Boosting the Capabilities of Compilers via Static Analysis

David Kozak[1,2]
PhD Student – 3rd year
Supervisor FIT: Tomas Vojnar[1]
Supervisor Oracle: Christian Wimmer[2]

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

[2]Oracle Labs

Activity in the Past Year

- **Accepted:**
  - Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image – **MPLR 23 (Core C)**
    - Small, **specialised**, strong participation of the community
  - Software Architecture Reconstruction for Microservice Systems using Static Analysis via GraalVM Native Image – **SANER 24 (Core A)**
    - With **Tomas Cerny** from Baylor University, now University of Arizona
  - Scaling Type-Based Points-to Analysis with Saturation – **PLDI 24 (Core A*)**
    - **Flagship conference** (Google Scholar H5: 50, #9 of all publication channels in Software Systems)
- **Submitted:**
  - Partially Flow-Sensitive Points-to Analysis using Predicates – **OOPSLA 24 (Core A)**
    - The **main focus** of my last year and this presentation

Topic Overview

Boosting the Capabilities of Compilers via Static Analysis

Boosting the Capabilities of Compilers via Static Analysis

- **Static analysis** embedded inside **compilers**

Boosting the Capabilities of Compilers via Static Analysis

- **Static analysis** embedded inside **compilers**
- **Old area** – the **genesis** of static analysis

Boosting the Capabilities of Compilers via Static Analysis

- **Static analysis** embedded inside **compilers**
- **Old area** – the **genesis** of static analysis
- **What is new?**

Boosting the Capabilities of Compilers via Static Analysis

- **Static analysis** embedded inside **compilers**
- **Old area** – the **genesis** of static analysis
- **What is new?**
- **Shift to the cloud** – make **small** applications that **start quickly**

Boosting the Capabilities of Compilers via Static Analysis

- **Static analysis** embedded inside **compilers**
- **Old area** – the **genesis** of static analysis
- **What is new?**
- **Shift to the cloud** – make **small** applications that **start quickly**
- **Closed-world ahead-of-time** compilation model

Boosting the Capabilities of Compilers via Static Analysis

- **Static analysis** embedded inside **compilers**
- **Old area** – the **genesis** of static analysis
- **What is new?**
- **Shift to the cloud** – make **small** applications that **start quickly**
- **Closed-world ahead-of-time** compilation model
  - **Whole-program analysis**

Boosting the Capabilities of Compilers via Static Analysis

- **Static analysis** embedded inside **compilers**
- **Old area** – the **genesis** of static analysis
- **What is new?**
- **Shift to the cloud** – make **small** applications that **start quickly**
- **Closed-world ahead-of-time** compilation model
  - **Whole-program analysis**
  - **Aggressive optimizations**

Boosting the Capabilities of Compilers via Static Analysis

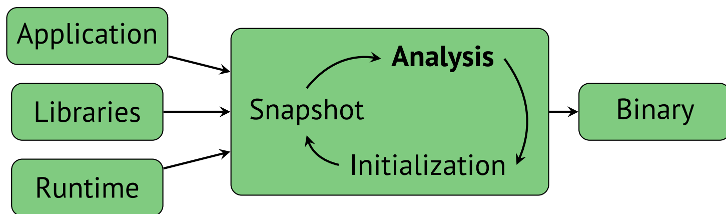
- **Static analysis** embedded inside **compilers**
- **Old area** – the **genesis** of static analysis
- **What is new?**
- **Shift to the cloud** – make **small** applications that **start quickly**
- **Closed-world ahead-of-time** compilation model
  - **Whole-program analysis**
  - **Aggressive optimizations**
- New use cases, e.g. **microservices**

GraalVM Native Image

- Research in collaboration with **Oracle Labs**

GraalVM Native Image

- Research in collaboration with **Oracle Labs**
- **GraalVM Native Image**
  - Ahead-of-time compiler for **Java bytecode**
  - Produces **self-contained** binaries



Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)
- Allows **optimizations** st. type-check elimination, devirtualization, ...

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)
- Allows **optimizations** st. type-check elimination, devirtualization, …
- Uses a **typeflow graph** modelling **interprocedural value propagation**

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)
- Allows **optimizations** st. type-check elimination, devirtualization, ...
- Uses a **typeflow graph** modelling **interprocedural value propagation**
  - **Nodes**
    - **Memory locations** – variables, parameters, object fields, ...

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)
- Allows **optimizations** st. type-check elimination, devirtualization, …
- Uses a **typeflow graph** modelling **interprocedural value propagation**
  - **Nodes**
    - **Memory locations** – variables, parameters, object fields, …
    - **Relevant instructions** – method invocations, type-checks, …

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)
- Allows **optimizations** st. type-check elimination, devirtualization, ...
- Uses a **typeflow graph** modelling **interprocedural value propagation**
  - **Nodes**
    - **Memory locations** – variables, parameters, object fields, ...
    - **Relevant instructions** – method invocations, type-checks, ...
  - **Edges**
    - **Use** – data-flow from sources (allocations) to sinks (usages)

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)
- Allows **optimizations** st. type-check elimination, devirtualization, ...
- Uses a **typeflow graph** modelling **interprocedural value propagation**
  - **Nodes**
    - **Memory locations** – variables, parameters, object fields, ...
    - **Relevant instructions** – method invocations, type-checks, ...
  - **Edges**
    - **Use** – data-flow from sources (allocations) to sinks (usages)
    - **Observer** – other dependencies, e.g. receiver to method invocation

Static Analysis in GraalVM Native Image

- Context-insensitive type-based **points-to analysis**
  - **Approximates** the values of **pointers/references** at runtime.
- Computes **reachable methods** from entry points (e.g. `main`)
- Allows **optimizations** st. type-check elimination, devirtualization, ...
- Uses a **typeflow graph** modelling **interprocedural value propagation**
  - **Nodes**
    - **Memory locations** – variables, parameters, object fields, ...
    - **Relevant instructions** – method invocations, type-checks, ...
  - **Edges**
    - **Use** – data-flow from sources (allocations) to sinks (usages)
    - **Observer** – other dependencies, e.g. receiver to method invocation
- Each **node** has a **value state** – a set of types

Partially Flow-Sensitive Points-to Analysis using Predicates

Submitted to OOPSLA 24

David Kozak[1,2] **Tomas Vojnar**[1] **Christian Wimmer**[2] **Codrut Stancu**[2]

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

[2]Oracle Labs

Introduction

- **Flow-sensitivite** analysis maintains a **program state** for each **program point**
  - Precise, expensive

Introduction

- **Flow-sensitivite** analysis maintains a **program state** for each **program point**
  - Precise, expensive
- **Flow-insensitive** analysis scales well, but is **imprecise**
  - Fast, lacks precision

Introduction

- **Flow-sensitivite** analysis maintains a **program state** for each **program point**
  - Precise, expensive
- **Flow-insensitive** analysis scales well, but is **imprecise**
  - Fast, lacks precision
- **Partial flow-sensitivity** as the middle ground

Introduction

- **Flow-sensitivite** analysis maintains a **program state** for each **program point**
  - Precise, expensive
- **Flow-insensitive** analysis scales well, but is **imprecise**
  - Fast, lacks precision
- **Partial flow-sensitivity** as the middle ground
  - Where to set the **threshold**?

Motivating Example 1 - Default Value

```
class Scene {
    void render(..., Display display) {
      if (display == null) {
        display = new FrameDisplay();
      }
      ...
    }
}
```

```
class BucketRenderer {
    void render(Display display) {
      ...
      display.imageBegin();
      ...
    }
}
```

Motivating Example 1 - Default Value

```
class Scene {
    void render(..., Display display) {
      if (display == null) {
        display = new FrameDisplay();
      }
      ...
    }
}
```

```
class BucketRenderer {
    void render(Display display) {
      ...
      display.imageBegin();
      ...
    }
}
```

- `FrameDisplay` is only **instantiated** iff `display` is `null`

Motivating Example 1 - Default Value

```
class Scene {
    void render(..., Display display) {
      if (display == null) {
        display = new FrameDisplay();
      }
      ...
    }
}
```

```
class BucketRenderer {
    void render(Display display) {
      ...
      display.imageBegin();
      ...
    }
}
```

- `FrameDisplay` is only **instantiated** iff `display` is `null`
- What if we **know** that `display` is actually never `null`?

Motivating Example 1 - Default Value

```
class Scene {
    void render(..., Display display) {
      if (display == null) {
        display = new FrameDisplay();
      }
      ...
    }
}
```

```
class BucketRenderer {
    void render(Display display) {
      ...
      display.imageBegin();
      ...
    }
}
```

- `FrameDisplay` is only **instantiated** iff `display` is `null`
- What if we **know** that `display` is actually never `null`?
- `FrameDisplay.imageBegin` makes Java GUI libraries **Swing** and **AWT** reachable

Motivating Example 2 - Optional Invocation

```
class SharedThreadContainer {
   Set<Thread> virtualThreads;
   public void onExit(Thread thread) {
     if (thread.isVirtual())
       virtualThreads.remove(thread);
   }
}
```

```
class Thread {
   public boolean isVirtual() {
     return this instanceof
          BaseVirtualThread;
   }
}
```

Motivating Example 2 - Optional Invocation

```
class SharedThreadContainer {
   Set<Thread> virtualThreads;
   public void onExit(Thread thread) {
     if (thread.isVirtual())
       virtualThreads.remove(thread);
   }
}
```

```
class Thread {
   public boolean isVirtual() {
     return this instanceof
           BaseVirtualThread;
   }
}
```

- Virtual threads are an **experimental feature**

Motivating Example 2 - Optional Invocation

```
class SharedThreadContainer {
   Set<Thread> virtualThreads;
   public void onExit(Thread thread) {
     if (thread.isVirtual())
       virtualThreads.remove(thread);
   }
}
```

```
class Thread {
   public boolean isVirtual() {
     return this instanceof
         BaseVirtualThread;
   }
}
```

- Virtual threads are an **experimental feature**
- **Not enabled** by default

Motivating Example 2 - Optional Invocation

```
class SharedThreadContainer {
   Set<Thread> virtualThreads;
   public void onExit(Thread thread) {
     if (thread.isVirtual())
       virtualThreads.remove(thread);
   }
}
```

```
class Thread {
   public boolean isVirtual() {
     return this instanceof
         BaseVirtualThread;
   }
}
```

- Virtual threads are an **experimental feature**
- **Not enabled** by default
- The block guarded by `if(thread.isVirtual())` is then **dead code**

Key Observations

Key Observations

- Conditions can often be **evaluated** with a simple **value flow analysis**
  - **No need** for SMT solving

Key Observations

- Conditions can often be **evaluated** with a simple **value flow analysis**
  - **No need** for SMT solving
  - We have to **encode** the relationship between the condition and its branches

Key Observations

- Conditions can often be **evaluated** with a simple **value flow analysis**
  - **No need** for SMT solving
  - We have to **encode** the relationship between the condition and its branches
- Considering only types is **not enough**, we have to propagate:
  - **nullability** of references
  - **primitive values** across method boundaries

Key Observations

- Conditions can often be **evaluated** with a simple **value flow analysis**
  - **No need** for SMT solving
  - We have to **encode** the relationship between the condition and its branches
- Considering only types is **not enough**, we have to propagate:
  - **nullability** of references
  - **primitive values** across method boundaries
- Conditions, e.g. null-check, **filter** their input values
  - More precise information is known within the **successor branches**

```
if(x instanceof A){
   foo(x); // here x is a subtype of A
}
```

Key Observations

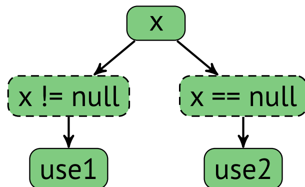
- Conditions can often be **evaluated** with a simple **value flow analysis**
  - **No need** for SMT solving
  - We have to **encode** the relationship between the condition and its branches
- Considering only types is **not enough**, we have to propagate:
  - **nullability** of references
  - **primitive values** across method boundaries
- Conditions, e.g. null-check, **filter** their input values
  - More precise information is known within the **successor branches**

```
if(x instanceof A){
   foo(x); // here x is a subtype of A
}
```

- We have **expressed** all the cases above as an **extension of points-to analysis**

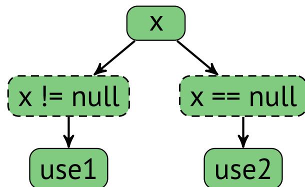
Branch-Specific FilterFlows

```
if (x != null) {
   use1(x);
} else {
   use2(x);
}
```



Branch-Specific FilterFlows

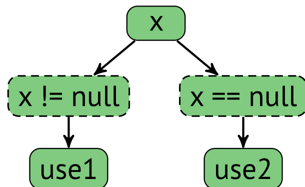
```
if (x != null) {
   use1(x);
} else {
   use2(x);
}
```



- Condition is **split** into multiple **FilterFlows**

Branch-Specific FilterFlows

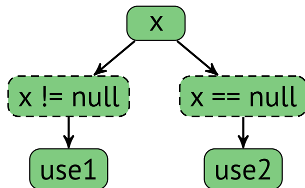
```
if (x != null) {
    use1(x);
} else {
    use2(x);
}
```



- Condition is **split** into multiple **FilterFlows**
- Each **FilterFlow** filters the input based on a **condition** (e.g. null-check)

Branch-Specific FilterFlows

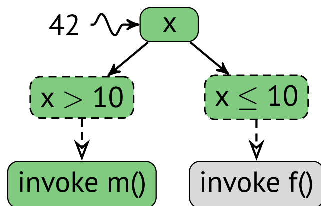
```
if (x != null) {
   use1(x);
} else {
   use2(x);
}
```



- Condition is **split** into multiple **FilterFlows**
- Each **FilterFlow** filters the input based on a **condition** (e.g. null-check)
- Nodes from individual branches **use** the nearest FilterFlow **instead of** x

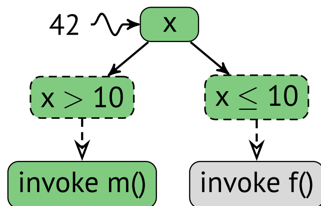
Predicate Edges

```
if (x > 10) {
   m();
} else {
   f();
}
```



Predicate Edges

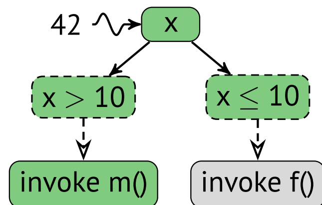
```
if (x > 10) {
   m();
} else {
   f();
}
```



- **Predicate edges** established between **conditions** and **nodes from branches**

Predicate Edges

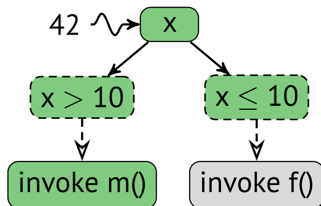
```
if (x > 10) {
   m();
} else {
   f();
}
```



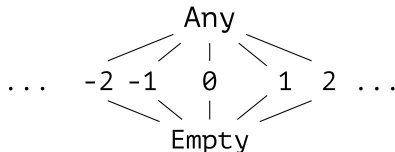
- **Predicate edges** established between **conditions** and **nodes from branches**
- **Target** of a predicate edge propagates value iff the **source** has **non-empty state**

Predicate Edges

```
if (x > 10) {
   m();
} else {
   f();
}
```



- **Predicate edges** established between **conditions** and **nodes from branches**
- **Target** of a predicate edge propagates value iff the **source** has **non-empty state**
- **Primitives** modelled using a simple **3-tier lattice**



Evaluation

- Prototype implemented in **GraalVM Native Image**

Evaluation

- Prototype implemented in **GraalVM Native Image**
- **Evaluated on**:
  - **Renaissance** 0.15.0 (R) – a well-established Java benchmark suite
  - **Dacapo** 9.12 (D) – a well-established Java benchmark suite
  - **Microservices** (M) – a set of microservice applications using various frameworks

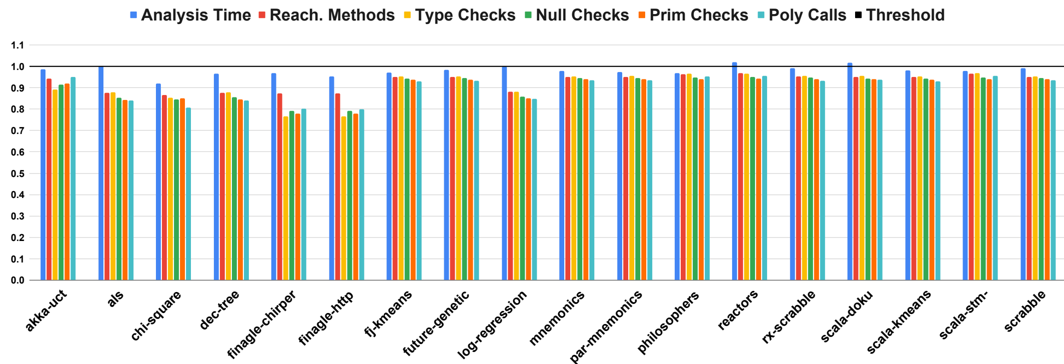
Evaluation

- Prototype implemented in **GraalVM Native Image**
- **Evaluated on**:
  - **Renaissance** 0.15.0 (R) – a well-established Java benchmark suite
  - **Dacapo** 9.12 (D) – a well-established Java benchmark suite
  - **Microservices** (M) – a set of microservice applications using various frameworks
- **Metrics**:
  - **Analysis Time**
  - **Reachable Methods**
  - **Counter Metrics** – how many instances of given instructions could not be optimized

Evaluation

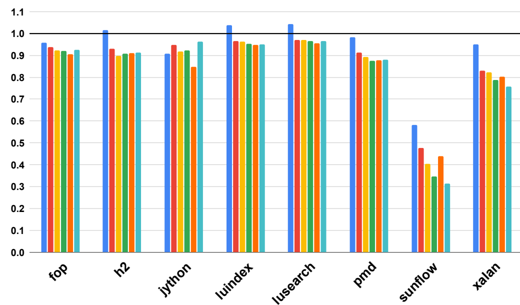
- Prototype implemented in **GraalVM Native Image**
- **Evaluated on**:
  - **Renaissance** 0.15.0 (R) – a well-established Java benchmark suite
  - **Dacapo** 9.12 (D) – a well-established Java benchmark suite
  - **Microservices** (M) – a set of microservice applications using various frameworks
- **Metrics**:
  - **Analysis Time**
  - **Reachable Methods**
  - **Counter Metrics** – how many instances of given instructions could not be optimized
    - Type Checks
    - Null Checks
    - Primitive Checks
    - Polymorphic Calls

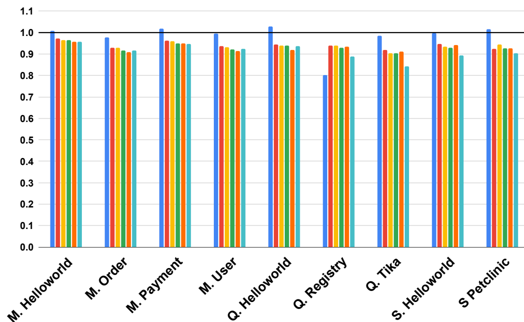
Normalized Metrics



Normalized Metrics

■ Analysis Time ■ Reach. Methods ■ Type Checks ■ Null Checks ■ Prim Checks ■ Poly Calls ■ Threshold





Overall Results

Reduction in **reachable methods** per bench suite (other metrics follow the same trend):

- **Renaissance** – max 13.5%, min 3.1%, avg 7.2%
- **Dacapo** – max 52.3%, min 3.1%, avg 12.9%
- **Microservices** – max 8.2%, min 2.7%, avg 5.8%

Overall **8.2%** reduction **without negatively impacting the analysis time**

Status Update

Status Update

- **Accepted Publications:**
  - Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image – **MPLR 23 (Core C)**
    - Small, **specialised**, strong participation of the community
  - Software Architecture Reconstruction for Microservice Systems using Static Analysis via GraalVM Native Image – **SANER 24 (Core A)**
    - With **Tomas Cerny** from Baylor University, now University of Arizona
  - Scaling Type-Based Points-to Analysis with Saturation – **PLDI 24 (Core A*)**
    - **Flagship conference** (Google Scholar H5: 50, #9 of all publication channels in Software Systems)
- **Submitted Publications:**
  - Partially Flow-Sensitive Points-to Analysis using Predicates – **OOPSLA 24 (Core A)**

Status Update

- **Accepted Publications:**
  - Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image – **MPLR 23 (Core C)**
    - Small, **specialised**, strong participation of the community
  - Software Architecture Reconstruction for Microservice Systems using Static Analysis via GraalVM Native Image – **SANER 24 (Core A)**
    - With **Tomas Cerny** from Baylor University, now University of Arizona
  - Scaling Type-Based Points-to Analysis with Saturation – **PLDI 24 (Core A*)**
    - **Flagship conference** (Google Scholar H5: 50, #9 of all publication channels in Software Systems)
- **Submitted Publications:**
  - Partially Flow-Sensitive Points-to Analysis using Predicates – **OOPSLA 24 (Core A)**
- **PhD Checklist:**
  - Time: Fulfilling the plan, want to finish in **1-2 years**
  - Publications: **3** accepted, **1** submitted
  - Quality Publications (Core B+): **2** accepted (A*,A), **1** submitted (A)
  - Internship, international projects: **3.5 years** at the **GraalVM** team at **Oracle Labs**

Appendix – All Publications

- **Accepted:**
  - Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image – **MPLR 23 (Core C)** – small, **specialised**, strong participation of the community
  - Software Architecture Reconstruction for Microservice Systems using Static Analysis via GraalVM Native Image – **SANER 24 (Core A)** – with **Tomas Cerny** from Baylor University, now University of Arizona
  - Scaling Type-Based Points-to Analysis with Saturation – **PLDI 24 (Core A*)** – **flagship conference** (Google Scholar H5: 50, #9 of all publication channels in Software Systems)
- **Submitted:**
  - Partially Flow-Sensitive Points-to Analysis using Predicates – **OOPSLA 24 (Core A)**
- **In the making:**
  - Extending the OOPSLA paper with **richer domains** for primitive values
  - Improving points-to analysis via **compiler optimizations**
  - Tools for **debugging** points-to analysis
- **Planned:**
  - **”Baseline”** Native Image points-to analysis paper
  - **Change-impact** analysis for microservices

Appendix II.

Running Example

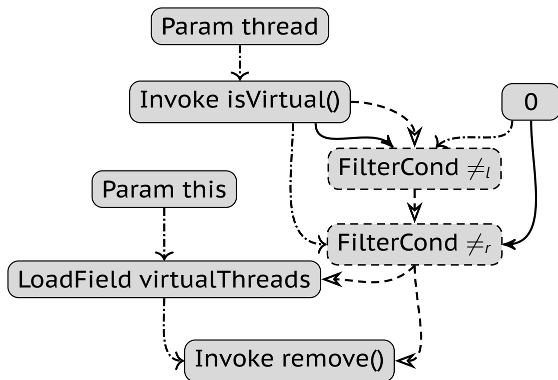
Running Example – Source

```
class SharedThreadContainer {
   Set<Thread> virtualThreads;
   public void onExit(Thread thread) {
     if (thread.isVirtual())
       virtualThreads.remove(thread);
   }
}
```

```
class Thread {
   public boolean isVirtual() {
     return this instanceof
          BaseVirtualThread;
   }
}
```

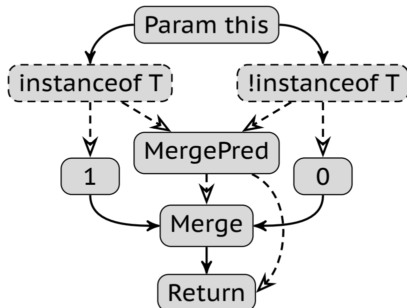
Running Example – TypeFlowGraph onExit

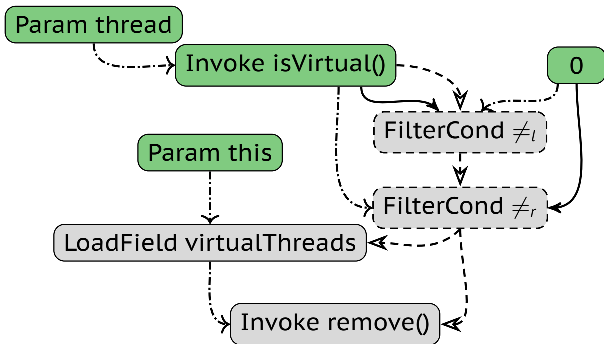
```
class SharedThreadContainer {
   Set<Thread> virtualThreads;
   public void onExit(Thread thread) {
     if (thread.isVirtual())
       virtualThreads.remove(thread);
   }
}
```



Running Example – TypeFlowGraph isVirtual

```
class Thread {
   public boolean isVirtual() {
     return this instanceof
          BaseVirtualThread;
   }
}
```

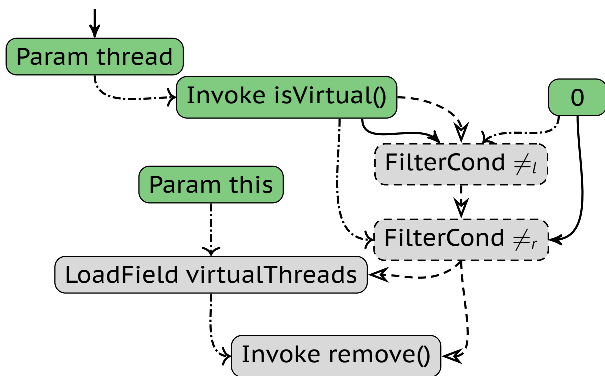





```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

```
class Thread {
    public boolean isVirtual() {
      return this instanceof
                BaseVirtualThread;
    }
}
```

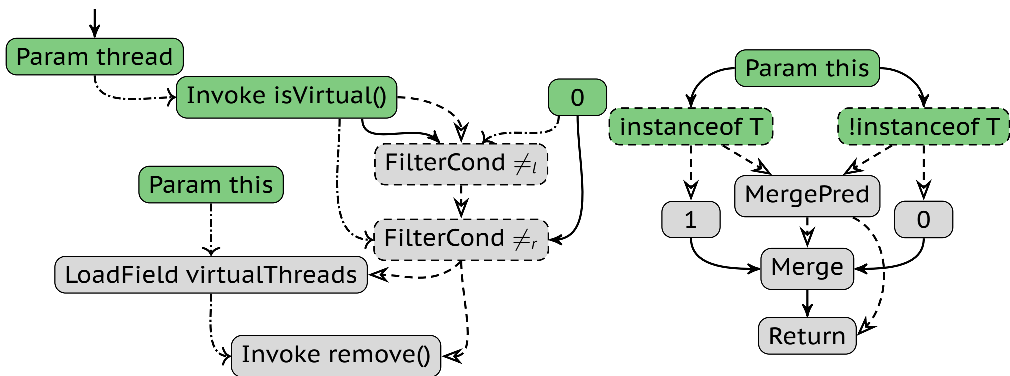
$\texttt{VirtualThread} \notin VS_{in}(Param\ thread)$




```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

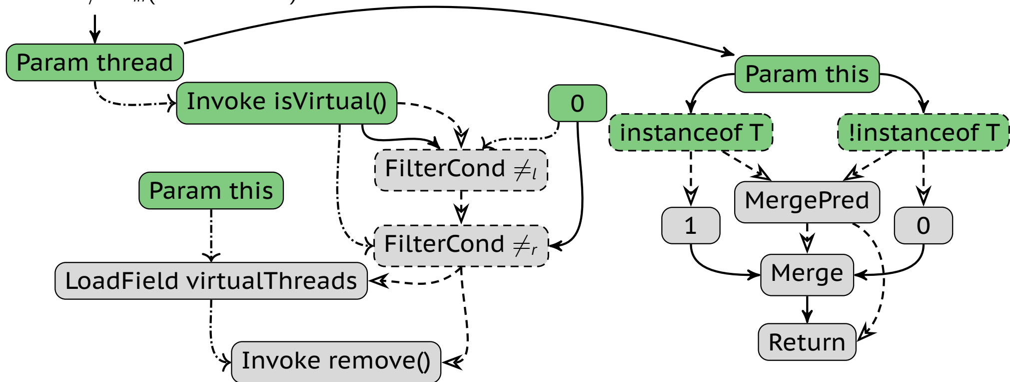
```
class Thread {
    public boolean isVirtual() {
      return this instanceof
              BaseVirtualThread;
    }
}
```

$\texttt{VirtualThread} \notin VS_{in}(\mathit{Param\ thread})$




```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if  (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

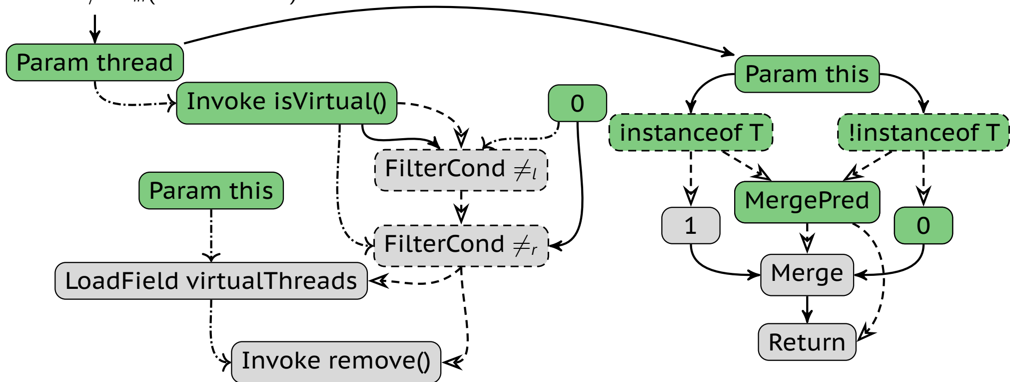
```
class Thread {
    public boolean isVirtual() {
      return this instanceof
              BaseVirtualThread;
    }
}
```




```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

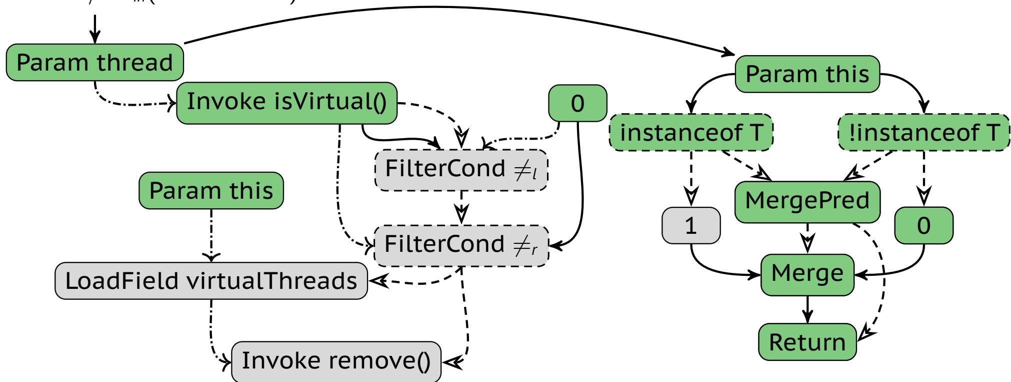
```
class Thread {
    public boolean isVirtual() {
      return this instanceof
            BaseVirtualThread;
    }
}
```

$\texttt{VirtualThread} \notin VS_{in}(Param\ thread)$



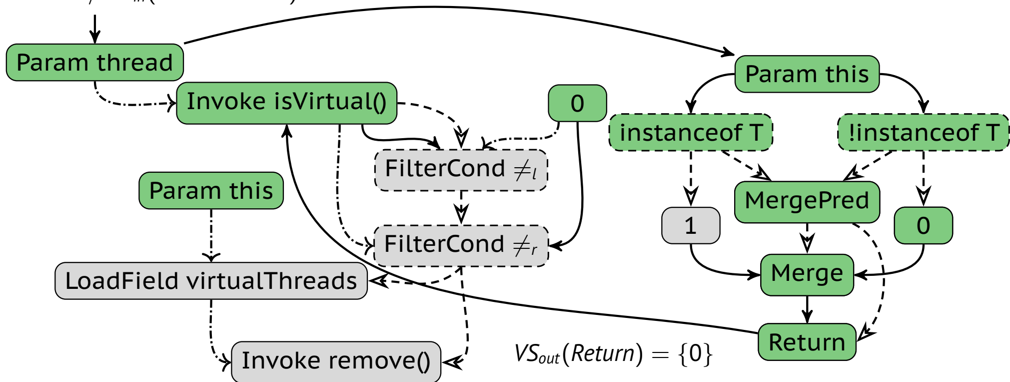

```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

```
class Thread {
    public boolean isVirtual() {
      return this instanceof
              BaseVirtualThread;
    }
}
```



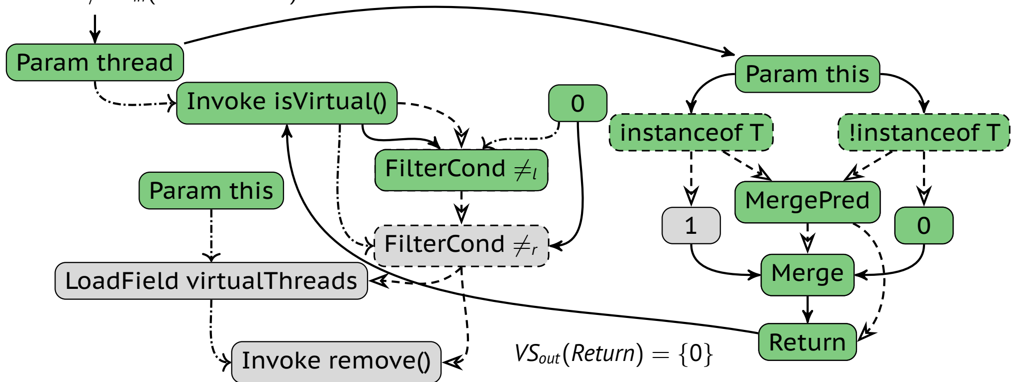

```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

```
class Thread {
    public boolean isVirtual() {
      return this instanceof
              BaseVirtualThread;
    }
}
```




```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

```
class Thread {
    public boolean isVirtual() {
      return this instanceof
               BaseVirtualThread;
    }
}
```




```
class SharedThreadContainer {
    Set<Thread> virtualThreads;
    public void onExit(Thread thread) {
      if (thread.isVirtual())
        virtualThreads.remove(thread);
    }
}
```

```
class Thread {
    public boolean isVirtual() {
      return this instanceof
                BaseVirtualThread;
    }
}
```