Comparison of Points-to Analyses

Tobias Gutzmann
Master Thesis

Comparison of Points-to Analyses

Tobias Gutzmann

2008

Institut für Programmstrukturen
und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe

Department of Computer Science
Växjö University

Prof. Dr. Welf Löwe
Prof. em. Dr. Dr. h.c. Gerhard Goos
Abstract

Points-to analysis is a static program analysis which computes possible reference relations between different parts of a program. It serves as input to many high-level analyses. Points-to analyses differ, among others, in flow- and context-sensitivity, program representation, and object abstraction. Most program representations used for points-to analysis are sparse representations which abstract from, e.g., primitive data types and intra-procedural control-flow. Thus, a certain degree of information is sacrificed for compact program representation, which results in scalable performance. In this thesis, we present a framework which allows building different versions of Points-to SSA (P2SSA), a sparse, Memory SSA based program representation. Distinct instantiations of P2SSA contain different levels of abstraction from a program's full representation. We present another framework which allows running Points-to analyses on these program representations. We use these two frameworks to instantiate different versions of P2SSA and compare them in terms of analysis precision and execution time.
## Contents

1 Introduction
   1.1 Problem and Motivation ............................................. 1
   1.2 Goal Criteria .................................................. 1
   1.3 Outline ......................................................... 2

2 State of the Art ..................................................... 3
   2.1 Concepts ....................................................... 3
   2.2 Existing Approaches ............................................. 4
   2.3 Conclusion ..................................................... 5

3 Framework ........................................................... 6
   3.1 Firm vs. P2SSA .................................................. 6
   3.2 Building ........................................................ 6
      3.2.1 Entity Mapping ............................................. 8
      3.2.2 Graph Mapping ............................................ 9
   3.3 Analysis ........................................................ 14
      3.3.1 Framework Contracts .................................... 14
      3.3.2 Native Methods .......................................... 18
   3.4 Conclusion ..................................................... 18

4 Implementation ...................................................... 19
   4.1 Run in Detail ................................................... 19
      4.1.1 Firm Build .................................................. 19
      4.1.2 Building P2SSA ........................................... 20
      4.1.3 Running Analyses ......................................... 21
   4.2 Interpretation of Values ........................................ 21
   4.3 Memory Modeling ................................................ 22
   4.4 Framework Instantiation ....................................... 22
      4.4.1 Graph Stabilizer .......................................... 22
      4.4.2 Context Strategies ....................................... 22
      4.4.3 Name Schemata ........................................... 24
      4.4.4 Semantics ................................................ 24
      4.4.5 Call Replacer ............................................. 30
      4.4.6 Native Methods .......................................... 31
   4.5 Optimizations .................................................. 31
      4.5.1 NoLoad Methods .......................................... 31
      4.5.2 Handling of null .......................................... 32
   4.6 Optimistic Approximations ..................................... 32
   4.7 Contributions to Third-Party Libraries ....................... 32
   4.8 Limitations .................................................... 34
      4.8.1 Exceptions ................................................ 34
      4.8.2 Threads .................................................... 34
      4.8.3 Reflection ................................................ 34
      4.8.4 Boolean Operations ...................................... 34
   4.9 Conclusion ..................................................... 34
## List of Figures

3.1 Java Source Code and Firm Graphs ........................................ 7
3.2 Mapped P2SSA Graphs from Figure 3.1 ................................. 7
3.3 Build framework .......................................................... 8
3.4 Identifying Reachable Blocks ............................................. 10
3.5 SSA program structure .................................................... 14
3.6 Analysis Framework ....................................................... 15
3.7 Analysis Run ............................................................... 16
4.1 Dangling Cast .............................................................. 21
4.2 Analysis framework instantiation ........................................ 23
4.3 (a) Common SSA-graph (b) SSA-graph with Special Node □ .... 29
4.4 Read-Read Conflict ....................................................... 29
4.5 Return Types of Native Methods ......................................... 31
5.1 Null Pointer Check ......................................................... 36
## List of Tables

3.1 Basic Nodes .................................................. 11
3.2 Nodes for Conditional Execution ......................... 12
3.3 Optional Nodes for Primitive Types .................... 12
3.4 Matching Patterns in Firm Graphs ...................... 12
3.5 Overview of the Main Configuration Options .......... 17
4.1 Nodes Introduced by Plugin System .................... 30
4.2 Optimistic Approximations .............................. 33
5.1 Types for Which to Use a Class Name Schema .......... 37
5.2 Evaluation Setup - Semantics and Context Sensitivity ...... 37
5.3 Benchmark Programs ................................... 38
5.4 Program Sizes ........................................ 38
5.5 Analysis Results - Performance ....................... 40
5.6 Analysis Results - Call Graph ........................ 41
5.7 Analysis Results - Other Metrics .................... 42
C.1 Options for Constructing P2SSA ......................... 51
C.2 Options for Configuring the Analysis .................. 52
1 Introduction

Points-to analysis is a good overview is given in [Hin01] is a static program analysis aiming at analyzing the reference structure of dynamically allocated objects at compile-time. It comes in many variants ranging from context- and flow-insensitive analyses, which are fast but imprecise, to context- and flow-sensitive analyses, which are exponential in time but very precise. The ultimate goal is, however, to find an analysis that is scalable for large programs and sufficiently precise for a client application.

1.1 Problem and Motivation

This thesis project evaluates two points-to analyses experimentally - both are context-insensitive and flow-sensitive. They differ in the program representation they are to: one operates on Points-to-SSA, a program representation in Static Single Assignment (SSA) form that abstracts from all but pointer-related operations and data-types. This makes the program representation very compact and, hence, program analysis fast. However, it also excludes some program simplifications such as, e.g., dead-code elimination, that may be enabled due to intermediate analysis results. Therefore, a second, yet to develop, program representation regards Boolean operations on pointers that are input to control statements.

The open research questions are:

- Which variant is more efficient in terms of execution time?
- Which variant is more effective in terms of analysis precision?

These questions cannot be decided theoretically. Instead, experiments on a set of standard benchmarks should give evidence, which variant is to prefer. We decided to execute these experiments on Java programs (as most recent work in the field does).

1.2 Goal Criteria

As the primary goal, the following tasks need to be performed:

- A front-end for reading Java source and byte codes needs to be connected to a library constructing SSA intermediate representations.
- Filtering Firm SSA and constructing the different variants of Points-to SSA. Although Points-to SSA exists, redesign seems to be appropriate since it was not designed for reuse in the first place.
- Implementing the context-insensitive and flow-sensitive Points-to analysis based on the two variants of Points-To SSA. The challenge is to interleave this analysis with dead-code elimination for optimizing the Points-To SSA variant containing Boolean operations on pointers and control statements.
- Running comparative measurements on the two versions of Points-to analysis. While the performance measurements are rather straightforward, the measurements of accuracy are a challenge in itself. Different metrics are defined in literature - they are motivated by different client applications - and one has to select a representative set of accuracy metrics.
As a secondary goal, the project aims at providing an experimental framework for more experiments of the same kind. Hence, the goal criteria are maintainability and reusability of the software to develop. Further, the analysis is aimed at software comprehension tools.

1.3 Outline
We will begin this thesis with an overview of previous work done in the field of points-to analysis in Chapter 2. In Chapter 3, we will then present two frameworks: One for building different versions of Points-to SSA, our intermediate SSA representation, and one for running analyses on Points-to SSA. We will present the use of these frameworks in Chapter 4. In Chapter 5, we will run benchmarks to compare different versions of Points-to SSA in terms of precision and runtime. In Chapter 6, we will summarize the results of this thesis and show possible further experiments that can be conducted with help of our frameworks.
2 State of the Art

In this chapter, we give an overview of the state of the art in static program analysis, with a focus on those related to points-to analysis. We begin with a summary of concepts, followed by concrete applications of these.

2.1 Concepts

Data flow analysis in general is a static analysis technique for gathering information about the possible set of values at various program points that might occur at runtime. A Points-to analysis in particular focuses on the analysis of operations that process pointers. Its goal is to compute, for each pointer-valued expression $e$ at each point in a program, the points-to set $\text{pt}(e)$ of possible objects it may refer to. For simplicity, we refer to an expression $e$ and its points-to set $\text{pt}(e)$ interchangeably in this thesis. In Java, which we analyze in this thesis, pointers do not exist, but references. We will still use the term points-to-analysis instead of references-analysis, as this is the common nomenclature in literature.

A data flow analysis needs to approximate its analysis results; an exact analysis is in general not possible, as all possible execution paths of a given program would need to be analyzed. Such an approximation is either optimistic or pessimistic, yielding either stronger or weaker conclusions than the exact statement, respectively. For a points-to analysis, this means either an under- or overestimation of the obtained points-to sets. We will refer to pessimistic approximations as conservative.

A program analysis needs to abstract from the values which expressions may take during a real application run in some way. The abstract values are chosen such that they form a value lattice $(V, \sqsubseteq, \sqcup, \sqcap, \bot)$ [NNH99]. Here, $(V, \sqsubseteq)$ are a partially ordered set, $\sqcap$ and $\sqcup$ compute the infimum and supremum, respectively. $\top$ and $\bot$ are the largest and smallest elements in $V$ ($\top$ represents all abstract values, $\bot$ none).

Literature has proposed several name schemata to select the elements for such a lattice. The simple class name schema abstracts all objects to their type. A more fine-grained, yet easy to implement and handle name schema identifies each abstract object with its syntactical allocation site [Tra01]. The approach of [MRR02] can even distinguish multiple abstract objects of one allocation site by taking contexts into consideration, see below. There is no limit to the granularity of a name schema, as long as a concrete runtime object cannot be given multiple names, i.e., when a name $n \in N$ is a classification of one or more run time objects $o(n)$, then the following must hold: $\forall n_1, n_2 \in N : n_1 \neq n_2 \Rightarrow o(n_1) \cap o(n_2) = \emptyset$.

A simple way to perform data flow analysis of programs is to set up data flow equations and solve them by repeatedly calculating them until the system stabilizes, i.e., it reaches a fixpoint. To guarantee termination, the transfer functions of the data flow equations are required to be monotone, i.e., for a function $f$ and two input values $x, y : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

If analysis results can be overwritten by subsequent analysis results, we speak of strong updates in contrast to weak updates, where this is not possible.

An important decision that affects the precision (and cost) of a points-to analysis is whether it is flow-sensitive or -insensitive. Flow-sensitivity defines whether or not an analysis takes the order of execution of operations into consideration. A flow-insensitive analysis, for example, will decide that after execution of $y=1; x=y; y=2;$, the variable $x$ will have a value of either 1 or 2, while a flow-sensitive analysis will compute the precise result. Flow-sensitive analyses are obviously more precise.
than flow-insensitive analyses, however, they are also more costly. A possibility to introduce local flow-sensitivity is by using Static Single Assignment (cf. appendix A) form \cite{CFR+89}, in short SSA, for intermediate program representation \cite{HH98}.

An operation \textit{op} on a syntactical location \textit{s} in a program may be reached through multiple predecessor statements, therefore an analysis may distinguish distinct execution paths in order to improve precision. This can either be done on an \textit{intra-procedural} level, in which case we speak of \textit{path-sensitivity}, or on an \textit{inter-procedural} level, in which case we speak of \textit{context-sensitivity}. Note that a path-sensitive dataflow analysis is necessarily flow-sensitive. An example for a path-sensitive implementation technique are \(\chi\)-terms \cite{Tra01}.

However, recent works have focused on inter-procedural context-sensitivity. To distinguish contexts, virtual copies of methods, called \textit{contours}, are created. Each contour obtains its own set of variables. The question is how such contours - and, especially, in what number - are to be created for a given method. There are two major approaches; the first is the \textit{call-string} approach, where method contexts are distinguished by the syntactical location sites they are called from; usually the call-stack of depth \(k\) is taken into consideration \cite{NNH99}. The other is the \textit{functional} approach, where contours are distinguished by some means of the actual arguments passed to the method.

### 2.2 Existing Approaches

A group of similar approaches that work on class level are \textit{Reachability Analysis}, \textit{Class Hierarchy Analysis}, \textit{Rapid Type Analysis}, and \textit{Separated Type Analysis}.

A simple flow- and context-insensitive algorithm for calculating the set of reachable methods is \textit{Reachability Analysis}. The set \(R\) of reachable methods is initialized with the program’s \textit{main}-method. With \(M.m \in R\) and \(e.n\) a method invocation in \(M.m\), it follows that \(\forall N \in \text{Program} : N.m \in R \land (M.m, N.n) \) is a possible call relation within the program.

Of course, it is also possible to additionally compute the set of reachable fields. Since fields can neither reference other members, nor are they polymorphic in almost any relevant programming language, we omit the discussion here. For the following algorithms, fields can be seen as a pair \(\text{getField}()\) and \(\text{setField}()\).

A refinement of Reachability Analysis is \textit{Class Hierarchy Analysis} (CHA), which was introduced in \cite{Dea96}. It takes into account the class hierarchy: Upon a member access, only the members of classes that are subclasses of the exactly referenced one are marked as reachable.

A further refinement of CHA is \textit{Rapid Type Analysis} (RTA). Additionally to the set of reachable methods, it also maintains a set of instantiated classes. If now a class \(A\) defines a method \(m\), and class \(B\) is a subclass of \(A\) and overrides \(m\), then, when \(A.m\) is invoked, \(A.m\) is added to set of reachable methods (and the worklist) only if the algorithm has seen an allocation statement for this class \(A\) before.

An extension to this approach is called \textit{Separated Type Analysis} (XTA). Instead of maintaining a global set of types that have been allocated, it is recorded per method which types can reach each method.

An approach that works with a more granular name schema and that is context-sensitive, but still flow-insensitive, is presented in \cite{MRR02}. The context-sensitivity here is called \textit{Object-sensitivity}, where contexts are distinguished by the (\(k\)-call stack of) receiver-objects of a method invocation; thus, it is a functional context sensitivity. A common implementation is 1-object-sensitivity, which effectively creates one
contour for each \((\text{obj}, \text{method})\)-pair.

A group of flow-sensitive algorithms are \emph{k-Control Flow Analyses} (k-CFA, [Shi88]). 0-CFA is an interprocedural flow-sensitive, but context-insensitive analysis. For \(k > 0\), contexts are determined by the last \(k\) syntactical call sites. A k-CFA analysis with a refined name schema is practically a points-to analysis in our sense.

A flow- and context-sensitive approach that works on SSA graphs is presented in [LL06]. Besides local flow-sensitivity, it also achieves a certain amount of interprocedural flow-sensitivity through \emph{Simulated Execution}, where an actual program run is simulated. The context-sensitivity used here is a variation of object-sensitivity, called \emph{this-sensitivity}. It uses the set of the receiver objects to distinguish contexts instead of creating a contour for each single object.

A similar approach for achieving local and inter-procedural flow-sensitivity is used in [Lie06]. Here, contexts are distinguished by their \emph{full} acyclic call string. This yields an extremely fine-grained distinction, resulting in very precise analysis results, but being very costly in terms of time and memory.

### 2.3 Conclusion

Many approaches differing in, e.g., context-sensitivity, flow-sensitivity, and object abstraction level, exist. They provide different granularity of precision, being differently costly in terms of time and memory. Thus, tradeoffs in between both must be made, depending on area of application. A general approach which is suitable for \emph{all} application areas is unknown.
3 Framework

In this thesis, we present two frameworks: A framework for building Points-To SSA (or, in short, P2SSA), and an analysis framework, which mainly consists of interfaces and their interaction contracts. Our analysis framework is instantiated with this SSA representation, so an instantiation of it will in principle be an intraprocedurally flow-sensitive analysis. However, no assumptions are made on side of the framework on implementation techniques. Where we think it is helpful for understanding, will we give references to concrete techniques described in Chapter 2.

We now give a short justification why we provide an own SSA representation, and do not reuse Firm or the existing Points-to SSA implementation. We will then describe the two frameworks. Following the natural order, we will start with our building framework, followed by the analysis framework.

3.1 Firm vs. P2SSA

Why an own SSA representation? With Firm, a full program representation in Memory SSA form [Tra01] exists, so the question pops up why we do not run our analysis on Firm directly. Firm is designed for optimizing transformations, which requires a flexible graph structure, as well as code generation, which requires the translation of symbolic references into machine addresses – e.g., monomorphic calls are translated into a memory address, while polymorphic calls are translated into a vtable lookup. For this, Firm requires multiple (at least two) nodes for a method call, cf. Figure 3.1. For our means, one node for a call is sufficient and reduces memory consumption and operating time. Further, we do not necessarily rely on Firm - it is just one possible frontend.

Figure 3.1 shows a simple Java method and the corresponding Firm graph. Figure 3.2 shows two possible ways to map it to P2SSA: Both with exception mapping disabled, and once each with and without basic block structure mapped.

Firm uses Proj-nodes to extract certain values (e.g., memory state or method parameters) from the result-tuple of a node. While this allows for great flexibility for transformations, it also greatly increases the number of nodes required. P2SSA uses Ports instead, which are strictly attached to their defining operations. Note that some ports may not be used. In Figure 3.2, this is, e.g., the case for the implicit this-argument of the Start-node.

We have further decided to redesign Points-to SSA, as it was never intended for reuse or flexibility, so that it cannot provide the flexibility we will need as described in the next sections.

3.2 Building

Our building framework allows for construction of P2SSA from an arbitrarily chosen frontend. We will however focus on mapping programs in the Memory-SSA based Firm representation to our Memory-SSA based, sparse P2SSA representation.

The framework requires a Firm program as input. We will describe, in Section 4.1, the steps taken to obtain such a program from given Java source- and bytecode.

Taken for granted that the Firm program is at hand, the first steps are mapping entities, i.e., classes, methods, and fields, to our own program structure, and then mapping each Firm graph to a corresponding P2SSA graph. These mappings are described in Sections 3.2.1 and 3.2.2.

Figure 3.2 shows the components involved in the mapping from Firm to P2SSA. The package framework defines two interfaces: BuildEntities, which defines an ab-
String foo(Object o) {
    return o.toString();
}

Figure 3.1: Java Source Code and Firm Graphs

Figure 3.2: Mapped P2SSA Graphs from Figure 3.1
abstract factory for constructing P2SSA entities, and BuildGraph, which defines the methods required for mapping to P2SSA graphs. The package framework.firmfrontend contains our firm frontend, which is described in the next two sections. The packages build and plugin are described in Section 4.1.2.

![Diagram of the Build framework](image)

Figure 3.3: Build framework

### 3.2.1 Entity Mapping

Entity mapping is a straightforward process, since it mainly includes a one-to-one mapping. The Firm program structure is mapped to the program structure of our target representation through the interface BuildEntities described in Figure 3.2. However, since there are naturally not many variation points, the target representation will most likely look like the one shown in Figure 3.5, cf. next section.

Some types may not be desired in the target P2SSA representation and will not be mapped. This is determined by the following algorithm:

```plaintext
function isMapped(type)
    if make primitive types then
        return true
    end if
    while type isArray do
        type = type.baseType
    end while
    if type isClass then
        return true
    else if type isPrimitiveBoolean ∧ make conditional execution then
        return true
    else
        return false
    end if
end function
```

Thus, all types are mapped if and only if the option `map primitive types` is set.

It is also possible to provide several instances of BuildEntityInterface. This enables to provide plugins, e.g., it may be useful to map a set of classes from the Java
Collection Framework to a single target class. Together with the possibility to use method handlers written in Java (cf. Section 3.3.2) instead of P2SSA graphs, this allows for performance improvements of this for many application crucial framework.

Information from the Firm program on which method is the program entry method - i.e., the program’s main-method - and which system initialization calls must be made before initialization - for Java, this is the method *System.initializeSystemClass* - is also obtained from the frontend.

### 3.2.2 Graph Mapping

The basic idea of the graph mapping algorithm is straight-forward: First, mapping the Firm nodes, i.e., the operations, to P2SSA nodes, and then assigning the control- and dataflow edges. This comes close to a one-to-one mapping. However, some features like primitive types or exception handling may not be desired by the target P2SSA representation, so we need to take some extra care.

As a first step, we do a coarse grained approximation of reducing the code we map to P2SSA by identifying a Firm subgraph which is sufficient to take into consideration. Next we will perform the node mapping from Firm to P2SSA. In a third step, we add the control and dataflow edges. We will now take a closer look at these three passes.

**Pass 1: Identification of Needed Blocks**

We perform this step to avoid removing unreachable nodes and their associated edges from the P2SSA graph later on.

We identify necessary blocks in two steps: (1) identifying basic blocks which can be reached through non-exception related control flow, and (2) identifying basic blocks which contain at least one semantically relevant operation.

*Finding reachable basic blocks*

This step is performed only if exception handling constructs shall not be mapped. The idea for finding non-exception related blocks is to identify all possible paths from the start block to the end block, taking only non-exception edges. Basic blocks on such paths are relevant. Figure 3.4 describes the general idea of the used algorithm.

*Finding basic blocks containing relevant nodes*

We narrow the set of relevant basic blocks received from the previous step further by identifying those that are *empty*. A basic block is empty if it does not contain any node we will attempt to map to our target representation, which depends on configuration options and will be discussed in the next section. This is just an approximation: As we will see, not all nodes that *can* be mapped *will* also be mapped. Hence, empty blocks may be present in the mapped SSA graph.

**Pass 2: Node Mapping**

Once we identified the basic blocks that need to be considered, node matching is basically a straight-forward process: We map Firm nodes to corresponding nodes of our intermediate representation, and possibly collect information from operand edges.

We also mark relevant Firm nodes for which we do not build a node in our target representation. We will need this for edge mapping, cf. Section 3.2.2.
function \text{FindNonExceptionBlocks}(start, end) \\
\quad U \leftarrow \emptyset \quad \triangleright \text{set of unreachable blocks} \\
\quad R \leftarrow start \quad \triangleright \text{initialize set of reachable blocks with start block} \\
\quad W \leftarrow end \quad \triangleright \text{initialize work list with end block} \\
\quad \text{while } W \neq \emptyset \text{ do} \\
\quad \quad c \leftarrow W.\text{peek} \quad \triangleright \text{c is current work block.} \\
\quad \quad isBad \leftarrow true \quad \triangleright \text{Boolean helper variable.} \\
\quad \quad \text{for all } e \in c.\text{NonExceptionInEdges} \text{ do} \\
\quad \quad \quad i \leftarrow e.\text{sourceBlock} \quad \triangleright \text{source block reachable?} \\
\quad \quad \quad \text{if } i \in R \text{ then} \\
\quad \quad \quad \quad R \leftarrow R \cup c \quad \triangleright \text{Add } c \text{ to reachable blocks,} \\
\quad \quad \quad \quad W \leftarrow W \setminus c \quad \triangleright \text{and remove from worklist.} \\
\quad \quad \quad \quad isBad \leftarrow false \\
\quad \quad \quad \text{else if } i \notin U \text{ then} \\
\quad \quad \quad \quad W \leftarrow W \cup i \quad \triangleright \text{Add to worklist. This block...} \\
\quad \quad \quad \quad isBad \leftarrow false \quad \triangleright \ldots \text{could be a valid predecessor of } c \\
\quad \quad \quad \text{end if} \\
\quad \quad \text{end for} \\
\quad \quad \text{if isBad then} \\
\quad \quad \quad U \leftarrow U \cup c \quad \triangleright \text{Mark as unreachable,} \\
\quad \quad \quad W \leftarrow W \setminus c \quad \triangleright \text{and remove from work list} \\
\quad \quad \text{end if} \\
\quad \text{end while} \\
\quad \text{return } R \\
end function

Figure 3.4: Identifying Reachable Blocks
Table 3.1 lists the nodes which must always be supported by the backend. With these nodes mapped, it is guaranteed that operands of type reference value and memory state are always available, see next section. A baseline points-to analysis can therefore be performed with just these node types. Table 3.2 lists the nodes which are required to support conditional execution. BasicBlock, Branch and Jmp nodes form the basic requirement, while all other nodes are optional. The graph builder implementation may return null for those. Table 3.3 lists the nodes which may be built if support for primitive types is enabled. None of them is mandatory: for example, it is possible to only support simple operations like 'plus' and 'minus'.

The signatures describe the operands and results of the nodes. $M$ stands for memory, $R$ for reference value, $I$ for integer value of arbitrary size (e.g., short, int, long, ...), $B$ for boolean value, $N$ for numerical value (e.g., integral, boolean, and floating point values), $V$ for any arbitrary value, and $X$ for right of execution. Square brackets stand for optional values, i.e., depending for Store, Load, and Call on whether a static member is referenced, for Call, Start and Return on the corresponding method's signature, and for the MemPhi and ValuePhi the number of data flow predecessors. Note that, for simplicity of implementation, our framework does not check if a mapped $\phi$-node will have at least two operands, as is usually demanded by definition.

<table>
<thead>
<tr>
<th>name</th>
<th>semantics</th>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc</td>
<td>object allocation</td>
<td>$M \rightarrow M \times R$</td>
</tr>
<tr>
<td>Call</td>
<td>method invocation</td>
<td>$M[\times V_1 \times \ldots \times V_n] \times R \rightarrow M[\times V]$</td>
</tr>
<tr>
<td>Cast</td>
<td>type cast</td>
<td>$R \rightarrow R$</td>
</tr>
<tr>
<td>Constant</td>
<td>constant declaration</td>
<td>$\bot \rightarrow V$</td>
</tr>
<tr>
<td>End</td>
<td>sink of graph</td>
<td>does not apply</td>
</tr>
<tr>
<td>Load</td>
<td>load value from memory</td>
<td>$M \times R \rightarrow M \times V$</td>
</tr>
<tr>
<td>MemPhi</td>
<td>phi for memory</td>
<td>$M_1[\times M_2 \times \ldots \times M_n] \rightarrow M$</td>
</tr>
<tr>
<td>Return</td>
<td>return from method</td>
<td>$M[\times V] \rightarrow X$</td>
</tr>
<tr>
<td>Start</td>
<td>head of graph</td>
<td>$\bot \rightarrow M \times X[\times V_1 \times \ldots \times V_n]$</td>
</tr>
<tr>
<td>Store</td>
<td>store value to memory</td>
<td>$M[\times R] \times V \rightarrow M$</td>
</tr>
<tr>
<td>ValuePhi</td>
<td>phi for value</td>
<td>$V_1[\times V_2 \times \ldots \times V_n] \rightarrow V$</td>
</tr>
</tbody>
</table>

Table 3.1: Basic Nodes

Table 3.4 describes the cases when we map patterns instead of single nodes. Here, we add information for which Firm uses separate nodes directly to the P2SSA node. For example, the target for a Call-node in Firm is either encoded in an EntitySelect or a Constant node, depending on whether the call is polymorphic or monomorphic.

A special case are proj-nodes. While we usually omit Proj-nodes and use Ports instead, they have a special meaning for Compare-nodes: Firm does not distinguish the nature of a comparison - e.g., equal, lower than, ... - by providing different node types, but encodes the selection of the result in a proj-node. Therefore, the last pattern listed in the table is mapped to different P2SSA nodes according to the nature of the proj-node.

**Implementation notes**
Java bytecode does not explicitly provide boolean values. Instead, semantics known from C is used within the virtual machine, meaning that any nonzero integral value
<table>
<thead>
<tr>
<th>name</th>
<th>semantics</th>
<th>signature</th>
<th>optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>boolean and</td>
<td>$B \times B \to B$</td>
<td>yes</td>
</tr>
<tr>
<td>BasicBlock</td>
<td></td>
<td>$X_1 \times X_2 \times \ldots \times X_n \to X$</td>
<td>no</td>
</tr>
<tr>
<td>Branch</td>
<td>conditional jump</td>
<td>$B \to X \times X$</td>
<td>no</td>
</tr>
<tr>
<td>EQ</td>
<td>values equal</td>
<td>$V \times V \to B$</td>
<td>yes</td>
</tr>
<tr>
<td>InstanceOf</td>
<td>Runtime type check</td>
<td>$R \to B$</td>
<td>yes</td>
</tr>
<tr>
<td>Jmp</td>
<td>unconditional jump</td>
<td>$\bot \to X$</td>
<td>no</td>
</tr>
<tr>
<td>NEQ</td>
<td>values unequal</td>
<td>$V \times V \to B$</td>
<td>yes</td>
</tr>
<tr>
<td>Not</td>
<td>boolean not</td>
<td>$B \to B$</td>
<td>yes</td>
</tr>
<tr>
<td>Or</td>
<td>boolean or</td>
<td>$B \times B \to B$</td>
<td>yes</td>
</tr>
<tr>
<td>XOR</td>
<td>boolean exclusive or</td>
<td>$B \times B \to B$</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3.2: Nodes for Conditional Execution

<table>
<thead>
<tr>
<th>name</th>
<th>semantics</th>
<th>signature</th>
<th>optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitAnd</td>
<td>bitwise and</td>
<td>$N \times N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>BitOr</td>
<td>bitwise or</td>
<td>$N \times N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>BitXOR</td>
<td>bitwise exclusive or</td>
<td>$N \times N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>Div</td>
<td>numerical division</td>
<td>$N \times N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>Modulo</td>
<td>integral modulo</td>
<td>$I \times I \to I$</td>
<td>yes</td>
</tr>
<tr>
<td>Multi</td>
<td>numerical multiplication</td>
<td>$N \times N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>Add</td>
<td>numerical addition</td>
<td>$N \times N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>Sub</td>
<td>numerical subtraction</td>
<td>$N \times N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>Negate</td>
<td>sign inversion</td>
<td>$N \to N$</td>
<td>yes</td>
</tr>
<tr>
<td>Shl</td>
<td>bitwise shift left</td>
<td>$I \times I \to I$</td>
<td>yes</td>
</tr>
<tr>
<td>Shr</td>
<td>bitwise shift right</td>
<td>$I \times I \to I$</td>
<td>yes</td>
</tr>
<tr>
<td>Shrl</td>
<td>rotate bits left</td>
<td>$I \times I \to I$</td>
<td>yes</td>
</tr>
<tr>
<td>BitInvert</td>
<td>invert (1-complement)</td>
<td>$I \to I$</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3.3: Optional Nodes for Primitive Types

<table>
<thead>
<tr>
<th>Firm pattern</th>
<th>P2SSA</th>
<th>remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntitySelect $ptr \leftarrow Call $</td>
<td>Call</td>
<td>polymorphic</td>
</tr>
<tr>
<td>Constant $ptr \leftarrow Call $</td>
<td>Call</td>
<td>monomorphic</td>
</tr>
<tr>
<td>EntitySelect $ptr \leftarrow Load $</td>
<td>Load</td>
<td>instance field</td>
</tr>
<tr>
<td>Const $ptr \leftarrow Load $</td>
<td>Load</td>
<td>class field</td>
</tr>
<tr>
<td>EntitySelect $ptr \leftarrow Load ; ref \leftarrow Load ; ptr \leftarrow ArraySelect $</td>
<td>Load</td>
<td>array</td>
</tr>
<tr>
<td>EntitySelect $ptr \leftarrow Store $</td>
<td>Store</td>
<td>instance field</td>
</tr>
<tr>
<td>Const $ptr \leftarrow Store $</td>
<td>Store</td>
<td>class field</td>
</tr>
<tr>
<td>EntitySelect $ptr \leftarrow Load ; ref \leftarrow Store ; ptr \leftarrow ArraySelect $</td>
<td>Store</td>
<td>array</td>
</tr>
<tr>
<td>Compare $\leftarrow Proj $</td>
<td>EQ,NEQ</td>
<td>compare nodes</td>
</tr>
</tbody>
</table>

Table 3.4: Matching Patterns in Firm Graphs
is considered to be true, see [LY99], Chapter 2.4.5. Therefore we map Firm integer constants zero and one to boolean constants false and true, respectively, unless full mapping of primitive types is enabled.

**Pass 3: Edge Mapping**

For edge mapping, we once again take a look at every node in the Firm graph that we have mapped to P2SSA.

How to find the operand for a node in P2SSA depends on the type of the edge, i.e., whether it is a memory, operand, or execution edge:

*Memory edges*

With \( N_x \) we denote a node in the Firm graph, and with \( N'_x \) the corresponding node in P2SSA, which may be \( \epsilon \) if we have not mapped \( N_x \). \( B_x \) is the basic block in the Firm graph that \( N_x \) belongs to, and \( r_x \) the fact whether or not this block is reachable as described above.

Let now \( N_1 \) be a node for which \( N'_1 \neq \epsilon \), and \( N_2 \) the defining node for a memory edge of \( N_1 \). There exists more than one such edge if and only if \( N_1 \) is a \( \phi \)-node, in which case we repeat the following steps for each of these edges.

If now \( N'_2 \equiv \epsilon \), there are two possibilities: (1) \( N_2 \) is not a \( \phi \)-node, or (2) it is a \( \phi \)-node. In the first case, \( r_2 \) must hold: either \( B_1 \equiv B_2 \), or \( B_2 \) is the only data flow predecessor of \( B_1 \). We then simply follow the memory edge recursively from \( B_2 \).

In the latter case, \( r_2 \equiv \text{false} \iff N'_2 \equiv \epsilon \), as \( \phi \)-nodes must always be mapped. If now \( r_2 \) does not hold, we simply omit the edge, as it is part of data flow confluence from exception handling.

*Operand edges*

Operand edges are mapped similar to memory edges, with one exception: If we encounter a Firm node \( N \) for which we could have (but haven’t) built a node \( N' \) in our target representation, we add an unknown operand instead of traversing the graph further up, as we would do with memory edges. This unknown operand acts as a placeholder and occurs when the input node is not covered by our target representation on account of settings, e.g., when mapping of primitive types is disabled. It is noteworthy that this never happens for edges carrying reference values, since all nodes processing such values must be mapped to P2SSA, as described in Section 3.2.2.

*Execution edges*

Execution edges appear between basic blocks and jump respectively branch nodes. For simplicity, we treat basic blocks like \( \phi \)-nodes, jumps and branches like operands, and use the same algorithm as for operand edges.

*Implementation details*

For \( \phi \) nodes it is possible to not add certain edges, i.e., reflexive edges and those that come from the same operand. While this gives a small advantage in performance and memory footprint, this option can be disabled when an analysis implements \( \chi \)-terms [Tra01], since then all \( \phi \)-nodes in a basic block should have the same number of operands.
3.3 Analysis

Our analysis framework is built on top of the program structure described in Figure 3.5. The program structure is obtained from the Firm program using the framework from the previous section.

We will first present a description of the interfaces and classes defined by our analysis framework. Then, we will take a short look on how handling of native methods is supported by our framework. Concrete implementations will be presented in the next chapter.

![Diagram](image)

Figure 3.5: SSA program structure

3.3.1 Framework Contracts

Figure 3.6 shows the most important interfaces and classes of our analysis framework.

The class `Analysis` with its method `run()` is the central entry point. It also provides the method `analyzeMethod()`, which uses the configuration options to properly set up the involved actors and starts the analysis of the given method.

An invocation of `run()` will use first analyze the system initialization methods obtained from the Firm program, followed by the program’s entry method.

An example for a method interpretation is given in Figure 3.3.1. In the following, we will further describe the involved classes.

This class also provides the convenience method `analyzeCInit()`, which takes a type as parameter and invokes that type’s class initialization method (clinit), unless analyzed beforehand. In addition, it foremost triggers interpretation of the class initialization method of the type’s superclass.

Each method is associated with a `MethodExecutionHandler`, whose `execute()` method is invoked for interpreting the semantics of the method. This abstract interpretation of the method is either described explicitly, e.g., in Java code, or triggers the interpretation of a P2SSA graph, which is the common case.

A `GraphStabilizer` - e.g., based on an underlying loop tree algorithm, cf. next chapter - takes care of the stabilization of an analysis when using an SSA graph. It calculates an evaluation order for the nodes, starting with the `Start`-node of a graph, and ending with its `End`-node. It then invokes the `accept()` methods for the nodes,
Figure 3.7: Analysis Run
<table>
<thead>
<tr>
<th>parameter</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>The P2SSA program which is to be analyzed.</td>
</tr>
<tr>
<td>nameSchema</td>
<td>The NameSchema class to use.</td>
</tr>
<tr>
<td>semantics</td>
<td>The Semantics class to use.</td>
</tr>
<tr>
<td>contextStrategy</td>
<td>The ContextStrategy to use.</td>
</tr>
<tr>
<td>fieldStrategy</td>
<td>The FieldStrategy to use (not shown in UML diagram), which is used for load and store operations.</td>
</tr>
<tr>
<td>graphStabilizer</td>
<td>The GraphStabilizer class to use for stabilizing SSA graphs. There may be more than one instance of this class during analysis.</td>
</tr>
</tbody>
</table>

Table 3.5: Overview of the Main Configuration Options

...each of which returns a boolean value, indicating whether or not the analysis value of the node has changed during its interpretation. This may be used to stabilize the dataflow analysis of loops.

The mapping from an allocation site to an abstract object is done by a name schema (interface NameSchema). Its method getAbstractObject() takes two parameters: the allocation site and the context. A simple class name schema will map an allocation site to its type, while an allocation site name schema will map an allocation site to itself. A more sophisticated name schema may create several abstract objects for an allocation site, depending on the context it is called in, cf. Chapter 2. For performance reasons, the name schema decides whether the underlying ReferenceStoringStrategy\(^1\), which is an abstraction usually for a bit vector implementation, shall use a prior-known set of abstract objects, or if the set of abstract objects may grow during analysis. The former is more performant, while the latter is more flexible. Should the name schema work with a fixed set of abstract objects, it is queried to compute the top-set of abstract objects from the program.

The distinction of contexts is handled by a ContextStrategy. The interpretation of a Call-node has to invoke its getContours() method, which returns one or more ContextQueryResults. The getContours() method takes the called method, parameters, memory state, calling node, and the context of the calling node as parameters. This allows for any well-known context strategy, e.g., object sensitivity or call site strings, and it also allows for easy experiments of combinations of them. The ContextStrategy does not take care of polymorphic call resolution, this is a task of the interpretation of call nodes.

A ContextQueryResult is a data structure, containing either the Contour under which a method is to be invoked, or, if the method has been analyzed with a matching contour before, the result of that analysis. A context insensitive strategy would, as an extreme, return just one ContextQueryResult for each query, while, as another extreme, a very fine-grained (and, therefore, expensive) strategy could return a ContextQueryResult for each combination of the passed parameters, cf. Chapter 2.

A Contour is used for the mapping of ports to values. Its method getValue() should be called by Semantics implementations only, because such an implementa-

\(^1\)This class is not shown in the diagram.
3.3.2 Native Methods
There are two ways to handle a native method in our framework: By implementing a native method handler - also called a method approximation -, and by replacing invocations of this method with a special node (cf. Section 3.2). While the former uses the regular context mechanism and is always possible (it works just as a graph method handler, except that semantics must be programmed "by hand"), there are some issues to look at with the latter. First, it is important to realize that using a special node instead of a regular call node, we inline the method in question. This can only be done under certain constraints, for example when the method is never called polymorphically (i.e., the method in question is final or static), and if it does not call itself recursively.

We provide both possibilities in our framework as the precision of our analysis can be improved substantially in certain cases: Consider the static method java.lang.System.arraycopy() - if we implemented it using the native method handler mechanism, we would mix up the contents of arrays used as input parameters to this method, unless using a very fine grained context strategy.

3.4 Conclusion
In this chapter, we have presented two frameworks: A framework that maps Firm to our coarse grained P2SSA representation. While - naturally - being rather stiff in the mapping of the program structure, it is very flexible when it comes to map SSA graphs: It optionally allows to exclude features such as exception handling or operations processing primitive types from the mapping process, while at the other hand being able to retain control flow related features such as the basic block structure.

We further provide a framework for running analyses on P2SSA. It allows to flexibly exchange main elements like context strategies, name schemata, node interpretations, and more. Native methods are supported in two ways: by implementing method approximations, and by replacing invocations of such.

In the next chapter, we will use these two frameworks to implement several Pointsto analyses.
4 Implementation

In the previous chapter, we have presented two frameworks: One for constructing our P2SSA representation, and one for running analyses on it. In this chapter, we present an exhaustive description of our implementation of these frameworks. In Section 4.1 we will outline the steps involved in leading from a program available in source code to the actual analysis result. We also describe the auxiliary steps that are not part of the core of our frameworks in more detail. Section 4.4 forms the main part of this chapter. There, we present the instantiation of our analysis framework. Afterwards, we present some optimizations performed (Section 4.5), and then summarize the issues concerning the soundness of our analysis (Section 4.6). We then give a note on the contributions we made for third party libraries (Section 4.7), and conclude this chapter with an overview of the limitations of our approach in Section 4.8.

4.1 Run in Detail

A rough classification of a complete run yields three parts: Building Firm (Section 4.1.1), mapping Firm to P2SSA (Section 4.1.2), and the actual analysis (Section 4.1.3).

4.1.1 Firm Build

Our implementation takes a program provided in Java source code as an initial set of classes and methods to consider. This program is translated to Firm using the Recoder [Rec] based Firm frontend Rec2Firm, which was developed within the Cate-project [GL06]. Since almost every Java program will reference third party libraries - at least the Java runtime library - for which source code is not available, we take the set of referenced bytecode methods and run a rapid type analysis [Bac97] on them in order to identify the set of methods and classes we have to build additional Firm code for. We build Firm code for these classes and methods using the BCEL [BCE] based Bc2Firm [Dre02] frontend, which is also being developed at the University of Karlsruhe. Finally, we serialize the Firm program to hard disk.

We now take a closer look at each of these steps.

Step 1: Rec2Firm

Rec2Firm is based on Recoder, a Java framework intended for tools aiming at analyzing and transforming Java source code. Rec2Firm uses Recoder to parse the Java source code and perform necessary analysis, e.g., establish call relations, field references, etc. It is noteworthy that Recoder focuses on Java source code and provides only rudimental support for analyzing bytecode: It stops parsing Java class-files at declaration level, i.e., no actual bytecode instruction is analyzed. Rec2Firm was initially implemented as a stand-alone tool and therefore used to process bytecode methods it references like native methods. Since this behavior does not suite us, we have modified Rec2Firm such that it builds Firm programs that allow for later adding method graphs for those methods.

Rec2Firm is invoked with a list of Java source files to build a Firm program for. The set of referenced bytecode methods for which, as mentioned, no Firm graphs can be built for, is afterwards retrieved from Recoder and passed to the next step, Rapid Type Analysis, which is described in the following.
Step 2: Rapid Type Analysis
Rapid type analysis - often referred to as RTA - is a static, flow- and context-insensitive program analysis for object oriented programs. Its algorithm takes a call graph generated by class hierarchy analysis [DGC95] as input. RTA then traverses the call graph beginning at the program’s entry method. At first, polymorphic calls are considered to be monomorphic. When the program traversal encounters the instantiation of a type, that type’s polymorphic methods that were left out before are also traversed, yielding an iteratively growing set of reachable methods. The algorithm is very fast, while still allowing us to greatly reduce the number of methods we need to build Firm code for, resulting in a very noticeable performance gain in terms of program size and running time.

For our purposes, we start from a set of program entry methods, namely the bytecode methods obtained from the previous step, plus the system initialization method System.initializeSystemClass. We also ensure that callback methods from doPrivileged calls, cf. Section 4.4.5, are being followed.

We run the rapid type analysis after running Rec2Firm, since usually most gain is achieved by the reduction of library functions [BS96]. It is of course possible to also implement a rapid type analysis on Recoder, but we have not done so in our implementation.

The results obtained from RTA are used by the Bc2Firm frontend, which is described in the next section.

Step 3: Bc2Firm
Bc2Firm is a frontend for building Firm from Java bytecode. As Rec2Firm, it was initially implemented as a stand-alone tool. Its algorithm takes an initial class file and builds Firm code for this class file. If it encounters a class reference - no matter if, e.g., through a method invocation or a field reference - it will add that class to its worklist and eventually also build Firm for that class. However, it will build graphs for all methods contained in this class. We therefore have adapted Bc2Firm so that it uses the results of our Rapid Type Analysis obtained in the previous step.

We invoke Bc2Firm with the class files that contain the bytecode methods referenced by our first step, Rec2Firm.

After this step, the construction of the Firm program is completed.

Step 4: Serialization
We use the Java serialization mechanism, which is described in [Mica], for simplicity. It is based on reflection and therefore not very efficient, but adequately fast for our needs. However, due to the possible size of Firm graphs, and since the underlying algorithm works recursively, we have encountered problems with stack overflows. This requires us to manually specify a bigger thread stack size when invoking the Java Virtual Machine - using the parameter -Xss - which does not seem to have any effect on the Java 5 Runtime Environment on Windows systems. However, Linux and Sun systems, as well as Java 6 (Beta) runtime environments on any tested platform, work as expected.

4.1.2 Building P2SSA
After deserialization of a Firm program from harddisk, we map it to P2SSA using the framework described in Section 3.2.
class A {
    int x = 3;
    static Object a = new A();
    static foo() {
        bar(((A)a).x)
    }
    static void bar(int x) {}
}

Figure 4.1: Dangling Cast

As an additional step after constructing each graph, dead nodes and their edges are removed from the graph. Dead nodes are those that do not have a path to the end node of our P2SSA representation. Figure 4.1 shows an example of how this condition may arise: The left graph shows a P2SSA graph which includes handling of primitive types. For the right graph, operations concerning types were not mapped, namely the Load-node of field \( x \). However, because the Cast-node operates on reference values, it was mapped; the only operation relying on this node, though, was the dropped Load-node.

Implementation notes
Our implementation of P2SSA graphs is based on the grail graph library [Pan05]. We have chosen grail as it provides an implementation of the loop tree algorithm (cf. Section 4.4.1) and visualization support. Besides that, our implementation is independent of any particular features. In fact, an underlying graph library is not necessary at all, as advanced features like graph manipulation are required only to a very small extend. Not using any graph library will come with lower memory consumption, but no disadvantage except some more implementation effort.

4.1.3 Running Analyses
An analysis is performed by instantiation of the Analysis-class described in Chapter 3. The main configuration options are discussed in upcoming sections, while an overview of these and further configuration options is given in Appendix C.

4.2 Interpretation of Values
Our implementation processes boolean lattices and reference lattices. The operation \( \sqcup \) on reference lattice elements (cf. Chapter 2) works analogously to the set operation \( \cup \), for boolean values the operation is defined in the following table:
### 4.3 Memory Modeling

We model memory in terms of size of the heap. Each time the memory is changed by a store operation, the heap size increases. Memory values are treated like any other lattice values. It ensures that upon a call, methods which have been seen with the same arguments before, will be processed again if the heap has changed. This is necessary as these methods may reference fields whose values have changed. However, the memory changes that occurred since the last invocation of a method in question may be unrelated to that method, causing the repeated analysis of the method to be in vain. Possible improvements are presented in Section 4.5.1 and Section 6.3.3.

### 4.4 Framework Instantiation

In this section, we will present the implementations we provide for the variation points of our analysis framework presented in the previous chapter. Most of them are straightforward, so the main focus will lie on the implementations of the Semantics interface. Figure 4.2 gives an overview of the analysis framework including the implementations we provide.

#### 4.4.1 Graph Stabilizer

We have implemented a graph stabilizer that uses the loop tree implementation of the grail library. Loop trees analysis was introduced by [Tra01] and is a refinement of the commonly known interval analysis. Instead of focusing solely on identifying loops on basic block level as interval analysis does, loop trees also take data flow dependencies into consideration. This promises a faster stabilization of an analysis, but constructing loop trees comes at a worst-case cost of $O(n^3)$, where $n$ is the number of nodes in a graph. Since the results of this thesis do not vitally depend on which stabilization strategy is used, we omit a further discussion of loop trees here.

#### 4.4.2 Context Strategies

As context strategies, we implemented a simple context insensitive strategy, and 1-object-sensitivity, as described in Chapter 2. A query to a context strategy is of the form $q = (\text{meth, this, arg}_1, \ldots, \text{arg}_n, \text{mem}, S_{\text{call}}, \delta_{\text{call}})$, where $\text{meth}$ is the method in question and $(\text{this, arg}_1, \ldots, \text{arg}_n, \text{mem})$ the method arguments to the call. $S_{\text{call}}$ is the syntactical position of the querying call, and $\delta_{\text{call}}$ the enclosing context under which the call itself is being analyzed. For each query, a context strategy returns one or more result items $q_{\text{res}}$, each denoting either a contour $c_{\text{meth}} = (\text{this, arg}_1, \ldots, \text{arg}_n, \text{mem})$ which needs to be analyzed, or a result pair $r_q = (r_{\text{val}}, r_{\text{mem}})$ obtained from a previous analysis under such a query $q$.

**ConIns**

Our context insensitive strategy retains one pair $(c_{\text{meth}}, r_{\text{meth}})$ for each method. $c_{\text{meth}}$ denotes the previously seen arguments, and $r_{\text{meth}}$ the analysis result from an analysis with such arguments. The following algorithm describes the context strategy’s behavior upon a query:

<table>
<thead>
<tr>
<th>$\bot$</th>
<th>false</th>
<th>true</th>
<th>$\top$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>false</td>
<td>true</td>
<td>$\top$</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>$\top$</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>$\top$</td>
<td>true</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
</tbody>
</table>
Figure 4.2: Analysis framework instantiation
function GETCONTOURS(meth, this, arg\_1, \ldots, arg\_n, mem, S\_call, δ\_call)
    if this ⊑ \texttt{this}\_meth ∧ arg\_1 ⊑ \texttt{arg}\_meth\_1 ∧ \ldots ∧ arg\_n ⊑ \texttt{arg}\_meth\_n ∧ mem ⊑ \texttt{mem}\_meth
        then
            return (r\_meth\_val, r\_meth\_mem)
        else
            \texttt{this}\_meth = \texttt{this}\_meth ∪ this
            \texttt{mem}\_meth = \text{max}(\texttt{mem}\_meth, mem)
            \textbf{for} i = 1, \ldots, n \textbf{do}
                arg\_meth\_i = arg\_meth\_i ∪ arg\_i
            \textbf{end for}
            \textbf{end for}
            q\_res = (\texttt{this}\_meth, arg\_meth\_1, \ldots, arg\_meth\_n, mem)
            return q\_res
    \textbf{end if}
\textbf{end function}

ObjSense
The 1-object-sensitive strategy distinguishes contexts for each \texttt{o} ∈ \texttt{this}, thus maintaining a set of (contour,result)-pairs, but besides that works just like the context insensitive strategy.

Note that neither of the two implementations we provide makes use of the \texttt{S}\_call and \texttt{δ}\_call arguments, which, e.g., can be used to implement a call site string strategy, cf. Chapter 2.

Contour Implementation
Our contours are implemented as mappings \texttt{Port} \mapsto \texttt{Value}, where the values can be of type boolean, reference, or memory. This allows for a method to be analyzed (indirectly) recursively under different contexts, without having to clone its entire graph or taking any other preparing actions. For performance reasons, ports have unique ids that form a continuous interval for each method graph, allowing the mapping to be implemented as an array of values. Basically the implementation of the value lookup for a port is implemented as values\{port.getId() − offset\}. The offset is necessary as the numbering of ports does not begin with zero for each method.

Each context strategy is also responsible for initializing the contours it returns with the arguments and memory state provided by the query.

4.4.3 Name Schemata
We implemented two name schemata: A class name schema, and an allocation site name schema. Both work with a fixed set of abstract objects which they, for performance reasons, compute prior to the analysis. The former uses the set of types in the SSA program, the latter determines its set of abstract objects by traversing all SSA graphs and collecting Alloc nodes. The allocation site name schema also optionally uses a class name schema for a configurable set of types.

4.4.4 Semantics
We provide two implementations of the Semantics interface: One that implements a baseline points-to analysis which can also handle conditional execution, and an extension of latter that makes further statements about dataflow values. We will
go into details of the idea behind the latter after we have discussed our baseline analysis.

Both implementations interpret a given program by simulated execution [Lun04]. The analysis simulates the actual execution of a program, where the processing of a method is interrupted when a call occurs, and resumed when that call returns. This gives our approach a certain amount of inter-procedural flow-sensitivity.

CESemantics
Our baseline implementation, CESemantics, supports all nodes shown in Tables 3.1 and 3.2. Before each node is analyzed, it is checked if the basic block that the node belongs to has at least one predecessor node that gives the basic block right of execution. This can either be a start, jump, or branch node.

Call A node \( \text{Call}_{\text{meth}, p}(\text{this}, \text{arg}_1, \ldots, \text{arg}_n, \text{mem}, S_{\text{call}}, \delta_{\text{call}}) : (r_{\text{val}}, r_{\text{mem}}) \) describes an invocation of the method \( \text{meth} \), where \( p \) is the predication of whether the call is polymorphic, \( \text{this} \) is the address parameter, \( \text{arg}_i \) the \( i \)-th argument to the call, and \( \text{mem} \) the state of memory. \( S_{\text{call}} \) is the node’s syntactical position, and \( \delta_{\text{call}} \) the enclosing context. \( r_{\text{val}} \) and \( r_{\text{mem}} \) denote the result value and memory state after invocation, respectively. \( r_{\text{val}} \) is neglected iff the return type of \( \text{meth} \) is considered irrelevant as described in Section 3.2.1. If \( \text{meth} \) is a static method, then \( \text{this} = \perp \) and \( p = false \). If otherwise \( p \) holds, then the call is transformed into multiple calls

\[ \text{Call}_{\text{meth}, i}(false, \text{this}_i, \text{arg}_1, \ldots, \text{arg}_n, \text{mem}, S_{\text{call}}, \delta_{\text{call}}) : (r_{\text{val}i}, r_{\text{mem}i}) \]

where the \( \text{meth}_i \) are the possible target methods, i.e., \( \text{meth} \) itself and the methods overwriting \( \text{meth} \), and their corresponding \( \text{this}_i \subseteq \text{this} \), with \( \text{this}_i \cap \text{this}_j \neq \emptyset \leftrightarrow i = j \).

For the tuple \((\text{this}, \text{arg}_1, \ldots, \text{arg}_n, \text{mem}_{\text{in}}, S_{\text{call}}, \delta_{\text{call}})\) the context strategy is queried for a set of context query results \( q_0, \ldots, q_m \). Each \( q_j \) is then processed separately: \( q_j \) may either denote a context \( \delta_j \) under which \( \text{meth} \) is to be analyzed, or the result \( r_j \) obtained by a previous analysis of \( \text{meth} \) under context \( \delta_j \). In the former case, \( \text{meth} \) is analyzed under context \( \delta_j \), also yielding a result \( r_j \). For each \( r_j \), \( r_{\text{val}} \) and \( r_{\text{mem}} \) are updated: With \( r_i = (r_{\text{val}i}, r_{\text{mem}i}) \), we get \( r_{\text{mem}} = \max(r_{\text{mem}}, r_{\text{mem}i}) \), and \( r_{\text{val}} = r_{\text{val}} \sqcup r_{\text{val}i} \).

Our implementation does not explicitly perform the transformation of polymorphic calls to monomorphic ones syntactically on the graph, but simulates the behavior during interpretation. It can also happen that \( \cup_{\text{this}} \subset \text{this} \), i.e., for some abstract objects, there may not be a matching method found during analysis. This occurs due to imprecision of the analysis, e.g., elements contained in arrays may be mixed up. Such an abstract object is neglected, as it is certain that no runtime object it abstracts from would be receiver of the call: The statically declared type of the receiving object must be of type, or subtype, of the method’s owner. If, thus, the runtime type of the object is not of type, or subtype, of the statically declared type, then, for Java, a \textit{ClassCastException} must have occurred prior to the call.

We similarly filter the arguments \( \text{arg}_1 \ldots \text{arg}_n \) to match the abstract objects’ types to the declared type of the method signature.
Note that if the call targets a static method, then \texttt{clinit()} from the method’s owner class is analyzed prior to any other action by calling the analysis framework’s \texttt{Analysis.analyzeCLInit()} method.

**Store** A node \texttt{Store\_field(addr, val, mem)}: \( r_{\text{mem}} \) stores the lattice value \( \text{val} \) into \texttt{field} of each object in \( \text{addr} \): \( \forall o \in \text{addr}: [o, \text{field}] = [o, \text{field}] \uplus \text{val} \).

**Load** A node \texttt{Load\_field(addr, mem)}: \((r_{\text{val}}, \text{mem})\) reads the lattice values stored in \texttt{field} associated with each abstract object in \( \text{addr} \):
\[
r_{\text{val}} = \sqcup_{o \in \text{addr}} [o, \text{field}].
\]
The memory value \( \text{mem} \) is just passed through, as it is not being altered.

**MemPhi** With our interpretation of memory as integer values, a node
\[
\text{MemPhi}(\text{mem}_1, \ldots, \text{mem}_n) : r_{\text{mem}} \text{ computes } r_{\text{mem}} = \max(\text{mem}_1, \ldots, \text{mem}_n).
\]

**ValuePhi** \texttt{ValuePhi(val}_1, \ldots, \texttt{val}_n) : \( r_{\text{val}} \) computes the operation \( r_{\text{val}} = \sqcup_{\text{val}_i} \), which computes as described in Section 4.2.

**Alloc** A node \texttt{Alloc\_type,S(mem)} : \((r_{\text{val}}, \text{mem})\) creates a new abstract object of type \texttt{type} by querying the name schema:
\[
r_{\text{val}} = \text{nameSchema.getAbstractObject}(S), \text{ where } S \text{ is the syntactical position of the node.}
\]

**Return** A node \texttt{Return(mem, val)} updates \( r_{\text{mem}} \) and, if the return type of the method is considered as relevant, \( r_{\text{val}} \).

**Start** A node \texttt{Start()} : \((\text{arg}_1, \text{arg}_n, X_{\text{true}}, \text{mem})\) is the head of each P2SSA graph.
It provides the arguments and memory state passed to the method, and also serves as a jump-node for initialization purposes. Arguments and memory state are initialized by the context strategies as stated above.

**End** An \texttt{End}-node marks the sink of a graph. No action is taken by our implementation upon analysis of this kind of node.

**Const** A node \texttt{Const\_val()} : \( r_{\text{val}} \) represents either a string constant, \texttt{null}, or a concrete boolean constant.

**Cast** A node \texttt{Cast\_type(op)} : \( r_{\text{ref}} \) filters the argument \( \text{op} \) such that
\[
r_{\text{ref}} = \{x | x \in \text{op} \land (x \text{ instanceof type} \lor x = \text{null})\}.
\]
In other words, it performs a type cast as described by the Java Language Specification [GJSB05].

Note that our implementation uses pre-computed filters for the underlying bitvector for performance reasons, practically mapping a type cast to a number of bitwise and operations.

**InstanceOf** A node \texttt{InstanceOf\_type(op)} : \( r_{\text{boolean}} \) checks if none, at least one, or all of the elements of \( \text{op} \) are of type \texttt{type} and sets \( r_{\text{boolean}} \) accordingly: With \( \text{op}' = \{x | x \in \text{op} \land x \text{ instanceof type}\} \), \( |\text{op}'| = 0 \) yields \( r_{\text{boolean}} = \text{false} \), \( |\text{op}'| = |\text{op}| \) yields \( r_{\text{boolean}} = \text{true} \), and \( |\text{op}'| \neq |\text{op}| \) yields \( r_{\text{boolean}} = \top \).

Our Implementation uses pre-computed filters similar to Cast-nodes.

26
**BasicBlock** A basic block $\text{BasicBlock}(\text{in}_1, \ldots, \text{in}_n) : r_x$ serves as phi-node for control flow. If any of the $n$ control flow predecessors give this basic block the right of execution, nodes belonging to this basic block will be interpreted. Formally said: $r_x = \sqcup \text{in}_i$, where the $\text{in}_i$ are boolean values.

**Branch** A node $\text{Branch}(\text{op}) : (r_{\text{false}}, r_{\text{true}})$ splits up the abstract boolean value $\text{op}$ into two concrete boolean values $r_{\text{true}}$ and $r_{\text{false}}$, given a right of execution to none, one, or both control flow successors of the node’s basic block as described in the following table:

<table>
<thead>
<tr>
<th>$\text{op}$</th>
<th>$r_{\text{false}}$</th>
<th>$r_{\text{true}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>$\top$</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**Jmp** A node $\text{Jmp}() : X_{\text{true}}$ gives the block targeted the jump node a right of execution. The only precondition is, as with any other node, that the Jump-node is to be analyzed itself.

**Or** A logical $\text{Or}(\text{op}_1, \text{op}_2) : r$ node with the abstract boolean operands $\text{op}_1$ and $\text{op}_2$ results in an abstract boolean result value $r$ as described in the following table:

<table>
<thead>
<tr>
<th>$\lor$</th>
<th>$\bot$</th>
<th>$\text{false}$</th>
<th>$\text{true}$</th>
<th>$\top$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>false</td>
<td>$\bot$</td>
<td>$\text{false}$</td>
<td>$\text{true}$</td>
<td>$\top$</td>
</tr>
<tr>
<td>true</td>
<td>$\bot$</td>
<td>$\text{true}$</td>
<td>$\text{true}$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
</tbody>
</table>

**And** A logical $\text{And}(\text{op}_1, \text{op}_2) : r$ node with the abstract boolean operands $\text{op}_1$ and $\text{op}_2$ results in an abstract boolean result value $r$ as described in the following table:

<table>
<thead>
<tr>
<th>$\land$</th>
<th>$\bot$</th>
<th>$\text{false}$</th>
<th>$\text{true}$</th>
<th>$\top$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>false</td>
<td>$\bot$</td>
<td>$\text{false}$</td>
<td>$\text{false}$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>true</td>
<td>$\bot$</td>
<td>$\text{false}$</td>
<td>$\text{true}$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$\bot$</td>
<td>$\top$</td>
<td>$\top$</td>
<td>$\top$</td>
</tr>
</tbody>
</table>

**EQ** An equality test node $\text{EQ}(\text{op}_1, \text{op}_2) : r_{\text{boolean}}$ checks for equality of $\text{op}_1$ and $\text{op}_2$, which may be either boolean or reference values. In the former case, the result is determined with the table below. In the latter case, $s = \text{op}_1 \cap \text{op}_2$ is computed, and $r_{\text{boolean}} = \text{false}$ if $|s| = 0$. Otherwise, $r_{\text{boolean}} = \top$, as we cannot make a statement about exact equality, since elements in $\text{op}_1$ and $\text{op}_2$ are abstract objects, which may represent more than one concrete object. However, an optimization is presented in Section 4.5.2.
NEQ An inequality test node \( NEQ(op_1, op_2) : r \) is a syntactical shortcut for \( Not(EQ(op_1, op_2)) \).

XOr A logical \( XOr(op_1, op_2) : r \) node with the abstract boolean operands \( op_1 \) and \( op_2 \) results in an abstract boolean result value \( r \) as described in the following table:

<table>
<thead>
<tr>
<th>xor</th>
<th>( \bot )</th>
<th>false</th>
<th>true</th>
<th>( \top )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>( \top )</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>( \top )</td>
<td></td>
</tr>
<tr>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
<td>( \top )</td>
</tr>
</tbody>
</table>

Not A logical \( Not(op) : r \) node with the abstract boolean operand \( op \) results in an abstract boolean result value \( r = \neg op \) as described in the following table:

<table>
<thead>
<tr>
<th>op</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>( \top )</td>
<td>( \top )</td>
</tr>
</tbody>
</table>

ValueAnnotationSemantics During the work on this thesis, we realized that besides using control-flow related operations to identify dead code, there is further potential for using these operations to improve the analysis result.

Figure 4.3(a) shows a common SSA-graph, where \( P \) and \( Q \) are two arbitrarily chosen operations. It is trivial that, if the values produced by \( P \) and \( Q \) have equal values in a real program run, \( R \) will use this very same value. During a static analysis, though, the values of \( P \) and \( Q \) will represent elements of a value lattice. Therefore, \( R \) gets executed if \( P \) and \( Q \) have at least one common element. However, our previously presented analysis will use \( Q \)’s output for \( R \)’s input, although, obviously, \( R_{in} = P_{out} \cap Q_{out} \), would be still conservative, but stronger information.

A solution to gain this information is to introduce a special node, \( \sqcap \), into the SSA-graph, as shown in Figure 4.3(b): A basic block that contains three nodes, two \( \sqcap \) and one \( Jmp \) node, is interposed in between the block containing the comparison and the regular control flow successor. Note that the \( \sqcap \)-nodes cannot be contained in the control flow successor itself, as that block may have more than one predecessor block. This improvement is analogously possible for other node types, e.g., InstanceOf nodes.

However, this optimization only works with local variables: Figure 4.4 shows a short listing, and its corresponding bytecode. The field \( a \) is referenced twice, resulting in two load operations; introducing a \( \sqcap \)-node would not yield any benefit in such a case. The result of this optimization therefore depends on the programs
that are to be analyzed - or rather, the programming style they are implemented in. The example could be adapted in two ways: Either by introducing a local copy of the field \( a \) into the method body, or by introducing optimizations: A compiler could recognize that the load operations of the field \( a \) will always yield the same result, therefore one of the memory accesses could be eliminated. In this case, this is obvious, as the two load-operations are directly memory dependend. However, such optimizations are neither trivial to implement in general nor always allowed by the Java Language Specification, as such a transformation may change the runtime behavior of multithreaded programs.

```java
A a;
void foo(A b) {
    if (a == b) {
        a.foo();
    }
}
```

Figure 4.4: Read-Read Conflict

We feel that nodes allowing this improvement should be introduced on the basic block graph structure that is not yet in SSA, as there it is very easy to add a redefinition for a variable. This means a proper implementation would include changes to two libraries - BCEL and LibFirm - as well as the two frontends Bc2Firm and Rec2Firm. Since this is beyond scope of this thesis project, we have decided to simulate the approach experimentally in our ValueAnnotationSemantics implementation. The basic idea is to annotate each basic block with information about what stronger statements it can make about the values associated with certain ports, i.e., it maintains a set of pairs \((port, value)\). Then, a basic block serves as a \( \phi \)-node for
these annotations, computing for all ports $p_i$ that have a redefinition in all reachable predecessor blocks $b_j, \sqcap v_j$, where $v_j$ is the redefinition of a ports value in $b_j$.

We will not give a more formal description for the extended interpretation, as our implementation serves as a proof of concept only, and, in our opinion, a better way to implement would be on a different level, namely SSA construction.

### 4.4.5 Call Replacer

The Firm-frontends use special methods to allocate and manipulate arrays, as the semantics of the regular Firm nodes for array manipulation does not provide boundary checks. Since these methods are static, using the SSA graphs the Firm frontends build for them will result in mixing up the contents of all arrays we use in our analysis, unless using an expensive context strategy. As an example for our SSA-build plugin mechanism, we have implemented replacements for calls to these array get-and set-methods. We further replace calls to the native `System.arraycopy()` and `AccessController.doPrivileged()` methods. As another optimization, we skip calls to the `hashCode()` method for all classes. While this is not a conservative replacement, it usually does not have any side effects that are of interest for software analysis tools. It improves, however, performance, since the implementation of the method for example in the collections framework is quite costly. Table 4.4.5 summarizes the additional nodes introduced and explains their semantics.

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SkipCall</td>
<td>Skips a call that is present in the intermediate Firm representation. The node takes an arbitrary number of parameters. The replaced method’s return type must not be relevant for the framework instantiation.</td>
</tr>
<tr>
<td>InitArray</td>
<td>Initializes an array previously created by an <code>Alloc</code> node with <code>null</code>.</td>
</tr>
<tr>
<td>ArrayGet</td>
<td>Reads a value from array. Since we do not take the index of the array access into consideration, this works similarly to a <code>Load</code> operation. For this, each array type is associated with a fake field. <code>ArrayGet</code> then uses the same read mechanism as the interpretation of the <code>Load</code> node does.</td>
</tr>
<tr>
<td>ArraySet</td>
<td>Writes a value into an array. Analogously to <code>ArrayGet</code>, this works similarly to a <code>Store</code> operation.</td>
</tr>
<tr>
<td>ArrayCopy</td>
<td>Copies the values of all arrays given as first operand to all arrays given as third operand. Calls to the <code>System.arraycopy</code> method are replaced by this node.</td>
</tr>
<tr>
<td>DoPrivileged</td>
<td>Call the <code>run</code> method of the passed <code>PrivilegedAction</code> or <code>PrivilegedExceptionAction</code> argument. While the plugin mechanism could as well replace the original call with a call to the argument’s <code>run</code> method, we keep the information of the callback, which can be used, e.g., for call graph construction.</td>
</tr>
</tbody>
</table>

Table 4.1: Nodes Introduced by Plugin System
```java
class A implements Cloneable {
    void foo () {
        A a1 = new A ();
        A a2 = (A) a1.clone ();
    }
}
```

Figure 4.5: Return Types of Native Methods

### 4.4.6 Native Methods

Besides replacing calls to native methods as described in the previous section, we have a native method handler for the method `java.lang.System.initProperties()`.

If we encounter a native method for which we do not have implemented support explicitly, we have three different ways of handling them: (1) In case that the return type of the method is irrelevant for our analysis, we ignore the call. (2) If, otherwise, we use a class name schema for the return type of the method (cf. Section 4.4.3), we return the corresponding abstract object for that type. (3) In all other cases we return `null`. All these approaches are obviously not conservative, as none of them takes possible side effects into consideration.

We have also experimented with another approach: For each type in the program that is the return type of at least one native method, we introduce an `UnknownObject`. Each native method that is not handled in any other way returns such an object, depending on its return type. However, two problems arise: First, the declared type may not be the actual runtime type. For an example, see Figure 4.4.6: The invocation of `clone()` in line 4 results in an object of type `A` for sure, so the cast is sound and safe; however, the declared return type of the method is `Object`. Therefore, using this approach, we would have to assume that an object returned by a native method could be of any of its subtypes. For the `clone()` method, which has a declared return type of type `Object`, that would result in an `UnknownObject` that could be casted to any type in the program.

A second problem is that we may give multiple names to a single concrete object, violating the general contract for name schemata. An example for this is the `String.intern(String)` method, which may actually return the passed argument.

For these two reasons have we decided not to use this approach.

### 4.5 Optimizations

In this section we describe two optimizations that we perform on a conceptual level: no-load methods and handling of `null`-references.

#### 4.5.1 NoLoad Methods

In Section 4.3, we described the problem that change of memory size forces re-evaluation of any method invocation, even if the targeted method has nothing to do with the modified memory locations in question. In order to reduce the effect on performance, we identify methods that do not access memory through a load-operation, either directly or transitively through other invoked methods. If such a method is targeted by an invocation, and the previous arguments differ in memory size only, the context strategy can safely decide that the method invocation does not need to be evaluated again.
Instead of identifying such methods statically, we do this dynamically during our simulated execution: At first, no method is considered to access memory through a load operation. Only when the execution of a load node, or a call that targets a method that is a load method itself, occurs, will we mark that this method reads from memory.

4.5.2 Handling of null

The null object is handled just like any other abstract object, except that there is exactly one instance of it, independent of the name schema in use. This means that the interpretation of two sets of abstract objects can be seen as equal if both sets contain the null object only.

The java language specification requests that any reference field is implicitly initialized with null. We initialize those fields that are not initialized by the constructor invocation following each Alloc-node. Should a constructor - or any method called (in-)directly by that constructor - read from an uninitialized field, we initialize that field with null on demand. While this is not strictly a conservative approach - pathological examples that create imprecise results are imaginable - the impact should be quite low. Further, the behavior can be switched back to the conservative approach through a configuration option, cf. Appendix C.

Would we initialize all fields to null directly after object allocation, we would never get rid of it on account of the absence of strong updates.

4.6 Optimistic Approximations

While our approach is conservative in general, we have made a few optimistic approximations. For these either simple workarounds exist, or implementation effort must be made to resolve them. Table 4.6 gives an overview of these approximations and points out possible solutions. Besides exception handling and native methods, though, we feel that for our main application area - program comprehension tools - the benefits in terms of execution time of these optimistic approximations outbalance the negative impact on the analysis result.

4.7 Contributions to Third-Party Libraries

For our implementation, we used several third party open source software, some of which is still at an early stage of development. We were able to make contributions in terms of stability, usability, and functionality.

As mentioned before, we have adapted the Rec2Firm and Bc2Firm frontends, which were both stand-alone tools, to work together. We have also integrated a rapid type analysis into Bc2Firm, cf. Section 4.1. Further have we adapted Rec2Firm to use the latest release of Recoder, version 0.81, which is capable of analyzing Java 5 source code. We have also adjusted the Bc2Firm frontend to accommodate the changes made to the Java 5 bytecode, as described in [Micb].

For the Grail [Pan05] graph library, version 1.9 build 29-20060809, we could make a suggestion for lowering the memory footprint: Grail stores properties - basically a mapping of property descriptions to actual values - of nodes and edges in hash maps for fast access. However, if a node or an edge contains only few such properties, a simple list based data structure appears more memory efficient. We therefore proposed to switch from a list based data structure to a hash map once the number of properties for a node or edges increases. In applications using Grail, it appears
<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
<th>Proposed Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>clinit() analyzed only once</td>
<td>We analyze the class initializer only once for each class, on the first reference to that class in our analysis. In a real application run, class initialization may occur in a different order, and class initializers may make assumptions about this order, e.g., a class initializer relies on side effects triggered by a different class’s initialization. While we cannot think about a realistic example where this would be necessary, our approach must still be considered optimistic.</td>
<td>A straight-forward, but rather expensive solution is to analyze the class initializer upon every access that may trigger the class initialization.</td>
</tr>
<tr>
<td>Field initialization with null</td>
<td>As described in Section 4.5.2, we initialize instance fields only after the constructor call to a corresponding Alloc-node, and only if they are not being initialized otherwise.</td>
<td>A configuration option exists for our implementation to switch to a conservative behavior, see appendix C.</td>
</tr>
<tr>
<td>Native methods</td>
<td>We have not implemented method stubs for all native methods.</td>
<td>This issue is discussed in Section 6.3.1, future work.</td>
</tr>
<tr>
<td>Exception handling</td>
<td>Our implementation currently leaves out the analysis of any exception-handling related program code.</td>
<td>This issue is discussed in Section 4.8.1.</td>
</tr>
<tr>
<td>hashCode()</td>
<td>We ignore calls to hashCode() by replacing it with a Skip-Call node for performance reasons.</td>
<td>The corresponding CallReplacer can be disabled through configuration options.</td>
</tr>
</tbody>
</table>

Table 4.2: Optimistic Approximations
many nodes and edges, which contain only few properties, exist, so our suggestion led to noteworthy memory saving.

4.8 Limitations
Our implementation is impacted by some limitations, which are discussed in this section.

4.8.1 Exceptions
While our framework is capable of dealing with exception handling, we have not included it in our implementation. An approach to include exception handling would be to consider exceptions as any other data values, propagating the set of possible raised exceptions just like any other data values.

4.8.2 Threads
Our implementation does not support multi-threaded programs. A possible implementation is to analyze two (w.l.o.g.) concurrent threads $T_1$ and $T_2$ in a virtual loop: $T_1, T_2, T_1, T_2, \ldots$, each run starting at thread creation, until the analysis stabilizes. However, all benefits of interprocedural flow-sensitivity would be lost, and on first glance, efficiency may be impacted intolerably. If information from the synchronization mechanisms provided by Java can be used to obtain a more efficient algorithm is an open research question.

4.8.3 Reflection
To our knowledge, no general approach for handling reflection exists. Since most methods related to reflection rely on the contents of character strings, e.g., methods for reflectively loading classes or invoking methods, an analysis would have to inspect the contents of such character strings. In our opinion, such an analysis is impractical.

4.8.4 Boolean Operations
As already mentioned in Section 3.2.2, Java bytecode does not provide support for the primitive type Boolean, and instead maps operations on them to integer operations. Since this means that no logical operations - e.g., and, or, not - are being constructed in Firm graphs derived from Java bytecode, our support for Boolean operations is more limited when using the Bc2Firm frontend than when using the Rec2Firm frontend.

4.9 Conclusion
In this chapter, we presented our implementation for the two frameworks introduced in the previous chapter. For our building framework, we presented two frontends that build Firm code: one that reads source code, and one that reads Java bytecode. While not originally implemented by us, we have made contributions that allow these two frontends to collaborate, and we have further reduced the number of classes and methods that Firm code needs to be built for by implementing a Rapid Type Analysis.

As instantiations of our analysis framework, we have implemented some well-known techniques for name schemata and context strategies. Our baseline interpretation for P2SSA graphs provides a novelty: It allows to extend well-known points-to analysis techniques with evaluating control flow, thus improving analysis precision by detecting dead code. Our second implementation is an extension of the former.
It adds the simulation of a novel idea for SSA construction which may be used to improve dataflow analysis in general.

We will compare different combinations of our implementation in the next chapter.
5 Evaluation

In this chapter, we will evaluate the different implementations for our analysis framework. We will first introduce the metrics we use to measure differences in analysis results. Then, we will run benchmark programs, and compare them in terms of computing time and analysis precision.

5.1 Metrics

In this section we present the high-level analyses, which we use as metrics for our evaluation.

5.1.1 Reachable Methods

We count the number of reachable methods, both total and application methods. Application methods are distinguished by a name prefix, and exclude third-party libraries like the Java runtime library.

We refer to this metric with reachable methods and reachable application methods.

5.1.2 Access Safety

With access safety we refer to member accesses, i.e., call, load and store operations, that occur in our intermediate representation, and for which we can say for sure that they will never raise a null-pointer exception. We especially expect our ValueAnnotationSemantics to improve this metric, as programming patterns like in Figure 5.1 are quite common. Note that a local dataflow analysis could also rule out a null pointer accesses in line 4, but would usually not be able to transfer intermediate analysis results interprocedurally to the call in line 9.

We count the number of member accesses that are contained in the set of application methods. An access is safe if its points-to set does not contain the null-object. This definition includes unreachable accesses, as they indicate dead code and such accesses are safe as well.

```java
1 class A {
2     foo(A a) {
3         if (a != null) {
4             a.foo();
5             bar(a);
6         }
7     }
8     bar(A a) {
9         a.foobar();
10     }
11 }
```

Figure 5.1: Null Pointer Check

5.1.3 Call Graph

As our most detailed metric, we construct two graphs: The Application Method Graph, and the Application Object Method Graph. The former consists of nodes \( m \) representing methods, and edges \( e \) representing call relations \( m_i \rightarrow m_j \). The latter
Table 5.1: Types for Which to Use a Class Name Schema

<table>
<thead>
<tr>
<th>java.lang.String</th>
<th>java.lang.StringBuffer</th>
<th>java.util.SimpleTimeZone</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.Locale</td>
<td>java.lang.Integer</td>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>java.lang.Double</td>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>java.lang.Character</td>
<td>java.lang.Short</td>
<td>java.lang.Class</td>
</tr>
<tr>
<td>java.lang.Thread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

consists of nodes \([o, m]\), which describe methods that are called on a specific abstract object, and edges \([o_i, m_p] \rightarrow [o_j, m_q]\) representing call relations between them.

Only call relations where, at least, either the caller or the receiver is an application method are added to the graph.

In our evaluation, we refer to the size of these graphs with Nodes, Edges, ONodes, and OEdges.

5.2 Setup

All tests were run on an Athlon64 3000+ machine with 2GB RAM, running Linux and a Java6(Beta) Runtime Environment. The java virtual machine is started with `-Xmx1400M -Xms1000M -Xss24M` as command line arguments. Each configuration was run four times, of which the first one is a warm-up run whose results are not counted for our measurements.

For all runs, we replace calls of array accesses, `System.arraycopy()`, and `hashCode()` with special nodes as described in Section 4.4.5. We have not included the `Security.doPrivileged()` call replacement because it affects performance in a disproportional manner, while initializing features like the security mechanism, which is unimportant for our purposes. The name schema is for all runs an allocation site name schema, with exception of the types listed in table 5.1. Each test program is run in all of the combinations shown in Table 5.2, resulting in six benchmark result sets. ValueAnnotationSemantics are not run in combination with ObjSense since its implementation is only prototypical. For example, our implementation stores per-method information instead of per-context information for simplicity.

Further, we have left out execution time benchmarks for ValueAnnotationSemantics for the same reason. Especially when using it with conditional execution disabled, such measurements would be hardly meaningful as - although not needed - the basic block structure, jump- and branch-nodes are still being built and evaluated.

A list containing the descriptions of the benchmark programs used is given in Table 5.3. All programs were compiled with the Java 2 Runtime Environment, version 1.4.2_08. Note that while the versions of most programs we use for our benchmark are of older age, this is for comparability to other publications only. Our implementation is capable of analyzing programs written for and compiled with recent Java versions.

<table>
<thead>
<tr>
<th>context sensitivity</th>
<th>semantics</th>
<th>cond. execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConIns</td>
<td>CESemantics</td>
<td>on and off</td>
</tr>
<tr>
<td>ObjSense</td>
<td>CESemantics</td>
<td>on and off</td>
</tr>
<tr>
<td>ConIns</td>
<td>ValueAnnotationSemantics</td>
<td>on and off</td>
</tr>
</tbody>
</table>

Table 5.2: Evaluation Setup - Semantics and Context Sensitivity
JavaCC 3.2
(org.javacc)

JavaCC - the Java Compiler Compiler - is a lexical analyzer generator and LL(k)-parser generator for use with Java applications. It reads a description language and generates Java code which will then read the specified language. It is available from https://javacc.dev.java.net/

Emma 2.0.5312
(com.vladium)

Emma is an open-source toolkit for measuring and reporting Java code coverage. It is available from http://emma.sourceforge.net/

JavaC 1.3.1
(com.sun.tools)

JavaC is the Sun Java compiler. It comes with the Java Development kit (JDK).

Obfuscator 0.73
(reco더)

Obfuscator is a source code obfuscating tool. It comes as an example program with the Recoder [Rec] framework.

Jython
(org.python)

Jython is an implementation of Python fully written in and for Java. It comes with the DaCapo Benchmark Suite, http://www.dacapobench.org.

Table 5.3: Benchmark Programs

5.3 Results

In this section, we compare the benchmarks results in terms of program representation size, performance, and analysis precision.

5.3.1 Program Representation Size

<table>
<thead>
<tr>
<th>program</th>
<th>CE</th>
<th># nodes</th>
<th># Firm nodes</th>
<th>Firm load</th>
<th>mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaCC</td>
<td>off</td>
<td>111819</td>
<td>696126</td>
<td>29.1 sec.</td>
<td>3.7 sec.</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>332579</td>
<td>876149</td>
<td>6.2 sec.</td>
<td></td>
</tr>
<tr>
<td>Emma</td>
<td>off</td>
<td>97337</td>
<td>541852</td>
<td>25.3 sec.</td>
<td>3.2 sec.</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>275909</td>
<td>694131</td>
<td>4.8 sec.</td>
<td></td>
</tr>
<tr>
<td>JavaC</td>
<td>off</td>
<td>102570</td>
<td>587612</td>
<td>26.5 sec.</td>
<td>3.3 sec.</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>294780</td>
<td>746227</td>
<td>5.0 sec.</td>
<td></td>
</tr>
<tr>
<td>Obfuscator</td>
<td>off</td>
<td>120319</td>
<td>651966</td>
<td>28.6 sec.</td>
<td>3.9 sec.</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>335085</td>
<td>835896</td>
<td>5.7 sec.</td>
<td></td>
</tr>
<tr>
<td>Jython</td>
<td>off</td>
<td>118306</td>
<td>671852</td>
<td>29.3 sec.</td>
<td>4.2 sec.</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>340872</td>
<td>857518</td>
<td>6.1 sec.</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Program Sizes

Table 5.4 shows the size of our SSA representation in terms of nodes, compared against the size of the corresponding Firm graphs, and also in comparison of enabled/disabled conditional execution. For the number of Firm nodes, we only count those nodes that are contained within blocks that we map to P2SSA, i.e., these numbers do not include nodes contained in exception handling related blocks, but otherwise, nodes manipulating primitive data values are counted.

Naturally, our SSA representation is much more compact than the Firm representation, considering that we do not map operations on primitive types, and, with conditional execution disabled, not even the basic block structure and jump nodes.
More interesting is the observation that the size of our representation grows approximately by factor three when enabling conditional execution. This is important since the used loop tree algorithm has a worst-case complexity of $O(n^3)$, so a slowdown in our benchmarks on that account is possible.

The table shows also the time needed to deserialize Firm from hardisk. This does not include the time required to build the Firm graphs, but only the time to deserialize Firm. Building Firm takes in between less than a minute (Emma), to more than ten minutes (Obfuscator). However, this is not in our hands, so we do not explicitly discuss these times. Additionally, the time required for mapping Firm to P2SSA representation is also listed. While the former has rather informal value - as said before, the serialization mechanism is far from efficient - the mapping process works in acceptable time.

### 5.3.2 Performance

Prefacing we have to make some remarks about Jython. The results for Jython with conditional execution enabled show an incredible precision gain in our metrics, e.g., a reduction of OEdges by 97.9%. The problem is that Jython makes heavy use of native methods which we currently do not support explicitly, e.g., for dynamic class loading. Jython also often checks the contents of variables for null references, and explicitly raises an exception in such cases. This then effectively ends the analysis of the current method, as no further basic block will be considered as reachable. Many times, the only abstract objects that could be propagated to these checks are those returned by native methods.

This can be seen as a false positive in terms of dead code elimination, which, of course, does not occur when conditional execution is disabled.

Therefore, the benchmark results for Jython are meaningless when conditional execution is enabled, with the exception that they tell us that thorough support for native methods is significant to our approach. Since we have no intention to only compare object sensitivity and context insensitivity with each other, we omit the results of Jython in the following.

Table 5.5 shows the total time required to run each benchmark, excluding the times required to build P2SSA. For the benchmarks with conditional execution enabled, we also list the factor compared to the benchmark to the conditional execution disabled. We further show the total time required to build all required loop trees, as well as their percentage of the total analysis time.

On first note, conditional execution slows down the analysis on average by approximately factor two. Only Emma shows a slight speedup.

The slowdown of JavaCC is above average on account for the loop tree algorithm. JavaCC contains methods that are remarkably big, these are generated methods – so the non-linearity of the loop tree algorithm affects JavaCC disproportionately. Besides that, the loop tree algorithm does not have an unreasonable impact on performance. It is quite possible that the benefits provided by faster analysis stabilization outweigh this.

One rather strange result is the increase in absolute time required to construct the loop trees when going from $\text{ConIns}$ to $\text{ObjSense}$ for some test cases. While the method graphs are independent of context sensitivity, and ObjSense should not reach a larger set of methods, the only explanation we have is that, due to higher memory consumption for ObjSense, the garbage collector interrupts the loop tree algorithm.
Table 5.5: Analysis Results - Performance

<table>
<thead>
<tr>
<th>program</th>
<th>configuration</th>
<th>total time</th>
<th>factor</th>
<th>loop tree</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaCC</td>
<td>ConIns, NoCE</td>
<td>25,3</td>
<td>6,2</td>
<td>24,5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>85,9</td>
<td>3,4</td>
<td>41,8</td>
<td>48,7</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>30,6</td>
<td>6,3</td>
<td>20,6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>107,2</td>
<td>3,5</td>
<td>41,7</td>
<td>38,9</td>
</tr>
<tr>
<td>Emma</td>
<td>ConIns, NoCE</td>
<td>394,3</td>
<td>8,4</td>
<td>2,1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>304,4</td>
<td>0,77</td>
<td>17,7</td>
<td>5,8</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>752,4</td>
<td>9,6</td>
<td>1,3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>656,2</td>
<td>0,87</td>
<td>18,7</td>
<td>2,8</td>
</tr>
<tr>
<td>JavaC</td>
<td>ConIns, NoCE</td>
<td>474,7</td>
<td>7,2</td>
<td>1,5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>783,9</td>
<td>1,65</td>
<td>15,8</td>
<td>2,0</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>1641</td>
<td>11,7</td>
<td>0,71</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>2607</td>
<td>1,59</td>
<td>21,7</td>
<td>0,83</td>
</tr>
<tr>
<td>Obfuscator</td>
<td>ConIns, NoCE</td>
<td>128,3</td>
<td>4,8</td>
<td>3,7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>246,4</td>
<td>1,92</td>
<td>14,5</td>
<td>5,9</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>331,3</td>
<td>6,4</td>
<td>1,9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>674,1</td>
<td>2,03</td>
<td>17,0</td>
<td>2,5</td>
</tr>
</tbody>
</table>

5.3.3 High Level Analyses

Table 5.6 shows the results obtained from the call graph construction.

JavaCC show nearly no precision gain from either object sensitivity or conditional execution. JavaC can hardly benefit from conditional execution and only little from ObjSense. On the other hand, for Emma, both object sensitivity and conditional execution provide substantial improvements in precision, while the latter brings a little more. Combined with each other, the result improve even more, which suggests that both approaches target different points of optimization potential.

Our ValueAnnotationSemantics without conditional execution shows little, but measurable benefits. However, we believe that once properly implemented, it also comes at nearly no cost.

Table 5.7 shows the metrics safe accesses and the two variations of reachable methods.

It comes at no surprise that ValueAnnotationSemantics has the best results in safe accesses, as discussed in Section 5.1.2. For JavaC and Obfuscator, enabling conditional execution reduces the number of safe accesses. Here, we assume that the analysis stabilizes slower with than without conditional execution enabled, and subsequently reducing the effect of the flow-sensitivity. Besides that, we see a noticeable drop in reachable methods for almost all test cases when conditional execution is enabled. While this suggests great potential in terms of dead-code analysis, the impact of side effects - as described for Jython above - needs to be eliminated by means which we described before.

5.4 Conclusion

In this chapter, we have compared different combinations of our implementations from the previous chapter. While we discovered that the results for conditional execution suffer from side-effects because of incomplete support for native methods, they still show promising precision gains. However, effort needs to be done to provide full support for native methods in our framework. The analysis itself is, on average,
<table>
<thead>
<tr>
<th>program</th>
<th>configuration</th>
<th>Nodes</th>
<th>Edges</th>
<th>ONodes</th>
<th>OEdges</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaCC</td>
<td>ConIns, NoCE</td>
<td>1023</td>
<td>3332</td>
<td>1851</td>
<td>7838</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>100%</td>
<td>99,9%</td>
<td>100%</td>
<td>99,9%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, NoCE</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>99,2%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, CE</td>
<td>100%</td>
<td>99,9%</td>
<td>100%</td>
<td>99,1%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., NoCE</td>
<td>100%</td>
<td>99,9%</td>
<td>100%</td>
<td>99,9%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., CE</td>
<td>100%</td>
<td>99,6%</td>
<td>99,8%</td>
<td>98,5%</td>
</tr>
<tr>
<td>Emma</td>
<td>ConIns, NoCE</td>
<td>1653</td>
<td>4576</td>
<td>3490</td>
<td>15376</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>93,1%</td>
<td>89,9%</td>
<td>87,9%</td>
<td>81,1%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, NoCE</td>
<td>99,5%</td>
<td>96,1%</td>
<td>96,3%</td>
<td>82,5%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, CE</td>
<td>92,4%</td>
<td>89,8%</td>
<td>86,2%</td>
<td>71,8%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., NoCE</td>
<td>100%</td>
<td>99,8%</td>
<td>99,9%</td>
<td>99,5%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., CE</td>
<td>92,3%</td>
<td>89,7%</td>
<td>87,6%</td>
<td>80,8%</td>
</tr>
<tr>
<td>JavaC</td>
<td>ConIns, NoCE</td>
<td>1630</td>
<td>6500</td>
<td>5822</td>
<td>136288</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>99,8%</td>
<td>99,9%</td>
<td>99,8%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, NoCE</td>
<td>100%</td>
<td>100%</td>
<td>99,5%</td>
<td>97,8%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, CE</td>
<td>99,8%</td>
<td>99,9%</td>
<td>99,4%</td>
<td>97,8%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., NoCE</td>
<td>99,9%</td>
<td>99,8%</td>
<td>99,9%</td>
<td>99,9%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., CE</td>
<td>99,7%</td>
<td>99,8%</td>
<td>99,7%</td>
<td>99,9%</td>
</tr>
<tr>
<td>Obfuscator</td>
<td>ConIns, NoCE</td>
<td>3318</td>
<td>10463</td>
<td>8427</td>
<td>98975</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>99,5%</td>
<td>97,9%</td>
<td>99,5%</td>
<td>99,1%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, NoCE</td>
<td>99,8%</td>
<td>99,8%</td>
<td>99,5%</td>
<td>99,1%</td>
</tr>
<tr>
<td></td>
<td>Obj'Sense, CE</td>
<td>99,4%</td>
<td>98,4%</td>
<td>99,1%</td>
<td>65,1%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., NoCE</td>
<td>99,9%</td>
<td>98,6%</td>
<td>99,1%</td>
<td>99,1%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot., CE</td>
<td>92,8%</td>
<td>93,0%</td>
<td>93,3%</td>
<td>95,5%</td>
</tr>
</tbody>
</table>

Table 5.6: Analysis Results - Call Graph
<table>
<thead>
<tr>
<th>program</th>
<th>configuration</th>
<th>safe accesses</th>
<th>meths</th>
<th>appl.meths</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaCC</td>
<td>ConIns, NoCE</td>
<td>68,3%</td>
<td>1675</td>
<td>921</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>68,3%</td>
<td>99,6%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>68,3%</td>
<td>97,0%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>68,3%</td>
<td>96,6%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,NoCE</td>
<td>69,9%</td>
<td>99,9%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,CE</td>
<td>69,9%</td>
<td>99,6%</td>
<td>100%</td>
</tr>
<tr>
<td>Emma</td>
<td>ConIns, NoCE</td>
<td>82,5%</td>
<td>3872</td>
<td>1090</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>83,9%</td>
<td>86,1%</td>
<td>96,1%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>83,7%</td>
<td>99,7%</td>
<td>99,9%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>85,1%</td>
<td>86,0%</td>
<td>96,1%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,NoCE</td>
<td>85,6%</td>
<td>99,1%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,CE</td>
<td>86,5%</td>
<td>86,1%</td>
<td>96,1%</td>
</tr>
<tr>
<td>JavaC</td>
<td>ConIns, NoCE</td>
<td>68,2%</td>
<td>2967</td>
<td>1437</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>68,3%</td>
<td>97,9%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>68,2%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>68,4%</td>
<td>97,9%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,NoCE</td>
<td>69,5%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,CE</td>
<td>69,1%</td>
<td>97,9%</td>
<td>100%</td>
</tr>
<tr>
<td>Obfuscator</td>
<td>ConIns, NoCE</td>
<td>80,1%</td>
<td>4029</td>
<td>3025</td>
</tr>
<tr>
<td></td>
<td>ConIns, CE</td>
<td>80,1%</td>
<td>99,4%</td>
<td>99,7%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, NoCE</td>
<td>81,1%</td>
<td>99,8%</td>
<td>99,7%</td>
</tr>
<tr>
<td></td>
<td>ObjSense, CE</td>
<td>81,0%</td>
<td>99,3%</td>
<td>99,5%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,NoCE</td>
<td>86,1%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td></td>
<td>ValueAnnot.,CE</td>
<td>84,7%</td>
<td>93,8%</td>
<td>93,5%</td>
</tr>
</tbody>
</table>

Table 5.7: Analysis Results - Other Metrics
slowed down by factor two when enabling conditional execution. This is on the one hand due to the increased program size, which yields higher execution times in the loop tree algorithm, and, on the other hand, due to an increased number of required computations.

Our novel suggestion to improve SSA construction shows noticeable precision gains. We believe that it will come at nearly no additional cost once not simulated on graph interpretation level, but implemented in an SSA construction algorithm.
6 Conclusion and Future work

In this chapter, we summarize the previous chapters. We then draw the conclusions from the work done and present thoughts and ideas for future work.

6.1 Summary

In Chapter 3, we have presented two frameworks. Our first framework allows for flexibly mapping Firm, a Memory-SSA representation, to different variations of P2SSA, our sparse Memory-SSA representation. The framework allows for configuring which kind of nodes are to be mapped, e.g., nodes related to primitive types or control-flow related statements are optionally omitted. Further, it allows to optionally map exception handling. Our analysis framework allows for flexibly exchanging core parts of a points-to analysis, like context sensitivity, name schema, and more.

In Chapter 4, we have used these frameworks to implement different variations of Points-to SSA. We have provided different name schemata - a class name schema, and an allocation site name schema - and two context strategies - a context insensitive strategy, and 1-object-sensitivity. Our baseline implementation for the interpretation of Points-to SSA supports the set of operations described in [Lun04], plus additionally operations for evaluating conditional execution.

In the same chapter, we have also presented a novel idea to use operations related to control flow for improving the precision of data flow analysis. While conceptually this is a modification of SSA construction, we simulate the effects with a variation of our node interpretation algorithm.

In Chapter 5, we compared different configurations of our frameworks in terms of analysis time and precision. Adding conditional execution to our P2SSA graphs increased the time needed to perform our Points-to analysis on average by factor two. One of the benchmark programs showed side-effects from missing support for native methods and exception handling when enabling conditional execution. Besides that, potential for precision gain is shown.

Our novel idea for SSA construction shows measurable gains in precision. While its benefits currently depend on the coding-style of programmers, we believe that it will come at no cost once implemented in SSA construction.

6.2 Conclusion

We have fulfilled our goal to provide a frontend to construct Firm SSA from Java source and bytecode by connecting two existing frontends, Rec2Firm and Bc2Firm. We have additionally improved the latter by implementing an optimization - rapid type analysis - which improves its performance by avoiding construction of unnecessary code.

We have redesigned Points-to SSA, a sparse Memory SSA representation, to allow for easily adding support for new node types, like nodes related to primitive types or exception handling. Our framework for building P2SSA from Firm allows to map to different versions of this P2SSA representation.

We provided an analysis framework for our Points-to SSA representation. We used this to implement two analyses, each context-insensitive and flow-sensitive. One of them is an implementation as described in [Lun04], the other, furthermore, supports Boolean operations, and performs dead-code analysis. We have additionally implemented 1-object-sensitivity using our analysis framework.

We have presented a novel idea for SSA construction which can improve dataflow analysis, and implemented a simulation of it by extending the semantics of our node
interpretation.

We compared the different implementations running a set of benchmark programs, measuring performance and several metrics.

6.3 Future Work

6.3.1 Analysis

In order to provide a complete analysis, it is indispensible to provide support for all native methods from the JDK. While it is possible to provide stubs with the mechanisms provided into our framework, a more practical approach would be to provide a description language for specifying the effects of external components. An example of how this could be done is given by Liekweg and Boesler, who implemented such an approach for the C implementation of libFirm [LB05].

Our implementation further needs to be extended to support exception handling in order to provide a more conservative analysis.

Further, more analysis techniques can be integrated in our implementation. For instance, implementing $\chi$-terms [Tra01] for providing intra-procedural path-sensitivity would bring benefits for analysis precision.

6.3.2 Object Equality

It is in general impossible to decide on the runtime-equality of abstract objects, since an abstract object may always denote multiple runtime objects. However, sometimes an abstract object denotes exactly one concrete runtime object: We have already implemented this for the null-object. Another condition that exists - but which we did not implement - are compile-time constants: For once, these are final class-fields. But the Java Virtual Machine Specification [LY99] allows for another condition for equality, namely String-objects that are compile time constants: The specification demands that a pool of objects is maintained for such character strings; i.e., two such constants that have equal value can also be regarded as equal in our analysis, even regardless of their syntactical creation site.

If further, more sophisticated conditions can be identified is an open research question. We just name the Singleton design pattern as a promising target.

6.3.3 Memory Partitioning

Firm provides the possibility to partition memory into regions in order to reduce the number of memory dependencies. The possibly faster stabilization of the data-flow analysis may yield a better overall performance, since the number of contours that have to be re-analyzed on account of memory state changes only may greatly reduce. This may even improve the precision of an analysis, as more benefits may be obtained from inter-procedural flow sensitivity.

The problem is that such a transformation requires the results of a points-to analysis itself. A possible approach is to run an imprecise, but fast points-to analysis in preparation for a precise, but expensive second analysis. Experiments will have to show if any benefit can be obtained.

6.3.4 Typing Arrays

The contents of distinct arrays may are mixed up when they are used as method arguments. We have discussed this problem for in Section 3.3.2. The problem gets worse because Firm does not provide typed arrays, i.e., an array of type $A$ and an array of type $B$ are not mapped to types $A[]$ and $B[]$ in Firm - all arrays of class
types are of the same type. Therefore, even the contents of arrays that contain elements of types that are in no subtype-relation may get mixed up. By adapting Firm to support typed arrays, this problem can be solved.
References


A SSA and Firm

**Static Single Assignment** form - in short, SSA - is an intermediate representation technique first developed by Cytron et al. ([CFR+89]). Every variable is assigned a value exactly ones. For each definition in original form, a new *version* of that variable is created during SSA construction. To decide what version of a variable is valid in SSA form, \( \phi \)-nodes are introduced: \( \phi \)-nodes are artificial operations that take the possible versions of a variable as arguments and decide, depending on control flow, which of these operands is the currently valid definition. SSA form provides many benefits for program analysis, for instance, use-def relations become explicit.

**Memory SSA** is an extension to SSA that was first introduced by Armbruster and von Roques in [AvR96] and refined by Trapp [Tra01]. In Memory SSA, the traditional operation ordering of a basic block structure is abandoned and replaced by a directed graph structure. Basic blocks no longer contain a sequence of instructions but dependencies are modeled by edges. For instance, memory access dependencies are modeled by memory edges, and local variables are modeled by data flow edges, thus making def-use relations explicit; local variables also no longer have a name.

**Firm** [TLB99] is an intermediate representation description based on Memory SSA. For Firm, implementations for C and Java exist.
B Used Libraries

This appendix gives a short overview of third-party libraries that we use for our implementation.

B.1 LibFirm
For our building framework, we use the Java implementation of Firm, which is shortly described in Appendix A.

B.2 Recoder
Recoder [Rec] is a Java framework for statical source code metaprogramming, providing analysis and transformation functionality. For this thesis project, it is the base of the Rec2Firm Java source code frontend for the Firm library.

B.3 BCEL
BCEL[BCE], the “Byte Code Engineering Library”, is a library developed by the Apache Jakarta project, intended for analyzing and transforming Java byte code. The Bc2Firm frontend we use relies on this library.

B.4 ASM
The ASM framework[ASM] is a tool similar to BCEL. Our implementation of the Rapid Type Analysis uses this library.

B.5 Grail
Grail [Pan05] is a graph library being developed at Växjö University. Our SSA representation is built on top of it, and we use the implementation of the loop tree algorithm[Tra01] that it provides.
## C Configuration Options

The following tables give an overview of the configuration options for the P2SSA construction process, and the analysis, respectively.

<table>
<thead>
<tr>
<th>option</th>
<th>effect</th>
<th>Sec./Ch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>make exceptions</td>
<td>Build exception handling nodes and edges.</td>
<td>3.2.2</td>
</tr>
<tr>
<td>make primitive types</td>
<td>Add support for primitive types.</td>
<td>3.2.1</td>
</tr>
<tr>
<td>make conditional execution</td>
<td>Map basic blocks structure; support for instanceof and compare nodes as well as nodes concerning boolean operations.</td>
<td>3, 4</td>
</tr>
<tr>
<td>add reflexive phi edges</td>
<td>Add reflexive edges for ( \phi )-nodes.</td>
<td>3.2.2</td>
</tr>
<tr>
<td>call replacers</td>
<td>list of plugins that replace calls.</td>
<td>4.4.5</td>
</tr>
</tbody>
</table>

Table C.1: Options for Constructing P2SSA
<table>
<thead>
<tr>
<th><strong>option</strong></th>
<th><strong>effect</strong></th>
<th><strong>section</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>semantics</td>
<td>The Semantics implementation to use.</td>
<td>3.3.1, 4.4.4</td>
</tr>
<tr>
<td>name schema</td>
<td>The NameSchema implementation to use.</td>
<td>3.3.1, 4.4.3</td>
</tr>
<tr>
<td>context strategy</td>
<td>The ContextStrategy to use.</td>
<td>3.3.1, 4.4.2</td>
</tr>
<tr>
<td>graph stabilizer</td>
<td>The GraphStabilizer to use.</td>
<td>3.3.1, 4.4.1</td>
</tr>
<tr>
<td>overwrite conditional execution</td>
<td>if set, the semantics implementations will not skip the interpretation of nodes due to conditional execution, even if the basic block structure has been mapped.</td>
<td>n/A</td>
</tr>
<tr>
<td>run VM initialization</td>
<td>Whether or not to run System.initializeSystemClass() before analyzing the target program.</td>
<td>n/A</td>
</tr>
<tr>
<td>instantly initialize fields</td>
<td>When to initialize object fields with <em>null</em>.</td>
<td>4.5.2, 4.6</td>
</tr>
<tr>
<td>use class name schema for</td>
<td>If AllocationSiteNameSchema is used, this specifies a list of types for which a class name schema is used instead.</td>
<td>4.4.3</td>
</tr>
<tr>
<td>reachable methods filter</td>
<td>prefix of type names to include in high-level analyses.</td>
<td>5.1</td>
</tr>
</tbody>
</table>

Table C.2: Options for Configuring the Analysis