Inter-procedural Data-flow Analysis with IFDS/IDE and Soot *

Eric Bodden

Secure Software Engineering Group
European Center for Security and Privacy by Design (EC SPRIDE)
Technische Universität Darmstadt
Darmstadt, Germany
eric.bodden@ec-spride.de

Abstract

The IFDS and IDE frameworks by Reps, Horwitz and Sagiv are two general frameworks for the inter-procedural analysis of data-flow problems with distributive flow functions over finite domains. Many data-flow problems do have distributive flow functions and are thus expressible as IFDS or IDE problems, reaching from basic analyses like truly-live variables to complex analyses for problems from the current literature such as typestate and secure information-flow.

In this work we describe our implementation of a generic IFDS/IDE solver on top of Soot and contrast it with an IFDS implementation in the Watson Libraries for Analysis (WALA), both from a user's perspective and in terms of the implementation. While WALA's implementation is geared much towards memory efficiency, ours is currently geared more towards extensibility and ease of use and we focus on efficiency as a secondary goal.

We further discuss possible extensions to our IFDS/IDE implementation that may be useful to support a wider range of analyses.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Design, Performance, Documentation

Keywords Inter-procedural static analysis, flow-sensitive analysis, IFDS, IDE

1. Introduction

The IFDS framework by Reps, Horwitz and Sagiv [7] is a conceptual framework for computing the results to inter-procedural, finite, distributive subset (IFDS) problems. In such problems, flow functions are defined over a finite domain D and have to be distributive over the merge operator "$\sqcap$", i.e., for any flow function f and any $a, b \in D$ it must hold that $f(a) \sqcap f(b) = f(a \sqcap b)$. Many data-flow problems do have distributive flow functions and are thus expressible as IFDS or IDE problems, reaching from basic analyses like truly-live variables [7] to complex analyses for problems from the current literature such as typestate [5] and information-flow [1].

As Reps et al. show, when these conditions are fulfilled, the inter-procedural data-flow analysis problem can be fully reduced to a graph reachability problem: the IFDS framework defines an algorithm operating on a so-called *exploded super graph*. In this graph, any node (s, d) is reachable from a special distinct start node if and only if the data-flow fact d holds at statement s.

The IDE framework for inter-procedural distributive environments extends IFDS to allow analyses to compute additional values from a domain V at the time at which reachability is decided. Instead of merely deciding whether a node is reachable, the IDE algorithm propagates additional V-type information along any path in question. In this setting, flow functions effectively become distributive environment transformers, transforming mappings from D to V.

Soot [6] is one one of the most widely used frameworks for the static analysis and transformation of Java programs. Over more than a decade, Soot has been maintained and extended by an active user community, including many users from research and some from industry. The prime motivation for creating Soot in the first place was to foster inter-operability between different static analyses developed in the research community, and to enable objective and realistic comparisons between different static-analysis algorithms. One important feature that Soot has been lacking ever since, however, is the implementation of an inter-procedural program-analysis framework.

* This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

We therefore developed an implementation of an IFDS/IDE solver on top of Soot [6]. In this paper we discuss the important design decisions we had to make and describe the implementation from a user's perspective. We think that with an IFDS/IDE solver, the inter-operability and comparability within Soot can be taken to another level. Multiple analyses based on the same IFDS/IDE solver should be able to integrate with ease. Moreover, general optimizations to this solver will benefit all those analyses.

We further contrast our implementation with an existing IFDS implementation in the Watson Libraries for Analysis (WALA) [10]. The WALA-based implementation is quite mature and has been successfully used to implement a number of analyses. It focuses very much on memory efficiency. While we describe some performance optimizations in this paper, our primary goals are extensibility and ease of use.

While our implementation is complete and tested, it has not yet been released as part of Soot. We hope that an active discussion at the SOAP workshop will allow us to obtain community feedback on our design decisions and on the implementation's usability. As part of this paper we also discuss possible extension to our implementation that could support many whole-program analyses. Our current implementation is available at: `http://bodden.de/ide/`

To summarize, this paper presents the following original contributions:

- a description of our IFDS/IDE implementation from a user's perspective,
- an account of important implementation details, particularly deviations from the original IFDS and IDE algorithms,
- a comparison with the IFDS implementation in WALA,
- a discussion of possibly useful future extensions.

We continue by describing the IFDS and IDE frameworks on a conceptual level. Section 3 describes our actual implementation; we discuss differences to WALA in Section 4. Section 5 discusses possible extensions and related work.

2. The IFDS and IDE Frameworks

The so-called IFDS framework by Reps, Horwitz and Sagiv [7] defines a general way to solve inter-procedural, flow-sensitive, context-sensitive analysis of finite distributive subset problems. The algorithm has worst-case complexity $O(ED^3)$, where E is the number of control-flow edges (or statements) of the analyzed program and D is the size of the analysis domain. As this estimate shows, the efficiency heavily depends on the size of the domain.

The main idea of the IFDS framework is to reduce any program-analysis problem formulated in this framework to a simple graph-reachability problem. The IFDS algorithm builds, based on the program's inter-procedural control-flow graph, a so-called "exploded super graph" in which

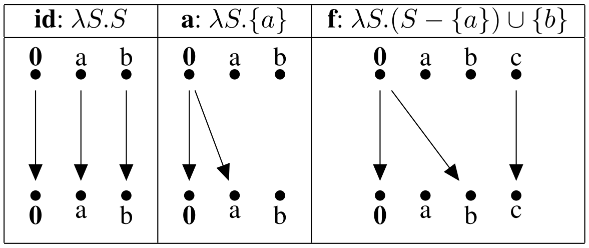


Figure 1: Function representation in IFDS, reproduced from [7]

a node (s, d) is reachable from a selected start node $(s_0, \mathbf{0})$ if and only if the data-flow fact d holds at s. (By "fact" we mean any logical statement, such as "variable v has definitely been initialized.") To achieve this goal, the framework has to encode data-flow functions as nodes and edges. Figure 1, reproduced from [7], shows how to represent compositions of typical *gen* and *kill* functions, as they are used in information-flow analysis. The function **id** is the identity function, mapping each data-flow fact before a statement onto itself. In IFDS, the value **0** represents an empty fact that is always valid, i.e., two nodes representing **0** will always be connected. This **0** value is used to generate data-flow facts unconditionally. The flow function **a** generates the data-flow fact a, and at the same time kills all other facts (such as b). Function **f**, on the other hand kills a, generates b and leaves all other values (such as c) untouched.

As an example, in Figure 2 we show the exploded supergraph for an information-flow analysis over the following simple program, in which we assume it to be a violation if the return value of `secret()` flows into `print`:

```
void main() {
  int x = secret();
  int y = 0;
  y = foo(x);
  print(y);
}
int foo(int p) { return p; }
```

The analysis in Figure 2 uses the program's variables as analysis domain. As we show in the figure, there are four different kinds of edges a user needs to define.

Call edges connect call sites to callees, passing information about code elements that concern the callee, e.g. actual method arguments.

Return edges pass information the other way around, e.g. about the return value.

Call-to-return edges pass information from directly before a call site to all of the call site's successor statements. Such edges typically pass information that do not concern the callee.

Normal edges for all other statements.

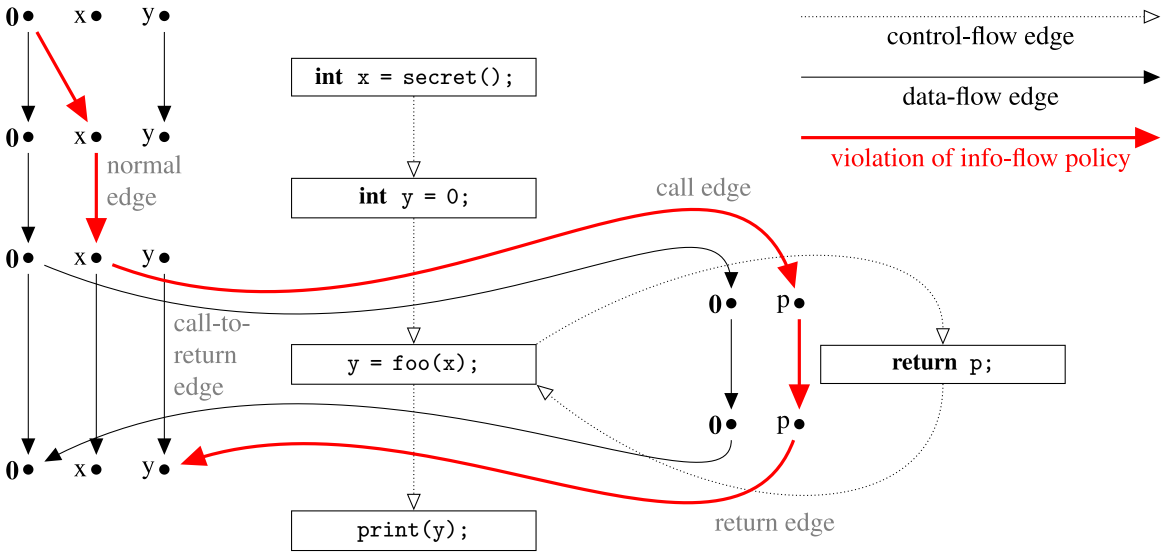
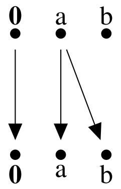


Figure 2: Exploded super-graph for an IFDS information-flow analysis

Separable vs. non-separable functions The flow functions in Figure 1 are what is frequently called "separable": their output only depends on the current statement but not on the input value. This can be seen by the fact that all edges contained in the graphs connect only to the unconditional input **0**. For separable IFDS problems, the IFDS algorithm has the better complexity bound $O(ED)$.



Many analysis problems, such as truly-live variables [7] or secure information flow [1], however, use flow equations that are non-separable. For instance, the function representation to the right could be chosen to model an assignment `b=a` in an information-flow analysis. Here, `a` has the same value as before the assignment, modeled by the arrow from a to a, and `b` obtains `a`'s value, modeled by the arrow from a to b. If `b` was previously holding a secret value, then it will only remain to do so if `a` contained a secret value as well. This is modeled by a missing arrow from b to b.

The IDE framework

The IDE framework for "inter-procedural distributed environment transformers" by the same authors [9] is an extension of IFDS that effectively allows a program analysis to extend the reachability to a value-computation problem. If a data-flow fact d from the domain D is reachable at a given statement, then the IDE algorithm will compute a value from a secondary domain V along all paths that reach d. IFDS can be modeled as a special case of IDE in which this "value domain" V is the binary domain $\{\top, \bot\}$. The complexity of the IDE algorithm is the same as for IFDS: $O(ED^3)$.

IDE can be quite useful from a performance point of view. For example, consider the problem of constant propagation. In such a setting, any statement s must be associated with information about a mapping from a finite set of variables $x \in \mathit{Var}$ in scope at s to values $val(x) \in \mathbb{N}$. In theory, such a problem could be solved in IFDS by using a finite domain $D := \mathit{Var} \times \mathbb{N}$, assuming that we put a finite upper bound on the representation of $\mathbb{N}$. However, as can easily be seen, this would cause the domain D to grow infeasibly large. In IDE, on the other hand, one can model the problem by choosing just $D := \mathit{Var}$ as the finite domain and $V := \mathbb{N}$ as the value domain. Since the size of V is irrelevant to the complexity of the IDE algorithm, IDE will terminate more quickly [9].

3. IFDS/IDE Implementation

We have implemented an IDE solver as an extension to Soot. The implementation is written in pure Java. The solver itself is also absolutely generic; it has no dependencies on Soot and can therefore, in principle, even be re-used for other static-analysis frameworks. We achieve this genericity through the use of Java's generic type parameters.

3.1 User perspective

Figure 3 shows how users define an IFDS problem; one simply creates a class implementing the interface shown, and passes it to an `IFDSSolver` object, followed by a call to `solve()`. Our actual solver is completely generic and has no dependencies on Soot. We achieve this genericity through type parameters. The parameters `N,D,M` represent nodes (typically `Unit`), data-flow facts (client specific) and methods respectively (typically `SootMethod`). The method `initialSeeds` returns the initial information used to boot-

```
interface IFDSTabulationProblem<N,D,M> {
  Multimap<M,D> initialSeeds();
  D zeroValue();
  InterproceduralCFG<N,M>
      interproceduralCFG();
  FlowFunctions<N,D,M> flowFunctions();  }
```

Figure 3: Interface for defining IFDS problems

```
public interface
    IDETabulationProblem<N,D,M,V>
  extends IFDSTabulationProblem<N,D,M>{
  EdgeFunctions<N,D,M,V> edgeFunctions();
  JoinLattice<V> joinLattice();
}
```

Figure 5: Interface for defining IDE edge functions

```
interface FlowFunctions<N, D, M> {
  public FlowFunction<D>
      getNormalFlowFunction(
    N curr, N succ);
  public FlowFunction<D>
      getCallFlowFunction(
    N callStmt, M destinationMethod);
  public FlowFunction<D>
      getReturnFlowFunction(
    N callSite, M calleeMethod,
    N exitStmt, N returnSite);
  public FlowFunction<D>
      getCallToReturnFlowFunction(
    N callSite, N returnSite);  }
```

Figure 4: Interface for defining flow functions

strap the analysis at program entry points (as defined through the result of `Scene.getEntryPoints()`). The method returns a multi map, associating entry-point methods with `D`-type facts that hold at the beginning of those methods. The method `zeroValue` returns the value representing the **0** node. (We experimented with using `null` as a representative for **0** but quickly found that this made the code harder to understand and also caused problems with some map-based data structures that do not accept `null` as a key.) Method `interproceduralCFG` returns an inter-procedural control-flow graph. To this end, we provide a default implementation in form of the class `DefaultInterproceduralCFG`, which implements an inter-procedural control-flow graph with node type `Unit` and method type `SootMethod`. The graph is implemented as a combination of `ExceptionalUnitGraphs` such that exceptional flow is handled properly be default. Clients can customize `DefaultInterproceduralCFG` to tailor it to their specific needs. It is the only Soot-specific code. Users would need to customize only this class to use our solver with a different analysis framework.

Method `flowFunctions` returns another object that instantiates flow functions for individual flow edges. We show the appropriate interface in Figure 4. As explained before, there are four categories of flow edges, and the interface mirrors this fact. One significant design decision is which kind of context information should be passed to clients such that they will be able to decide what particular flow function must be instantiated. As we will explain in Section 4, our particular choice differs slightly from the one in WALA, but we nevertheless think that our choice is appropriate (see Fig. 4).

An individual flow function is simply a function object with the following signature:

```
Set<D> computeTargets(D source);
```

For each source node, the function returns all target nodes which flow edges connect to the given source.

Defining IDE problems

Users define IDE problems as extensions to IFDS problems as shown in Figure 5, this time passing it to an `IDESolver` object. (`IFDSSolver` is actually just a subclass of the more generic `IDESolver`, converting an IFDS problem into an IDE problem over a two-element domain.) The interface contains a method `edgeFunctions` returning “edge functions”, a term by which we denote the `V`→`V` type functions that compute `V`-typed values along edges between `D`-typed nodes. The instantiation of edge functions (not shown) happens similar to the one of flow functions, except that the respective methods obtain additional `D`-typed source and target nodes as inputs. As for flow functions, there are four methods for the four different categories of functions. The method `joinLattice` returns a lattice object defining two `V`-typed top and bottom elements as well as a join function over `V`. The bottom element is used to initialize the function computation at entry points. The top element is used at merge points and is typically the neutral element of the join operator, which is used to merge `V`-type values a those points.

The definition of edge functions (Figure 6) requires a bit more input from the analysis clients, who must not just define the function itself but also how to compose, join and compare one function with another. The join operation for edge functions must be consistent with the definition of the join lattice. At this point, to obtain a reasonably efficient analysis, it is paramount that edge functions can be composed in place. As an example, consider the analysis problem of linear-constant propagation, in which we may wish to compose functions $\lambda x.\ x + 2$ and $\lambda x.\ x + 3$. In this case, one should not return a function object stored as an explicit composition, i.e., as $\lambda x.\ (x + 2) + 3$ but rather immediately reduce this function object to $\lambda x.\ x + 5$. If this rule is not obeyed, then function definitions could grow unduly large.

In related work, we use our IDE support to implement an information-flow analysis for software product lines [1].

```
public interface EdgeFunction<V> {
  V computeTarget(V source);
  EdgeFunction<V>
      composeWith(EdgeFunction<V>
      secondFunction);
  EdgeFunction<V>
      joinWith(EdgeFunction<V>
      otherFunction);
  public boolean equalTo(EdgeFunction<V>
      other);
}
```

Figure 6: Interface for defining a single IDE edge function

3.2 Important implementation details

We next discuss a few important differences between our implementation and the algorithms that Reps, Horwitz and Sagiv originally proposed.

The original formulation of the IFDS and IDE algorithms [7, 9] requires that flow functions are invertible because one part of the algorithm computes flow backwards [4]. This puts an extra burden on the analysis client, who thus has to explicitly define the inverse of each flow function. We use a trick originally proposed by Naeem et al. [4] that allows IFDS and IDE implementations to circumvent this problem. The same authors also proposed another trick that we make use of: to compute the program's inter-procedural super graph on the fly.

Another problem with the original algorithms is that they store summaries of all computed paths in the form of so-called path edges (or jump functions in the case of IDE) in a single list that is not indexed. We instead use a special indexed data structure (implemented in a class `JumpFunctions`) that allows $O(1)$ access to groups of path edges according to various keys such as source statement and node or just target statement. We implemented a similar data structure for the summary edges that the algorithms store. Those indexing data structures currently rely on Google's Guava collection library [3], but this dependency could easily be removed.

3.3 On the finiteness of the analysis domain

The original formulation of the IFDS and IDE algorithms demands that the analysis domain D be finite. Interestingly, we found that this is not a real requirement. Since both algorithms only explore those parts of the exploded super graph that are actually reachable, it is sufficient if the abstraction adheres to the so-called "ascending-chain condition", i.e., every ascending chain obtained by applying the flow functions to elements of the domain must eventually terminate (e.g., by reaching the top element). This is important to know, as it may be hard to impossible to enumerate large domains, for instance all possible alias sets of a program. Nevertheless, enumerating only the ones that do arise along a particular path may well be tractable.

3.4 Backward Analyses

One question we could not yet find answered in the current scientific literature is how to conduct a backwards analysis using IFDS/IDE. We found out that we could apply a simple trick: we implemented a customized version of `DefaultInterproceduralCFG` that instead of creating normal `ExceptionalUnitGraphs` creates reverted versions of those graphs, swapping heads with tails and successors with predecessors. We then modified our original IDE solver to accept multi-headed unit graphs—a quite natural extension of the original IDE algorithm.

4. Comparison to IFDS in WALA

The T.J. Watson Libraries for Analysis (WALA) [10] is another Java analysis framework, originally developed by IBM and now maintained as an open-source project. WALA's design differs to Soot in several respects, however, also WALA uses an intermediate representation for its analyses, in this case a representation in SSA form. WALA also features an implementation of an IFDS solver. We briefly outline the main differences in its design compared to our solver, and the consequences of those decisions.

Our own implementation was designed for maximal extensibility, maintainability and ease of use. Efficiency to us is a secondary concern. WALA's IFDS implementation is tuned to be highly memory efficient. It uses bit vectors to represent analysis domains. As a result, WALA users define flow functions not directly in terms of elements from the analysis domain but rather in terms of integer numbers. This can be quite efficient in cases where flow functions can directly operate on those numbers (e.g. the special value **0** is represented by the number 0), but can be awkward in other cases, in which clients need to map those numbers back to domain objects. We are yet looking for a way to achieve similar efficiency while at the same time hiding such complex implementation details.

WALA also provides additional context information for instantiating flow functions: it passes a return site to the method that creates call flow functions. This means that clients can create different call flow functions for each possible successor of this call statement. This is useful in backwards analyses such as in Snugglebug [2]. WALA further supports special call edges for the unresolved calls that have no callees. In our approach, we instead assume that users would propagate appropriate information using call-to-return edges.

The original IFDS and IDE algorithms, as well as our implementation, generate summary edges that connect call sites with their successor statements, summarizing the effects of any potentially called method. WALA instead stores summary edges on the side of the callee, which increases sharing of those edges, decreasing memory consumption. We plan to implement similar support in the future.

A final important difference is that WALA's solver only supports IFDS problems while we support full IDE.

5. Extensions and Related Work

We briefly explain a set of possible extensions that we think may be useful for implementing certain classes of analyses, some of which have been proposed in related work.

Support for branched analyses Some forward analyses are branched, i.e., propagate different information along different branches. For instance, a nullness analysis will propagate "`x` is definitely `null`" along the `true` branch of a conditional `if(x==null)` and the fact "`x` is definitely not `null`" along the `false` branch. One can currently implement branched analyses by analyzing the structure of the source statement in the method that creates "normal" flow functions. However, this is rather verbose and it may be desirable to have a cleaner interface for branched analyses.

Exceptional control flow A similar problem arises with exceptional control flows. Currently, we treat exceptional edges as normal flow edges. This is usually the right choice for a conservative analysis but may not be desirable for a data-flow analysis targeted towards analyzing exceptions. Again, it may be desirable to support such analyses through a dedicated interface.

Combination of multiple analyses It can easily be envisioned that in the future, researchers will implement a wide range of analyses in our IFDS/IDE framework. At this point it would be desirable to have a means to easily combine multiple analyses with each other, and to increase the reuse potential by pre-defining certain frequently-used domains and flow functions. This is exactly the kind of reuse that is currently hard to achieve in Soot because there is no pre-defined framework for inter-procedural analyses.

Persisting summaries Rountev et al. have developed a mechanism for persisting IDE-based summary information such that it can be re-used when re-running the analysis [8]. As the authors showed, this approach promises significant performance gains when used to persist information for frequently used large libraries. We believe that this would be a quite useful extension that would immediately benefit all analysis clients.

A concurrent solver Rodriguez and Lhoták have proposed a concurrent version of the IFDS algorithm based on Scala's actor framework. We believe that this idea could relatively easily be ported to IDE and also to Java's concurrency API. Just as with persisting summaries, we believe that this would be a useful extension that would benefit all analysis clients and could promise significant speedups on multi-core machines.

Return Flow Functions Naeem et al. proposed an extension to return flow functions that allows clients to map information from a callee method back to its caller based on caller-side information at the original call site [4]. We have not currently implemented this feature.

6. Conclusion

We have presented our implementation of an IFDS/IDE solver in Soot, from a user perspective and in terms of important implementation details. Further, we have contrasted our implementation with an existing IFDS implementation in WALA. We hope that our solver implementation will serve as a basis for future collaborative research in the area of whole-program analysis. While not all analyses problems may fit the IFDS/IDE framework, the ones who do could benefit from mutual reuse and from additional performance optimizations applied to the solver itself.

Acknowledgements. We thank Steven Fink for clarifications of some design decisions taken in WALA as well as Bruno Dufour and the anonymous reviewers for their constructive comments on an earlier draft of this paper.

References

[1] Eric Bodden. Position paper: Static flow-sensitive & context-sensitive information-flow analysis for software product lines. Workshop on Programming Languages and Analysis for Security (PLAS 2012), June 2012. To appear.

[2] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI'09*, pages 363–374, New York, NY, USA, 2009. ACM.

[3] The Google Guava collection library. `http://guava-libraries.googlecode.com/`.

[4] Nomair Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *Compiler Construction*, volume 6011 of *LNCS*, pages 124–144, 2010.

[5] Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, pages 347–366, New York, NY, USA, 2008. ACM.

[6] Vijay Sundaresan Patrick Lam Etienne Gagnon Raja Vallée-Rai, Laurie Hendren and Phong Co. Soot - a Java optimization framework. In *CASCON'99*, pages 125–135, 1999.

[7] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61, New York, NY, USA, 1995. ACM.

[8] Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 53–68, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT'95*, pages 131–170, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[10] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net/`.