (see Fig. 9) specializes the conditional-value evaluator $\mathtt{CEval}_\psi$ with a set $[g_m(x)]_{S_x}$ of conditional values that itself is specialized with the statically available dependencies (S_x). On the other hand, if S_x is empty (that is, the program element x does not have any static dependency), then the first AM projection would specialize $\mathtt{CEval}_\psi$ just with $g_m(x)$, similar to the way the first Futamura projection specializes the interpreter (*evaluator* of programs) with a given source program $\mathtt{P_S}$ (see Fig. 6).

Corollary 1. *The Partial-Result Evaluator specialized just with $g_m(x)$ is similar to the specialized interpreter obtained by the first Futamura projection, as both need to take the complete static+dynamic input (dependencies) for generating the final output (analysis result).*

Lemma 2. *Partial evaluation of a program with a statically available input implies that the program is specialized to the extent possible (that is, maximally specialized) with respect to that input.*

Proof Sketch. A partial evaluator specializes a program P with respect to some input $\mathtt{in_1}$ by precomputing all expressions that depend on $\mathtt{in_1}$, and then folding P to the extent possible [14], that is, in a loop until fixed point. Thus, the output of a partial evaluator is *maximal* in terms of evaluation of the program for the given set of inputs.

Corollary 2. *If the first Futamura projection uses the partially evaluated program $\mathtt{P_{in_1}}$ (instead of the original program P) for specialization, then the obtained interpreter is maximally specialized with respect to $\mathtt{P_{in_1}}$.*

Theorem 1. *For a given program element and its statically available dependencies, the partial-result evaluator obtained by the first AM projection is maximal in terms of the conditional-value evaluation that can be performed statically.*

Proof Sketch. Follows from Lemma 1 and Lemma 2, for a given set of statically available dependencies passed in for generating the partial-result evaluator.

Theorem 2. *For any program element, the analysis results generated by a whole-program analysis and by the corresponding staged analysis (as summarized in Fig. 10), are the same.*

Proof Sketch. Recall the whole-program analysis schema from Fig. 4. For a given program element x, its final result $f_m(x)$ after performing a whole-program analysis ψ can be obtained by evaluating the set $g_m(\mathrm{x})$ of conditional values with respect to all the dependencies $\mathtt{IN}_{g_m(x)}$. Now consider the staging of the analysis ψ as shown in Fig. 10. Here, we have broken down the set $\mathtt{IN}_{g_m(x)}$ of dependencies into the set S_x of statically available dependencies and the set D_x of dynamically available dependencies. The static component is a specializer (`Mix`) that generates the specialized conditional-value evaluator (`Partial-Result Evaluator`) by specializing the whole-program analysis evaluator $\mathtt{CEval}_\psi$ with respect to the partial result $[g_m(x)]_{S_x}$. The generated `Partial-Result Evaluator` forms the dynamic component of the staged analysis, and generates $f_m(\mathrm{x})$ using D_x. As established by the theory of partial evaluation, the result obtained by evaluating a program with respect to all of its inputs is same as the result obtained by evaluating the specialized program with respect to its dynamic input. It can be seen by construction that the way Fig. 10 stages a whole-program analysis is similar to the way partial evaluation stages the evaluation of a program. Hence, the analysis result generated by our staged analysis would be the same as the one generated by the whole-program analysis. This equality can be summarized as follows:

$$[\![g_m(x)]\!](\mathtt{IN}_{g_m(x)}) = f_m(x) = [\![[g_m(x)]_{S_x}]\!](D_x)$$

Theorem 2 establishes the correctness and precision of the proposed staging scheme, and Theorem 1 establishes the efficiency (indicating maximal evaluation during static compilation) achieved by using the staging scheme. Observe that the proofs became straightforward due to two important illustrations: (i) that a whole-program analysis could be modeled as the evaluation of dependencies across program elements; and (ii) that for languages like Java with statically unavailable program parts, the evaluation of static and dynamic dependencies could be modeled based on the theory of partial evaluation.

We next describe our experience implementing a specializer for generating partial-result evaluators from given conditional-value evaluators for the whole program, similar to the `Mix` described by Jones in his classic book on partial evaluation. We approach this problem by describing a language of conditional values, such that programs in the world of partial analysis become sets of conditional values and program interpretation becomes conditional-value evaluation.

5   Specializers for Partial-Result Evaluation

In this section, similar to a language of programs, we first describe a language of conditional values that could be used to generate sets of conditional values (denoting dependence on various kinds of elements) for different program elements (Sect. 5.1). Next, similar to program interpreters, we design a simple

```
<Start>     ::= <ProgElem> : <CV>*
<ProgElem> ::= <Class> <Method> <Type> <Ref> <Fields>
<CV>        ::= <ProgElem> <DepVal> <ResVal>
<Type>      ::= LOCAL | PARM | ARG | RETVAL | FIELD
<DepVal>   ::= D | E
<ResVal>   ::= D | E
                        (a)
<Start>     ::= <ProgElem> : <TaggedCV>*
<TaggedCV> ::= <Tag> <CV>
<Tag>       ::= STATIC | DYNAMIC
                        (b)
```

Fig. 11. (a) The grammar for our division prepass; (b) extended grammar for conditional-value evaluation.

evaluator that could resolve those dependencies to generate the final analysis result for various program elements. We then mirror the process of specializing a program interpreter by designing a Mix that could specialize conditional-value evaluators to generate partial-result evaluators (Sect. 5.2). We also compiled and ran the generated partial-result evaluators, by supplying the statically evaluated values of Java libraries (computed using the same conditional-value evaluator discussed above); Sect. 5.3 describes our experience with the same.

Note that though our efforts are independent of the program analysis being staged, we need a set of conditional values generated for a given program analysis. In this section, we have chosen a publicly available conditional-value generator for escape analysis, Stava [27], written in Soot [29]. Stava generates a list of dependencies for each abstract object (program element), denoting its dependence on other program elements towards computing its escape status: one of *Escapes* (E) and *DoesNotEscape* (D).

5.1   A Grammar for Conditional Values

Based on the kinds of elements on which the analysis result for a given program element could depend on, Fig. 11 shows a grammar to generate sets of conditional values (denoting those dependencies) for various program elements. Each program element ProgElem belongs to a Class and a Method, and could be of one of the five Types: (i) local object in current method; (ii) parameter taken by current method; (iii) argument passed to another method; (iv) return value of current method; and (v) field of any other element. Ref is a number, denoting the line number of allocation for LOCAL, parameter and argument number respectively for PARM and ARG, and simply a filler for the rest. Fields contains a list of fields (e.g., f1, f2 to denote the element pointed to by X.f1.f2 for any abstract object X). A conditional value (CV) denotes dependence on a program element. Essentially, a conditional value <P1 X1 X2> evaluates to X2 if the element P1 resolves to X1; see Sect. 2.1). For example, for escape analysis, DepVal and ResVal could either be D or E. Thus, pairs comprising of program elements

1 Procedure CEval($g_m(x)$, $\text{IN}_{g_m(x)}$)

2 | Initialize a list L of statically known dependencies.

3 | **foreach** $d \in \text{IN}_{g_m(x)}$ **do**

4 | | Add d to L.

5 | | Add the transitive dependencies of d to L.

6 | Form strongly connected components (SCCs) in the list L.

7 | **repeat**

8 | | **foreach** *strongly connected component S formed above* **do**

9 | | | **if** $\nexists e \in S$ *s.t. e depends on another SCC* **then**

10 | | | | $\forall e \in S$, resolve e to $\perp$.

11 | | | Take a meet of the resolved values in each SCC as per the meet operation defined in Section 2.2.

12 | **until** *fixed point*;

Fig. 12. Algorithm to perform conditional-value evaluation.

and the conditional values generated for those elements (e.g., by `Stava`) are the valid members of the language generated by this grammar. (Note that we had to make cosmetic changes in `Stava` to print its output in a form that can be parsed by our conditional-value grammar.)

5.2   Conditional-Value Evaluators and Specialization

Specialization using the theory of partial evaluation uses an auxiliary "division" routine to classify program inputs as static and dynamic. Having described a language of conditional values in the previous section, we next wrote a division prepass that classifies each conditional value as static or dynamic (based on whether it denotes a library dependency while analyzing applications, and vice-versa). We have implemented the division prepass as a JavaCC [26] visitor (about 300 lines of code) over the abstract syntax trees generated for the sets of conditional values prescribing to the grammar described above. We encode the result of division by prefixing each conditional value with a tag `STATIC` or `DYNAMIC`, resulting into a set of conditional values recognizable using the extended conditional-value grammar shown in Fig. 11(b). This extended grammar describes the language of conditional values that can be evaluated by our conditional-value evaluator `CEval`$_\Psi$.

Figure 12 gives an overview of the computation performed by `CEval`$_\Psi$. The evaluator takes as input the set of conditional values $g_m(x)$ for a program element x belonging to a method m and its dependencies $\text{IN}_{g_m(x)}$, and generates the partial result $[g_m(x)]_{S_x}$. First, the evaluator transitively adds all the dependencies of the given element into a list L (lines 2–5). Next, treating the various dependencies as a graph, the evaluator forms strongly-connected components (SCCs), denoting sets of equivalent resolved values. In case no element in an SCC depends on an element from another SCC, all the elements in that SCC are resolved to the bottom (most precise value) of the lattice of the analysis

```
1    class SpecializedCode1 {
2      public static void main(String[] args) {
3          // Read the values for dynamic dependencies
4          x1 = Resolved value of <L.lib, r1> // first dependence
5          x2 = Resolved value of <L.lib, r1> // second dependence
6          res = x1 ⊓ea x2
7          print(res);
8        }
9    }
```

Fig. 13. Schema of the partial-result evaluator emitted for $g_{\texttt{A.foo}}(O_4)$.

under consideration (lines 9–10). Finally, for the program element x, the evaluator takes a meet as described in Sect. 2.2, generating partial result in case of dynamic dependencies (line 11). The last two steps (lines 8 to 11) are performed till a fixed point. Our evaluator is also implemented as a JavaCC pass over the extended conditional-value grammar (from Fig. 11(b)), and spans about 1000 lines of code. Note that the only part of the evaluator that depends on the analysis for which the conditional-values are generated is the meet operation (to access its lattice); thus, the evaluator is essentially parametric over the analysis being performed. Hence, for escape analysis (ea), we denote the evaluator as $\texttt{CEval}_{ea}$.

Next we have implemented a $\texttt{Mix}$ that takes as input our conditional-value evaluator and a partial result $[g_m(x)]_{S_x}$, and specializes the evaluator for the given partial result. Our $\texttt{Mix}$ works similar to the partial-evaluation Mix proposed by Jones [14], but for Java programs of the kind of $\texttt{CEval}_{ea}$ (we could not find any existing implementation that we could reuse). Our $\texttt{Mix}$ is implemented in JavaCC (for a grammar that covers the subset of Java required to parse $\texttt{CEval}_{ea}$), and spans about 1500 lines of code. The output of our $\texttt{Mix}$ is a partial-result evaluator for the given partial result.

The fundamental idea behind specializing the evaluator is that its code should be executed as much as possible for the static dependencies, and the residual code should take the evaluated values of dynamic dependencies as input to generate the final analysis-result. Observe that Line 11 in Fig. 12 involves taking the meet (over the lattice $\texttt{Val}$) of the resolved dependencies. However, if any dependence is dynamic, its resolution cannot be performed statically. Hence, in order to specialize the evaluator for a given partial result, we first check the kind of dependence (populated by the division prepass), and for each dynamic dependence, we emit code to read and resolve the same. Next, we emit code to perform meet over the resolved values obtained therein. Finally, we enclose the emitted code as the $\texttt{main}$ method of a uniquely named specialized-code class (say $\texttt{SpecializedCodeN}$ for a unique $\texttt{N}$), to obtain the corresponding partial-result evaluator. For example, Fig. 13 shows the schema of the code emitted by our $\texttt{Mix}$ for the element $g_{\texttt{A.foo}}(O_4)$ (see Eq. 4, Sect. 4.1).

Before we could use the partial-result evaluators generated for different program elements, we need to obtain the evaluated values of the dynamic dependen-
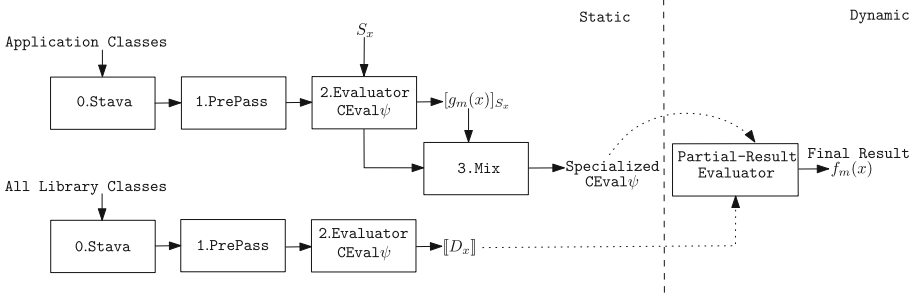
Fig. 14. Schema for prototype implementation of staged analysis

cies (libraries in case of Java). We do so by running an offline pass of $\mathtt{CEval}_{ea}$ on the libraries, and obtaining the evaluated values therein. Finally, one can supply these evaluated values to the specialized codes comprising the partial-result evaluators for various program elements, to obtain the final analysis-result. Figure 14 shows the complete scheme of our prototype implementation for generating and executing partial-result evaluators.

Observe that partial-result evaluation over the evaluated values of dynamic dependencies, as per our proposed model, can be performed by executing the specialized code as soon as the dynamic dependencies can be resolved. Thus, we have the following options:

– Simply place the partial-result evaluators in a JIT compiler.
– As the partial-result evaluators are generated statically for all the program elements, place them in a VM agnostic to the methods/code-portions that are JIT compiled.
– For the evaluated dependencies corresponding to each version of the libraries, invoke the partial-result evaluators ahead-of-time (i.e., statically itself).

The first option can be mapped to the kind of scheme adopted by staging frameworks such as PYE [28]; however, it would mean we can perform optimizations for only those elements that belong to methods/regions that are JIT compiled (usually very few in tiered runtime systems such as the HotSpot JVM [21]). On the other hand, the second option could be used to obtain analysis results in the VM irrespective of the tiered nature and mode of compilation (by implementing corresponding optimization passes). Finally and most interestingly, the third option makes the process of generating analysis results even independent of the runtime system, and can also be used to demonstrate the full impact a staging scheme could make over performing a whole-program analysis.

5.3   Running the Partial-Result Evaluators

In this section, we try to validate the observation staging schemes reduce the amount of computation one may have to perform at run-time to obtain whole-program analysis results, significantly. In order to do so, we tried to perform

a static whole-program escape analysis of a small benchmark `moldyn` from the JGF suite [10] (12 application and 3509 referenced library classes), by forcing `Stava` to analyze both application and libraries together. We found that for the whole-program analysis, though `Stava` generated conditional values for all the program elements in 4 h, their evaluation (using $CEval_{ea}$) did not terminate even in 20 h (on an Intel Xeon E5-2630 2.4 GHz system with 32 cores and 64 GB RAM, running Cent OS version 7), owing to the large number of residual dependencies (in the order of tens of thousands).

On the other hand, using the staged scheme (where we evaluate application and library dependencies separately), `Stava` took ∼30 s to generate the conditional-values for the application code, the division prepass (Step 1 in Fig. 14) took just a second, the evaluation of statically available dependencies (Step 2) took ∼3 min, the generation of partial-result evaluators (Step 3) took about a minute – all performed statically. Correspondingly, for the library classes referenced to by `moldyn`, Stava took ∼30 min, division and the evaluation took ∼90 min each – again performed statically. Finally, the execution of partial-result evaluators to generate final analysis-results, given the partial results for `moldyn` and the evaluated values of libraries – that is, the computation that needs to be performed dynamically – took only ∼35 s.

Thus, we can notice that a staged scheme may not only allow one to obtain the results of otherwise infeasible whole-program analysis during run-time, but also do so efficiently, that is, by reducing the actual amount of computation to be performed during run-time significantly. However, we looked into the list of dynamic dependencies generated for various program elements and found a scope to improve this time even further. In particular, we found that multiple program elements shared several dependencies; we attribute this to the fact that multiple parts of a Java application might use common library methods (for example, multiple objects being passed to the method `add` of `java.util.ArrayList`). To leverage this commonality, we modified our `Mix` to enclose the code generated for each partial-result evaluator in a separate function, and to concatenate all those functions in a single `PartialResultEvaluator` class. To obtain the final analysis results for all the program elements, we invoke the individual partial-result-evaluator functions present therein, using the Java reflection API; this reduced the time required to a mere ∼4 s.

Noting the improvement in the potential time required during run-time using the staged approach, we conclude that staging partial analyses into static and dynamic components, backed by a theoretical model presented in our paper, not only opens up avenues in languages with managed runtimes to perform existing optimizations more aggressively than present, but can also be used to perform novel analyses and optimizations that were otherwise practically infeasible.

6   Directions and Connections

Having drawn parallels between the classic theory of partial evaluation and a promising way to stage program analyses into static and dynamic components,

we now turn our attention to various recent features and challenges pertaining to contemporary programming languages. In particular, we address topics of interest that could lead to further work built upon our model of partial analysis for languages with static and dynamic compilers. We also discuss connections with few other ways of performing program analysis in managed runtimes. In effect, we believe this discussion would be useful in driving multiple future directions, not only for the Java world but also for the wider community interested in.

6.1   Runtime Features: Challenges and Possibilities

In this section, we discuss few challenges posed by modern static+dynamic compilation systems, and discuss the kinds of techniques that could be used to improve the precision of the analysis results obtained therein.

As described in Sect. 2 (see Eqs. 2 and 3), standard resolution and conditional-value evaluation assume that all the dependencies can be resolved at run-time (that is, *closed world assumption*). However, there could be scenarios because of the way tiered JIT compilation works in modern JVMs where the results for few dependencies may not be available at the point of need. For example, if the program being analyzed uses reflective calls, state-of-the-art static-analysis frameworks (such as Soot [29]) miss edges in the call-graph for the program, as a result of which few methods may not get analyzed altogether. On another note, features such as dynamic classloading [15] in JVMs may bring up classes in a statically non-deterministic order. For example, in the partial result for element O_4 (see Eq. 4), if the class L is not loaded when the corresponding evaluation is invoked, the conditional values dependent on L1.lib cannot be resolved. The only possible sound option in this case would be to take the least precise value E as the resolved value.

On the other hand, for analyses such as control-flow analysis that have a richer lattice, it is possible to improve the precision for unresolvable dependencies. Consider the code snippet shown in Fig. 15. In method A.foo, the set of conditional values representing the possible types of objects that can be pointed-to by the reference variable z is $\{\langle\langle \text{A.foo}, \text{b}\rangle, \text{B}, \text{Z1}\rangle, \langle\langle \text{A.foo}, \text{b}\rangle, \text{C}, \text{Z2}\rangle\}$. Under the existing evaluation scheme, where all the required dependencies are available, the type of z will get assigned to either Z1 if b's type is B, or to Z2 if b's type is C. However, say in presence of dynamic classloading, if the class C has not been loaded when the conditional values for z need to be evaluated, then assuming the least precise value of the lattice (which, for control-flow analysis, is the set of all types in the program) is highly conservative and may affect many other optimizations such as method inlining and virtual-call resolution. We now discuss a solution to address this problem.

While performing partial-result evaluation, in case a certain dependency cannot be resolved, instead of always falling back to the least precise value in the lattice, we can statically generate some "fallback values" that can be used as the fallback option during resolution. Observe that the set of possible values that can be obtained as the analysis result for a given program element x, can be formed only from the third elements (say resolution values) of the tuples present

```
 1 class A {                          11 class B {
 2    void foo(B b) {                 12    Z bar() {
 3       Z z = b.bar();               13       return new Z1() ;
 4    }                               14    }
 5 }                                  15 }
 6                                    16 class C extends B {
 7 interface Z {...}                  17    Z bar() {
 8 class Z1 implements Z { ... }      18       return new Z2();
 9 class Z2 implements Z { ... }      19    }
10 class Z3 implements Z { ... }      20 }
```

Fig. 15. A Java code snippet to demonstrate control-flow analysis.

in the set of conditional values for x. Thus, we can obtain a fallback value $fb(x)$ by taking the meet of all such resolution values:

$$fb(x) = \sqcap_{\forall \mathcal{T} \in g_{m(x)}} \mathcal{T}[3]$$

where $\mathcal{T}[3]$ gives the third element in each conditional value. As an example, for the reference variable z in Fig. 15, $fb(z)$ would be the set $\{Z1, Z2\}$.

In order to support fallback values, the staging scheme presented in Sect. 4 (see Fig. 10) can be modified as follows. In the static component, for each element x, apart from the partial result $[f_m(x)]_{S_x}$, we additionally need to store $fb(x)$. On the other hand, in the dynamic component, we can modify the partial-result evaluation as:

$$f_m(x) = \sqcap_{\forall \mathcal{T} \in g_{m(x)}} [\![\mathcal{T}]\!]$$

$$[\![\langle \langle \mathbf{n}, \mathbf{y} \rangle, v, v' \rangle]\!] = (is\_available(f_{\mathbf{n}}(y))) \; ? \; ((f_{\mathbf{n}}(\mathbf{y}) == v) \; ? \; v' \; : \; \bot) \; : \; fb(x)$$

Here, for a given element x, while trying to resolve a conditional value $\mathcal{T} = \langle \langle n, y \rangle, v, v' \rangle \in g_m(x)$, we first check if $f_n(y)$ is available; if yes, we proceed with normal resolution, else we use $fb(x)$ as the fallback value. Thus, for the example shown in Fig. 15, even if the runtime has no information about the caller of $\mathtt{A.foo}$ (that is, no knowledge about the type of the objects pointed-to by $\mathtt{b}$), using statically computed $fb(z)$, we can resolve the conditional values for z generated above to obtain the set $\{Z1, Z2\}$ as the analysis result for z.

Apart from the challenges posed by runtime features discussed above, another important consideration for static+dynamic analyses is to guarantee/verify that the static-analysis results correspond to the bytecodes being executed. In case of a difference, one may need to invalidate the partial-result evaluator for the corresponding and dependent methods. This can at a simpler level be done by maintaining a list of affected methods with the statically resolved dependencies, and be improved by precisely identifying the effect on various program elements. We believe this to be an interesting future research direction.

6.2   Drawing Newer Connections

In this section, we first discuss few interesting aspects related to cross-pollination of ideas between Futamura and AM projections, along with few subtle points related to generation of conditional values in partial analysis. We then highlight how our staged analysis scheme can be used along with various other applications involving static and dynamic analyses.

1. Cross-pollination of Specialization Ideas. We have mentioned previously that AM projections are similar to yet different from Futamura projections. To elucidate this point, note that the partial-result evaluator generated by the first AM projection is a result of specializing the conditional-value evaluator for an already specialized set of conditional values (see Fig. 9). On the other hand, the output of the first Futamura projection is a result of specializing the interpreter for the original source program (see Fig. 6). It is possible to take cue from the first AM projection and modify the first Futamura projection to specialize the interpreter too with a partially evaluated program. Doing so would generate a faster interpreter, as the input program can anyway be specialized with the statically available input. Similarly, the compiler generated by the second Futamura projection can also take a specialized program as input to efficiently generate the specialized interpreter obtained in the first Futamura projection.

As another possibility to explore in the space of specialization, it can be seen in Sect. 4 (Figs. 4 and 10) that the input taken by our model of whole-program as well as partial analyses is the set of conditional values, denoting dependencies, for a given program element. It is possible to visualize the process of generating these conditional values: from a given program analysis specified as an abstract interpreter, we can identify the set of statements required to compute the final analysis result for a particular program element x, as the dependencies of x. This process of generating conditional values can be made faster: one could model program analyses as "conditional-value generators", and then specialize them with respect to a given program, similar to the specialization of conditional-value evaluators done by AM projections. Also note that though this "per element" modeling is a bit different from the way usual iterative dataflow analyses [20] are implemented (as aggregate transfer functions over all the variables in the domain), it fits well with various recently popular ways of writing program analyses, as discussed next.

2. Query- and Feedback-Driven Analyses. In general, whole-program analyses generate results for all the program elements in all the methods of a program. On the other hand, one of the growingly popular ways to scale precise analyses in resource-constrained environments is to compute information only for elements that are of interest, often specified as a set of queries generated by various client optimizations [13,24]. These analyses are called "query-driven analyses". Staged schemes of the kind proposed in this paper can be integrated directly with such analyses: First, the client can generate the list of program elements that are of interest for the query under consideration, based on which the partial analysis can generate the relevant sets of conditional values. After-

wards, our staging scheme can be used to specialize the corresponding set of conditional values and further generate the `Partial-Result Evaluator` only for the elements of interest.

For languages that support static+dynamic compilation, one of the ways to improve the outcomes produced by static analyses is to perform profiling in the dynamic compiler and give feedback to the static compiler [6,11,30]. As an instance, Bastani et al. [6] make optimistic assumptions for the unavailable portions of a program, detect during runtime if an assumption goes wrong, abort execution if it does, and then refine the static analysis accordingly; this process is repeated until no assumptions fail. The staged scheme of our kind suits such analyses particularly well: the feedback from runtime can be used to obtain the list of affected elements that need to be re-specialized by the static analysis for subsequent runs of the re-created partial evaluator, thus requiring the (partial) static analysis to be re-performed only for the affected elements.

7   Related Work

In this section, we discuss relevant related work in four categories: (i) partial evaluation; (ii) partial program analysis; (iii) other applications of partial evaluation; and (iv) staged analysis. To the best of our knowledge, ours is the first scheme that maps the staging of whole-program analysis across static and dynamic compilation to the theory of partial evaluation.

7.1   Partial Evaluation

Partial evaluation is a well-known technique to specialize programs with statically available inputs. Jones [14] formalized the theory of partial evaluation in context of constructing compilers and compiler generators by specializing subsequent levels of interpreters, using Futamura projections [12]. Perugini and Williams [23] underlined the difficulty in understanding partial evaluation and Futamura projections, and devised a diagrammatic approach to visualize the working of partial evaluation, by modeling program execution using a box-substitution notation. In this paper, we have also tried to explain the three Futamura projections, particularly by showing the connections among the outputs and inputs of the subsequent projections. Our goal behind this visualization is to later build a mapping from our proposed model of partial analysis, to generate partial-result evaluators.

7.2   Partial Program Analysis

The idea of analyzing partial programs was first formalized in a tool called PPA [9], wherein the goal was to infer types for incomplete Java programs. In presence of ambiguities, PPA uses heuristics based on the structure of a program to generate imprecise but sound results. Similarly, Melo et al. [18] generate missing type annotations for incomplete C programs, while handling challenges

imposed by C's weak type system using "placeholder pre-types". In presence of ambiguities, Melo et al. fill the placeholder with an "orphan" type in the lattice of pre-types. Our staged scheme, though performs an analysis of the partial program to generate partial results statically, is able to generate sound as well as precise answers based on dynamic inputs, wherein the components needed to evaluate and complete partial results are generated statically, using novel AM projections based on the theory of partial evaluation.

There have been prior works [19, 22, 32] that generate results for incomplete programs by trying to obtain possible solutions based on examples and then ranking them for suitability. The limitation of these techniques is that they may generate unsound results. Our staged scheme, on the other hand, would always generate sound results, with varying precision based on the dynamic features supported by a given runtime.

Allen et al. [4] present a scheme to analyze Java libraries in absence of application code. They model concrete objects using allocation sites, while approximating unknown objects using static types, thus generating a combined lattice. Our approach, though different in the sense that it works for both application and library code, uses the idea of using static types as fallback values in absence of dynamic inputs. On the other end of the spectrum, Ali and Lhoták [3] generate call-graphs to analyze Java application code in absence of libraries, by approximating library methods as stubs. In comparison, our staging approach, instead of approximating the libraries, uses their analysis results (obtained offline) to complete application results during dynamic compilation.

7.3   Other Applications of Partial Evaluation

There have been works that use partial evaluation to speed up different parts of a program's execution lifecycle. One such implementation [16] speeds up the execution of code during JIT compilation by specializing the AST interpreter for a given language specified in the Truffle [31] framework. This avoids redundancy in code generation during JIT compilation. Marr and Ducasse [17] compare the performance of the previous approach with that of tracing JIT compilation (which optimizes a program by JIT-compiling traces obtained by profiling). On the other hand, in this paper, we have used the idea of partial evaluation to speed up the process of obtaining whole-program analysis results (possible during or just before JIT compilation). Our scheme can be augmented to the Truffle approach by performing partial analysis of the available program during AST specialization, and refining the results during JIT compilation.

7.4   Staged Analysis

Staging, though a general idea, has not often found place in static+dynamic analysis systems. Chug et al. [8] compute and check information-flow properties for Javascript programs statically, while leaving residual checks that depend on dynamic inputs for the runtime. Albarghouthi et al. [1] develop specifications for unknown methods in context of program synthesis. In context of Java, Thakur

and Nandivada [28] recently proposed the PYE framework, which uses the idea of conditional values to denote dependencies on dynamic input and resolves them to generate final results during JIT compilation. In this paper, we have proposed a general staging framework based on the theory of partial evaluation that can be used not only to model both the above works, but also to establish and prove the correctness and precision of the same. Importantly, our base scheme (Sect. 4) is independent of the language and framework under consideration, and the extensions for Java runtimes (Sect. 6) can be used to devise corresponding strategies for other runtimes with novel dynamic features.

8   Conclusion

In this paper, we presented a formal model to stage whole-program analyses into static and dynamic components. Our staging scheme took inspiration from the theory of partial evaluation and specialized the evaluators for whole-program analysis with partial results to generate partial-result evaluators. Similarly, based on the notion of Futamura projections for partial evaluation, we proposed a novel notion of AM projections that describe the generation of partial-result evaluators and their generators. The generated partial-result evaluators can also be placed in managed runtimes to generate final analysis results using the dependencies that become available dynamically. This model allowed us to establish the correctness and precision of the idea of staging in a straightforward manner. Moreover, in order to address the challenges presented by the dynamic nature of modern tiered runtimes, we also discussed possible future directions to extend the staging scheme further. To the best of our knowledge, ours is the first scheme that backs the staging of whole-program analysis into static and dynamic components, using the established theory of partial evaluation. We envisage that our formulated theory of partial analysis would be used not only to promote the design of staged partial analyzers, but to also perform erstwhile infeasible optimizations, for and beyond Java.

References

1. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 2016, pp. 789–801. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2837614.2837628
2. Ali, K.: The Separate Compilation Assumption. Ph.D. thesis, University of Waterloo, Waterloo, Ontario, Canada (2014)
3. Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 378–400. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_16
4. Allen, N., Krishnan, P., Scholz, B.: Combining type-analysis with points-to analysis for analyzing Java library source-Code. In: Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis. SOAP 2015, pp. 13–18. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2771284.2771287

5. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed OO languages. In: Proceedings of the 2007 Symposium on Dynamic Languages. DLS 2007, pp. 53–64. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1297081.1297091

6. Bastani, O., Sharma, R., Clapp, L., Anand, S., Aiken, A.: Eventually sound points-to analysis with specifications. In: Donaldson, A.F. (ed.) 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 11:1–11:28. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.11

7. Blanchet, B.: Escape analysis for JavaTM: theory and practice. ACM Trans. Program. Lang. Syst. **25**(6), 713–775 (2003). https://doi.org/10.1145/945885.945886

8. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged information flow for Javascript. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2009, pp. 50–62. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1542476.1542483

9. Dagenais, B., Hendren, L.: Enabling static analysis for partial Java programs. SIGPLAN Not. **43**(10), 313–328 (2008). https://doi.org/10.1145/1449955.1449790

10. Daly, C., Horgan, J., Power, J., Waldron, J.: Platform independent dynamic Java virtual machine analysis: the Java grande forum benchmark suite. In: Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande. JGI 2001, pp. 106–115. ACM, New York (2001). https://doi.org/10.1145/376656.376826

11. Dean, J., Chambers, C., Grove, D.: Selective specialization for object-oriented languages. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation. PLDI 1995, pp. 93–102. Association for Computing Machinery, New York (1995). https://doi.org/10.1145/207110.207119

12. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Higher Order Symbol. Comput. **12**(4), 381–391 (1999). https://doi.org/10.1023/A:1010095604496

13. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. PLDI 2001, pp. 24–34. Association for Computing Machinery, New York (2001). https://doi.org/10.1145/378795.378802

14. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. **28**(3), 480–503 (1996). https://doi.org/10.1145/243439.243447

15. Kotzmann, T., Mössenböck, H.: Escape analysis in the context of dynamic compilation and deoptimization. In: Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments. VEE 2005, pp. 111–120. ACM, New York (2005). https://doi.org/10.1145/1064979.1064996

16. Latifi, F.: Practical second futamura projection: partial evaluation for high-performance language interpreters. In: Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. SPLASH Companion 2019, pp. 29–31. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3359061.3361077

17. Marr, S., Ducasse, S.: Tracing vs. partial evaluation: comparing meta-compilation approaches for self-optimizing interpreters. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015, pp. 821–839. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2814270.2814275

18. Melo, L.T.C., Ribeiro, R.G., de Araújo, M.R., Pereira, F.M.Q.A.: Inference of static semantics for incomplete C programs. Proc. ACM Program. Lang. **2**(POPL) (2017). https://doi.org/10.1145/3158117
19. Mishne, A., Shoham, S., Yahav, E.: Typestate-based semantic code search over partial programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA 2012, pp. 997–1016. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2384616.2384689
20. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, Burlington (1997)
21. Paleczny, M., Vick, C., Click, C.: The Java hotspot server compiler. In: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium. JVM 2001, vol. 1, p. 1. USENIX Association, USA (2001)
22. Perelman, D., Gulwani, S., Ball, T., Grossman, D.: Type-directed completion of partial expressions. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2012, pp. 275–286. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2254064.2254098
23. Perugini, S., Williams, B.: Revisiting the Futamura projections: a diagrammatic approach. Theor. Appl. Inform. **28**, 15–32 (2017). https://doi.org/10.20904/284015
24. Rama, G.M., Komondoor, R., Sharma, H.: Refinement in object-sensitivity points-to analysis via slicing. Proc. ACM Program. Lang. **2**(OOPSLA), 142:1–142:27 (2018). https://doi.org/10.1145/3276512
25. Sharma, R., Kulshreshtha, S., Thakur, M.: Can we run in parallel? Automating loop parallelization for TornadoVM (2022). https://doi.org/10.48550/arXiv.2205.03590
26. Succi, G., Wong, R.: The application of JAVACC to develop a C/C++ preprocessor. ACM SIGAPP Appl. Comput. Rev. **7**, 11–18 (1999). https://doi.org/10.1145/333630.333633
27. Nikhil, T.R., Yadav, D., Thakur, M.: Stava (2021). https://github.com/CompL-IITMandi/stava
28. Thakur, M., Nandivada, V.K.: PYE: a framework for precise-yet-efficient just-in-time analyses for Java programs. ACM Trans. Program. Lang. Syst. **41**(3), 16:1–16:37 (2019). https://doi.org/10.1145/3337794
29. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON 1999, pp. 13–23. IBM Press (1999). http://dl.acm.org/citation.cfm?id=781995.782008
30. Vivien, F., Rinard, M.: Incrementalized pointer and escape analysis. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2001, pp. 35–46. ACM, New York (2001). https://doi.org/10.1145/378795.378804
31. Würthinger, T., et al.: Practical partial evaluation for high-performance dynamic language runtimes. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2017, pp. 662–676. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3062341.3062381
32. Zhong, H., Wang, X.: Boosting complete-code tool for partial program. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE 2017, pp. 671–681. IEEE Press (2017)