Science of Computer Programming 78 (2013) 1809–1827

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Extensible intraprocedural flow analysis at the abstract syntax tree level

Emma Söderberg [a,*], Torbjörn Ekman [b], Görel Hedin [a], Eva Magnusson [a]

[a] *Department of Computer Science, Lund University, Lund, Sweden*
[b] *Semmle Limited, Oxford, United Kingdom*

ARTICLE INFO

Article history:
Received 11 March 2010
Received in revised form 5 December 2011
Accepted 3 February 2012
Available online 18 February 2012

Keywords:
Declarative
Dataflow
Analysis
Control-flow
Java
Compiler
Attribute grammars

ABSTRACT

We have developed a new approach for implementing precise intraprocedural control-flow and dataflow analyses at the abstract syntax tree level. Our approach is declarative, making use of reference attribute grammars augmented with circular attributes and collection attributes. This results in concise executable specifications of the analyses, allowing extensions both to the language and with further source code analyses.

To evaluate the new approach, we have implemented control flow, dataflow and dead assignment analyses for Java, by extending the JastAdd Extensible Java Compiler. We have compared our results to several well-known analysis frameworks and tools, using a set of Java programs as benchmarks. These results show that our approach performs well concerning both efficiency and preciseness.

1. Introduction

Control-flow and dataflow analyses are key elements in many static analyses, and useful for a variety of purposes, e.g., code optimization, refactoring, enforcing coding conventions, bug detection, and metrics. Often, such analyses are carried out on a normalized intermediate code representation, rather than on the abstract syntax tree (AST). This simplifies the computations by not having to deal with the full source language. However, doing these analyses directly at the AST level can be beneficial, since the high-level abstractions are not compiled away during the translation to intermediate code. This is particularly important for tools that are integrated in interactive development environments, such as refactoring tools and tools supporting bug detection and coding convention violations.

In this paper, we present a new approach for computing intra-procedural control-flow and dataflow at the AST level. Our approach is declarative, making use of attribute grammars. Advantages include compact specification and modular support for language extensions, while giving sufficient performance for practical use.

To make the approach work, we rely on a number of extensions to Knuth's original attribute grammars [18]: *Reference attributes* [15] allow the control-flow edges to be represented as references between nodes in the AST. *Higher-order attributes* [24] are used for reifying entry and exit nodes in the control-flow graph as objects in the AST. *Circular attributes* [13,20] are used for writing down mutually recursive equations for dataflow as attributes, automatically solved through fixed-point iteration. Finally, *collection attributes* [6,19], enable the simple specification of reverse relations, for example, computing the set of predecessors, given the set of successors. These mechanisms are all supported in the JastAdd system [11], which we have used to implement our approach.

[*] Corresponding author. Tel.: +46 462223075.
E-mail addresses: emma.soderberg@cs.lth.se (E. Söderberg), torbjorn@semmle.com (T. Ekman), gorel.hedin@cs.lth.se (G. Hedin), eva.magnusson@cs.lth.se (E. Magnusson).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.02.002

```
public Set<CFGNode> CFGNode.succ();
public Set<CFGNode> CFGNode.pred();

public CFGNode MethodDecl.entry();
public CFGNode MethodDecl.exit();
```

Fig. 1. The generated Java API for the control flow graph of a method.

As a case study, we have implemented control-flow graphs and dataflow analysis for Java by extending JastAddJ (the JastAdd Extensible Java Compiler) [12]. The control flow graph is precise: it is implemented at the expression level and covers non-trivial control flow including Java exception handling, taking exception types into account, and short-circuited boolean expressions. For dataflow, we have implemented both liveness analysis and reaching definition analysis. As an example of a tool-oriented analysis, we have implemented a detector of dead assignments to local variables.

The implementation is modular and extensible. Similar to the internal modularization of JastAddJ [12], each module can be viewed as an object-oriented framework, with a client API representing the result of the analysis, and an extension API for the attributes that need to be defined by a language extension module. In many cases, new language features can reuse the existing analyses as they are, but for language constructs affecting control-flow, rules need to be added. We exemplify this by considering the effect on the analyses when extending Java 1.4 to Java 5.

These are the main contributions of this paper:

- We present a new approach to implementing precise control-flow graphs at the AST level, using reference attribute grammars. An attribute framework for control-flow graphs is presented that allows the modular addition of language constructs, classified into non-directing, internal flow, and abruptly completing constructs. We furthermore provide attribute grammar solutions for specifying precise control flow of exceptions and short-circuiting of boolean expressions.
- We present how the control-flow framework can be modularly extended with liveness analysis and reaching definition analysis. These dataflow analyses are specified using circular attributes, resulting in declarative implementations very similar to textbook definitions.
- We have implemented control flow graphs and dataflow analysis using our approach for full Java 1.4 and with a modular extension to support Java 5. The implementation is available at the JastAdd site [17].
- We report performance and preciseness results of our approach by comparing it to three well known analysis frameworks and tools for Java: Soot [23], PMD [9], and FindBugs [3]. This is done by comparing the results from a dead assignment analysis (implemented on top of the dataflow analyses) on a set of benchmark Java programs from the DaCapo suite [4], the largest being 130 000 lines of code. Our results show that our approach present precise results on par with Soot, and provides better performance than the selected set of tools for almost all selected benchmarks.

The rest of this paper is structured as follows. The implementation of control-flow analysis is described in Section 2, and the dataflow analyses in Section 3. An application doing dead assignment analysis is given in Section 4, and Section 5 discusses how to extend the analysis when the source language is extended. Section 6 provides a performance evaluation of our method. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2. Control-flow analysis

In control-flow analysis, the goal is to build a control-flow graph (CFG) where nodes represent blocks of executable code, and successor edges link the blocks in their possible order of execution. The nodes typically correspond to basic blocks, i.e., linear sequences of program instructions with one entry and one exit point [1]. Each node n has a set of immediate successors, $succ(n)$, and a set of immediate predecessors, $pred(n)$, both of which can be empty.

2.1. Control-flow API

In JastAdd, a program is represented as an AST, with nodes that are objects with attributes. To represent the CFG, we superimpose it on the AST, treating statement and expression nodes as nodes in the CFG. We represent the *succ* and *pred* sets as attributes on an interface **`CFGNode`** implemented by expressions and statements. To represent the entry and exit points of a method, we add synthetic empty statements to the method declaration.

JastAdd builds on Java, and generates an ordinary Java API for the AST and its attributes. Fig. 1 shows the generated Java API for the CFG of a method. JastAdd specs can use this API to specify additional analyses, for example dataflow. The API can also be used by ordinary Java code, for example, an integrated development environment implemented in Java.

2.2. Language structure

Fig. 2 shows an example Java method and parts of its corresponding AST. We will use this as an example to illustrate how the control-flow graph is superimposed on the AST. To keep the example concise, we have omitted parameters and local declarations in the code.

```
void m() {
  if(c > 2)
    x = c;
  while(c < 10) {
    x += p();
    c++;
  }
}
```

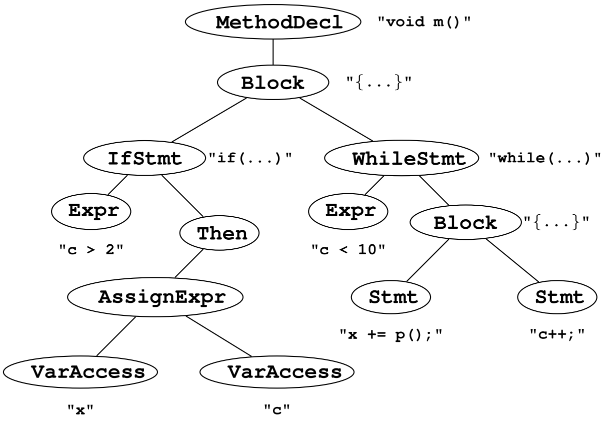

Fig. 2. Sample Java method and its abstract syntax tree.

```
MethodDecl ::= ParamDecl* Block;
ParamDecl  ::= <Type:String> <Name:String>;

abstract Stmt;
Block      : Stmt ::= Stmt*;
IfStmt     : Stmt ::= Expr Then:Stmt [Else:Stmt];
WhileStmt  : Stmt ::= Expr Stmt;
ExprStmt   : Stmt ::= Expr;
VarDecl    : Stmt ::= <Type:String> <Name:String> [Init:Expr];
ReturnStmt : Stmt ::= [Expr];
EmptyStmt  : Stmt;

abstract Expr;
AssignExpr : Expr ::= LValue:Expr RValue:Expr;
VarAccess  : Expr ::= <Name:String>;
MethodCall : Expr ::= <Name:String> Arg:Expr*;
```

Fig. 3. Simplified parts of the Java abstract grammar in Fig. 2.

A simplified part of the abstract grammar for Java is shown in Fig. 3. It is written in an object-oriented form with abstract classes `Stmt` and `Expr`, and subclasses for the individual statements and expressions such as `WhileStmt` and `VarAccess`.

The grammar uses a typical syntax with the Kleene star for list children, angle brackets for tokens, and square brackets for optional children. Children are either named after their types, such as a `Block` child of a `MethodDecl`, or with given names preceding the type name. For example, the left and right children of an `AssignExpr` are named `LValue` and `RValue`.

Certain constructs in Java can act as both expressions and statements, for example assignments and method calls. They are represented as expressions in the grammar, for example `AssignExpr`, and the class `ExprStmt` serves the purpose of adapting such expressions to serve as statements. The full grammar for Java is available at the JastAdd web site [17].

2.3. The control-flow graph

Fig. 4 shows how the AST has been attributed with successor edges and synthetic nodes, to form the CFG for the example method. The statement nodes constitute the nodes of the CFG, and reference attributes represent the successor edges. Two synthetic nodes are added to represent the entry and exit of the graph.

Some nodes can be viewed as explicitly transferring control, whereas others merely let the control flow through them. For example, the `AssignExpr` in Fig. 4 transfers the control first to its right-hand side (the read of `c`) and then to its left-hand side (the assignment to `x`). After that, it transfers control to some location decided by its context (to the `WhileStmt` in this case). For a `VarAccess`, the control simply flows through, transferring to a location decided by its context. Based on this observation, we distinguish between the following three categories of nodes.

Non-directing nodes which merely transfer control to the next node, as decided by their context. A `VarAccess` is an example of a node in this category.

Internal flow nodes which may transfer control to and between their children. Examples of nodes in this category are `Block`, `WhileStmt`, and `AssignExpr`.

Abruptly completing node which may transfer control to a specific location outside itself, in effect ending the execution of one or more enclosing nodes. Examples of such nodes in Java include *breaks*, *throws*, *returns* and method calls [14].

```
void m() {
  if(c > 2)
    x = c;
  while(c < 10) {
    x += p();
    c++;
  }
}
```

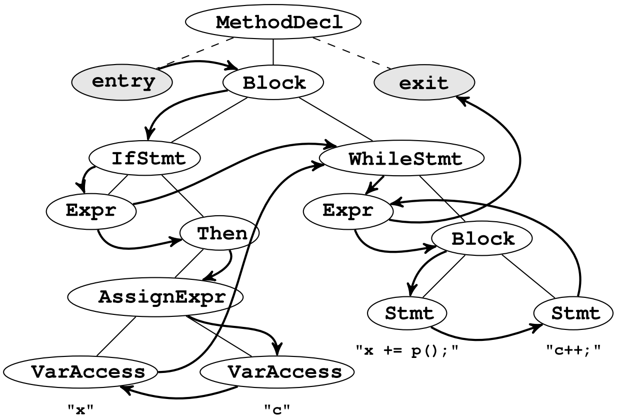

Fig. 4. Example method and its CFG, excluding control flow internal to omitted children. Successors are shown as directed edges. Synthetic nodes are gray and the dashed lines show parent–child relations to these nodes.

```
// The successor edges in the CFG
syn Set<CFGNode> CFGNode.succ();

// Nodes that follow a node, as seen from its context
inh Set<CFGNode> CFGNode.following();

// By default, they are the same.
eq CFGNode.succ() = CFGNode.following();

// The following node for conditional branches.  By default, these are empty
inh CFGNode.followingTrue();
inh CFGNode.followingFalse();
```

Fig. 5. The attribution framework for successors.

In the following subsections, we will discuss how the different parts of the CFG are specified, and how these different categories of nodes are handled.

2.4. The successor framework

Fig. 5 shows a small attribution framework for the successor edges. It specifies the behavior for non-directing nodes, and can be specialized to handle internal flow and abruptly completing nodes. The framework introduces four attributes: **succ**, **following**, **followingTrue** and **followingFalse**. The **succ** attribute is a set of references to nodes, and represents the successor edges in the CFG. The **following** attribute of a node *n*, is its set of successors as seen from its enclosing node, i.e., without any knowledge of the internal flow or possible abruptly completing nodes inside *n*.

The attributes **followingTrue** and **followingFalse** are used for handling control flow of short-circuited boolean expressions. For instance, in "`e1 && e2`", the evaluation of `e2` should be skipped if `e1` is *false*. If this boolean expression is enclosed in some other boolean expression or conditional construct, the place to skip to may be a different one from the ordinary following set. The attributes capture the appropriate place to skip to.

In the framework, **succ** is defined to be equal to **following**, thus capturing the behavior of non-directing nodes. Subclasses of **Expr** and **Stmt** can override this definition to cater for internal flow or abrupt completion.

The attribute **succ** is *synthesized*, whereas **following**, **followingTrue** and **followingFalse** are *inherited*.[1] The difference is that synthesized attributes must be defined in the node in which they are declared, whereas inherited attributes must be defined in an ancestor node. So, **succ** is defined by an equation in **CFGNode**, and can have overriding equations in subclasses of **Expr** and **Stmt**, similar to ordinary virtual methods. The attribute **following** of a node *n*, must instead be defined by one of the ancestor nodes of *n*. So to use this framework, equations must be provided that define the value of **following** for all possible nodes. The same applies to **followingTrue** and **followingFalse**.

The table in Fig. 6 shows how the CFG is achieved by extending the successor framework: For non-directing nodes, no additional equations are needed. For internal-flow nodes, the equation for **succ** needs to be overridden, and equations may need to be added for constituents' **following**, **followingTrue** and **followingFalse** attributes. For abruptly completing nodes, **succ** is overridden.

[1] Note that this use of the term *inherited* stems from Knuth [18] and is unrelated to and different from the object-oriented use of the term.

Node kind	Examples	`succ`	`following*`
Non-directing	variable	–	–
Internal-flow	block if assignment	direct flow to an internal node	possibly redefine for internal nodes

Abruptly completing	break
	return
	throw
		direct flow to a special location	–

Fig. 6. How different kinds of nodes extend the successor framework to achieve the control-flow graph.

```
eq Block.succ() =
   (getNumStmt() = 0) // no children
   ? following()
   : singleton(getStmt(0));

eq Block.getStmt(int i).following() =
   (i = getNumStmt()-1) // last child
   ? following()
   : singleton(getStmt(i+1));
```

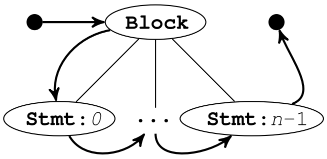

Fig. 7. Specializing the successor framework for **`Block`**.

```
eq IfStmt.succ() = singleton(getExpr());
eq IfStmt.getExpr().followingTrue() = singleton(getThen());
eq IfStmt.getExpr().followingFalse() = hasElse() ?
   singleton(getElse()) : following();
eq IfStmt.getExpr().following() =
   getExpr().followingTrue().union(getExpr().followingFalse());
```

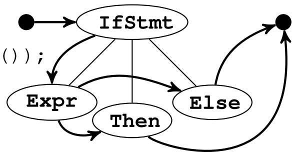

Fig. 8. Specializing the successor framework for **`IfStmt`**.

As an example of an internal-flow node, consider the **`Block`** whose CFG specification is shown in Fig. 7. To capture the internal flow, **`Block`** overrides the definition of its own **`succ`** attribute, transferring control to its first internal statement, if there is one. Since a block has a list of statement children, it must also define the value of **`following`** for each of these children. This is done by the equation **`Block.getStmt(int i).following = ...`** which applies to the **`i`**:th statement child of a block. For the last child, **`following`** is simply the same as for the block itself. For other children, **`following`** contains a reference to the next child in the block. The function **`singleton`** used in this definition returns a set containing a single given reference.

Another example of an internal-flow node is the **`IfStmt`**, whose CFG specification is shown in Fig. 8. The equation overriding **`succ`** states that control will be transferred to the **`Expr`** part (the condition). To allow boolean expressions in the condition to short-circuit to the correct branch, equations are given defining the **`followingTrue`** and **`followingFalse`** attributes. For normal (non-short-circuited) control flow, transfer is possible to both branches as defined by the equation for the **`following`** attribute.

Note that it is not necessary to define the **`following`** attribute for the **`Then`** and **`Else`** parts, since they should have the same value as **`following`** for the **`IfStmt`** itself, so the same equation in some ancestor applies to these parts.

Before we give examples of abruptly completing statements, we will introduce the framework for entry and exit nodes.

2.5. The entry and exit framework

To make sure there will always be well-defined entry and exit nodes, even for empty methods, we add two synthetic empty statements to each method. Nodes can be added declaratively to an AST by means of *higher-order attributes*, also known as *non-terminal attributes* (NTAs) [24]. An NTA is like a non-terminal in that it is a node in the AST. However, instead of being constructed as part of the initial AST, typically built by a parser, it is defined by an equation, just like an attribute. So in this sense, it is both an attribute and an AST node, hence the term higher-order. The right-hand side of an equation for an NTA must denote a *fresh* object, i.e. an object not already part of the AST, typically computed by a *new* expression.

Fig. 9 shows the attribution framework defining the entry and exit nodes. Since the method declaration is the parent of both the entry and exit nodes, as well as of the main block, it furthermore needs to define their **`following`** attributes. Naturally, the entry is followed by the main block, which is followed by the exit node, which in turn has no following

```
syn nta Stmt MethodDecl.entry() = new EmptyStmt();
syn nta Stmt MethodDecl.exit() = new EmptyStmt();

eq MethodDecl.entry().following() = singleton(getBlock());
eq MethodDecl.getBlock().following() = singleton(exit());
eq MethodDecl.exit().following() = empty();

inh Stmt Stmt.exit();
eq MethodDecl.getBlock().exit() = exit();
eq MethodDecl.entry().exit() = exit();
eq MethodDecl.exit().exit() = exit();
```

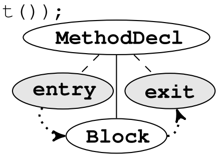

Fig. 9. Attribution framework for entry and exit nodes. Dotted directed edges indicate elements in the `following` sets.

```
eq ReturnStmt.succ() = exit();
```

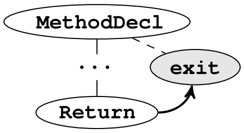

Fig. 10. Using the entry and exit framework to abruptly transfer control from return statements to the end of the method. (Simplified definition that ignores Java exceptions.)

```
try {
  return;
} finally {
  n();
}
```

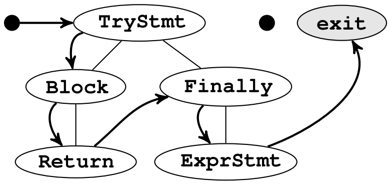

Fig. 11. The control flow from a `return`, in the presence of a `finally` block.

statements, as specified in the equations. The function `empty`, used when defining `following` for the exit node, simply returns the empty set.

The framework additionally defines an inherited attribute `exit` which gives all nodes access to the `exit` node. This is useful for abruptly completing nodes which need to transfer control directly to the `exit` node.

As a simple example of an abruptly completing node, consider the `return` statement. Fig. 10 shows how it directs the control flow directly to the `exit` node by overriding the `succ` attribute. This definition is simplified, however, and does not take Java exception handling into account. A full treatment of these issues is given in the next section.

2.6. Handling Java exceptions

The Java statements `break`, `throw`, `continue` and `return` are abruptly completing nodes, transferring control to a specific location outside of themselves.

The successor of an abrupt node is called the *target* node. For example, the target of a `return` statement is normally the `exit` node, as was shown in Fig. 10. However, if the abrupt node is inside the `try` block of a Java exception handler with a `finally` block, the `finally` block will intercept control before transferring control to the normal target(s). Fig. 11 shows an example.

In a similar way, other abrupt nodes also have a normal target to which control is transferred if there are no enclosing `try` statements with `finally` blocks. For `throw` it is a matching `catch`, or the `exit` node. For `break` the normal target is the statement following a matching enclosing loop or labeled statement. For `continue` the normal target is the first part of a matching enclosing loop. Fig. 12 shows example normal control-flow (without `finally` blocks).

We will now show how control flow of abrupt nodes is handled in the presence of `finally` blocks. As an example, we will take a closer look at the `break` statement. The other abrupt nodes are handled in an analogous way. We introduce an inherited attribute `breakTarget`, returning a singleton set with the matching target, or the empty set if no target is found (corresponding to a compile-time error). For the `break` statement, this attribute will be the true successor, i.e., either the normal target (e.g., a while loop), or a `finally` block.

The attribute `breakTarget` is also defined for the `try` statement, by which the `finally` block can find its successor, i.e., usually the normal target. This solution works also for nested `try` statements with `finally` blocks, in which case control is transferred from the `break` statement, through all the `finally` blocks of enclosing `try` statements, and finally to the normal target.

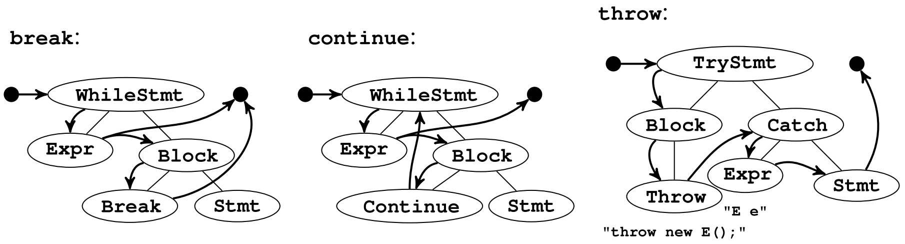

Fig. 12. Control flow for some abrupt nodes.

```
eq BreakStmt.succ() = breakTarget(this);

inh Set BreakStmt.breakTarget(BreakStmt stmt);
inh Set TryStmt.breakTarget(BreakStmt stmt);

// Equations for breakTarget
eq BodyDecl.getChild().breakTarget(BreakStmt stmt) = empty();
eq BranchTargetStmt.getChild().breakTarget(BreakStmt stmt) =
    targetOf(stmt)
    ? following()
    : breakTarget(stmt);
eq TryStmt.getBlock().breakTarget(BreakStmt stmt) =
    hasFinally()
    ? singleton(getFinally())
    : breakTarget(stmt);
```

Fig. 13. Specializing the successor framework for **BreakStmt**. The **targetOf** attribute is defined in the compiler frontend.

```
{
    while (..) {
        try {
            break;
        } finally {
            n();
        }
    }
    m();
}
```

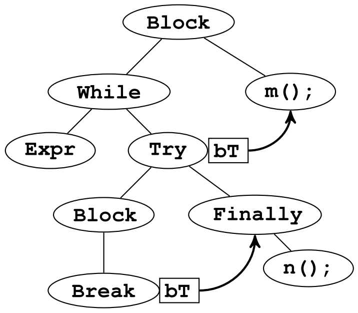

Fig. 14. Values of the **breakTarget** attribute (**bT**).

The **breakTarget** attribute is parameterized by the **BreakStmt** to allow the target for the correct **BreakStmt** to be found. This attribution solution, using parameterized inherited attributes, is similar to the JastAdd implementation of Java name analysis, as presented in [10].

The successor of a **BreakStmt** is now simply defined as the **breakTarget** of itself. Fig. 13 shows the specification. There are several equations defining **breakTarget**, and if there is more than one in a chain of ancestors, the closest equation applies. Therefore, if a **BreakStmt** is enclosed by a **TryStmt**, and then by a **BranchTargetStmt** (e.g., a while loop), the equation in the **TryStmt** will hold. If the **BreakStmt** is not enclosed by any of these kinds of statements, the equation defined in **BodyDecl** will hold, defining the target to be the empty set. To illustrate how this works, consider Fig. 14, showing the values of **breakTarget** for an example program.

To handle the remaining abrupt statements, **continue**, **return**, and **throw**, we define one target attribute for each of them and use them in a similar fashion. With this approach we end up with potentially several abrupt nodes transferring control to the **finally** block. The potential successors of the **finally** block is thus the set of normal targets for all these intercepted abrupt nodes. For this reason, we introduce an attribute **interceptedAbruptNodes** which contains references to these nodes. Given this attribute, the **TryStmt** can define the **following** attribute for its **finally** block, as shown in

```
eq TryStmt.getFinally().following() {
    Set flw =
      (getFinally().canCompleteNormally())
      ? following()
      : empty();
    for (Stmt abrupt : interceptedAbruptStmts) {
      flw = flw.union(abrupt.targetAt(this));
    }
    return flw;
}

syn Set Stmt.targetAt(TryStmt t) = empty();
eq BreakStmt.targetAt(TryStmt t) = t.breakTarget(this);
eq ContinueStmt.targetAt(TryStmt t) = t.continueTarget(this);
...
```

Fig. 15. Specializing the successor framework for **TryStmt**.

```
coll Set CFGNode.pred() [empty()] with add;                  // (1)
Stmt contributes this to CFGNode.pred() for each succ();     // (2)
Expr contributes this to CFGNode.pred() for each succ();     // (3)
```

Fig. 16. Using a collection attribute to define the predecessors.

Fig. 15. Here, the attribute **targetAt** uses the double dispatch pattern [16] to let each kind of abrupt node decide how to compute its target.[2]

Handling unchecked exceptions

In addition to explicitly thrown exceptions, using the **throw** statement, exceptions can be thrown implicitly by the runtime system at runtime errors such as null pointer dereferencing, division by zero, out of memory, etc. Unless these errors are caught, they are propagated back to the calling method, making also method calls a source of such implicit exceptions. So in this sense, more or less every expression and statement can have abrupt completion. Instead of adding explicit successor edges for all these possible control paths, we define an inherited attribute **uncheckedExceptionTarget** for **Expr** and **Stmt** nodes, and in that way make all nodes aware of these potential successors. By default, this attribute is a set containing the **exit** node. But if there are **catch** clauses that match **RuntimeException** or **Error**, these clauses are also added.

This approach is inspired by the factored control-flow graph explained in [8] where unchecked exception branches are summarized at the end of basic blocks to limit the number of branches.

2.7. Predecessors

To complete the implementation of the control-flow API, we now define the set of predecessors. This is simply the inverse of the successor relation, so if there is a successor edge from *a* to *b*, there will be a predecessor edge from *b* to *a*. Such inverse relations are easily defined using *collection* attributes [6,19]. The attributes we have seen so far have been defined using an equation located in an AST node. A collection, in contrast, is an attribute whose value is defined by the combination of a number of *contributions*, distributed over the AST. This way, we can define the predecessor sets by letting each node contribute itself to the predecessor sets of its successors. Fig. 16 shows the JastAdd specification.

Rule (1) is the declaration of the collection attribute **pred** for **CFGNode**. The rule states the type of the attribute (**Set**), the initial value (the empty set), and the operation used to add contributions (add). For correct evaluation, it is assumed that the operation is commutative, i.e., that the order of adding the contributions is irrelevant, which is indeed the case for the add method for the Java class **Set**.

Rules (2) and (3) declare that each **Stmt** and **Expr** node contributes itself (**this**) to the **pred** attribute of each of its successors. A more detailed presentation of collection attributes and their evaluation in JastAdd is available in [19].

3. Dataflow analysis

We want to analyze dataflow on the control-flow graph defined in the previous section. Two typical examples of dataflow analyses are liveness analysis and reaching definition analysis. We describe our implementation of these analyses using JastAdd in the following two subsections.

[2] The equation for `following` uses an assignment and a for loop which might be surprising since our approach is declarative. However, because we use Java method body syntax to define attribute values, it is natural to use imperative code here. This is perfectly in agreement with the declarative approach as long as that code has no net side effects, i.e., only local variables are modified.

```
public boolean VarAccess.isDest();
public boolean VarAccess.isSource();
public Decl VarAccess.decl();
```

Fig. 17. JastAddJ API used by liveness analysis.

```
// def
syn Set<Decl> CFGNode.def();
eq Stmt.def() = empty();
eq Expr.def() = empty();
eq VarAccess.def() = isDest() ? singleton(decl()) : empty();
eq VarDecl.def() = singleton(this);
eq ParamDecl.def() = singleton(this);

// use
syn Set<Decl> CFGNode.use() = empty();
eq VarAccess.use() = isSource() ? singleton(decl()) : empty();
```

Fig. 18. Implementation of *def* and *use* for liveness analysis.

3.1. Liveness analysis

A variable is *live* at a certain point in the program, if its assigned value will be used by successors in the control-flow graph. If a variable is assigned a new value before an old value has been used, the old assignment to the variable is unnecessary, also called *dead*.

We express liveness in the same fashion as Appel in [2] using four sets — *in*, *out*, *def* and *use*. The *def* set of a node *n* contains the variables assigned a value in *n*, and the *use* set contains the variables whose values are used in *n*. From these two sets we calculate the *in* and *out* sets, i.e., variables live into a node and variables live out of a node, using the following equations:

Definition 3.1. Let *n* be a node and *succ*[*n*] the value of the **succ** attribute for the node *n*:

$$\begin{aligned} in[n] &= use[n] \cup (out[n] \setminus def[n]) \\ out[n] &= \bigcup_{s \in succ[n]} in[s] \end{aligned}$$

We note that the equations for the *in* and the *out* sets are recursive and mutually dependent, i.e. they have a circular dependency to each other. Equations like these are usually solved by iteration until a fixpoint is reached, which is guaranteed if all intermediate values can be organized in a finite height lattice and all operations are monotonic on that lattice. We will explain how circular equations like these can be implemented as circular attributes in JastAdd [20].

3.1.1. The use and def sets

The main challenge in computing the *use* set for each node, is to support all kinds of statements and expressions in the source language. A complex language such as Java has more than 20 statements and 50 expressions. Fortunately, it is quite easy to support all these constructs in JastAddJ (the JastAdd Extensible Java Compiler), since each expression that accesses a local variable encapsulates a **VarAccess** node performing the actual binding. Moreover, each **VarAccess** node has two **boolean** attributes, **isDest** and **isSource**, determining whether the access acts as a definition (*l-value*) or use (*r-value*). Some nodes actually act as both. For example, a **VarAccess** that is the child of the post increment operator '**++**', will both read from and write to the variable. JastAddJ also defines an attribute **decl** for **VarAccess** nodes, referring to the appropriate declaration node. Fig. 17 summarizes the JastAdd API used.

In the liveness analysis, we represent *use* and *def* as sets of references to declaration nodes in the AST. We implement them using synthesized attributes, and let **VarAccess** nodes add themselves to the appropriate collection, depending on their role as an *r-value* and/or *l-value*. The variable, parameter and field declarations are also viewed as assignments, so they contribute themselves to their own *def* set. Fig. 18 shows the implementation of these attributes.

These two attributes effectively compute the *use* and *def* sets for all intraprocedural control-flow nodes in Java. If we add a new language construct that modifies a local variable we need only make sure it encapsulates a **VarAccess** and provide equations for the inherited attributes **isDest** and **isSource**, which are needed elsewhere in the frontend anyway, and the *use* set and *def* set attributes are still valid.

3.1.2. The in and out sets for liveness

The equations for the *in* set and *out* set in Definition 3.1 are mutually dependent. As mentioned earlier, such equations can be solved by iteration as long as the values form a finite height lattice and all functions are monotonic. This is clearly the

```
// in
syn Set<Decl> CFGNode.live_in() circular [empty()] =
    use().union(live_out().compl(def()));

// out
syn Set<Decl> CFGNode.live_out() circular [empty()] {
    Set<Decl> set = empty();
    for(Stmt s : succ()) {
        set = set.union(s.live_in());
    }
    return set;
}
```

Fig. 19. Implementation of liveness *in* and *out* sets, using circular attributes.

```
coll Set<Decl> CFGNode.live_out() circular [empty()] with add;
Stmt contributes live_in() to CFGNode.live_out() for each pred();
Expr contributes live_in() to CFGNode.live_out() for each pred();
```

Fig. 20. Alternative implementation of the *out* set, using a circular collection.

case for our equations since the power set of the set of local variables, ordered by inclusion, forms a finite lattice, with the empty set as bottom, on which union is monotonic. A fixpoint will thus be reached if we start with the bottom value and iteratively apply the equations as assignments until no values change.

JastAdd has explicit support for fixpoint iteration through circular attributes, as described in [20]. If we declare an attribute as circular and provide a bottom value, then the attribute evaluator will perform the fixpoint computation automatically. This allows us to implement the *in* and *out* sets using circular attributes, resulting in a specification very close to the textbook definition, as shown in Fig. 19.

In our actual implementation, we use an even more concise specification of the *out* set by defining it as a collection attribute, reversing the direction of the computation by making use of the predecessors instead of the successors. See Fig. 20.

An alternative to using circular attributes would be to manually implement the fixpoint computation imperatively. Such a solution requires manual book keeping to keep track of change, which significantly increases the size of the implementation and the essence of the algorithm gets tangled with book keeping code. Also, it is necessary to either statically approximate the sets of attributes involved in the cycle to iterate over, or to manually keep track of such dependences dynamically. This is all taken care of automatically by the attribute evaluation engine in JastAdd when using circular attributes.

3.2. Reaching definition analysis

In computing *reaching definitions*, we are interested in sets of *definitions* (assignments), rather than in sets of variable declarations. Because definitions may occur in several different syntactic constructs, not just in assignment statements, we define an interface **`Definition`** to abstract over the relevant AST classes, namely **`VarAccess`**, **`VarDecl`**, and **`ParamDecl`**. Not all variable accesses are definitions, but the **`isDest`** attribute can be used to decide this.

A definition of a variable is said to *reach* a use of a variable if there is a path in the control-flow graph from the definition to the use. A variable use may be reached by more than one variable definition in which case the actual value of the variable cannot be decided statically. For cases where there is only one reaching definition the use might be replaceable with a constant, a property typically used in, for example, constant propagation.

We define five sets — *defs*, *gen*, *kill*, *in* and *out*, in the same fashion as Appel [2]. The *defs* set of a variable declaration v contains all definitions of that variable. The *gen* set of a node n contains the definitions in n, i.e., corresponding to the new variable values generated by that node. The *kill* set of a node n is the set of definitions killed by definitions made in n. Consider a definition d of a certain variable v. The *kill* set for a definition d is the *defs* for v, minus the definition d itself, see Definition 3.2. The *kill* set for a statement is simply the union of the *kill* sets of its *gen* set.

The *in* set of a node n is the set of definitions that reach the beginning of n, and *out* is the set that reaches the end of n. Given the *kill* and *gen* sets, *in* and *out* are defined as shown in Definition 3.3. Note that the equations for *in* and *out* are recursive and mutually dependent, hence requiring a fixpoint iteration for evaluation.

Definition 3.2. Let d be a definition of a variable v:

$$d : v \leftarrow \ldots : \mathit{kill}[d] = \mathit{defs}[v] \setminus \{d\}$$

Definition 3.3. Let n be a node and $\mathit{pred}[n]$ the value of the **`pred`** attribute for the node n:

$$\begin{aligned} \mathit{in}[n] &= \bigcup_{p \in \mathit{pred}[n]} \mathit{out}[p] \\ \mathit{out}[n] &= \mathit{gen}[n] \cup (\mathit{in}[n] \setminus \mathit{kill}[n]) \end{aligned}$$

```
coll Set<Definition> Variable.defs() [empty()] with add;
VarAccess contributes this
    when isDest() to Variable.defs() for decl();
VarDecl contributes this to Variable.defs() for this;
ParDecl contributes this to Variable.defs() for this;
```

Fig. 21. Implementation of *defs* using attributes.

```
// gen
syn Set<Definition> CFGNode.gen();
eq Stmt.gen() = empty();
eq Expr.gen() = empty();
eq VarAccess.gen() = isDest() ? singleton(this) : empty();
eq VarDecl.gen() = singleton(this);
eq ParamDecl.gen() = singleton(this);

// kill
syn Set<Definition> CFGNode.kill();
eq Stmt.kill() = empty();
eq Expr.kill() = empty();
eq VarAccess.kill() = isDest() ? defs().compl(this) : empty();
eq VariableDeclaration.reaching_kill() = defs().compl(this);
eq ParameterDeclaration.reaching_kill() = defs().compl(this);
```

Fig. 22. Implementation of *gen* and *kill*.

```
// out
syn Set<Definition> CFGNode.reach_out() circular [empty()];
eq CFGNode.reach_out() = gen().union(reach_in().compl(kill()));

// in
coll Set<Definition> CFGNode.reach_in() circular [empty()] with add;
Stmt contributes reach_out() to CFGNode.reach_in() for each succ();
Expr contributes reach_out() to CFGNode.reach_in() for each succ();
ParamDecl contributes reach_out() to CFGNode.reach_in() for each succ();
```

Fig. 23. Implementation of the *in* and *out* sets for reaching definitions.

3.2.1. The defs set

To implement the *defs* set, we use a collection attribute on **`Variable`**, which is an interface implemented by **`VarDecl`** and **`ParamDecl`**. We then let the definitions contribute themselves to their declaration. Contributing **`VarAccess`** nodes check that they are actually acting as definitions using the attribute **`isDest`**. The implementation is shown in Fig. 21.

3.2.2. The gen and kill sets

The *gen* set of a node contains all the definitions inside the node. We use a synthesized attribute to implement this set and let variable declarations, parameter declarations and **`VarAccess`** nodes, that serve as definitions, contribute themselves to their own *gen*. The *kill* set is implemented using the same strategy, see Fig. 22.

3.2.3. The in and out sets for reaching definitions

In Definition 3.3 the sets *in* and *out* are defined as two mutually dependent equations using the *kill* and *gen* sets. Again we use circular attributes, obtaining an implementation very similar to the textbook definition of these sets. See Fig. 23.

4. Dead assignment analysis

To evaluate the efficiency and scalability of our approach, we have implemented a simple intraprocedural analysis for Java which detects dead assignments. In more detail, we locate assignments whose values are not used later in a body declaration, i.e., in a method, constructor, instance initializer, static initializer, or field declaration. We only include assignments to local non-constant variables and parameters in the analysis:

```
syn lazy boolean CFGNode.includeInDeadAssignAnalysis() = false;
eq VarAccess.includeInDeadAssignAnalysis() =
    isDest() && isLocalStore();
```

```
eq VarDecl.includeInDeadAssignAnalysis() =
    hasInit() && isLocalVariable() && !isConstant();
```

We try out two versions on this selection: one based on liveness analysis, and one combining liveness analysis with analysis of reaching definitions.

4.1. Collecting dead assignments

To collect all dead assignments of a compilation unit, we add a collection (**coll**) attribute **deadAssignments** to the **CompilationUnit** class. This class represents a file with one or more classes which might contain one or more body declarations (methods, constructors etc.):

```
coll Set<Stmt> CompilationUnit.deadAssignments() [empty()] with add;
```

The **CompilationUnit** class is connected to the grammar in Fig. 3 as follows (here, only including methods):

```
CompilationUnit ::= ClassDecl*;
ClassDecl       ::= MethodDecl*;
MethodDecl      ::= ...
```

Dead assignments contribute themselves to the collection of their enclosing **CompilationUnit** using a **contributes** clause. The reference to the **CompilationUnit** node is propagated to descending statement nodes using an inherited attribute **enclosingCompilationUnit**:

```
VarAccess contributes this
    when includeInDeadAssignAnalysis() && isDeadAssign()
 to CompilationUnit.deadAssignments()
    for enclosingCompilationUnit();
VarDecl contributes this
    when includeInDeadAssignAnalysis() && isDeadAssign()
    to CompilationUnit.deadAssignments()
    for enclosingCompilationUnit();
```

Each of these nodes, **VarAccess** and **VarDecl**, contribute to the collection, if they are included in the selection of the analysis, and their **isDeadAssign** attribute is true. We define this attribute to be false by default for all control flow nodes:

```
syn boolean CFGNode.isDeadAssign() = false;
```

4.2. Analyzing using liveness

Definition 4.1. If a variable is defined, but not live immediately after the node, the assignment is considered dead in the sense that the assignment is unnecessary. That is, an assignment *a* is dead when:

$$kill[a] \neq \emptyset \wedge kill[a] \cap out[a] = \emptyset$$

Using liveness analysis, we can define an assignment to be dead when a defined variable is not live after the assignment, as defined in Definition 4.1. With this in mind, we can define a very useful attribute **isDead** which we can use to define equations for **isDeadAssign** as follows:

```
syn lazy boolean CFGNode.isDead();
eq CFGNode.isDead() = !def().compl(liveness_out()).isEmpty();

eq VarAccess.isDeadAssign() = isDead();
eq VarDecl.isDeadAssign() = isDead();
```

4.3. Analyzing using liveness and reaching definition

We can combined liveness analysis with reaching definition analysis, by adding a condition to the equations of the **isDeadAssign** attribute, as follows:

```
eq VarAccess.isDeadAssign() = isDead() || allReachedUsesAreDead();
eq VarDecl.isDeadAssign() = isDead() || allReachedUsesAreDead();
```

The consequence of combining these two analyses, is that we can find additional dead assignments on the form:

```
a = 0; // Also dead because b is dead (the reached use)
b = a; // b is dead
```

Here, the assignment to a is dead because the assignment to b is dead, which is the only reached use of a. To get this behavior, we need to define the attribute **allReachedUsesAreDead**, which investigates whether all reached uses are dead:

```
syn boolean ReachingDef.allReachedUsesAreDead() circular [false];
eq Stmt.allReachedUsesAreDead() {
 for (ReachedUse use : reachedUses())
  if (!use.inDeadAssign())
   return false;
 return true;
}
```

The **reachedUses** attribute is defined on an interface **ReachingDef**, implemented by nodes defining values, and it returns a set of reached uses, implementing an interface **ReachedUse**. Nodes implementing the **ReachedUse** interface has an additional attribute **inDeadAssign** returning true if the use is in the right-hand side of an assignment that is dead:

```
inh boolean ReachedUse.inDeadAssign();
eq VarDecl.getInit().inDeadAssign() = isDead();
eq AssignExpr.getSource().inDeadAssign() =
    getDest().isLocalStore() && getDest().isDead();
eq Program.getChild().inDeadAssign() = false; // default value
```

It might be the case that an assignment that is dead has, for instance, a method call on its right-hand side, but we do not want to consider variables given to the method as dead. To avoid cases like these, we can add an equation to, for example, a method call as follows:

```
eq MethodAccess.getArg(int i).inDeadAssign() = false;
```

5. Language extensions

The previous examples have illustrated how the control-flow specification for individual statements can be written modularly. Similarly, the control-flow implementation for Java 1.4 can be extended modularly to support Java 1.5. The only new language constructs that affect the CFG are the new enhanced **for** statement and **enum** constant, which is a new kind of body declaration. As an example we will considering the enhanced **for** statement in more detail, which has the following abstract syntax:

```
EnhancedFor : BranchTargetStmt ::= VarDecl Expr Stmt;
```

This statement iterates over the elements in the iterable object denoted by **Expr**. In each iteration, a new element is assigned to **VarDecl**, and the **Stmt** is executed. To capture this flow, we let the **EnhancedFor** itself represent the initialization of the iterator. We provide equations defining the **succ** attribute for **EnhancedFor** and the **following** attributes of its constituents. Fig. 24 shows the specification.

Note that since the analyses of liveness, reaching definitions, and dead assignments are defined in terms of the control-flow graph, they will work automatically also for these new constructs.

6. Evaluation

To evaluate our approach, we have run the dead assignment analysis on a set of Java benchmark applications, and compared the results and performance to other analysis tools. We have also measured the size of our specification modules in order to evaluate development effort, and compared them to another tool.

```
eq EnhancedForStmt.succ() = singleton(getExpr());
eq EnhancedForStmt.getExpr().followingTrue() =
    singleton(getVarDecl());
eq EnhancedForStmt.getExpr().followingFalse() = following();
eq EnhancedForStmt.getExpr().following() =
    getExpr().followingTrue().union(
        getExpr().followingFalse());
eq EnhancedForStmt.getVarDecl().following()=
    singleton(getStmt());
eq EnhancedForStmt.getStmt().following()=
    singleton(getExpr());
```

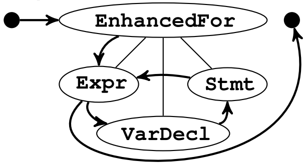

Fig. 24. Control flow for `EnhancedFor`.

Name	Version	Lines of Code	Candidates	# Flows	Avg. Flow Size
ANTLR	2.7.7	37 730	3 826	3 332	47.0
Bloat	1.0	38 581	5 740	5 095	136.0
Chart	1.0	9 968	1 818	1 469	39.0
FOP	0.95	130 300	18 203	19 632	110.0

Fig. 25. Java benchmarks. *Candidates* are the number of local variable declarations and assignments in an application. The last two columns show the number of intraprocedural flows (methods etc.) in an application and the average flow size, i.e. the average number of nodes in a flow.

6.1. Setup

6.1.1. Selection of benchmarks

For evaluating our analyses, we have selected four Java applications of varying size from the DaCapo benchmark suite [4]: **ANTLR**, **Bloat**, **Chart** and **Apache FOP**. ANTLR is a parser and translator generator, Bloat is a byte-code level optimization and analysis tool, Chart is a charting utility tool and Apache FOP is a print formatting tool. Fig. 25 gives an overview of the selected benchmarks with regard to size (lines of code), number of flows (methods, instance initializers etc.), and average size of these flows (number of nodes in the control-flow graph). ANTLR and Bloat are of similar size, but we include both because they differ substantially in their average flow size.

The figure also shows the number of possible dead assignments, or *candidates*, in each application. For a node to be a candidate it needs to be either a variable declaration with an initializing assignment, or an assign expression. For reason of comparison, we exclude constants of primitive types (integer, double etc.) and strings from the set of candidates. Constants like these may be removed by default by some analysis tools, excluding them from the dead assignment analysis that we want to compare to.

6.1.2. Selection of analysis tools

We compare our JastAdd-based analysis results to those of **Soot** (2.4.0), **FindBugs** (1.3.9) and **PMD** (4.3.5). Soot is a very well known Java optimization framework, working at the byte code level [23]. It is interesting for comparison as it can be expected to have very high precision and correctness. FindBugs [3] and PMD [9] are two well known tools for detection of bugs and anomalies in Java source code. They are interesting for us to compare to since they exemplify the developer-oriented tools we have in mind for our AST-based analysis. FindBugs performs the analysis on byte code, whereas PMD analyzes the source code directly.

All these tools support a number of different analyses, but for our comparison we are only interested in dead assignment analysis. In order to get these results from each tool we have used the following configurations:

Soot The Soot framework is made up by a set of phases, each connected to a certain kind of analysis. For example, there is a phase called `jb` which translates input to a three-address code called *jimple*, and there are phases for whole program analysis, for example, `cg`, `wjtp`, `wjop`. We are interested in the intra-procedural analyses found in a phase called `jop`. So we disable all other phases, except for the `jb` phase. Inside a phase there are several packs, one for each analysis. For the `jop` phase, we are only interested in the `jop.dae` pack, performing dead assignment elimination, and hence we disable all other packs in the `jop` phase.

We want to easily find which assignments that Soot wants to eliminate. With this in mind, we have added a flag `-only-tag` to the `jop.dae` pack, which causes the analysis to tag an assignment rather than removing it. This way, we can print out the jimple code and find which assignments are detected as dead.

Since Soot operates on a jimple representation it might find a lot of dead assignments to temporaries in its own representation which do not correspond to assignments in the source. To partly deal with this issue we only consider assignments on the line of a source assignment, i.e., on the line of a candidate. However, given that one source assignment may be represented on several lines in Soot, it is still possible that Soot will remove an assignment but not the actual source assignment. For cases like these, i.e., where Soot does not remove all jimple lines of a candidate, we do a manual check.

FindBugs FindBugs performs a number of identifications of so called *bug patterns*, i.e., patterns in the code possibly corresponding to a bug. One such bug pattern identifies dead local stores (DLS), that is, dead assignments. We have configured FindBugs to only include the DLS pattern in its analysis. The results are given on a file and source line basis which makes it easy for us to map the result to candidates in a benchmark. To get as good precision as possible and to find all pattern matches for DLS we run FindBugs with the `-effort:max` and `-low` flags.

PMD PMD supports the definition of rules using Java or XPath, but also provides a default set of rules. One such rule set looks for so called dataflow anomalies of three kinds, and two of these locate dead assignments — DU and DD. DD by identifying when a variable is assigned twice in a row with out a use in between, and DU by identifying if an assigned value is not used in the scope it is defined. The third finds undefined variables (UR) which is not interesting for our comparison. Results are obtained on a file and line basis which makes it easy for us to map the results to candidates in a benchmark.

6.1.3. Comparison of results

In order to compare the results of different tools we need a unified way to identify which assignments that are found to be dead. To accomplish this we pretty-print the source code of each benchmark and let each assignment start on a new line. This way we can identify a candidate by file name and source line.

In the case where the analysis is not performed on source code, we may need to maintain a mapping to source. For Soot we maintain a mapping between each source line and its corresponding jimple lines, to know if an assignment has been found completely dead or partially dead. In the case with FindBugs, which analyses bytecode, the result includes information of source lines and no extra mapping is required.

6.1.4. Performance measurement

All performance measurements have been performed on a Lenovo Thinkpad X61 running Ubuntu 10.10 (Maverick Meerkat). For comparison between tools, we use the average time of 10 runs from a terminal, measuring execution time with the Unix command `time`. For JastAdd, we also provide performance measurements using the multi-iteration approach with a pre-heated VM, as presented in [5].

6.2. Correctness and precision

Fig. 26 shows the number of dead candidate assignments found by each tool. The results are grouped into four subfigures, one for each Java benchmark.

For JastAdd we only include the results for JA$_{live}$, i.e., the dead assignment analysis only using liveness. The results for JA$_{live+reaching}$ are slightly more precise, but at a substantial additional cost in execution time. JA$_{live+reaching}$ only identified an additional three cases, one in ANTLR and two in Chart, all on the form:

```
s = s + a; // dead in JA_live
s = b; // also dead in JA_live+reaching
```

None of the other tools found any of these cases.

JastAdd and Soot both find very similar numbers of dead assignments among the selected candidates. JastAdd finds a few dead assignments that Soot does not find, and we have manually verified that they are indeed dead. Soot also finds a few dead assignments that JastAdd does not find. We have looked at each of these manually. One of these cases correspond to a true dead assignment at the source level:

```
int a = 0;
while (expr) {
    a++; // dead in Soot but not in JastAdd
}
```

Here, `a` is kept alive in the JastAdd analysis, while not in Soot.

In the other cases where Soot identifies dead candidate assignments, and JastAdd not, it is actually not the source level assignment that is detected, but assignments to temporary variables introduced in the jimple code. These do thus not correspond to dead assignments at the source level.

There are some cases where both Soot and JastAdd have identified a dead assignment, but Soot has only removed some of the corresponding jimple lines. This is because the right-hand side is a construct that might have side effects, typically a method call, and the call is therefore still present. The behavior for these cases is equivalent for JastAdd and Soot, but we needed to look at the jimple code manually to determine this.

Dead Assignments Found (#)

Tool: A,B	only A	both	only B
JA, Soot	8	308 (22)	3
JA, PMD	57	259	658
JA, FB	278	38	0
Soot, PMD	57	254 (21)	663
Soot, FB	276	35 (21)	3
PMD, FB	885	32	6

(a) **Results for ANTLR**

Tool: A,B	only A	both	only B
JA, Soot	8	78 (26)	3
JA, PMD	32	54	466
JA, FB	58	28	0
Soot, PMD	31	50 (6)	470
Soot, FB	59	22 (10)	6
PMD, FB	502	18	10

(b) **Results for Bloat**

Tool: A,B	only A	both	only B
JA, Soot	8	22 (4)	0
JA, PMD	0	30	104
JA, FB	19	11	0
Soot, PMD	0	22 (4)	112
Soot, FB	16	6 (3)	5
PMD, FB	123	11	0

(c) **Results for Chart**

Tool: A,B	only A	both	only B
JA, Soot	13	226 (31)	6
JA, PMD	22	217	1705
JA, FB	193	46	0
Soot, PMD	10	222 (27)	1700
Soot, FB	191	41 (21)	5
PMD, FB	1884	38	8

(d) **Results for Apache FOP**

Fig. 26. Results The numbers show the number of dead candidate assignments found by pairs of tools: Soot, JA$_{live}$ (JA), FindBugs (FB) and PMD. For each tool pair, the number of assignments only found in one of the tools and the number of assignments found in both are shown. For the assignments found by both tools, where one tool is Soot, the number of cases where Soot only removed some jimple lines are shown within parentheses.

Benchmark	JA$_{live}$	JA$_{live+reach}$	Soot	FindBugs	PMD
ANTLR	11.8 ± 0.3	22.3 ± 0.2	26.0 ± 5.2	105.6 ± 18.1	17.9 ± 2.4
Bloat	15.0 ± 0.4	46.8 ± 11.8	37.0 ± 8.9	115.5 ± 14.8	61.9 ± 10.1
Chart	7.4 ± 0.2	17.2 ± 4.4	20.2 ± 5.2	53.0 ± 12.0	7.6 ± 0.1
FOP	59.4 ± 11.6	278.9 ± 27.3	256.3 ± 2.6	250.3 ± 38.3	38.9 ± 9.3

Fig. 27. Average total execution time (in seconds).

In addition to the dead assignments found on candidates shown in the figure, Soot also finds an additional number of dead assignments not matching candidates (ANTLR = 256, Bloat = 902, Chart = 106 and FOP = 5212). We have not been able to manually check all these assignments, but after looking at many of them, we have only found cases that are either due to constant propagation (which we do not do), or to temporary variables introduced in the jimple code.

PMD reports very many dataflow anomalies of type DD and DU. After inspecting several of those that are neither reported by Soot nor JastAdd, we have only found false positives. It seems that arrays appear to be treated as ordinary variables, and that the control-flow is not fine enough, ignoring, for example, short-circuiting of boolean expressions. Like Soot, PMD reports dead assignments for non-candidates (ANTLR = 18, Bloat = 99, Chart = 22, FOP = 625). These non-candidates may, for example, be fields. It should be pointed out that the DD and DU reports are described by PMD to be anomalies that are *potentially* dead assignments. PMD does not claim that they are dead.

FindBugs finds comparatively few dead assignments, and reports no dead assignments for non-candidates. All the dead assignments found by FindBugs are found also by JastAdd.

6.3. *Performance*

Fig. 27 shows average total execution times in seconds for all tools, measured using `time`. All average times are given with a confidence interval of 95%.

JA$_{live}$ is faster than Soot on all four applications, and it is the fastest tool for three of the applications, with the exception of FOP where PMD is faster. For PMD, the performance for Bloat sticks out, which may be due to the large average flows in Bloat. FindBugs generally gets the worst performance, except for FOP where both JA$_{live+reach}$ and Soot are worse. Both

Benchmark	Plain	JA$_{live}$	JA$_{live+reach}$
ANTLR	1.7 ± 0.1	2.9 ± 0.06	9.1 ± 0.04
Bloat	2.3 ± 0.1	3.6 ± 0.06	17.5 ± 0.08
Chart	1.0 ± 0.07	1.3 ± 0.1	9.3 ± 0.2
FOP	10.0 ± 0.05	16.2 ± 0.07	182.4 ± 16.5

Fig. 28. In-memory performance for JastAdd. In seconds, using a pre-heated VM. *Plain* is the static-semantic analysis only.

Modules			Number of Rules				
Name	**Version**	**LOC**	`syn`	`inh`	`eq`	`coll`	`contr.`
Java Frontend	1.4	10 352	471	168	1 453	0	0
	1.5	4 909	166	48	588	0	0
Control Flow	1.4	444	17	26	185	2	5
	1.5	20	0	0	9	0	0
Liveness	1.4	29	4	1	10	1	3
Reaching	1.4	96	8	1	30	3	7
Helpers	1.4	33	11	1	13	1	1
Dead assignment	1.4	25	3	1	5	2	5

Fig. 29. Size of modules using lines of code (LOC) and number of JastAdd rules separated into different columns for – **`syn`**, **`inh`**, **`eq`**, **`coll`**, **`contributes`**. The modules for the alternative variants of liveness (JA$_{live}$ and JA$_{live+reaching}$) have the same size and are only included once.

JastAdd and PMD perform analysis on source which is likely to result in smaller control-flow graphs with less nodes. This might also explain the difference in performance between Soot/FindBugs and JastAdd/PMD.

One motivation for doing this type of analysis on source rather than on byte code is the applicability in interactive settings, for example, in editors. In an editing scenario a model of the edited program will be kept in memory. This model, which is typically an AST, will be updated in response to user actions, like code modifications. The time needed for re-computation of information will affect the response time experienced by the user, and a translation to byte code would potentially slow down performance. Fig. 28 shows JastAdd performance measures for an in-memory AST with a pre-heated VM. We show both our analyses, JA$_{live}$ and JA$_{live+reach}$, as well as a plain analysis only doing semantic analysis. These numbers show the performance for a full analysis of the whole benchmark application. Preferably, in an editing scenario each edit action should not trigger a full analysis of the application being edited, but employ some incremental evaluation mechanism for limiting unnecessary re-computations.

6.4. *Effort*

In order to estimate effort of implementation, we look at the actual size of the implementations. By making use of higher-level abstractions in the form of attributes, our wish is to decrease the development effort needed for the analyses.

Fig. 29 shows an overview of the different modules for the JastAdd approach, including the frontend of JastAddJ. Each module is separated into two rows when there is a modular extension from Java version 1.4 to Java version 1.5. For cases where such an extension is unnecessary due to reused behavior, only numbers for version 1.4 are given. Besides size, we also show the number of JastAdd rules divided into different columns depending on rule type. For completeness, the size of a Helpers module, needed by the Control Flow, Liveness, Reaching Definition and Dead Assignment modules, is also included.

The total number of lines for the JastAdd analyses is 647. In comparison, the corresponding Soot implementation is 1308 lines of code, i.e., more than twice as large. This includes 186 for the dead assignment analysis, 481 for dataflow analysis, and 641 for control-flow including the handling of exceptions. We have not found it meaningful to compare with the implementation sizes of PMD and FindBugs, since the results they report are so different.

7. Related work

Silver is a recent attribute grammar system with many similarities to JastAdd, but which does not support circular attributes. It has also been applied for declarative flow analysis [26], but using a different approach than ours. In Silver, the specification language itself is extended to support the specification of control-flow and dataflow analyses. The actual dataflow analysis is not carried out by the attribute grammar system, but by an external model checking tool. This approach is motivated by the difficulty of declaratively specifying dataflow analysis on the same program representation as, for example, type analysis. No performance figures for this approach are reported. In contrast, we have shown how both control flow and dataflow can be specified in a concise way directly using the general attribute grammar features of JastAdd, in particular relying on the combination of reference attributes, circular attributes and collection attributes.

Farrow introduced circular attributes, and used liveness as a motivating example [13]. He builds on traditional attribute grammars without reference attributes, and does therefore not build any explicit control-flow graph. The dataflow analysis is instead defined directly in terms of the underlying syntax, with rules for each kind of statement.

Another declarative approach to dataflow analysis (both inter- and intra) is to use techniques based on logic programming and deductive databases, running queries on a database of facts extracted from the program code [21]. Deductive database languages like Datalog have been used for interprocedural flow analyses of Java [25,7]. In this approach, the source program needs to be preprocessed, for example to resolve names, in order to extract the relevant facts. In contrast, the attribute grammar approach can be used seamlessly for all analysis after parsing. However, it should be pointed out that our current implementation concerns *intraprocedural* flow analysis only. Implementation of interprocedural flow analyses using reference attribute grammars is still future work.

Soot, [23], is a framework for optimizing, analyzing, and annotating Java bytecode. The framework provides a set of inter- and intraprocedural program optimizations with a much wider scope than the analyses presented in this paper. Soot is based on several kinds of intermediate code representations, including typed three-address code, and provides seamless translations between the different representations. Java source code is first translated into one of these representations in which some high-level structure is lost. The control-flow and data-flow frameworks in Soot are indeed quite powerful with reasonably small APIs. A major difference, as compared to our approach, is that the Soot approach is not declarative and therefore relies on manual scheduling when combining analyses, or adding new analyses as new specializations of the framework.

Schäfer et al. have used a variant of our analyses modules in the implementation of experimental refactoring tools for Java. They report performance on par with industrial strength refactoring tools [22].

8. Conclusions

Control-flow and intraprocedural dataflow analyses are important for source-level tools like bug detectors and refactoring tools. Doing such analysis at the source level, rather than at the level of intermediate code, is desirable from a tool integration point of view. The downside of working at the source level is that it requires all language constructs to be taken into account, and that the analyses need to be extended when new language features are added.

In this paper, we have presented a new approach to source level control-flow and dataflow analyses, based on reference attribute grammars that are augmented with circular attributes and collection attributes. We argue that this provides an excellent foundation for implementing these analyses, leading to concise specifications that are close to text book definitions, and that are easy to extend modularly when the language evolves.

We have demonstrated that the approach works well for practical applications by implementing control flow, dataflow, and dead assignment analyses for Java, and comparing with Soot (a well known Java optimization framework), and with PMD and FindBugs (both well known tools for bug and code anomaly detection).

Our evaluation shows that JastAdd analyses are concise and easy to extend modularly. The JastAdd specification for Java 1.4 is only 627 lines for Java 1.4, and only a 20-line module is needed to extend the control-flow analysis to Java 5, and the other analyses can be reused as they are. In comparison, the corresponding Soot implementation is 1308 lines.

To evaluate correctness and precision, we compared the results of dead assignment analysis on a number of Java benchmark programs, the largest being over 130 000 lines of code. Due to its focus on optimization, Soot can be expected to have both high correctness and preciseness, and manual inspection of deviating results between the tools confirmed this. We found that the results from JastAdd and Soot were almost identical, with both finding a very small number of dead assignments that the other did not find. Both PMD and FindBugs found substantially fewer dead assignments, and PMD had many false positives.

Concerning performance, our JastAdd-based solution is between four and nine times faster than FindBugs, and at the same time more precise. The performance comparison between JastAdd and the other two tools, Soot and PMD, is less clear cut. While JastAdd is the fastest on most benchmarks, Soot and PMD find dead assignments also outside the candidate set we have tested. While we believe that most of these reports are due to constant propagation and internal optimizations of jimple code for Soot, and false positives for PMD, this would need to be manually verified.

We implemented two variants of dataflow analysis: liveness only, and liveness combined with reaching definitions. The difference between these variants was extremely small: adding the reaching definitions analysis accounted for merely three additional dead assignments detected in the four benchmark programs together, and which none of the other tools detected. The performance cost was quite large, however, resulting in an analysis that was between two to five times slower.

There are several interesting ways to continue this work. One is to investigate more advanced interactive tool support that need precise intraprocedural dataflow analysis. For example, more advanced bug and code anomaly detectors. Another direction is to extend the work to interprocedural analyses, in particular to object-oriented call graph construction and interprocedural points-to analysis. We already have promising work in the direction of call graphs and simple whole program devirtualization analysis [19]. Because evaluation of reference attribute grammars is demand-driven, they should lend themselves to interprocedural analyses.

A third direction is to apply these results to analysis on intermediate code, and to develop declarative frameworks building SSA form and declarative implementation of related analyses.

Acknowledgements

We are very grateful to Max Schäfer for refactorings and improvements to the Control Flow Graph module.

References

[1] Frances E. Allen, Control flow analysis, SIGPLAN Notices 5 (7) (1970) 1–19.
[2] Andrew W. Appel, Modern Compiler Implementation in Java, Cambridge University Press, 2002.
[3] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, Using static analysis to find bugs, IEEE Software 25 (5) (2008) 22–29.
[4] S.M. Blackburn, R. Garner, C. Hoffman, A.M. Khan, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: Proceedings of OOPSLA'06, ACM Press, October 2006.
[5] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Lee, J. Eliot, B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, Ben Wiedermann, Wake up and smell the coffee: evaluation methodology for the 21st century, Communications of the ACM 51 (8) (2008) 83–89.
[6] John Tang Boyland, Remote attribute grammars, Journal of the ACM 52 (4) (2005) 627–687.
[7] Martin Bravenboer, Yannis Smaragdakis, Strictly declarative specification of sophisticated points-to analyses, in: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, ACM, 2009, pp. 243–262.
[8] Jong-Deok Choi, David Grove, Michael Hind, Vivek Sarkar, Efficient and precise modeling of exceptions for the analysis of java programs, SIGSOFT Software Engineering Notes 24 (5) (1999) 21–31.
[9] T. Copeland, PMD Applied, Centennial Books, 2005.
[10] Torbjörn Ekman, Görel Hedin, Modular Name Analysis for Java Using JastAdd, in: Generative and Transformational Techniques in Software Engineering, vol. 4143/2006, Springer, Berlin/Heidelberg, 2006, pp. 422–436.
[11] Torbjörn Ekman, Görel Hedin, The jastadd system — modular extensible compiler construction, Science of Computer Programming 69 (1–3) (2007) 14–26.
[12] Torbjörn Ekman, Görel Hedin, The JastAdd extensible java compiler, in: OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications, 2007, pp. 1–18.
[13] Rodney Farrow, Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars, in: Proceedings of CC'86, ACM Press, 1986, pp. 85–98.
[14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Java (TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)), Addison-Wesley Professional, 2005.
[15] Görel Hedin, Reference attributed grammars, Informatica (Slovenia) 24 (3) (2000) 301–317.
[16] D.H.H. Ingalls, A simple technique for handling multiple polymorphism, in: Proceedings of OOPSLA'86, 1986, pp. 347–349.
[17] JastAdd, 2007–2011. http://jastadd.org.
[18] Donald E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145. Correction: Mathematical Systems Theory 5 (1) (1971) 95–96.
[19] Eva Magnusson, Torbjörn Ekman, Görel Hedin, Extending attribute grammars with collection attributes — evaluation and applications, in: Proceedings of Seventh IEEE Working Conference on Source Code Analysis and Manipulation, September 2007.
[20] Eva Magnusson, Görel Hedin, Circular reference attributed grammars — their evaluation and applications, Science of Computer Programming 68 (1) (2007) 21–37.
[21] Thomas W. Reps, Solving demand versions of interprocedural analysis problems, in: Compiler Construction, 5th International Conference, CC'94, in: Lecture Notes in Computer Science, vol. 786, Springer, 1994, pp. 389–403.
[22] Max Schäfer, Torbjörn Ekman, Oege de Moor, Sound and extensible renaming for Java, in: Gregor Kiczales (Ed.), 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, ACM Press, 2008.
[23] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, Phong Co, Soot — a Java optimization framework, in: Proceedings of CASCON 1999, 1999, pp. 125–135.
[24] H.H. Vogt, S.D. Swierstra, M.F. Kuiper, Higher order attribute grammars, in: Proceedings PLDI'89, ACM Press, 1989, pp. 131–145.
[25] John Whaley, Monica S. Lam, Cloning-based context-sensitive pointer alias analysis using binary decision diagrams, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2004, ACM, 2004, pp. 131–144.
[26] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, Derek Bodin, Attribute grammar-based language extensions for Java, in: Proceedings of ECOOP'07, in: LNCS, Springer, 2007.