Principles of Staged Static+Dynamic Partial Analysis

29th Static Analysis Symposium

Aditya Anand and Manas Thakur

Indian Institute of Technology Mandi

December 7th, 2022

Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.

Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.
- Performing precise program-analysis during JIT Compilation – highly inefficient.

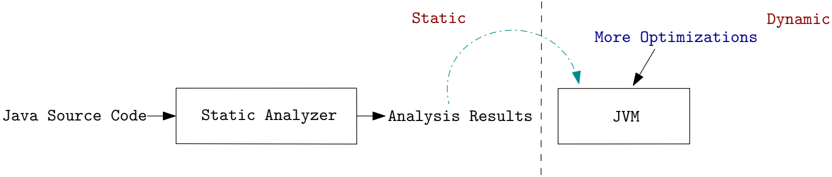
Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.
- Performing precise program-analysis during JIT Compilation – highly inefficient.
- Most JIT compilers sacrifice precision for efficiency.

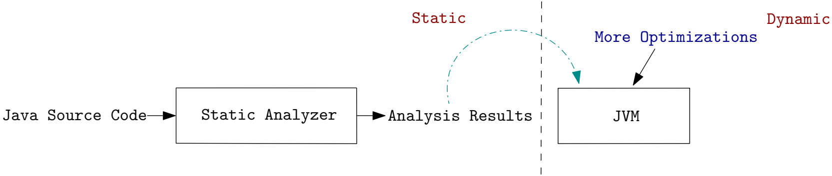
Introduction

- Several popular languages (such as Java, C#, Scala) use just-in-time (JIT) compilation.
- Performing precise program-analysis during JIT Compilation – highly inefficient.
- Most JIT compilers sacrifice precision for efficiency.
- **Can we use static analysis to impart precision in JIT analyses?**

Using Static Analysis Results in JIT Compilers

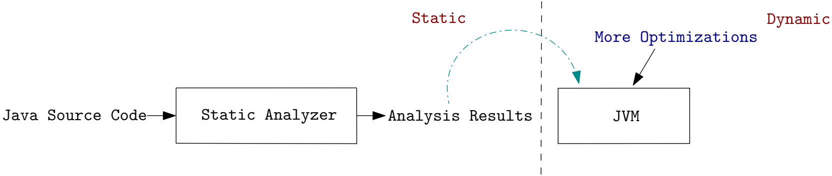


Using Static Analysis Results in JIT Compilers



- ❖ Challenge:
  - ✧ Library code needed to perform whole-program analysis.

Using Static Analysis Results in JIT Compilers



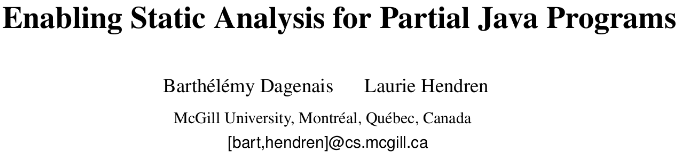
❖ Challenge:

- ✧ Library code needed to perform whole-program analysis.
- ✧ Imprecise results due to conservative assumptions.

Partial Program Analysis

Partial Program Analysis

- Proposed: OOPSLA 2008



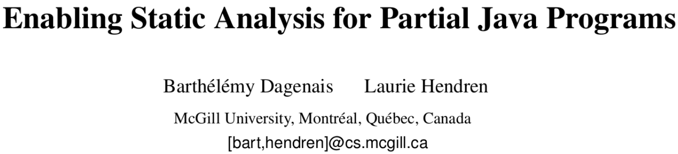
Enabling Static Analysis for Partial Java Programs

Barthélémy Dagenais Laurie Hendren

McGill University, Montréal, Québec, Canada
[bart,hendren]@cs.mcgill.ca

Partial Program Analysis

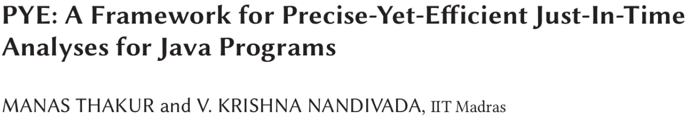
- ❖ Proposed: OOPSLA 2008



Enabling Static Analysis for Partial Java Programs

Barthélémy Dagenais Laurie Hendren
McGill University, Montréal, Québec, Canada
[bart,hendren]@cs.mcgill.ca

- ❖ Applied to JIT compilers: TOPLAS 2019



PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs

MANAS THAKUR and V. KRISHNA NANDIVADA, IIT Madras

PYE Framework and Conditional Values

"Precise-Yet-Efficient" framework generates highly precise analysis results for JIT compilers at a very low cost:

- ❖ Offloads expensive analysis to static compiler (javac) and generates *conditional values*.
- ❖ JIT component evaluates the conditional values at run-time and generates final analysis result.

PYE Framework and Conditional Values

"Precise-Yet-Efficient" framework generates highly precise analysis results for JIT compilers at a very low cost:

- Offloads expensive analysis to static compiler (javac) and generates *conditional values*.
- JIT component evaluates the conditional values at run-time and generates final analysis result.

- Essentially, conditional values for a program element enlist the dependencies of that element.

Dependencies in form of Conditional Values

```
class A {
   void foo(B b) {
      A a1 = new A(); // Object O3
      A a2 = new A(); // Object O4
      a1.bar(a2);
      L l1 = new L(); // Object O6
      l1.lib(a2); }
   void  bar(A p1) {
      // no assignment to p1
   } }
```

```
class L {
   // A library class
   void lib(A r1) {
      // A library method
      ...
   } }
```

Dependencies in form of Conditional Values

```java
class A {
   void foo(B b) {
      A a1 = new A(); // Object O3
      A a2 = new A(); // Object O4
      a1.bar(a2);
      L l1 = new L(); // Object O6
      l1.lib(a2); }
   void bar(A p1) {
      // no assignment to p1
   } }
```

```java
class L {
   // A library class
   void lib(A r1) {
      // A library
         method
      ...
   } }
```

❖ Conditional values for object O_4 (for escape analysis):

Dependencies in form of Conditional Values

```java
class A {
   void foo(B b) {
      A a1 = new A(); // Object O3
      A a2 = new A(); // Object O4
      a1.bar(a2);
      L l1 = new L(); // Object O6
      l1.lib(a2); }
   void bar(A p1) {
      // no assignment to p1
   } }
```

```java
class L {
   // A library class
   void lib(A r1) {
      // A library method
      ...
   } }
```

- Conditional values for object O_4 (for escape analysis):

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

Partial Result

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \ \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

Partial Result

$$[\![g_{\texttt{A.foo}}(O_4)]\!] = \ \sqcap_{ea}\{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so = \perp\ i.e\ D)$$

Partial Result

$$[\![g_{\mathtt{A.foo}}(O_4)]\!] = \sqcap_{ea} \{D, \langle\langle \mathtt{L.lib}, \mathtt{r1}\rangle, D, D\rangle, D, \langle\langle \mathtt{L.lib}, \mathtt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so = \perp \; i.e \; D)$$

$$[\![g_{\mathtt{A.foo}}(O_4)]\!] = \sqcap_{ea} \{D, \langle\langle \mathtt{L.lib}, \mathtt{r1}\rangle, D, D\rangle, \langle\langle \mathtt{L.lib}, \mathtt{r1}\rangle, E, E\rangle\}$$

Partial Result

$$\llbracket g_{\texttt{A.foo}}(O_4) \rrbracket = \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so =\perp\ i.e\ D)$$

$$\llbracket g_{\texttt{A.foo}}(O_4) \rrbracket = \sqcap_{ea} \{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because D \sqcap D = D) \tag{2}$$

Partial Result

$$\llbracket g_{\texttt{A.foo}}(O_4)\rrbracket = \ \sqcap_{ea}\{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because f_n(y) \neq v, so =\perp\ i.e\ D)$$

$$\llbracket g_{\texttt{A.foo}}(O_4)\rrbracket = \ \sqcap_{ea}\{D, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\}$$

$$(\because D \sqcap D = D) \tag{2}$$

- Partial Result:

$$\boxed{f_{\texttt{a.foo}}(O_4) = \ \sqcap_{ea}\ \{\langle\langle \texttt{l.lib}, \texttt{r1}\rangle, D, D\rangle, \langle\langle \texttt{l.lib}, \texttt{r1}\rangle, E, E\rangle\}}$$

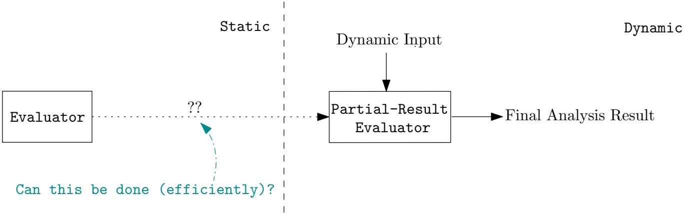
Existing PYE-like Staged Static+JIT Analyses

Existing PYE-like Staged Static+JIT Analyses

- Is it correct to stage whole-program analysis across static and JIT compilation?

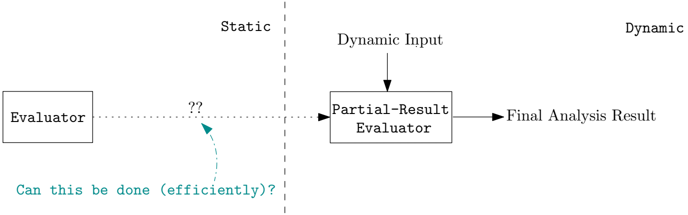
Existing PYE-like Staged Static+JIT Analyses

- Is it correct to stage whole-program analysis across static and JIT compilation?
    - What is the form of the evaluator needed during JIT compilation?



Existing PYE-like Staged Static+JIT Analyses

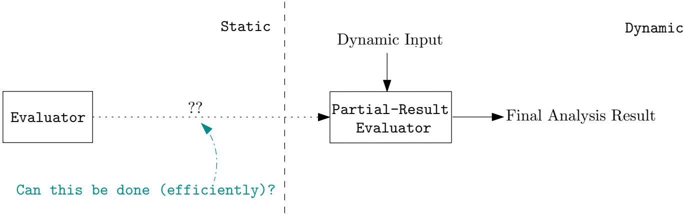
- Is it correct to stage whole-program analysis across static and JIT compilation?
  - What is the form of the evaluator needed during JIT compilation?



- Does the precision of staged analysis remain same as whole-program analysis?

Existing PYE-like Staged Static+JIT Analyses

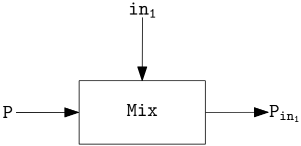
- Is it correct to stage whole-program analysis across static and JIT compilation?
  - What is the form of the evaluator needed during JIT compilation?



- Does the precision of staged analysis remain same as whole-program analysis? What about its efficiency?

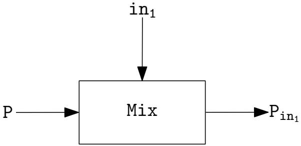
Partial Evaluation

Partial evaluation [Jones 1996] specializes a given program with its statically available inputs. The resultant partially evaluated program can later be executed with the dynamic inputs to generate the final output.



Partial Evaluation

Partial evaluation [Jones 1996] specializes a given program with its statically available inputs. The resultant partially evaluated program can later be executed with the dynamic inputs to generate the final output.



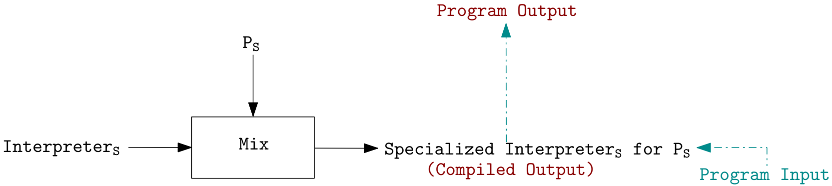
- Advantage: The specialized program $\mathtt{P}_{\mathtt{in}_1}$ often executes faster compared to executing the original program $\mathtt{P}$ provided both static and dynamic inputs together.

Futamura Projections

- Partial evaluation has also been used to specialize interpreters and their generators, based on the notion of Futamura projections.

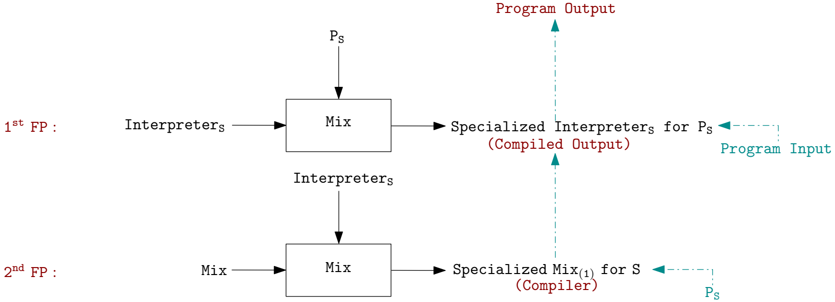
Futamura Projections

- Partial evaluation has also been used to specialize interpreters and their generators, based on the notion of Futamura projections.



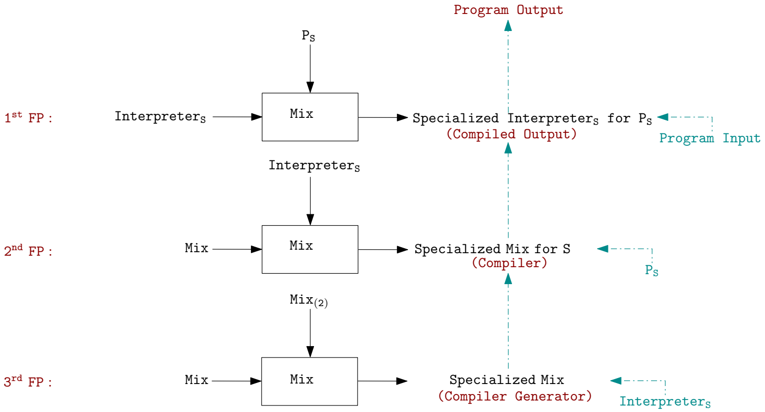
Futamura Projections

- Partial evaluation has also been used to specialize interpreters and their generators, based on the notion of Futamura projections.



Futamura Projections

- Partial evaluation has also been used to specialize interpreters and their generators, based on the notion of Futamura projections.



Modeling Partial Analysis based on Partial Evaluation

Mapped Notation

Source Program → $g_m(x)$ (Conditional Values)

Static Inputs → S_x (Resolved Values)

Runtime Inputs → D_x (Unresolved Values)

Evaluator → $\mathtt{CEval}_{\psi}$

Language for Conditional Values

- Conditional Values:

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

Language for Conditional Values

- Conditional Values:

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

- Language for Conditional Values:

```
<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
```

Language for Conditional Values

- Conditional Values:

$$\begin{aligned} g_{\texttt{A.foo}}(O_4) = \{&\langle\langle \texttt{A.bar}, \texttt{p1}\rangle, D, D\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, D, D\rangle, \\ &\langle\langle \texttt{A.bar}, \texttt{p1}\rangle, E, E\rangle, \langle\langle \texttt{L.lib}, \texttt{r1}\rangle, E, E\rangle\} \end{aligned}$$

- Language for Conditional Values:

```
<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
```

- Updated Conditional Values:

$$\begin{aligned} \langle \texttt{A foo LOCAL 4}\ \langle\texttt{nil}\rangle\rangle : \{&\langle\langle \texttt{A bar PARM 1}\langle\texttt{nil}\rangle\rangle\ \texttt{D D}\rangle, \langle\langle \texttt{L lib RETVAL 1}\langle\texttt{nil}\rangle\ \texttt{D D}\rangle\}, \\ \{&\langle\langle \texttt{A bar PARM 1}\langle\texttt{nil}\rangle\rangle\ \texttt{E E}\rangle, \langle\langle \texttt{L lib RETVAL 1}\langle\texttt{nil}\rangle\ \texttt{E E}\rangle\} \end{aligned}$$

Division of Conditional Values

Division of Conditional Values

❖ Tagged Conditional Values:

$\langle$`A foo LOCAL 4` $\langle$`nil`$\rangle\rangle$:

$\{\langle\langle$`STATIC A bar PARM 1`$\langle$`nil`$\rangle\rangle$ `D D`$\rangle$, $\langle\langle$`DYNAMIC L lib RETVAL 1`$\langle$`nil`$\rangle$ `D D`$\rangle$,

$\langle\langle$`STATIC A bar PARM 1`$\langle$`nil`$\rangle\rangle$ `E E`$\rangle$, $\langle\langle$`DYNAMIC L lib RETVAL 1`$\langle$`nil`$\rangle$ `E E`$\rangle\}$

Division of Conditional Values

- Tagged Conditional Values:

$$\langle \texttt{A foo LOCAL 4}\ \langle \texttt{nil} \rangle\rangle :$$

$$\{\langle\langle \texttt{STATIC A bar PARM 1} \langle \texttt{nil} \rangle\rangle\ \texttt{D D}\rangle, \langle\langle \texttt{DYNAMIC L lib RETVAL 1} \langle \texttt{nil} \rangle\ \texttt{D D}\rangle,$$
$$\langle\langle \texttt{STATIC A bar PARM 1} \langle \texttt{nil} \rangle\rangle\ \texttt{E E}\rangle, \langle\langle \texttt{DYNAMIC L lib RETVAL 1} \langle \texttt{nil} \rangle\ \texttt{E E}\rangle\}$$

- Language for Tagged Conditional Values:

Division of Conditional Values

- Tagged Conditional Values:

⟨A foo LOCAL 4 ⟨nil⟩⟩ :

{⟨⟨STATIC A bar PARM 1⟨nil⟩⟩ D D⟩, ⟨⟨DYNAMIC L lib RETVAL 1⟨nil⟩ D D⟩,
⟨⟨STATIC A bar PARM 1⟨nil⟩⟩ E E⟩, ⟨⟨DYNAMIC L lib RETVAL 1⟨nil⟩ E E⟩}

- Language for Tagged Conditional Values:

```
<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)

Conditional Value Evaluator

```
Procedure CEval($g_m(x)$, $\texttt{IN}_{g_m(x)}$)
    Initialize a list $L$ of statically known dependencies.
    foreach $d \in \texttt{IN}_{g_m(x)}$ do
        Add $d$ to $L$.
        Add the transitive dependencies of $d$ to $L$.
    Form strongly connected components (SCCs) in the list $L$.
    repeat
        foreach strongly connected component $S$ formed above do
            if $\nexists e \in S$ s.t. $e$ depends on another SCC then
                $\forall e \in S$, resolve $e$ to $\perp$.
            Take a meet of the resolved values in each SCC
    until fixed point;
```

Conditional Value Evaluator

```
Procedure CEval($g_m(x)$, $\texttt{IN}_{g_m(x)}$)
    Initialize a list $L$ of statically known dependencies.
    foreach $d \in \texttt{IN}_{g_m(x)}$ do
        Add $d$ to $L$.
        Add the transitive dependencies of $d$ to $L$.
    Form strongly connected components (SCCs) in the list $L$.
    repeat
        foreach strongly connected component $S$ formed above do
            if $\nexists e \in S$ s.t. $e$ depends on another SCC then
                $\forall e \in S$, resolve $e$ to $\bot$.
            Take a meet of the resolved values in each SCC
    until fixed point;
```

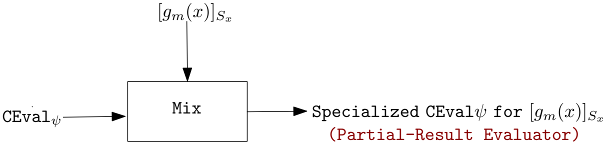
- Evaluated Conditional Values:

$$= \sqcap_{ea} \{\langle\langle \texttt{DYNAMIC L lib RETVAL 1}\langle\texttt{nil}\rangle\ \texttt{D D}\rangle, \langle\langle \texttt{DYNAMIC L lib RETVAL 1}\langle\texttt{nil}\rangle\ \texttt{E E}\rangle\}$$

AM Projections

AM Projections

- Specialize the evaluator with the partial result.



$[g_m(x)]_{S_x}$

`CEval`$_\psi$

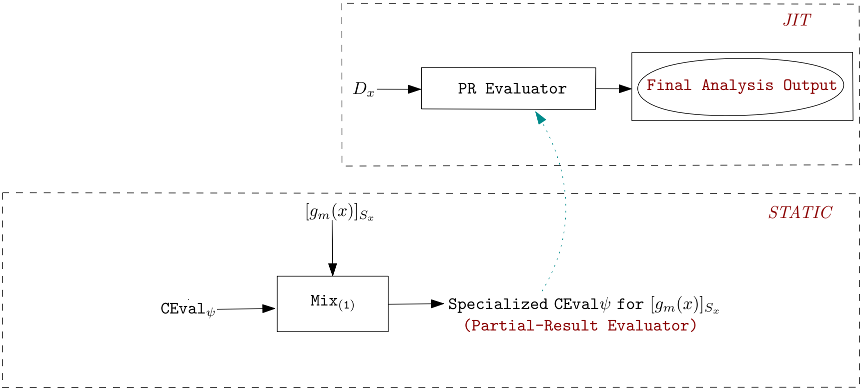
`Mix`

`Specialized CEval`ψ `for` $[g_m(x)]_{S_x}$

`(Partial-Result Evaluator)`

AM Projections

- Specialize the evaluator with the partial result.



JIT

D_x PR Evaluator Final Analysis Output

STATIC

$[g_m(x)]_{S_x}$

$\texttt{CEval}_{\psi}$ $\texttt{Mix}_{(1)}$ Specialized CEvalψ for $[g_m(x)]_{S_x}$
(Partial-Result Evaluator)

Generated Partial Result Evaluator

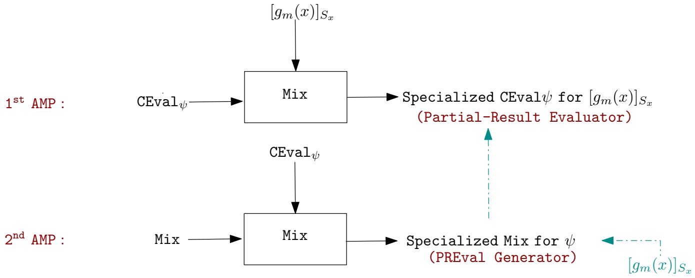
```
class PartialResultEvaluator {
   public static void main(String[] args) {
      // Read the values for dynamic dependencies
      x1 = Resolved value of <L.lib, r1> // first dependence
      x2 = Resolved value of <L.lib, r1> // second dependence
      res = x1 ⊓_ea x2
      print(res);
   }
}
```

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

- Can be placed in any VM to obtain the final analysis result for O_4.

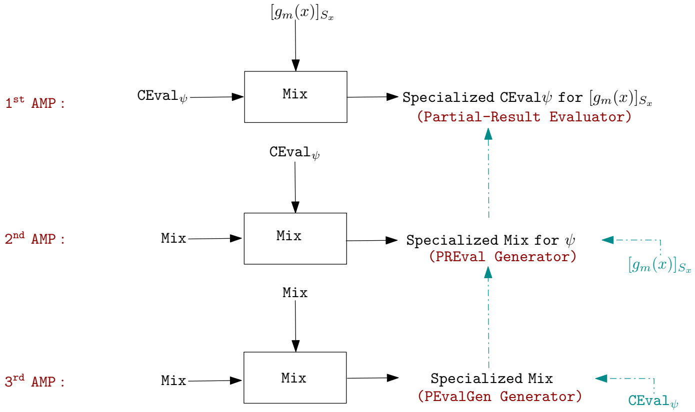
2^{nd} AM Projection

❖ Specialize Mix with the conditional-value evaluator.



3^{rd} AM Projection

- Specialize Mix with Mix.



Efficiency, Correctness and Precision of Staging

Efficiency, Correctness and Precision of Staging

- Lemma 1. Statically available Input

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.

Efficiency, Correctness and Precision of Staging

- Lemma 1. Statically available Input

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.

- Lemma 2. Maximal Specialization

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.

Efficiency, Correctness and Precision of Staging

- Theorem 1. Efficiency

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.

Efficiency, Correctness and Precision of Staging

- Theorem 1. Efficiency

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.

- Theorem 2. Precision and Correctness

For any program element, the analysis results generated by a whole-program analysis and by the corresponding staged analysis are the same.

Future Directions and Connections

Future Directions and Connections

- Runtime Features: Challenges and Possibilities.

Future Directions and Connections

- Runtime Features: Challenges and Possibilities.
    - Fallback values in case of other analyses.

Future Directions and Connections

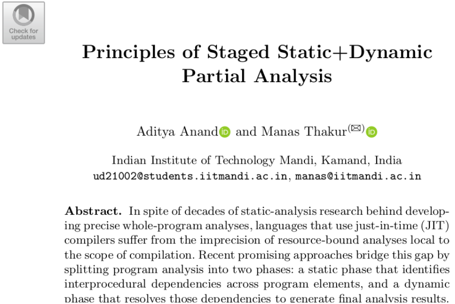
- Runtime Features: Challenges and Possibilities.
    - Fallback values in case of other analyses.
- Cross-pollination of Specialization Ideas

Future Directions and Connections

- Runtime Features: Challenges and Possibilities.
    - Fallback values in case of other analyses.
- Cross-pollination of Specialization Ideas
- Query- and Feedback-Driven Analyses.

Future Directions and Connections

- Runtime Features: Challenges and Possibilities.
  - Fallback values in case of other analyses.
- Cross-pollination of Specialization Ideas
- Query- and Feedback-Driven Analyses.



Principles of Staged Static+Dynamic Partial Analysis

Aditya Anand and Manas Thakur(✉)

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.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.
- Projections for efficiently generating the evaluators (and their generators) for partial results.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.
- Projections for efficiently generating the evaluators (and their generators) for partial results.
- Proved the correctness and precision of staged static+dynamic analysis.

Conclusion

- Formalized the theory of staged static+dynamic partial analysis based on the theory of partial evaluation.
- Projections for efficiently generating the evaluators (and their generators) for partial results.
- Proved the correctness and precision of staged static+dynamic analysis.

Thank You!