Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image

David Kozak [1,2], Vojin Jovanovic [2], Codrut Stancu [2], Tomáš Vojnar [1], Christian Wimmer [2]

[1] Brno University of Technology, Faculty of Information Technology

[2] GraalVM, Oracle Labs

ikozak@fit.vutbr.cz

October 22, 2023

Motivation - Call Graph Construction

- Call graph construction for ahead-of-time compilers.

Motivation - Call Graph Construction

- Call graph construction for ahead-of-time compilers.
  - Used to detect reachable program elements.

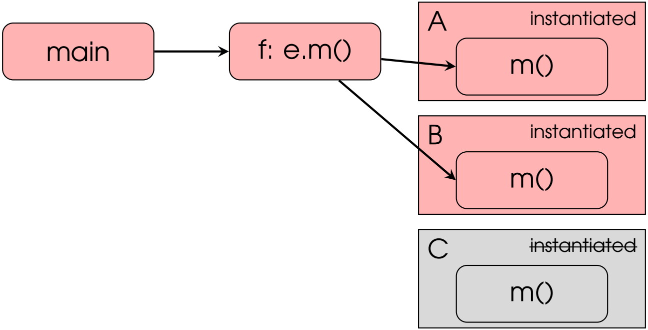
Motivation - Call Graph Construction

- Call graph construction for ahead-of-time compilers.
  - Used to detect reachable program elements.
  - Key difficulty - virtual methods.

Motivation - Call Graph Construction

- Call graph construction for ahead-of-time compilers.
  - Used to detect reachable program elements.
  - Key difficulty - virtual methods.
  - For each virtual invoke $e.m()$ in each reachable method, determine all callees that can be called at runtime.

- Call graph construction for ahead-of-time compilers.
  - Used to detect reachable program elements.
  - Key difficulty - virtual methods.
  - For each virtual invoke *e*.*m*() in each reachable method, determine all callees that can be called at runtime.



- Ahead-of-time (AOT) compiler for Java.

GraalVM Native Image

- Ahead-of-time (AOT) compiler for Java.
- Closed-world assumption:
  - All code that can be executed at runtime has to be available at compile time.

- Ahead-of-time (AOT) compiler for Java.
- Closed-world assumption:
  - All code that can be executed at runtime has to be available at compile time.
  - All dynamic features, e.g. reflection, proxy, dynamic class loading, have to be explicitly configured.

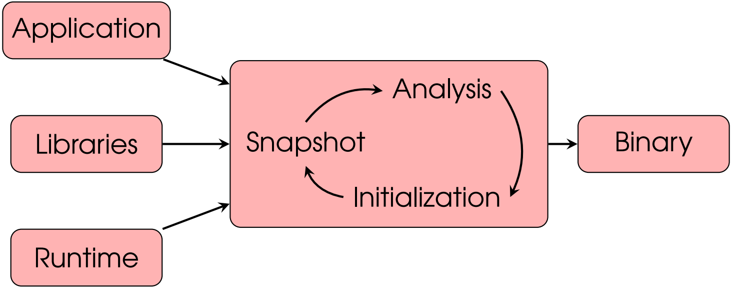
GraalVM Native Image

- Ahead-of-time (AOT) compiler for Java.
- Closed-world assumption:
  - All code that can be executed at runtime has to be available at compile time.
  - All dynamic features, e.g. reflection, proxy, dynamic class loading, have to be explicitly configured.
- Produces standalone binaries containing the application along with all dependencies and runtime components.

GraalVM Native Image

- Combination of points-to analysis, class initialization at build time and heap snapshotting.

- Combination of points-to analysis, class initialization at build time and heap snapshotting.



- Fixed-point computation starting from a set of entrypoints.

Motivation

However, points-to analysis can be time-consuming.

Motivation

However, points-to analysis can be time-consuming.

Analysis of `Spring Petclinic` can take up to **159 seconds**.

Motivation

However, points-to analysis can be time-consuming.

Analysis of `Spring Petclinic` can take up to **159 seconds**.

Main Research Question

Is it possible to reduce the analysis time by replacing points-to analysis with an alternative, cheaper analysis?

Motivation

However, points-to analysis can be time-consuming.

Analysis of `Spring Petclinic` can take up to **159 seconds**.

Main Research Question

Is it possible to reduce the analysis time by replacing points-to analysis with an alternative, cheaper analysis?

Criteria:

However, points-to analysis can be time-consuming.

Analysis of `Spring Petclinic` can take up to **159 seconds**.

Main Research Question

Is it possible to reduce the analysis time by replacing points-to analysis with an alternative, cheaper analysis?

Criteria:

- Scalability - debug mode for Native Image.

Motivation

However, points-to analysis can be time-consuming.

Analysis of `Spring Petclinic` can take up to **159 seconds**.

Main Research Question

Is it possible to reduce the analysis time by replacing points-to analysis with an alternative, cheaper analysis?

Criteria:

- Scalability - debug mode for Native Image.
- Reasonable precision - do not increase the workload for the compilation phase too much.

Motivation

However, points-to analysis can be time-consuming.

Analysis of `Spring Petclinic` can take up to **159 seconds**.

Main Research Question

Is it possible to reduce the analysis time by replacing points-to analysis with an alternative, cheaper analysis?

Criteria:

- Scalability - debug mode for Native Image.
- Reasonable precision - do not increase the workload for the compilation phase too much.
- Fit into the environment of Native Image - interact with features such as heap snapshotting.

However, points-to analysis can be time-consuming.

Analysis of `Spring Petclinic` can take up to **159 seconds**.

Main Research Question

Is it possible to reduce the analysis time by replacing points-to analysis with an alternative, cheaper analysis?

Criteria:

- Scalability - debug mode for Native Image.
- Reasonable precision - do not increase the workload for the compilation phase too much.
- Fit into the environment of Native Image - interact with features such as heap snapshotting.
- Incrementality - reuse results from previous compilations.

Call Graph Construction

There is a plethora of call graph construction algorithms:

- Unique Name Analysis,

There is a plethora of call graph construction algorithms:

- Unique Name Analysis,
- Class Hierarchy Analysis,

Call Graph Construction

There is a plethora of call graph construction algorithms:

- Unique Name Analysis,
- Class Hierarchy Analysis,
- Rapid Type Analysis, and

Call Graph Construction

There is a plethora of call graph construction algorithms:

- Unique Name Analysis,
- Class Hierarchy Analysis,
- Rapid Type Analysis, and
- 0-CFA (Control Flow Analysis).

There is a plethora of call graph construction algorithms:

- Unique Name Analysis,
- Class Hierarchy Analysis,
- Rapid Type Analysis, and
- 0-CFA (Control Flow Analysis).

After careful examination, we chose Rapid Type Analysis, which offers a good tradeoff between precision and scalability.

The basic effect of Rapid Type Analysis can be summarized using the following equations. For R denoting a set of reachable methods and I denoting a set of instantiated types:

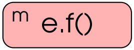
The basic effect of Rapid Type Analysis can be summarized using the following equations. For R denoting a set of reachable methods and I denoting a set of instantiated types:

1. `main` $\in R$.

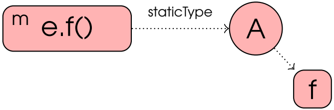
Rapid Type Analysis

The basic effect of Rapid Type Analysis can be summarized using the following equations. For R denoting a set of reachable methods and I denoting a set of instantiated types:

1. $\texttt{main} \in R$.
2. $\forall m \in R\ \forall e.f() \in \mathit{CallExpr}(m)$
   $\forall t \in \mathit{Subtypes}(\mathit{StaticType}(e))$.
   $t \in I \wedge \mathit{StaticLookup}(t, f) = m'$
   $\implies m' \in R$.
3. $\forall m \in R\ \forall \mathit{new}\ C() \in \mathit{InstExpr}(m)$. $C \in I$.



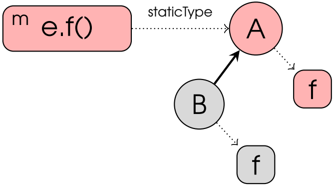
The basic effect of Rapid Type Analysis can be summarized using the following equations. For R denoting a set of reachable methods and I denoting a set of instantiated types:



1. $\mathtt{main} \in R$.
2. $\forall m \in R\ \forall e.f() \in CallExpr(m)$ $\forall t \in Subtypes(StaticType(e))$. $t \in I \wedge StaticLookup(t, f) = m'$ $\implies m' \in R$.
3. $\forall m \in R\ \forall new\ C() \in InstExpr(m)$. $C \in I$.

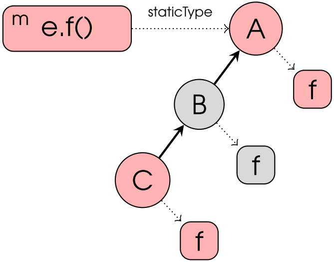
The basic effect of Rapid Type Analysis can be summarized using the following equations. For R denoting a set of reachable methods and I denoting a set of instantiated types:

1. $\texttt{main} \in R.$
2. $\forall m \in R\ \forall e.f() \in CallExpr(m)$
   $\forall t \in Subtypes(StaticType(e)).$
   $t \in I \wedge StaticLookup(t, f) = m'$
   $\implies m' \in R.$
3. $\forall m \in R\ \forall new\ C() \in$
   $InstExpr(m).\ C \in I.$



The basic effect of Rapid Type Analysis can be summarized using the following equations. For R denoting a set of reachable methods and I denoting a set of instantiated types:

1. $\texttt{main} \in R$.
2. $\forall m \in R \; \forall e.f() \in CallExpr(m)$
   $\forall t \in Subtypes(StaticType(e))$.
   $t \in I \wedge StaticLookup(t, f) = m'$
   $\implies m' \in R$.
3. $\forall m \in R \; \forall new\, C() \in InstExpr(m)$. $C \in I$.



Rapid Type Analysis has been well-studied, so let us now clearly distinguish our contributions from the state of the art:

Rapid Type Analysis has been well-studied, so let us now clearly distinguish our contributions from the state of the art:

- We designed an extension of rapid type analysis suited for the context of GraalVM Native Image.

Rapid Type Analysis has been well-studied, so let us now clearly distinguish our contributions from the state of the art:

- We designed an extension of rapid type analysis suited for the context of GraalVM Native Image.
- We extended the proposed algorithm to be parallel and incremental.

Rapid Type Analysis has been well-studied, so let us now clearly distinguish our contributions from the state of the art:

- We designed an extension of rapid type analysis suited for the context of GraalVM Native Image.
- We extended the proposed algorithm to be parallel and incremental.
- The incrementality was achieved using method summaries that sum up the effect of each analyzed method.

Rapid Type Analysis has been well-studied, so let us now clearly distinguish our contributions from the state of the art:

- We designed an extension of rapid type analysis suited for the context of GraalVM Native Image.
- We extended the proposed algorithm to be parallel and incremental.
- The incrementality was achieved using method summaries that sum up the effect of each analyzed method.
- We provided a detailed comparison of the new variant of RTA with a points-to analysis for ahead-of-time compilation of Java.

RTA - Core Algorithm

- The core algorithm starts with a set of root methods that are considered reachable.

RTA - Core Algorithm

- The core algorithm starts with a set of root methods that are considered reachable.
- Each reachable method is parsed into the intermediate representation from which the method summary is extracted.

RTA - Core Algorithm

- The core algorithm starts with a set of root methods that are considered reachable.
- Each reachable method is parsed into the intermediate representation from which the method summary is extracted.
- The summary is then used to update the state of the analysis, possibly making new methods reachable.
  - The update is done by calling specialized `register*` methods.

RTA - Core Algorithm

- The core algorithm starts with a set of root methods that are considered reachable.
- Each reachable method is parsed into the intermediate representation from which the method summary is extracted.
- The summary is then used to update the state of the analysis, possibly making new methods reachable.
  - The update is done by calling specialized `register*` methods.
- Once a given method is made reachable, the decision is never reverted.

RTA - Core Algorithm

- The core algorithm starts with a set of root methods that are considered reachable.
- Each reachable method is parsed into the intermediate representation from which the method summary is extracted.
- The summary is then used to update the state of the analysis, possibly making new methods reachable.
  - The update is done by calling specialized `register*` methods.
- Once a given method is made reachable, the decision is never reverted.
- The analysis stops once a fixed-point is reached.

Algorithm 1 RTA Core Algorithm.

```
for m ∈ rootMethods do
    registerAsInvoked(m)
procedure REGISTERASINVOKED(m)
    if mark(m.isInvoked) then                        ▻ atomic check
        schedule(() → onInvoked(m))                  ▻ schedule new task
procedure ONINVOKED(m)
    irGraph ← parseMethod(m)
    s ← extractSummary(irGraph)
    applySummary(s)
```

- All `register*` methods guarded by atomic checks.
  - Each class/method/field is processed only once.
- Non-trivial operations scheduled as separate tasks.

Method Summaries for RTA

- The effect of each method can be described using a method summary consisting of sets containing the following information:
  - directly invoked methods,
  - virtually invoked methods,
  - accessed types,
  - instantiated types,
  - read fields,
  - written fields, and
  - embedded constants.

Method Summaries for RTA

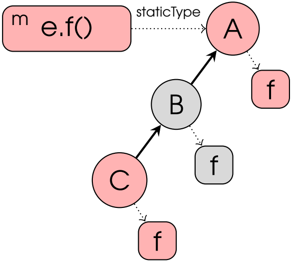
- The effect of each method can be described using a method summary consisting of sets containing the following information:
  - directly invoked methods,
  - virtually invoked methods,
  - accessed types,
  - instantiated types,
  - read fields,
  - written fields, and
  - embedded constants.
- These summaries can be extracted by a linear pass over the intermediate representation.

Handling of Virtual Methods

Algorithm 2 RTA handling of virtual methods.

```
procedure REGISTERASVIRTUALINVOKED(m)
    if mark(m.isVirtInvoked) then
        t ← m.declType
        t.virtInvoked.add(m)

        for subt ∈ t.instSubtypes do
            res ← subt.resolveMethod(m)
            registerAsInvoked(res)
```

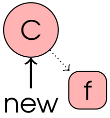


- First, method *m* is marked as virtual invoked.
- Then, all instantiated subtypes of the declaring type are considered when computing the call targets.

Handling of Virtual Methods

Algorithm 3 RTA handling of virtual methods.

```
procedure REGISTERASINSTANTIATED(t)
    if mark(t.isInst) then
        for st ∈ t.superTypes do
            st.instSubtypes.add(t)
        for st ∈ t.superTypes do
            for m ∈ st.virtInvoked do
                res ← t.resolveMethod(m)
                registerAsInvoked(res)
```

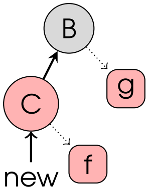


- First, the type t is marked as instantiated in all its supertypes.
- Then, all supertypes of the declaring type are traversed and each of their virtual invoked methods is processed.
- Note that type t is always used for the resolution.

Handling of Virtual Methods

Algorithm 4 RTA handling of virtual methods.

```
procedure REGISTERASINSTANTIATED(t)
    if mark(t.isInst) then
        for st ∈ t.superTypes do
            st.instSubtypes.add(t)
        for st ∈ t.superTypes do
            for m ∈ st.virtInvoked do
                res ← t.resolveMethod(m)
                registerAsInvoked(res)
```

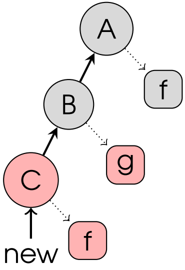


- First, the type *t* is marked as instantiated in all its supertypes.
- Then, all supertypes of the declaring type are traversed and each of their virtual invoked methods is processed.
- Note that type *t* is always used for the resolution.

Handling of Virtual Methods

Algorithm 5 RTA handling of virtual methods.

```
procedure REGISTERASINSTANTIATED(t)
    if mark(t.isInst) then
        for st ∈ t.superTypes do
            st.instSubtypes.add(t)
        for st ∈ t.superTypes do
            for m ∈ st.virtInvoked do
                res ← t.resolveMethod(m)
                registerAsInvoked(res)
```



- First, the type *t* is marked as instantiated in all its supertypes.
- Then, all supertypes of the declaring type are traversed and each of their virtual invoked methods is processed.
- Note that type *t* is always used for the resolution.

```java
public class Hello {
    public static void main() {
        new Hello().foo(new A());
        log();
    }
    static void log(){
        new B();
    }
    void foo(I i){
        i.bar();
    }
}
interface I { void bar(); }
class A implements I {...}
class B implements I {...}
```

Method Summaries:

[1] New represents instantiated types

Running Example

```
public class Hello {
    public static void main() {
        new Hello().foo(new A());
        log();
    }
    static void log(){
        new B();
    }
    void foo(I i){
        i.bar();
    }
}
interface I { void bar(); }
class A implements I {...}
class B implements I {...}
```

Method Summaries:

- `main`
  - New[1] - `Hello, A`
  - Direct invoke - `log`
  - Virtual invoke - `foo`

[1] New represents instantiated types

Running Example

```java
public class Hello {
    public static void main() {
        new Hello().foo(new A());
        log();
    }
    static void log(){
        new B();
    }
    void foo(I i){
        i.bar();
    }
}
interface I { void bar(); }
class A implements I {...}
class B implements I {...}
```

Method Summaries:

- `main`
  - New[1] - `Hello, A`
  - Direct invoke - `log`
  - Virtual invoke - `foo`
- `log`
  - New - `B`

[1] New represents instantiated types

Running Example

```
public class Hello {
    public static void main() {
        new Hello().foo(new A());
        log();
    }
    static void log(){
        new B();
    }
    void foo(I i){
        i.bar();
    }
}
interface I { void bar(); }
class A implements I {...}
class B implements I {...}
```

Method Summaries:

- main
  - New[1] - Hello, A
  - Direct invoke - log
  - Virtual invoke - foo
- log
  - New - B
- foo
  - Virtual invoke - bar

[1] New represents instantiated types

Table: Results of the analyses on the running example.

Analysis	Results	
	Instantiated types	Invoked methods
PTA	Hello, A, B	log, foo, A.bar
RTA	Hello, A, B	log, foo, {A, B}.bar

Incrementality

- Summaries contain mainly classes, methods, and fields.

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.

Incrementality

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.
  - ClassId - fully qualified name.

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.
  - ClassId - fully qualified name.
  - MethodId - ClassId + name + signature.

Incrementality

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.
  - ClassId - fully qualified name.
  - MethodId - ClassId + name + signature.
  - FieldId - ClassId + name.

Incrementality

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.
  - ClassId - fully qualified name.
  - MethodId - ClassId + name + signature.
  - FieldId - ClassId + name.
- Constraints:
  - Stable identifiers - issues with generated classes.

Incrementality

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.
  - ClassId - fully qualified name.
  - MethodId - ClassId + name + signature.
  - FieldId - ClassId + name.
- Constraints:
  - Stable identifiers - issues with generated classes.
  - Trivial embedded constants - only primitives and well-known types such as `java.lang.String`.

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.
  - ClassId - fully qualified name.
  - MethodId - ClassId + name + signature.
  - FieldId - ClassId + name.
- Constraints:
  - Stable identifiers - issues with generated classes.
  - Trivial embedded constants - only primitives and well-known types such as `java.lang.String`.
- Issues:
  - Parsing a method in Native Image has side effects.

Incrementality

- Summaries contain mainly classes, methods, and fields.
  - They can be represented textually.
  - ClassId - fully qualified name.
  - MethodId - ClassId + name + signature.
  - FieldId - ClassId + name.
- Constraints:
  - Stable identifiers - issues with generated classes.
  - Trivial embedded constants - only primitives and well-known types such as `java.lang.String`.
- Issues:
  - Parsing a method in Native Image has side effects.
  - Many summaries contain non-trivial constants.

ORACLE | FIT

Table: Detailed statistics of the evaluated benchmarks.

Suite	Benchmark	Reachable Methods PTA	Reachable Methods RTA	Analysis Time (s) PTA	Analysis Time (s) RTA	Total time (s) PTA	Total time (s) RTA	Binary size (MB) PTA	Binary size (MB) RTA
Console	helloworld	18	**+17%**	14	**+21%**	36	**+17%**	13	**+23%**
Dacapo	avrora	24	**+25%**	12	-8%	51	+6%	23	**+30%**
	fop	94	+4%	46	-30%	128	-10%	105	+11%
	jython	71	+8%	55	**-35%**	140	**-26%**	134	+9%
	luindex	26	+23%	13	-8%	54	**+7%**	32	+25%
Microservices	micronaut-helloworld-wrk	74	+4%	34	-32%	88	-9%	45	+18%
	mushop:order	168	+2%	102	-59%	209	-30%	104	+13%
	mushop:payment	82	+4%	36	-33%	91	-10%	50	+14%
	mushop:user	115	+3%	57	-44%	135	-18%	76	+13%
	petclinic-wrk	207	+4%	159	**-64%**	297	**-35%**	144	+15%
	quarkus-helloworld-wrk	52	+6%	18	-22%	69	-3%	50	+4%
	quarkus:registry	111	+5%	49	-39%	126	-16%	69	**+19%**
	spring-helloworld-wrk	67	+4%	30	-33%	87	-10%	47	+13%
	tika-wrk	82	**+6%**	29	-28%	117	-6%	88	+6%
Renaissance	chi-square	173	+8%	129	-60%	260	-30%	100	+17%
	dec-tree	324	+6%	2009	-95%	X	X	X	X
	future-genetic	27	**+22%**	15	0%	44	+5%	19	**+21%**
	gauss-mix	189	+8%	146	-61%	286	**-32%**	107	+17%
	log-regression	334	+7%	2215	**-95%**	X	X	X	X
	page-rank	171	+8%	129	-60%	258	-31%	119	+13%
	reactors	30	+13%	19	+16%	47	+11%	19	+21%
	scala-stm-bench7	30	+20%	19	**+26%**	49	**+14%**	19	+21%

Scalability In Number of Cores

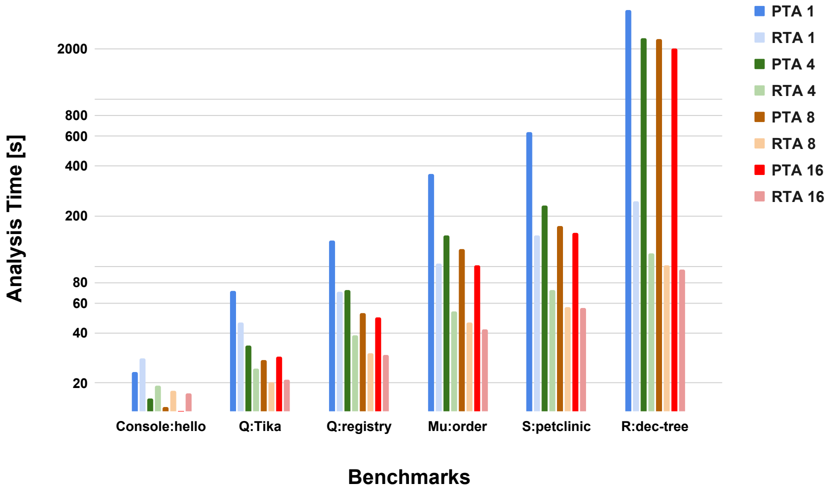


Figure: Scalability in number of cores.

- We designed an extension of rapid type analysis for the context of GraalVM Native Image.

Conclusion

- We designed an extension of rapid type analysis for the context of GraalVM Native Image.
- We extended the proposed algorithm to be parallel and incremental.

Conclusion

- We designed an extension of rapid type analysis for the context of GraalVM Native Image.
- We extended the proposed algorithm to be parallel and incremental.
- The incrementality was achieved using method summaries that sum up the effect of each analyzed method.

Conclusion

- We designed an extension of rapid type analysis for the context of GraalVM Native Image.
- We extended the proposed algorithm to be parallel and incremental.
- The incrementality was achieved using method summaries that sum up the effect of each analyzed method.
- For Spring Petclinic, we reduced the analysis time by **64%**, the overall build time by **35%** at the cost of increasing the image size by **15%**.

Conclusion

- We designed an extension of rapid type analysis for the context of GraalVM Native Image.
- We extended the proposed algorithm to be parallel and incremental.
- The incrementality was achieved using method summaries that sum up the effect of each analyzed method.
- For Spring Petclinic, we reduced the analysis time by **64%**, the overall build time by **35%** at the cost of increasing the image size by **15%**.
- Next steps:
  - Extend the support for incremental analysis.
  - Combine points-to analysis with rapid type analysis.

Thank You For Your Attention.