Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 238 (2009) 155–171
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Declarative Intraprocedural Flow Analysis of Java Source Code

Emma Nilsson-Nyman [1] Görel Hedin [3] Eva Magnusson [4]

Department of Computer Science
Lund University
Lund, Sweden

Torbjörn Ekman [2]

Programming Tools Group
University of Oxford
Oxford, United Kingdom

Abstract

We have implemented intraprocedural control-flow and data-flow analysis of Java source code in a declarative manner, using reference attribute grammars augmented with circular attributes and collection attributes. Our implementation is built on top of the JastAdd Extensible Java Compiler and we have run the analyses on medium-sized Java programs. We show how the analyses can be built using small concise composable modules, and how they provide extensible frameworks for further source code analyses. Preliminary measurements indicate that there is little difference in execution time between our declarative data-flow analysis and an imperative implementation.

Keywords: declarative, data-flow, analysis, control-flow, Java, compiler, attribute grammars

1 Introduction

Control-flow and data-flow analysis form the foundation for many static analyses of source code, e.g., code optimization, refactoring, enforcing coding con-

[1] Email: emma.nilsson_nyman@cs.lth.se
[2] Email: torbjorn.ekman@comlab.ox.ac.uk
[3] Email: gorel.hedin@cs.lth.se
[4] Email: eva.magnusson@cs.lth.se

1571-0661 © 2009 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2009.09.046

ventions, and metrics. Both analyses are usually carried out on a normalized intermediate code representation rather than on a more high-level abstract syntax tree (AST) representation, to simplify the analyses by not having to deal with the full source language. However, doing these analyses directly on the source representation can be beneficial, since the high-level abstractions are not compiled away during the translation to intermediate code. This is particularly important for analysis tasks requiring some kind of interaction with the user, such as refactoring and reporting violations of coding conventions.

In this paper we show how control-flow and data-flow analysis can be performed directly on the AST used in a compiler frontend while retaining good performance and compact specification size. Moreover, our approach is completely declarative, enabling both modular and extensible specifications. We have implemented the analyses on top of the JastAdd Extensible Java Compiler [7], using an approach based on extended attribute grammars. The analyses reported in this paper are intra-procedural, i.e., local to a method. Extending the technique to support inter-procedural analysis is part of our ongoing work.

We use a number of extensions to the traditional Knuth style attribute grammars [10] that turn out to be extremely useful for flow analyses. Attribute values may be *references* to distant tree nodes [9], which is particularly useful to superimpose graph structures on top of the dominant AST structure, e.g., to refer to predecessor and successor nodes in the control flow graph. *Circular* attributes [8,12] allow attribute equations to be mutually dependent as long as there is a well-defined fixpoint. They enable us to declaratively specify traditional data-flow properties such as the *in* and *out* sets used in liveness analysis. The concept of *collection* attributes [6,11] allows an attribute to contribute a partial value to a collection in a distant tree node through a reference attribute. For example, a name may contribute itself to its declaration's set of uses. Collection attributes are also very convenient for defining reverse relations. For example, the set of predecessors can be computed from the set of successors using a single equation.

For each analysis, we present an object-oriented framework that specifies how a set of AST nodes collaborate in computing the desired property. Implementing these frameworks using attribute grammars makes them declarative, i.e., the values of the properties are defined, but not the order in which they are computed. Data-flow and control-flow analysis can therefore be specified in isolation for each kind of statement and expression in a syntax-directed fashion. The framework for control-flow analysis specifies how to provide equations for each statement that affects control flow, e.g., a looping construct. Simi-

larly, for expressions altering the flow of a particular kind of data, e.g., uses and definitions of local variables, the framework for data-flow analysis specifies how such expressions alter the data flow. These equations are then used by the provided framework code to compose a global solution. Statements that do not affect the control flow and expressions that do not affect the data being analyzed, can reuse the default behavior provided by the framework. This allows us to specify control-flow and data-flow analysis at the source level, even for complex languages such as Java, with moderate effort. The frameworks also enable simple extension to the analyses if the language evolves, e.g., when new statements or expressions are added. If the new language feature affects control flow or data flow then those effects can be specified modularly in a syntax-directed style, and if they have no effect on the analyses then the existing framework code can be reused as is.

To evaluate the efficiency and scalability of our presented approach we implemented an analysis for Java to detect dead assignments to local variables. The control-flow framework is less than 300 lines of code (LOC), and the data-flow framework is less than 30 LOC. The dead-assignment analysis adds an additional 8 LOC which gives us all in all less than 340 LOC. We ran the dead-assignment analysis on real Java applications of sizes around 40000 LOC, with execution times less than 9 seconds, including static-semantic checking, e.g., name binding, type checking, etc. Initial experiments indicate that there is little difference in execution time when comparing our declarative implementation of data-flow analysis to an imperative implementation using explicit fixpoint iteration.

The rest of this paper is structured as follows. We first describe the implementation of the control-flow analysis in Section 2. Then we present the data-flow analysis in Section 3 and show how both analyses can be used to analyze real-life applications in Section 4. We compare our work to related approaches in Section 5 and conclude and outline future work in Section 6.

2 Control Flow Analysis

In JastAdd, a program is represented as an AST. The AST nodes are represented by objects with attributes. Given this representation, we want to provide a reusable and extensible implementation of the intra-procedural control-flow graph. Control flow is typically defined over basic blocks, i.e., linear sequences of program instructions having one entry point [5]. The typical definition is as follows:

Definition 2.1 Control flow can be described as a directed graph $G = (B, E)$ where B is the set of nodes $\{b_1, b_2, \ldots, b_n\}$ and E is the set of directed edges

$\{(b_i, b_j), (b_k, b_l), \ldots\}$. Nodes represent basic blocks and edges represent control-flow paths. Each node has a set of *immediate successors* and *immediate predecessors* which both can be empty. The immediate successor set is given by the function $\Gamma_G(b_i) = \{b_j | (b_i, b_j) \in E\}$. The inverse of the successor function gives the set of immediate predecessors, $\Gamma_G^{-1}(b_j) = \{b_i | (b_i, b_j) \in E\}$.

At the AST level, we can view each statement as representing a basic block. The start and end of a control-flow graph are traditionally referred to as the *entry* and *exit* block. In our case, we will have an entry statement and an exit statement for each method. To represent the control-flow graph for a method, we represent the nodes by AST statement nodes, and the directed edges by references between these nodes. In JastAdd, these references can be defined declaratively, using attributes and equations.

The definition of the control flow for a method amounts to defining the following attributes:

```
Set Stmt.succ(); // Each statement has a set of successors
Set Stmt.pred(); // Each statement has a set of predecessors
Stmt MethodDecl.entry(); // Each method has an entry statement
Stmt MethodDecl.exit(); // Each method has an exit statement
```

In the following subsections, we will look at how these attributes are defined using JastAdd.

2.1 Language structure

We will use the code in Figure 1 as a running example to explain how a control-flow graph is superimposed on top of the existing AST. The goal is to add the successor edges shown in Figure 2 to the tree in Figure 1 using attributes. There are a few things worth noticing about these edges. Some statements are active in deciding where to transfer the control. For example, the `IfStmt` transfers control to its `Then` branch, which in this case starts with an `ExprStmt`. Some statements complete without explicitly transferring control to another statement but rather implicitly they transfer control depending on their context. For example, the `ExprStmt` contained in the `IfStmt` continues with the statement following the `IfStmt`, while the `ExprStmt` contained in the `WhileStmt` goes back and re-evaluates the `WhileStmt`. We first present the AST structure and then go on to define attribute equations for successors, predecessors, and finally the entry and exit nodes.

The language structure is defined by the abstract grammar in Figure 3. We will briefly go through the structure since all our attributes and equations will be defined in terms of this grammar. There are `Stmt`s and `Expr`s which are abstract entities. A `Block` is a subclass of `Stmt` and holds a list of `Stmt`s as

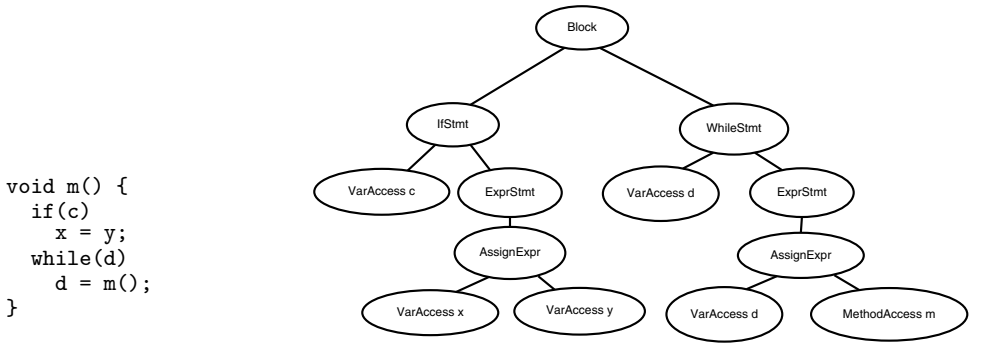

Fig. 1. Sample method body and its abstract syntax tree.

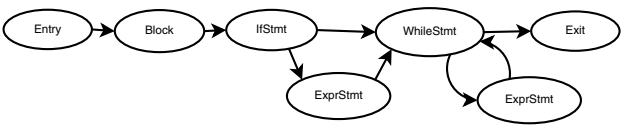

Fig. 2. The control flow graph for the example in Figure 1.

its children. An `IfStmt` has a condition `Expr`, a `Then` branch, and an optional `Else` branch. A `WhileStmt` has a condition `Expr` and a `Stmt` which is executed each iteration in the loop. Certain `Expr`s can act as `Stmt`s and we therefore introduce an `ExprStmt` turning an expression into a statement. The value of an `AssignExpr` is its `RValue` and the `LValue` is assigned that value as a side-effect. A `VarAccess` refers to a variable and can act as both an `LValue` and an `RValue`. A `MethodAccess` is a method invocation with a list of arguments. A more thorough introduction to the abstract syntax definition is available at [2].

```
abstract Stmt;
Block : Stmt ::= Stmt*;
IfStmt : Stmt ::= Expr Then:Stmt [Else:Stmt];
WhileStmt : Stmt ::= Expr Stmt;
ExprStmt : Stmt ::= Expr;

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

Fig. 3. The abstract grammar for the language in Figure 1.

2.2 Successors

We use a *synthesized* [5] attribute called `Stmt.succ()` to define the immediate successor function Γ_G, as described in Definition 2.1. The default behavior is that a statement completes normally and continues with the following statement. We therefore introduce a default equation for the `succ` attribute using an *inherited* [6] helper attribute `following`, representing the next statement in the current context. This behavior is desirable for statements that do not affect the flow, such as the `ExprStmt` which continues with the next statement. However, statements such as `IfStmt`, `WhileStmt`, and `Block`, need to be treated separately.

```
syn Set Stmt.succ() = following();
inh Set Stmt.following();
```

An `IfStmt` with both a `Then` branch and and an `Else` branch will transfer control to either statement, but an `IfStmt` with only a `Then` branch will jump to either the `Then` statement or to the following statement in the flow. We can capture that behavior with the equation below. Sets are formed by starting with the *empty* set and using the union operator to add additional members. The sets are immutable and each operation returns a new set.

```
eq IfStmt.succ() = hasElse() ?
  empty().union(getThen()).union(getElse())
 : following().union(getThen());
```

The successor to a `WhileStmt` is its contained statement when the condition is true and the following statement when the condition is false. The equation below therefore defines the successors to be the union of the *following* set and the contained statement. The contained statement is treated slightly differently compared to the `IfStmt`. If the contained statement completes normally then the following statement is to go back to evaluate the condition again. The `WhileStmt` needs therefore define that the *following* set for its child should include the `WhileStmt` itself rather than the current *following* set. We therefore need two equations, one for the successor of `WhileStmt` and one to redefine the context, i.e., the `following` attribute, for the contained statement:

```
eq WhileStmt.succ() = following().union(getStmt());
eq WhileStmt.getStmt().following() = empty().union(this);
```

[5] A synthesized attribute of an AST node is defined by an equation in the same node.
[6] An inherited attribute of an AST node is defined by an equation in an ancestor node.

The successor to a `BlockStmt` is the first contained statement or the *following* set if the block is empty. The `BlockStmt` acts like a mediator giving each contained statement permission to execute in order. If statement i in a `Block` completes then the successor is statement $i + 1$, unless i is the last statement in which case the block completes and gives control to the *following* set.

```
eq Block.succ() = getNumStmt() != 0 ?
  empty().union(getStmt(0)) : following();
eq Block.getStmt(int i).following() = i != getNumStmt() - 1 ?
  empty().union(getStmt(i+1)) : following();
```

Some nodes need thus define the successor attribute, others both the successor and following attributes, while others can reuse the default behavior. To generalize the given examples we define three roles which can be used to characterize a statement in determining which equations need to be provided.

explicit jump Statements which are executed and then locally define their successors. A statement in this category needs to give an equation for the `succ` attribute. Typical examples of statements include `IfStmt` but also statements such as `BreakStmt` and `ThrowStmt`.

call subroutine Statements which temporarily give up control to another statement to execute but then reclaim control by defining the successors for that statement. A statement in this category needs to give equations for both the `succ` and the `following` attribute. Typical examples of statements are `Block` and `WhileStmt`.

complete normally Statements for which the next statement to execute is given by the current context e.g., the statement's location in a block with respect to the ordering of the statement sequence of that block. An `ExprStmt` is a typical example in this category.

Only statements in the first two categories need to provide equations for control-flow analysis, whereas statements in the third category, e.g, the `ExprStmt`, can reuse the framework as is. The first two roles also form an extension framework for the analysis. If we add a new kind of statement we need only determine if it plays either of the roles above. If it does then we need to provide one or two equations to extend the control-flow analysis, which can otherwise be reused as is.

2.3 Predecessors

The attributes presented so far only deal with the set of successors but in many analyses it is necessary to have the set of predecessors as well. To define

the immediate predecessor function, Γ_G^{-1}, we use a collection attribute `pred` defined as the inverse of the immediate successor function, using the following two rules.

```
coll Set Stmt.pred() [empty()] with add;
Stmt contributes this to Stmt.pred() for each succ();
```

The first rule declares a collection attribute (`coll`) called `pred` for `Stmt` nodes. Its value is of type `Set`, and is defined as the call to `empty` (a method on `Set`) combined with a number of *contributions*, each added by a call to `add` (also a method on `Set`). (The contributing method of a collection attribute, `add` in this case, must be such that the order of adding the contributions does not matter.)

The second rule defines the contributions (`contributes ... to ... for each`). This rule says that a `Stmt` contributes itself (`this`) to the predecessor set (`Stmt.pred`) of each of its successors (`succ`). A more detailed presentation of collection attributes and their evaluation in JastAdd is available in [11].

2.4 Entry and exit nodes

From a flow analysis point of view it is often convenient to have explicit entry and exit nodes in the control-flow graph. Since the AST may not have unique entry and exit nodes we attach them to the AST using non-terminal attributes (also called higher-order attributes) [15]. Like other attributes they are defined using equations, but are considered as higher-order in that they also act as nodes in the AST and can themselves have attributes. For each `MethodDecl` we define two non-terminal attributes: `entry` and `exit`. Then we add the method block to the *following* of the entry node and add the exit node to the *following* of the method block. This effectively attaches the entry node before the block and if the block completes then it binds to the exit node. We also provide an equation that propagates a reference to the exit node to the method block. That way statements that want to transfer control to the end of the method, e.g., a `ReturnStmt`, simply adds the exit reference to its successor set.

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

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

eq MethodDecl.getBlock().exit() = exit();
```

2.5 Advanced Language Constructs

When constructing a control-flow graph for a language such as Java, it is necessary to deal with exceptions and language constructs such as `try-catch-finally` and `throw`. Especially the `finally` block of the `try-catch` statement affects the control flow related to `break`, `continue` and `return`. Of the approximately 300 LOC required to define the attributes for the control-flow graph, around 240 LOC are directly related to these language constructs.

We use synthesized attributes to propagate information upwards in the AST. For example, sets of unmatched `throw` statements can be acquired via an attribute called `uncaughtThrows` defined for all nodes of type `Stmt`. Information about, e.g., enclosing `TryStmt`s, are broadcasted downwards in the AST using inherited attributes such as `enclosingTryStmt`. These synthesized and inherited attributes are matched against each other. For example, the set given by the `uncaughtThrows` attribute is matched against `catch` clauses in the closest enclosing `try` statement given by the `enclosingTryStmt` attribute. If no match is found, the control flow is directed to the `finally` block of the enclosing `TryStmt`, if there is such a block, otherwise it is passed on to the next enclosing `TryStmt`, and so on.

Similar techniques are used to deal with `break`, `continue` and `return` statements. For example, after a `break` statement has been executed, all enclosing `finally` blocks between the `break` statement and its enclosing target statement, e.g., a `while` statement, need to be executed before the target statement. The control flow for a `return` statement is just a special case of this scenario, with the exit node as the constant target statement.

3 Data-Flow Analysis

We want to analyze data flow on our control-flow graphs defined in the previous section. A typical data-flow analysis is liveness analysis. We use the following definition of liveness, based on the definition in [4]:

Definition 3.1 Let $in(b)$ be the set of variables live immediately before block b and let $out(b)$ be the set of variables live immediately after block b. Let $def(b)$ be the set of assigned variables in b and $use(b)$ be the set of used variable in b. The $def(b)$ and $use(b)$ relates to the $in(b)$ and $out(b)$ sets in the following way:

$$in(b) = use(b) \cup (out(b) \setminus def(b))$$

$$out(b) = \bigcup_{x \in succ(b)} in(x)$$

A variable is live if its assigned value will be used by successors in the control-flow graph. If the variable is assigned a new value before the old value has been used the old assignment can be considered unnecessary.

Since our control flow analysis operates on statements rather than blocks we need to define the set of used variables for each statement, and the set of assigned variables for each statement. These sets combined with the set of successors enable us to express the *in* set and *out* set purely in terms of statements. Defining these attributes for each statement will thus make the liveness analysis valid for all kinds of statements. We also notice that the *in* set and the *out* set are defined using recursive equations which are mutually dependent. Such equations 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.

3.1 Use sets and definition sets

The main challenge in computing the sets of accessed variables is to support all kinds of statements and their enclosed expressions in the source language. A complex language such as Java has more than 20 statements and 50 expressions. Fortunately, it turns out that it is quite easy to support all these constructs in the JastAdd Extensible Java Compiler, as we will now explain.

All expressions that access a local variable encapsulate a `VarAccess` node performing the actual binding. Moreover, each `VarAccess` node has two boolean attributes, `isDest` and `isSource`, determining whether the access acts as an LValue or an RValue. 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.

The use sets can be computed by collecting all `VarAccess` nodes acting as an RValue that are enclosed by a particular statement. This can be done very conveniently using collection attributes, as shown below. Each `VarAccess` contributes its corresponding declaration to the enclosing statement's *use* set if it acts as an RValue. The *def* sets are computed using the same strategy. The binding from an expression to its enclosing statement is computed using an inherited attribute. Each statement provides an equation for all its children that states that it is the enclosing statement. This single equation propagates the binding down to expressions at arbitrary depths, since equations for inherited attributes in JastAdd are valid for all nodes in a subtree and not only for the immediate children.

```
coll Set Stmt.use() [empty()] with add;
VarAccess contributes decl()
  when isSource() && decl().isLocalVariable()
```

```
  to Stmt.use() for enclosingStmt();

coll Set Stmt.def() [empty()] with add;
VarAccess contributes decl()
  when isDest() && decl().isLocalVariable()
  to Stmt.def() for enclosingStmt();

inh Stmt Expr.enclosingStmt();
eq Stmt.getChild().enclosingStmt() = this;
```

These three attributes effectively computes the *use* sets and *def* sets for all statements in Java. Consider for instance a `MethodAccess` with the structure described in Figure 3. Its arguments may very well contain uses and definitions, since both are expressions in Java. However, we need not provide any additional equations for that language construct since contributions from each `VarAccess` are collected automatically, and inherited attributes are valid for all descendants and not only the immediate children. These ways to abstract over the tree structure are the reason that three equations are sufficient to handle all kinds of expressions in a complex language such as Java. This is also important from an extension point of view. 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 anyways, and the *use* set and *def* set attributes are still valid. In Section 2 we showed how the control-flow analysis could be extended to support new statements by only adding a few equations as well. This means that a few equations are all that is needed to extend both the control-flow and data-flow analysis to make them support a language extension, regardless of if we add new statements or expressions.

3.2 In sets and out sets

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 case for our equations since the power set of the set of local variables ordered by inclusion forms a lattice with the empty set as bottom and 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 such iteration through circular attributes as described in [12]. 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 specify the *in* and *out* sets in a style very close to their formal definition using the following two circular

attributes: [7]

```
syn Set Stmt.in() circular [empty()] =
  Stmt.in() = use().union(out().compl(def()));

syn Set Stmt.out() circular [empty()] {
  Set set = empty();
  for(Iterator iter = succ().iterator(); iter.hasNext();) {
    Stmt stmt = (Stmt)iter.next();
    set = set.union(stmt.in());
  }
  return set;
}
```

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:

```
coll Set Stmt.out() circular [empty()] with add;
Stmt contributes in() to Stmt.out() for each pred();
```

An alternative to using circular attributes would be to manually implement the fixpoint computation imperatively. For comparison, we have implemented the imperative liveness analysis algorithm given in [4]. 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 functions 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 when using circular attributes. In Section 4 we give a more quantitative comparison of both implementations in terms of size and speed.

4 An Application

To evaluate the efficiency and scalability of our approach, we have implemented a simple analysis for Java which detects dead assignment of local variables, i.e., the assignment of variables whose values are not used later in the program. This analysis can easily be added as an extension to the liveness analysis described in the previous section. We use the following definition to detect dead assignments:

[7] The equation for `out` 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.

Table 1
The size of the control-flow analysis (CFA) is not given for the imperative analysis since it uses the attributes from the declarative implementation. The two other columns show the size of the data-flow analysis (DFA) and the size of the dead assignment analysis (DAA)

Implementation	**CFA** (LOC)	**DFA** (LOC)	**DAA** (LOC)
Declarative	300	30	8
Imperative	—	190	8

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

$$\textbf{not}\ def(s) \subseteq out(s)$$

This definition detects unnecessary assignments. Dead assignments might still have right-hand side expressions which need to be evaluated to preserve program behavior. In JastAdd, the definition is expressed as follows.

```
syn boolean Stmt.isDead() = !def().subset(out());
```

To collect all dead assignments in a compilation unit, we add a collection attribute `deadCode()` to the `CompilationUnit` AST node. The dead assignments contribute themselves to the collection of their compilation unit using a contributes clause. The reference to the compilation unit is propagated to the statement nodes using an inherited attribute called `enclosingCompilationUnit`.

```
coll HashSet CompilationUnit.deadCode() [new HashSet()]
  with add root CompilationUnit;
Stmt contributes this when isDead() to
  CompilationUnit.deadCode() for enclosingCompilationUnit();
inh CompilationUnit Stmt.enclosingCompilationUnit();
eq CompilationUnit.getTypeDecl(int i).enclosingCompilationUnit()
  = this;
```

All together, the complete declarative extension from liveness analysis to dead assignment analysis adds up to a mere 8 LOC. As a comparison, we extended the imperative implementation of liveness, from the previous section, to include a search for dead assignments. The size of this extension is the same: 8 LOC. These numbers are summarized in Table 1. Both implementations depend on the declarative control-flow implementation. The declarative control-flow analysis which covers all of Java1.4 adds up to only 300 LOC, which is a small number considering the complexity of the language.

The size of the different liveness implementations differs by a factor of approximately six while the sizes of the dead assignment implementations are the same.

Table 2
Lines of code (LOC), compilation time (CT) without any additional analysis, compilation time with imperative dead code analysis (CT + IA), compilation time with declarative dead code analysis (CT + DA) and the number of dead statements (DS) found.

Project	**LOC**	**CT** (s)	**CT + IA** (s)	**CT + DA** (s)	**DS** (#)
antlr	42000	8.73	10.70	10.90	96
bloat	39000	8.37	10.90	11.17	11

To test our implementations and to measure execution time we have chosen to look for dead assignments in the following Java projects:

- **antlr** (v. 2.7.7) - A parser and translator generator [1].
- **bloat** (v. 1.0) - A byte-code level optimization and analysis tool [3].

Execution time with and without analysis for both implementations along with the number of dead assignments found are summarized in Table 2. Both implementations run in approximately the same time. We found 96 dead assignments in **antlr** and 11 in **bloat**. In **bloat** half of the dead assignments are `null` assignments while in **antlr** there were only a few dead null assignments. Two thirds of the dead assignments found in **antlr** are due to one frequent, but unused, variable.

5 Related Work

Silver is a recent attribute grammar (AG) system with many similarities to JastAdd, but which does not support circular attributes. Silver has also been applied for declarative flow analysis [16], but using a different approach than ours. In the Silver approach, the specification language itself is extended to support the specification of control-flow and data-flow analysis. The actual data-flow 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 data-flow 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 data flow can be specified in a concise way directly using the general AG features of JastAdd, in particular relying on the combination of reference attributes, circular attributes and collection attributes.

Morgenthaler [13] has developed static analysis techniques for source-to-source tools. To reduce the cost, techniques for efficient demand-driven analyses are proposed as opposed to traditional exhaustive methods. These techniques operate directly on the AST, the most appropriate data structure for

a source-to-source tool architecture. No explicit control flow representation is built. Instead, a so called *virtual control flow* is constructed by demand-driven computations of all possible control successors and predecessors. Functions realizing this scheme for the C language, implemented in C++, required about 1000 lines of code. A major difference between this approach and ours is that in using JastAdd, the demand-driven evaluator is automatically constructed from concise declarative grammar specifications.

Soot, [14], is a framework for optimizing, analyzing, and annotating Java bytecode. The framework provides a set of intra-procedural and whole program optimizations with a wider scope than the analyses presented in this paper. Soot is based on several kinds of intermediate code representations, e.g., 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.

6 Conclusions

Control-flow and data-flow analysis are usually cumbersome to implement for source level analyses of complex languages such as Java. The main reason is the tedious work to implement analyses that support all language constructs in today's mainstream languages. Moreover, since languages constantly evolve there is a need to update the analyses accordingly.

We have shown how reference attributed grammars augmented with circular attributes and collection attributes provide an excellent foundation for declaratively specifying control-flow and data-flow analysis. The specifications are concise and close to text book definitions, yet the generated analyzers are sufficiently efficient for real applications. The specifications are also extensible in that the analyses can be extended modularly when new features are added to a language.

These are the main contributions of this paper:

- We present how to concisely specify control-flow and data-flow analysis *declaratively* using attribute grammars extended with reference attributes, circular attributes, and collection attributes.
- We provide declarative frameworks for control-flow and data-flow analysis. The frameworks provide default behavior and therefore only require equa-

tions for statements that affect control flow and expressions that manipulate the desired data kind. The equations are specified in a syntax directed fashion and can be extended modularly for new language constructs.

- We have implemented the described frameworks and analyses on top of the JastAdd Extensible Java Compiler and evaluated the implementation on real world applications of around 40000 lines of code.

There are several interesting ways to continue this work. The design ideas and frameworks presented in this paper are general and it would be interesting to see how they extend to more advanced analyses, e.g., object-oriented call graph construction and inter-procedural points-to analysis. We already have promising work in this direction, for example simple whole program devirtualization analysis [11]. We would also like to design and implement declarative frameworks for other traditional backend analyses such as translation to SSA-form. Another interesting area would be to apply the same techniques to do domain-specific source level analyses, for example, enforcing framework conventions.

References

[1] Antlr, 2007. http://www.antlr.org/.

[2] JastAdd, 2007. http://jastadd.org.

[3] Purdue Bloat, 2007. http://sourceforge.net/projects/javabloat/.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, 1986.

[5] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[6] John Tang Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

[7] Torbjörn Ekman and 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*, pages 1–18, 2007.

[8] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of CC'86*, pages 85–98. ACM Press, 1986.

[9] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[10] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[11] Eva Magnusson, Torbjörn Ekman, and 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.

[12] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, August 2007.

[13] J D Morgenthaler. *Static Analysis for a Software transformation Tool.* Ph.D. thesis, University of San Diego, California, 1997.

[14] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[15] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings PLDI'89*, pages 131–145. ACM Press, 1989.

[16] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute Grammar-based Language Extensions for Java. In *Proceedings of ECOOP'07*, LNCS. Springer, 2007.