Soot
A Java Bytecode Optimization Framework

Sable Research Group

School of Computer Science

McGill University

Goal

- Provide a Java framework for optimizing and annotating bytecode
- provide a set of API’s easy to use and efficient enough for developing competitive optimizers

Outline

- Contribution
- Framework overview
- Intermediate representations
- Transformations
- Optimizations
- Conclusion

Currently used methods for improving Java performance

- JIT compilers
- Way-Ahead-Of-Time Java compilers
- <u>Optimizing bytecode directly</u>
  - Must address expensive bytecode operations : virtual method call, interface call, object allocation
- <u>Annotating the bytecode</u>
  - Statically checking the safety of memory accesses and annotating the bytecode (eg: array bounds)

Contributions

- 3 intermediate representations used for bytecode optimizations and de-compilations
- Support for both intra-procedural and whole program optimizations
- Able to add future support for bytecode annotation

Framework overview

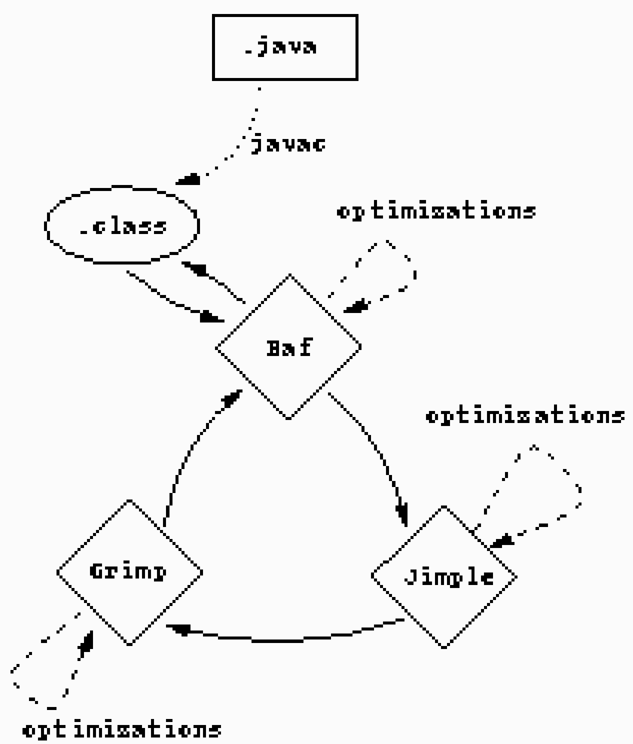


Figure 1: The Soot Optimization Framework consists of three intermediate representations: Baf, Jimple and Grimp.

- Baf - streamlined representation of bytecode
- Jimple - typed 3 - address code suitable for optimizations
- Grimp - aggregated version of Jimple suitable for decompilation and bytecode codification

Intermediate representations

- Baf
  - motivation
    - easier to manipulate than the bytecode (abstracts away the constant pool)
    - some bytecode instructions are untyped (dup ,swap) : difficult to estimate their effect and therefore optimize
  - description
    - stack-based
    - fully typed instructions
    - untyped variables

Intermediate representations(cont.)

- Jimple
  - motivation
    - stack code optimization is difficult
    - 2 types of variables : locals and stack locations
    - untyped nature of stack
  - description
    - 3 address code
    - stack replaced by local variables
    - untyped instructions
    - typed local variables
    - ideal for optimizations

Intermediate representations(cont.)

- Grimp
  - motivation
    - IR difficult to read
    - 3 address code difficult to deal in some cases (eg. Generating good stack code)
  - description
    - compacted version of Jimple : flattened expressions, new and invokespecial compacted to new
    - looks like a partially decompiled Java code

```
public int stepPoly(int x)
{
    if(x < 0)
    {
        System.out.println("foo");
        return -1;
    }
    else if(x <= 5)
        return x * x;
    else
        return x * 5 + 16;
}
```

Figure 2: `stepPoly` in its original Java form.

```
public int 'stepPoly'(int)
{
    word r0, i0

    r0 := @this
    i0 := @parameter0
    load.i i0
    ifge label0
    staticget java.lang.System.out
    push "foo"
    virtualinvoke println
    push -1
    return.i

 label0:
    load.i i0
    push 5
    ifcmpgt.i label1
    load.i i0
    load.i i0
    mul.i
    return.i

 label1:
    load.i i0
    push 5
    mul.i
    push 16
    add.i
    return.i
}
```

Figure 3: `stepPoly` in Baf form.

```
public int 'stepPoly'(int)
{
    Test r0;
    int i0, $i1, $i2, $i3;
    java.io.PrintStream $r1;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    $r1 = java.lang.System.out;
    $r1.println("foo");
    return -1;

 label0:
    if i0 > 5 goto label1;

    $i1 = i0 * i0;
    return $i1;

 label1:
    $i2 = i0 * 5;
    $i3 = $i2 + 16;
    return $i3;
}
```

Figure 4: `stepPoly` in Jimple form. Dollar signs indicate local variables representing stack positions.

```
public int stepPoly(int)
{
    Test r0;
    int i0;

    r0 := @this;
    i0 := @parameter0;
    if i0 >= 0 goto label0;

    java.lang.System.out.println("foo");
    return -1;

 label0:
    if i0 > 5 goto label1;

    return i0 * i0;

 label1:
    return i0 * 5 + 16;
}
```

Figure 5: `stepPoly` in Grimp form.

Transformations

- Bytecode -> Baf
  - stack simulation : types of untyped instructions
  - distributing the constant pool
- Baf -> Jimple
  - produce naïve 3 address code
  - type the local variables (paper)
  - clean up the code (simply collapsing def-use pairs)

Transformations(cont.)

- Jimple -> Grimp
  - aggregate expressions
  - fold constructors
  - aggregate expressions
- Grimp -> Baf
  - expression trees converted to stack based code
- Baf ->Bytecode
  - Pack local variable for placing onto Frame
  - Optimize load/stores (eliminate redundancies)
  - Compute maximum stack height (required by JVM)
  - Produce the bytecode

Optimizations

- Scalar optimizations (implemented)
  - constant propagation and folding
  - conditional and unconditional branch elimination
  - copy propagation
  - dead assignment and unreachable code elimination
  - expression aggregation
- Scalar optimizations (future)
  - common sub-expression elimination
  - loop invariant removal

Optimizations(cont.)

- Whole program optimizations (OOP)
  - call graph based
  - methods for constructing the call graph
    - class hierarchy analysis
    - rapid type analysis
    - variable type analysis (*)
  - methods inlining

Experimental results

	# Jimple Stmts	Base Execution			Speed up: ⟶		Speed up: -O		Speed up: -W	
		Int.	JIT	Int./JIT	Int.	JIT	Int.	JIT	Int.	JIT
_201_compress	3562	441s	67s	6.6	0.86	0.97	1.00	1.00	0.98	1.21
_202_jess	13697	109s	48s	2.3	0.97	0.99	0.99	0.98	1.03	1.03
_205_raytrace	6302	125s	54s	2.3	0.99	0.99	1.00	0.97	1.08	1.10
_209_db	3639	229s	130s	1.8	0.98	1.03	1.01	1.02	1.00	1.03
_213_javac	26656	135s	68s	2.0	0.99	1.01	1.00	1.00	1.01	1.00
_222_mpegaudio	15244	374s	54s	6.9	0.94	0.97	0.99	1.00	0.96	1.05
_227_mtrt	6307	129s	57s	2.3	0.99	1.01	1.00	0.99	1.07	1.10
_228_jack	13234	144s	61s	2.4	0.99	0.97	0.99	0.99	1.00	0.98
sablecc-j	25344	45s	30s	1.5	0.98	1.01	0.99	0.99	1.00	1.04
sablecc-w	25344	70s	38s	1.8	1.00	1.00	1.00	1.01	0.98	1.04
soot-c	39938	85s	49s	1.7	0.98	0.99	0.98	1.00	1.03	0.96
soot-j	39938	184s	126s	1.5	0.98	0.99	0.99	0.99	1.02	1.01

Figure 6: Benchmark characteristics and speed-up results. ⟶, -O and -W represent no optimizations, intraprocedural optimizations, and whole program optimizations, respectively. The programs were executed on a 400Mhz dual Pentium II machine running GNU/Linux with Linux JDK1.2, pre-release version 1.

Conclusions

- Soot : framework for optimizing bytecode
- 3 IRs
- transformations between IRs
- useful for optimizing and decompilation
- Speedup for both interpreter and JIT
- Present work
  - Eliminating redundant loads/stores from baff
  - Adding new optimizations(loop invariant removal , common subexpression elimination)

Intra-procedural Inference of Static Types for Java Bytecode

Sable Technical Report No. 1999-1

McGill Univarsity

Sable Research Group

Outline

- Introduction
- Challenges of Types
- Typing algorithm
- Extending the algorithm for arrays
- Program transformations
- Experimental results
- Conclusions

Introduction

- Bytecode
  - target IR for a variety of compilers (Ada ,ML,Scheme,Eiffel)
  - "well-behaved"(verifiable) - checked by bytecode verifier (eg. each method invocation has the correct number of arguments, arguments are well-typed)
  - bytecode verification
    - static : flow analysis for local type estimation (well typed instructions) and not global
    - dynamic (eg. Array bounds checks)
  - local variables of each method kept on the frame of the method and accessed by index (are not typed)

Introduction(cont.)

- Drawbacks of bytecode
  - stack-based
    - complicates the analysis
    - doesn’t map nicely on existing architectures
    - not easy readable
  - local variables not typed (could be used for both analysis and decompilation)
- Addressing the drawbacks
  - IR representations (Jimple)
  - this paper : typing Jimple

```
public java.lang.String f()
{   <unknown> a;
    <unknown> b;
    <unknown> c;
    <unknown> s;

s1: c = new C();
s2: b = new B();
    if ( ... )
s3:   a = c;
    else
s4:   a = b;
s5: s = a.toString();
s6: return(s);
}
```

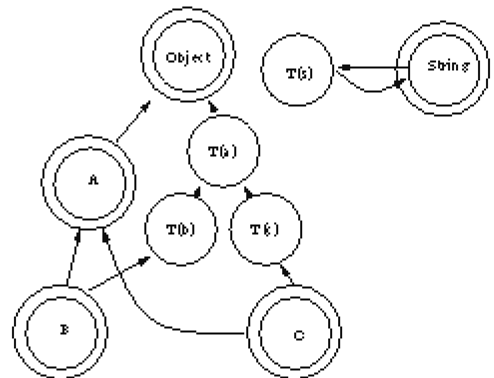
(a) untyped method

```
class A extends Object
{ ... }

class B extends A
{ public String toString() ...;
  ...
}

class C extends A
{ public String toString() ...;
  ...
}
```

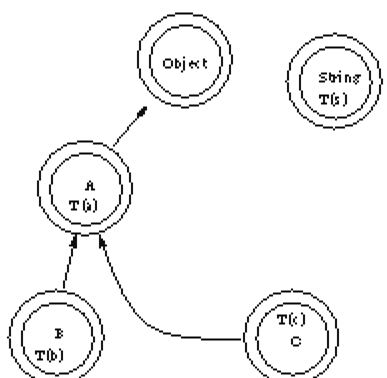
(b) class hierarchy

s1: $T(c) \leftarrow C$
s2: $T(b) \leftarrow B$
s3: $T(a) \leftarrow T(c)$
s4: $T(a) \leftarrow T(b)$
s5: $Object \leftarrow T(a)$
$T(s) \leftarrow String$
s6: $String \leftarrow T(s)$

(c) constraints



(d) graph problem



(e) solution

```
public java.lang.String f()
{   A a;
    B b;
    C c;
    java.lang.String s;

s1: c = new C();
s2: b = new B();
    if ( ... )
s3:   a = c;
    else
s4:   a = b;
s5: s = a.toString();
s6: return(s);
}
```

(f) typed method

Figure 1: Simple example of static typing

Introduction (cont.)

- Problem definition
  - Given: an untyped Jimple method
  - Find : static types of local variables
- Modeled as a graph problem
  - *hard nodes* : types in the declared hierarchy
  - *soft nodes* : type variables (to be determined)
  - *directed edges* : constraints between 2 nodes
    - *constraint* : denoted a<-b if b is *assignable to a*

Challenges of Types

- Declared types versus types at program points
  - verifier checks the type at each local point (are the operands of the instruction right ?)
  - at control flow merge verifier takes LCAC (least common ancestor class of the types of the branch)
  - program from fig.2 will verify , but there is no static solution (a solution where copies are introduced will be presented later)

```
void m() {
     <unknown> a;
     if (...)
       { a = new A();
 s1:     a.f(); // invokevirtual A.f()
       }
     else
       { a = new B();
 s2:     a.g(); // invokevirtual B.g()
       }
 s3: a.toString();
       // invokevirtual Object.toString()
 }
```

(a) untyped method

```
class Object {
   public String toString() { .... }
   ... }

class A extends Object {
   public void f() { .... }
   ... }

class B extends Object {
  public void g() { .... }
  ... }
```

(b) hierarchy

Figure 2: Different types needed at different program points

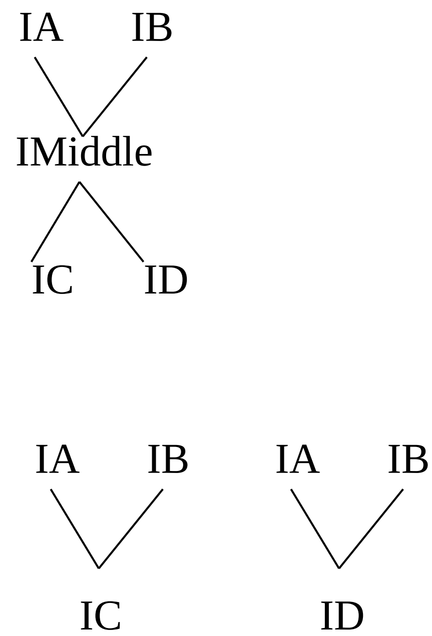
Challenges of Types(cont.)

- Type problems due to interfaces (multiple inheritance)
  - LCAC strategy to resolve types from different branches breaks for multiple inheritance
  - Java verifier checks at run-time
  - Hierarchy I - statically typeable (but can be expensive in the presence of many ancestors)
  - Hierarchy II - not statically typeable (extra-copies can solve the problem)

```
class CC implements IC
{ void f() {}
  void g() {}
}

class CD implements ID
{ void f() {}
  void g() {}
}

class Hard
{ IC getC() { return new CC(); }
  ID getD() { return new CD(); }

  void test()
  { <untyped> a;

    if( ... )
s1:   a = getC();
    else
s2:   a = getD();

s3: a.f(); // invokeinterface IA.f
s4: a.g(); // invokeinterface IB.g
  }
}
```

(a) untyped program

```
[Hierarchy I]
Interface IA { void a(){} }
Interface IB { void b(){} }
Interface IMiddle extends IA, IB {}
Interface IC extends IMiddle {}
Interface ID extends IMiddle {}


[Hierarchy II]
Interface IA { void a(); }
Interface IB { void b(); }
Interface IC extends IA, IB {}
Interface ID extends IA, IB {}
```



(b) hierarchy I and II

Figure 3: Typing interfaces

Typing algorithm

- Algorithm overview
  - abstract problem into a constraint system (directed graph problem)
  - restrict the problem to programs w/o arrays
  - apply simplifying transformations on the graph
  - if no solution found so far perform an exponential search (algorithm is shown to be NP-hard)

Typing algorithm(cont.)

- Building the constraint system
  - a<-b , where a,b - nodes and b is *assignable to a*
    - simple assignment a=b => T(a)<-T(b)
    - binary expression assignment a=b+3 => T(a)<-T(b), T(a)<-int and T(b)<-int
    - method invocationa=b.equals© => T(a)<-int , java.lang.Object<-T(b) and java.lamg.Object<-T(c)

Typing algorithm(cont.)

- Transformations
  - connected components
  - merging primitive types
  - transitive constraints

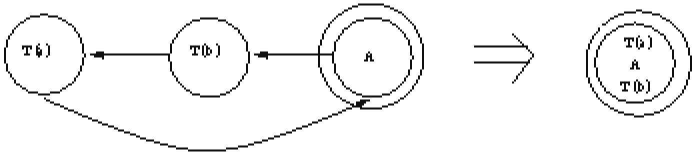


Figure 5: Merging Connected Components

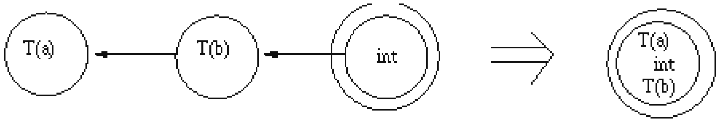


Figure 6: Merging Primitive Types

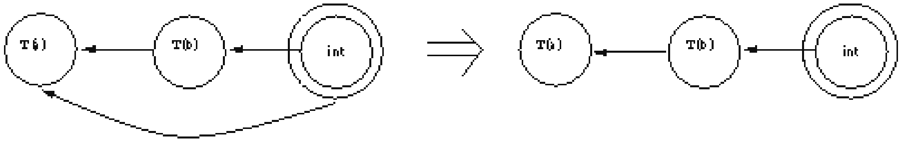


Figure 7: Merging Transitive Constraints

Typing algorithm(cont.)

- merging single constraints
  - *single parent constraint* x if y<-x and x is not parent of anybody else
  - *single child constraint* y if x<-y and x is not child of anybody else
  - transformations
    - merge all single child constrains
    - merge all soft parent constraints
    - merge with LCA
    - merge all remaining parent constraints
- if no solution found perform an exhaustive search

Extending the algorithm to arrays

- Definitions
  - *array constraint a-->b* means a is an array whose type is b
  - *array depth :* number of indirections necessary to get a non-array type (eg. A[][] has depth 2)

Extending the algorithm to arrays(cont.)

- Algorithm
  - starting from hard nodes
    - follow parent constraints : modify the parent depths s.t. they are <= than child’s depth
    - follow array constraints : assign to element type array depth -1
  - propagate array constraints on arrays
    - propagate a constraint between 2 nodes at equal depth to a constraint between their depth 0 element types
    - change a constraint between 2 nodes of different depth to a constraint between the depth 0 element type of lowest depth node and java.lang.Clonable
  - find a solution using the non-array algorithm and only 0-depths nodes
  - propagate the solution back to array depths

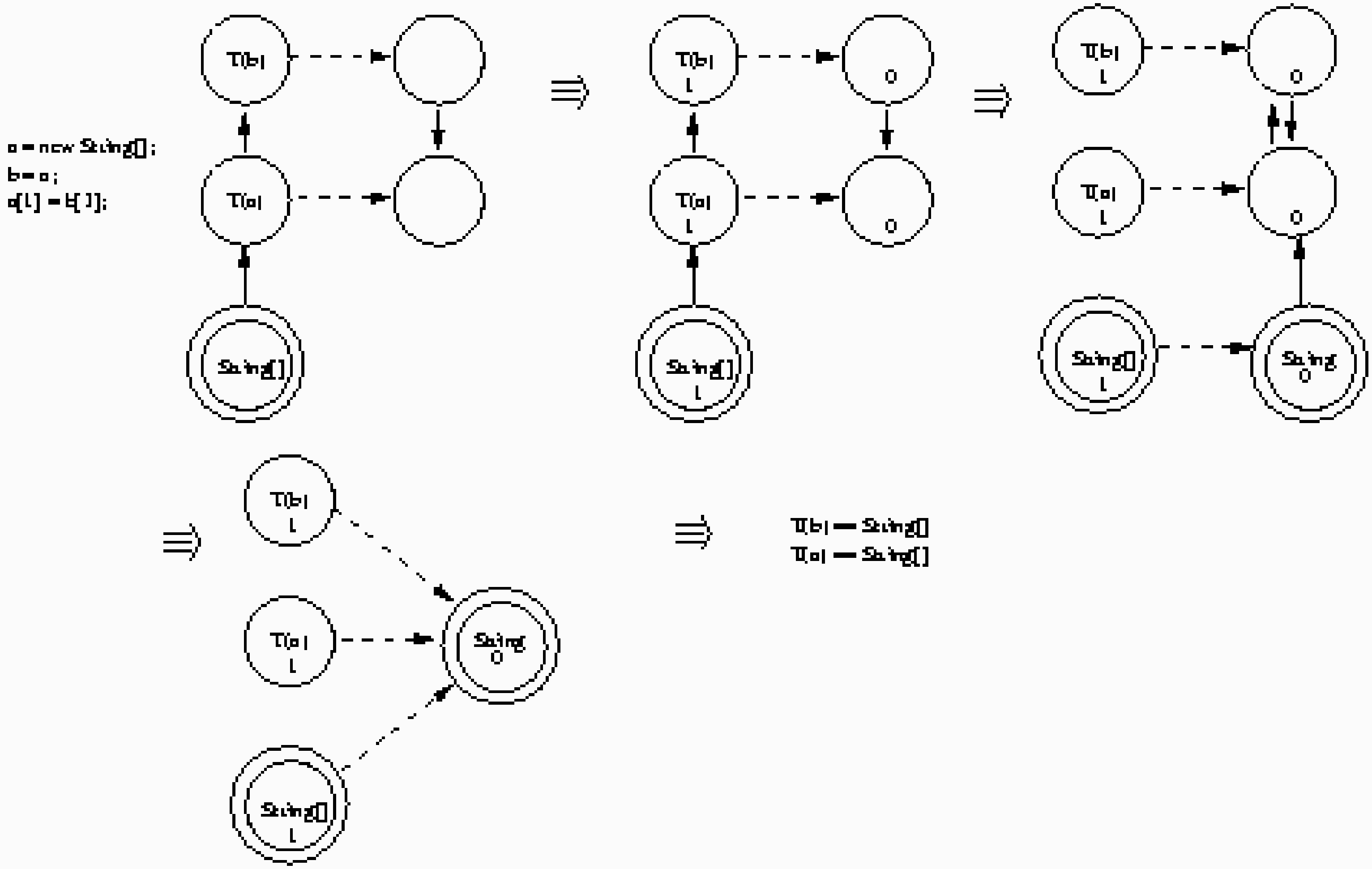


Figure 9: Solving Array Constraints

Program transformations

– Performed when there is no static type solution

- Type casts

  s3: ((IA)a).f();

  – makes fig. 3 program typeable
  – but adds run-time overhead

- Copy statements
  – introducing copy statements following new statements to take care of the common case of the creation of instances on 2 branches
  – well known techniques can get rid of extra copy statements(copy propagation)

```
void m() {
     <unknown> a;
     <unknown> b;
     <unknown> c;
     if (...)
       { a = new A();
 s1:     a.f(); // invokevirtual A.f()
         c = a; // Extra copy
       }
     else
       { b = new B();
 s2:     b.g(); // invokevirtual B.g()
         c = b; // Extra copy
       }
```

```
void m() {
     <unknown> a;
     if (...)
       { a = new A();
 s1:     a.f(); // invokevirtual A.f()
       }
     else
       { a = new B();
 s2:     a.g(); // invokevirtual B.g()
       }
 s3: a.toString();
       // invokevirtual Object.toString()
 }
```

(a) untyped method

```
class Object {
   public String toString() { .... }
   ... }

class A extends Object {
   public void f() { .... }
   ... }

class B extends Object {
  public void g() { .... }
  ... }
```

(b) hierarchy

Figure 2: Different types needed at different program points

Experimental results

Typing Java bytecode

Language	Benchmark	# methods	# transf.	conn. comp.	single cons.	exhaust.
java:	javac	1179	3	383	796	0
java:	jdk1.1	5060	14	2832	2228	0
ada:	kalman	735	10	473	262	0
eiffel:	compile_to_c	7621	0	1558	5959	0
ml:	lexgen	209	0	140	69	0
scheme:	boyer	2255	2	820	1439	0

Table 1: Required steps

Experimental results

Improving Class Hierarchy analysis

–receiver type more accurately determined

source language	program name	call-graph edges untyped Jimple (#)	call-graph edges typed Jimple (#)	Reduction (%)
java:	jack	10583	10228	3
java:	javac	26320	23625	10
java:	jimple	51350	33464	35
ada:	rudstone	8151	7806	4
eiffel:	illness	3966	3778	5
ml:	nucleic	5009	4820	4

Table 2: Call Graph reduction

Conclusion

- Static type inference algorithm for typing Java bytecode
- emphasized the difference between well-behaved and well-typed bytecode
- experimental results show how the algorithm improves the results of further analysis