DISSERTATION

# Analysis of Object-Oriented Programs with Flow-Shape Graphs

ausgeführt zum Zwecke der Erlangung des akademischen Grades
Doktor der technischen Wissenschaften

eingereicht am Institut für Informationstechnologie
der Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik

von
Markus Schordan, Dipl.-Ing.

im Mai 2001

1. Begutachter: o.Univ. Prof. Dr. Laszlo Böszörményi
2. Begutachter: o.Univ. Prof. Dr. Manfred Brockhaus

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Schrift verfasst und alle ihr vorausgehenden oder sie begleitenden Arbeiten durchgeführt habe. Die in der Schrift verwendete Literatur sowie das Ausmaß der mir im gesamten Arbeitsvorgang gewährten Unterstützung sind ausnahmslos angegeben. Die Schrift ist noch keiner anderen Prüfungsbehörde vorgelegt worden.

Unterschrift

**Abstract**

We present a new algorithm for static analysis of object-oriented programs. The analysis is formulated as an instance of a monotone framework and operates on finite flow-shape graphs to approximate conservatively the possible "shapes" that heap-allocated data structures can take on. Flow-shape graphs cover alias information, reaching definition information, and type information simultaneously. May-alias information as well as must-alias information is implicitly kept in the graphs, allowing strong updates and, thus, leading to high precision. By using the type information of the graphs we are able to determine which overriding method may be invoked at run time and integrate method invocation resolution. The reaching definition information is essential to optimizing algorithms which rely on precise dependence information of programs.

## Kurzfassung

Wir stellen einen neuen Algorithmus für die statische Analyse von objektorientierten Programmen vor. Die Analyse ist als Instanz eines monotonen Systems formuliert and operiert auf endlichen Flow-Shape Graphen, um eine konservative Abschätzung der möglichen "Shapes", die eine im Heap allozierte Datenstruktur annehmen kann, berechnen zu können. In Flow-Shape Graphen sind Alias Informationen, Erreichende Definitionen, und Typinformationen enthalten. May-Alias Information als auch Must-Alias Information ist implizit in den Graphen enthalten, so dass Strong Updates durchgeführt werden können, wodurch eine hohe Genauigkeit der Analyse erreicht wird. Durch Verwendung der Typinformation in den Graphen kann bestimmt werden, welche der überschriebenen Methoden zur Laufzeit aufgerufen werden können und somit integrieren wir auch das Auflösen von Methodenaufrufen. Die Erreichenden Definitionen sind essentiell für optimierende Algorithmen, die präziser Abhängigkeitsinformation von Programmen bedürfen.

# Acknowledgements

I would like to thank my advisor Laszlo Böszörményi for a liberal supervision in the development of this program analysis and for his ongoing encouragement and patience. In our numerous discussions he has provided me with inspirations, comments and insight. I am also grateful to my co-advisor Manfred Brockhaus for his continued interest in the development of the algorithm and his support through out the years which has contributed a lot to my interest in compiler construction.
I have also profited greatly from the probing questions and comments of Roland T. Mittermeir on aspects of the program analysis in local PhD meetings. I would like to thank Wolfram Amme for all the time he has dedicated to our PAOLA project and I wish to thank Wilhelm Rossak for supporting the PAOLA project and our cooperation. I would also like to thank Dominik Rauner-Reithmayer for the numerous discussions about topics of compiler construction. And I would like to thank Harald Kosch for all the discussions at the early stage of the development of this thesis.

# Contents

# List of Figures

# Chapter 1

# Introduction

Object-oriented languages owe their increasing popularity important programming concepts, like abstraction, encapsulation of data and code, inheritance, polymorphism and dynamic binding. Availability of these concepts often allows a programmer to build on object-oriented systems in an elegant and convenient manner. In the last decade, object-oriented programming has become a widely-used, important programming paradigm that is supported in many different languages. The C++ programming language is currently the most widely available and used language, and Java is a relatively new type-safe language which has become popular recently.

In object-oriented languages, objects are mostly accessed via references. Languages like Java allow to perform destructive updates on variables by assigning references to variables. Therefore, precise program analysis of object-oriented languages must be able to describe properties of linked data structures (e.g. lists, trees, graphs), as well as the intermediate states that arise when such structures are destructively updated. For automatic optimization of object-oriented programs precise dependence information must be provided by a program analysis. Furthermore it must deal with types and overriding of methods.

Program analysis offers static techniques for predicting safe and computable approximations to the set of values or behaviors arising dynamically during execution of programs.

We present a new combined algorithm for the analysis of object-oriented programs which computes a precise characterization of the shapes of object-oriented data structures and computes in the same pass precise reaching definition information for each access path in a program. Additionally, type information is computed for each represented value. The type information is used to perform a method invocation resolution (see section 1.2.4) in the same pass.

This work addresses problems that can be looked at - depending on one's point of view - as pointer analysis problems, alias analysis problems, shape analysis problems, or dependence analysis problems.

The goal of the analysis is to estimate the shape of the data structure accessible from a given heap-directed pointer and to relate the shape of a data structure with the dependence information and type information for a given program.

The information gathered by analyzing a given program is three-fold: For every value, or approximated set of values, the shape a data structure may take on at a program point is represented as a directed graph. For each statement in the program a graph is computed that describes which path expressions may refer to the same objects (and which never do). Secondly, the reaching definitions (assignments) are computed for each access path and thirdly, the set of types an object may have, is integrated in the graph. All information is represented in a so called flow-shape graph (FSG ). Our analysis combines these different informations in a uniform framework and computes by a fix-point algorithm all three types of information in the same pass.

## 1.1 Motivation

Without any knowledge of pointer values, compilers typically make worst-case assumptions about aliasing and generate very conservative code. By providing the missing information, pointer analysis can enable compilers to perform more aggressive optimizations and to take advantage of parallelism in programs. Besides its use in compilers, pointer analysis is also needed by tools for debugging, performance analysis and other tasks that require knowledge of pointer values.

Alias analysis is needed to uncover parallelism in programs. Modern processors rely heavily on instruction-level parallelism to obtain their high performance. Unless the compiler is able to recognize parallelism and generate code to take advantage of the parallel hardware, super-scalar and VLIW processors cannot reach their full potential. With the help of a precise alias analysis, the compiler can find parallelism spread across large sections of a program and can perform high-level program transformations to expose that parallelism. A prerequisite for effective instruction scheduling is to disambiguate memory references so that references to distinct memory locations can be reordered. The analysis presented in this thesis provides exactly the information necessary for memory disambiguation.

Shape analysis has many uses beyond improving program performance. Any tool involving automatic program analysis is likely to benefit from information about shapes of data structures. Software engineering techniques such as program slicing need also precise dependence information [Ste98, ADS91]. Debugging tools can use dependence information to help detect errors such as memory leaks [SBF00], dangling pointers and using uninitialized data. The analysis presented, will allow us to statically detect errors like dereferencing a null-pointer - this is guaranteed to give rise to dynamic error - and a warning can be issued. Perhaps more surprisingly, the analysis allows us to validate certain properties of the shape of data structures manipulated by the program. We will also show that our algorithm can handle the possible different alias configurations and data dependencies on different execution paths of the same variables with a high amount of precision although all information is represented in one single graph for each statement.

## 1.2 Challenges of Shape Analysis

Because references, heap-allocated storage, and destructive updating are all mechanisms that introduce aliasing, the formal treatment of shape analysis is notationally somewhat formidable.

### 1.2.1 Representation of Data Structures

With an appropriate amount of precision an analysis should be able to represent the gathered information so that a characterization of data structures as basic data structures such as lists, trees, or graphs is possible. A main goal in this respect is that modifications of such data structures are handled by the analysis in such a way that enough information about the data structure is preserved so that properties like "listness" or "treeness" can be preserved. Hand in hand with that goes a precise dependence analysis. Each access path for a given data structure and variables refering to elements of the data structure can be accessed and assigned a value. The more precise are the dependencies reported by a program analysis, the more freedom is given to an optimizing algorithm for reorganizing the code.

One important key-aspect in a precise alias analysis is that of strong nullification.

### 1.2.2 Strong Nullification

To model the effect of an assignment of a null-pointer to a variable $x$ is one of the major challenges in alias analysis. It requires to remove all values represented in our graph that have been stored in the variable $x$. At the same time the reaching definition information is killed and the number of dependencies reduced. The type information of objects is represented by the set of types of all values refering to an object. If several edges with reaching definition information and type information can be removed from the graph, the precision of the dependence information and the type information is improved at the same time.

### 1.2.3 Object Hierarchies

Objects (as well as structures) can be used to build hierarchies. The general problem with such hierarchies is that the size is not fixed and that is why $k$-bounded approaches [LH88, AZ97, AZ98, JM79, JM81] fail to track possible alias configurations as soon as the height of the object hierarchy exceeds the fixed value $k$. In our analysis, we are able to track alias configurations as long as any combination of variables directly refer to objects of such a hierarchy. The maximum length of a path in the graph depends on the number of variables.

### 1.2.4 Overridden Methods

If a method call $x.m(\ldots)$ of a type safe language like Java is analyzed a type-oriented conservative approach is to assume that methods of all subtypes of the declared type of $x$ may be invoked during execution of the program. This set of potentially invoked methods can be reduced if the set of subtypes that objects may have, which are referred to by $x$, can be reduced. Since our analysis computes precise alias information and at the same time the respective type information, the set of potentially invoked methods can be reduced in some cases. Only those types of objects to which a variable $x$ may refer at a given program point are relevant to the selection of the method. Our analysis integrates this aspect and performs a method invocation resolution. The transfer functions that are mapped to entry and exit nodes in an inter-procedural control flow graph (ICFG) ensure that information is propagated only to those paths in the ICFG which represent methods that may be invoked according to the set of types that is computed for an object. This is explained in detail in section 3.3.2, where we show that entry nodes can be understood as acting like a filter that does not propagate any information to paths that represent declared overriding methods that cannot be invoked by a given program.

## 1.3 A New Approach to Object-Oriented Analysis

The usual way for analyzing program dependencies in programs with references is to perform an alias analysis as a first step. A precise alias analysis needs to be context sensitive. Based on this alias information the reaching definitions of a program can be determined. For object-oriented programs it is necessary to compute type information to deal with subtyping and gather the necessary information for method invocation resolution. Our approach integrates all three analyses into one uniform framework. Although a precise shape analysis algorithm alone requires already a somewhat formidable formalism our approach of determining multiple summary nodes depending on the possible alias configurations is vital to ease the definition of those parts of the algorithm that perform a reaching definition analysis and a type analysis.

The naming scheme for multiple summary nodes gives us a high level of accuracy not only for the computation of alias configurations but also for keeping the sets of reaching definitions from different variables apart. By this we are able to discriminate between objects that are transitively pointed to by different collections of variables or via different selectors. Since this provides a sufficient level of precision it is not necessary to enumerate summary nodes by type information as proposed in [SRW98] or to use allocation-sites to identify shape nodes as in [LR92, Ruf95].

### 1.3.1 Representation of Data Structures

Data structures are represented by graphs where each node represents a single object or a set of objects. To identify the nodes in a graph we use a naming scheme for the nodes such that each node is identified by the collection of variables that directly or transitively refer to it. The explicit representation we use is similar to [SRW98].

In fig. 1.1 two data structures are represented in the same flow-shape graph: an arbitrary graph referred to by $p$ and $q$ and a list referred to by $x$. The edges are labeled by variable names, reaching definition information and type information. The naming scheme of flow-shape graphs will be introduced in sections 2.2 and 2.3.

### 1.3.2 Integration of Reaching Definition Information

For each value we compute the set of reaching definitions where this value may have been created. This does not effect the number of nodes in the graph. Each edge in the graph has attached a triple consisting of a variable name, a reaching definition, and a type. In general, we compute a graph that represents all possible access paths that may be used at a statement to

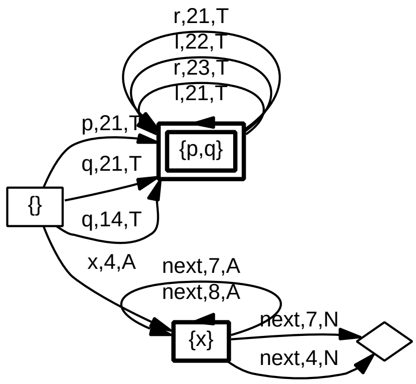
| Pos | Flow-Shape Graph |
|---|---|
| 40 |  |



Figure 1.1: An arbitrary graph and an unshared list

access a value and for each "part" of the access path we also compute the definition (assignment) where the value representing this "part" of the access path may have been assigned.

### 1.3.3 Integration of Type Information

In object-oriented programs it is necessary for method invocation resolution to have precise information about the set of subclasses an object may belong to. We do not store the type of the object itself in a flow-shape graph but the types of references that may refer to an object. The set of all these types equals the set of types an object may be of. Hence, the number of different types in a program does not affect the number of nodes in a graph but the size of sets of types that are related to an edge.

## 1.4 Overview of our Algorithm

The key for computing precise dependence information in programs with dynamically allocated objects in the heap is to represent the aliasing and sharing information such that regions of the heap which never interfere with each other are kept separate in the analysis. We represent the data structures in the heap as a graph. The nodes of the graph are named by sets of local variables. Local variables are declared in blocks of methods like in Java ([GJSB00] 14.4). Different kinds of variables - local variables and instance variables - are represented by different kinds of edges in a flow-shape graph and are introduced in section 2.3.1. Each statement is mapped to a transfer function modeling the semantics of the statement. The transfer functions are proved to be monotone and since the lattice of FSGs is bounded (see section 2.3.8) we can prove that our algorithm always terminates by finding a fix-point.

All flow-shape graphs shown in this work have been computed by our prototype implementation and are visualized using the tool dot.

To illustrate what the algorithm achieves, we use a program that generates a list and briefly discuss the computed flow-shape graph (FSG ) shown in fig. 1.3. We will discuss in detail how our algorithm computes the FSGs in sections 2 and 3.

Let variable i be of any value and of type integer, and variables x, y, and z of type A. The generated list, referred to by variable x, has length $l \geq 1$. An FSG represents all possible data structures that may be created at any program path without taking into account the concrete values of scalar variables. It is necessary to be able to represent approximations of data structures of potentially infinite size. We need to represent infinite lists, trees, and graphs. We use two different kinds of nodes: single nodes and summary nodes. Single nodes represent one object that is referred to by at least one reference. A summary node represents a set of objects where those edges that start at the summary node and point to the node itself (self-edges) represent references between those objects that are represented by the summary node. Summary nodes are printed bold in the graphs.

Various aspects of data structures and how they are connected via pointers can be understood from flow-shape graphs.

```
    class A {
      A next;
      ...
    }

 1: A x,y,z;
 2: int i;
 3: i = read();
 4: x = new A;
 5: y = x;
 6: while (i>0) {
 7:   z = new A;
 8:   y.next = z;
 9:   y = z;
10:   i = i-1;
    }
11: y=null;
```

Figure 1.2: Source: creation of a list

For the given program which generates a list referred to by x we obtain the result shown in figure 1.3 at program position 11.

For the given example program the list created is of minimum length 1. For $i \geq 0$ the length of the list is $i+1$. All possible shapes of the created list are represented by the nodes $\{x\}$ and $\{x, z\}$ and the edges refering to and from these nodes. The node $\{x\}$ is called summary node because it represents a set of objects, here the first $i$ elements of the list. The node $\{x, z\}$ is called single node because it represents one object, here the last element of the list. An edge refering to node $\diamond$ represents either a null-value or a scalar value. The different kinds of nodes and edges are explained in detail in sections 2.3.2 and 2.3.1.

The reaching definition information is computed for each value in the flow-shape graph. A value is represented by an edge. For example, the *reference* to the first element of the list is created at program position 4. The edge from node $\{\}$ to node $\{x\}$ represents this reference. If the list created has more than one element then variable z refers to the last element of the list at

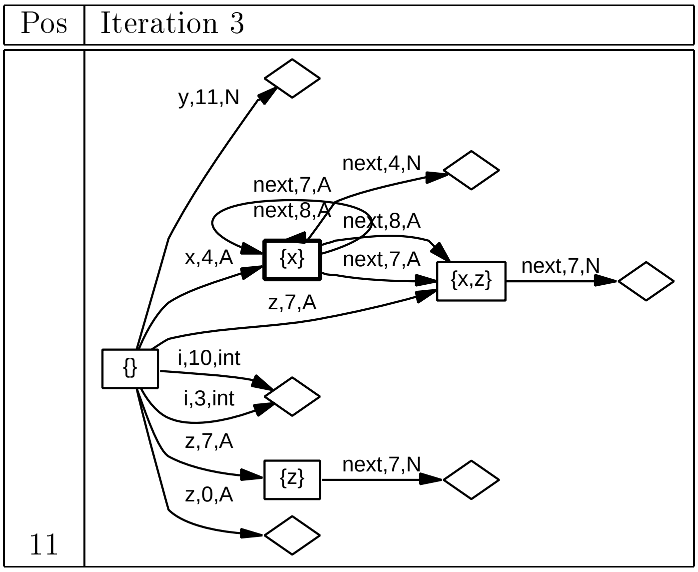



Figure 1.3: Creation of a list, iteration 3/4, out-information at pos 11

position 11. Possible values of the instance variable next are represented by edges labeled by next. This program is discussed in detail in section 2.5.1.

Important to many parallizing algorithms is whether an object (or a set of objects) is shared by different variables. Similar to [SRW98] we use a separate component to keep track of sharing information. In difference to their approach we integrate multiple summary nodes to separate regions of the heap. In fig. 1.1 two different data structures are represented by two different summary nodes. The objects of the arbitrary graph are represented by node $\{p, q\}$ which represents a shared data structure (double boxed) whereas the objects of the list are represented by the non-shared node $\{x\}$.

We will show that the use of multiple summary nodes is essential to the computation of precise dependence information when performed in the same pass as the shape analysis.

## 1.5 Contributions

Our algorithm computes three essential properties of object-oriented programs in the same pass:

- An approximation of all possible data structures that may exist at a program point.
- Reaching definition information for each approximated path that may be accessible at a program point.
- Type information for each reference and scalar value.

For each program position we compute one single graph that represents all possible data structures that may exist at this program position. A node in a flow-shape graph is named by the set of variables directly or indirectly refering to it. For each reference value we compute the definition where this value was last assigned and the type of that value. We do not compute the type of the objects itself because this can always be obtained from the set of references refering to this object and all the types of those references.

The analysis integrates essential aspects such as:

- Strong update (strong nullification)
- Multiple summary nodes to separate regions of the heap that are transitively pointed to by sets of variables
- Integration of the computation of reaching definitions of local variables and instance variables.
- Method invocation resolution of overridden methods
- Explicit representation of a null-value.

Strong update is achieved by always removing all edges in a graph that represent a local variable value that is killed by an assignment. This is essential to keep track of all kill aspects of definition and type information.

The integration of multiple summary nodes allows to separate regions of the heap that may be pointed to transitively by different sets of variables and/or different sets of fields of objects. By this we are able to keep reaching definition sets separate even in the case of combinations of non-cyclic and cyclic data structures (see fig. 1.1).

We integrate the computation of reaching definition information for local variables and instance variables, either being reference variables or scalar variables. For languages like Java it is necessary to also cope with scalar values of objects since scalar values may be stored in fields of objects which are elements of arbitrary data structures in the heap.

We also integrate the computation of types that any reference to an object may have. Based on this information we are able to perform a method invocation resolution. We define transfer functions for entry and exit nodes of functions in section 3.3.2.

Although the null-value and scalar values are not references to other objects they are represented as single directed arrows to the special node $\diamond$ in the graph. It holds that there are never any out-edges from this special node which models the fact that null-values and scalar values can never be dereferenced.

# Chapter 2

# Intraprocedural Analysis

In this chapter we introduce the monotone framework for the intraprocedural analysis. We will define Flow-Shape Graphs (FSG ) as the data the analysis operates on. A FSG represents all possible alias configurations at one program point as a single graph, including type information and definition information. Each graph represents a finite characterization of shapes that data structures may take on at a program point. We will associate a transfer function with each statement and prove the monotonicity of each transfer function. We will show that our analysis performs a may analysis which includes some aspects of must analysis. The intraprocedural analysis will be presented as a flow-sensitive forward analysis.

## 2.1 Language Definition

We are going to analyze object-oriented programs which can be defined as a subset of languages like Java or Modula-3. The language aspects that are to be analyzed are conditionals, loops, the effects of assignments of references and scalar values on data structures, objects, type hierarchies, and overridden method invocation. Therefore we define a simple language, which covers exactly those constructs, which are of importance for us.

The language we define is similar to the object-oriented language $\mathcal{L}$ in [TS00] but with access paths defined similar to Java. We will not cope with aspects of variables and methods that can be eliminated by renaming of variables and method names. For example, shadowed instance variables in Java are assumed to be renamed and to be unique (i.e. do not exist); likewise with overloaded methods. The problem of approximating the set of potentially invoked overriding methods is integrated in our analysis. We shall define transfer functions for method invocation resolution in section 3.

For sake of simplicity, we associate data flow information with single assignment statements, skip statements, and the tests that appear in conditionals and loops of methods that belong to a class. We require a method to identify these. The most convenient way of doing this is to work with a labeled program - as indicated in the syntax below. We will often refer to the labeled items as *elementary blocks*. We assume that distinct elementary blocks are initially assigned distinct labels.

We use the following syntactic categories:

$$\begin{array}{lcll} a & \in & \mathbf{AExp} & \text{arithmetic expressions} \\ b & \in & \mathbf{BExp} & \text{boolean expressions} \\ S & \in & \mathbf{Stmt} & \text{statements} \end{array}$$

We assume that some countable set of variables is given; numerals and labels will not be further defined and neither will the operators:

$$\begin{array}{lcll} x, y & \in & \mathbf{LVar} & \text{local variables} \\ sel & \in & \mathbf{IVar} & \text{instance variables} \\ n & \in & \mathbf{BExp} & \text{numerals} \\ \ell & \in & \mathbf{Lab} & \text{labels} \\ t & \in & \mathbf{SType} \cup \mathbf{CType} & \text{scalar types and class types} \\ p & \in & \mathbf{PExp} & \text{pointer expressions} \end{array}$$

Local variables are declared by a declaration as defined in section 2.4.10. When we are interested in variables, which are either local variables or instance variables we use $Var = LVar \cup IVar$.

An object is a class instance. The reference values (often just references) are pointers to these objects, and a special null reference *null*, which refers to no object.

Some languages, such as C++ and Oberon, expose the distinction between attribute records and references to attribute records in order to distinguish between stack-allocated and heap-allocated objects. Other languages, such as Java, Simula, Smalltalk, and Modula-3, hide this distinction and deal exclusively with heap allocated objects.

The design of our analysis does not make any distinction whether objects are created on the heap or on the stack because representants of objects are defined by the set of variables that refer to them but not by their concrete location.

The semantics of assignments are the same as in Java. Instance variables of objects can be accessed by *pointer expressions* like x.next: this will first determine the object pointed to by the variable x (dereference operation) and then return the value of the instance variable next. The null-reference is denoted as null and is a pointer expression. We will discuss in detail in section 2.4.2 why we treat the null-reference similar to scalar values in our analysis.

The operators are defined as the following syntactic categories and are not further defined because our analysis computes the types and definitions of assignments but does not integrate scalar values or possible ranges of these.

$$\begin{array}{lcll} op_a & \in & \mathbf{Op}_a & \text{arithmetic operators} \\ op_b & \in & \mathbf{Op}_b & \text{boolean operators} \\ op_r & \in & \mathbf{Op}_r & \text{relational operators} \\ op_p & \in & \mathbf{Op}_p & \text{reference operators} \end{array}$$

The syntax of the language[1] is given by an *abstract syntax*. We start with defining expressions and statements and continue with the definition of programs and blocks.

The abstract syntax for statements of a program has the form:

[1] We will extend the language in section 3 by methods and method calls.

$$
\begin{array}{lll}
p & ::= & x \mid x.sel \mid \text{null} \\
a & ::= & p \mid n \mid a_1 \; op_a \; a_2 \\
b & ::= & \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \; op_b \; b_2 \mid a_1 \; op_r \; a_2 \mid op_p \; p \\
S & ::= & R; \mid S \; S \\
 & & \mid \text{if } ([b]^\ell) \; \{S_1\} \text{ else } \{S_2\} \\
 & & \mid \text{while}([b]^\ell)\{S\} \\
 & & \mid B \\
R & ::= & [p = a]^\ell \mid [\text{skip}]^\ell \mid [\text{return } a]^\ell
\end{array}
$$

Arithmetic expressions include reference expressions and an arithmetic expression can also include the constant null. The binary operations $op_a$ do *not* allow pointer arithmetic. The boolean expression such as the relational operators $op_r$ allow to test for the equality of references. Note that arithmetic as well as boolean expressions can only access objects in the heap, they cannot create new objects nor update existing objects. We write elementary blocks $\alpha$ (see section 2.1.2) associated with a label $\ell$ as $[\alpha]^\ell$.

A block $B$ is defined to start with a sequence of variable declarations followed by a sequence $S$ of statements:

$$B \;::=\; [\{]^{\ell_b} \; D \; S \; [\}]^{\ell_e}$$

The label $\ell_b$ of $[\{]$ marks the entry to the block and the label $\ell_e$ of $[\}]$ marks the exit of the block. Please note that we define a block to be a statement to allow arbitrary nesting of scopes. This will allow us to discuss the effects of un-declaring variables at the end of scopes on the size of flow-shape graphs.

Blocks allow to declare variables with textual scoping. A variable declared within a block can be accessed within the scope of the block which starts at $\ell_b$ and ends at $\ell_e$.

A sequence of variable declarations $D$ is defined as:

$$
\begin{array}{lll}
D & ::= & t \; L; \mid D \; D \\
L & ::= & x \mid L, L
\end{array}
$$

where a type $t$ is followed by a list $L$ of variables.

A Program, $P$, has the form

$$P \ ::= \ C\ B$$

where $C$ is a sequence of class declarations, and $B$ is a block. A class declaration has the form:

$$C \ ::= \ \text{class } c_1 \text{ extends } c_0 \ \{\ D\ \} \mid C\ C$$

where $c_1$ and $c_2$ are class names and $D$ is a sequence of variable declarations.

The extends clause in a class declaration specifies the direct superclass of the current class. A class is said to be a direct subclass of the class it extends. The direct superclass is the class from whose implementation the implementation of the current class is derived (similar to [GJSB00] 8.1). The superclass of all classes is denoted $O$ in our analysis.

We extend the definition of classes in section 3 to include methods and extend all definitions to cope with method calls for interprocedural analysis.

### 2.1.1 Initial and Final Labels

When presenting our analysis we will use a number of operations on programs and labels. The first of these is

$$init : \mathbf{Stmt} \rightarrow \mathbf{Lab}$$

which returns the *initial label* of a statement:

$$\begin{array}{rcl}
init([p = a]^{\ell};) & = & \ell \\
init([\text{return } a]^{\ell};) & = & \ell \\
init([t\ L]^{\ell};) & = & \ell \\
init([\text{skip}]^{\ell};) & = & \ell \\
init(S_1\ S_2) & = & init(S_1) \\
init(\text{if}([b]^{\ell})\ \{S_1\} \text{ else } \{S_2\}) & = & \ell \\
init(\text{while}([b]^{\ell})\{S\}) & = & \ell \\
init([\{]^{\ell_b}\ D\ S\ [\}]^{\ell_e}) & = & \ell_b
\end{array}$$

We will also need a function

$$final : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

which returns the set of *final labels* in a statement; whereas a sequence of statements has a single entry, it may have multiple exits (as for example in the conditional):

$$\begin{array}{rcl}
final([p = a]^{\ell};) &=& \{\ell\} \\
final([\text{return } a]^{\ell};) &=& \emptyset \\
final([t\ L]^{\ell};) &=& \{\ell\} \\
final([\text{skip}]^{\ell};) &=& \{\ell\} \\
final(S_1\ S_2) &=& final(S_2) \\
final(\text{if}([b]^{\ell})\ \{S_1\} \text{ else } \{S_2\}) &=& final(S_1) \cup final(S_2) \\
final(\text{while}([b]^{\ell})\ \{S\}) &=& \{\ell\} \\
final([\{]^{\ell_b}\ D\ S[\}]^{\ell_e}) &=& \ell_e
\end{array}$$

The analysis shall be defined by a pair of functions that map labels to the appropriate data-flow graph; one function in each pair specifies information that is true on *entry* to the elementary block, the second specifies information that is true at the *exit*.

### 2.1.2 Elementary Blocks

To access the statements, or *elementary blocks*, of the form $[p = a]^{\ell}$ or $[skip]^{\ell}$ as well as the set of tests of the form $[b]^{\ell}$ the function *elementary* is defined as follows:

$$\begin{array}{rcl}
elementary([p = a]^{\ell};) &=& \{[p = a]^{\ell}\} \\
elementary([\text{return } a]^{\ell};) &=& \{[\text{return } a]^{\ell}\} \\
elementary([t\ L]^{\ell};) &=& \{[t\ L]^{\ell}\} \\
elementary([\text{skip}]^{\ell};) &=& \{[\text{skip}]^{\ell}\} \\
elementary(S_1\ S_2) &=& elementary(S_1) \cup elementary(S_2) \\
elementary(\text{if}([b]^{\ell})\ \{S_1\} \text{ else } \{S_2\}) &=& \{[b]^{\ell}\} \cup \\
&& elementary(S_1) \cup elementary(S_2) \\
elementary(\text{while}([b]^{\ell})\ \{S\}) &=& \{[b]^{\ell}\} \cup elementary(S) \\
elementary([\{]^{\ell_b}\ D\ S\ [\}]^{\ell_e}) &=& \{[\{]^{\ell_b}, [\}]^{\ell_e}\} \cup \\
&& elementary(D) \cup elementary(S)
\end{array}$$

Then the set of *labels* occuring in a statement is given by:

$$labels : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

where

$$labels(S) = \{\ell | [\alpha]^\ell \in elementary(S)\}$$

Clearly $init(S) \in labels(S)$ and $final(S) \subset labels(S)$.

The set of labels of return statements is given by:

$$retlabs(S) = \{\ell | [\text{return } a]^\ell \in elementary(S)\}$$

We fully integrate the return statement in our analysis in sections 3.2.2 and 3.3.2 when we extend the control flow graph and introduce the transfer functions for method entry and return.

### 2.1.3 Control Flow Graph

For our analysis it is customary to think of a program as a graph: the vertices are the elementary blocks and the arcs describe how control might pass from one elementary block to another. We will need to operate on these arcs, or control flows, between labels in a program, when analyzing a program. We define a function

$$flow : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$$

which maps statements to sets of flows:

$$
\begin{array}{rcl}
flow([p = a]^{\ell};) & = & \emptyset \\
flow([\text{return } a]^{\ell};) & = & \emptyset \\
flow([t\ L]^{\ell};) & = & \emptyset \\
flow([\text{skip}]^{\ell};) & = & \emptyset \\
flow(S_1\ S_2) & = & flow(S_1) \cup flow(S_2) \\
 & & \cup\{(\ell, init(S_2))|\ell \in final(S_1)\} \\
flow(\text{if}([b]^{\ell})\ \{S_1\} \text{ else } \{S_2\}) & = & flow(S_1) \cup flow(S_2) \\
 & & \cup\{(\ell, init(S_1))\} \cup \{(\ell, init(S_2))\} \\
flow(\text{while}([b]^{\ell})\ \{S\}) & = & flow(S) \cup \{(\ell, init(S))\} \\
 & & \cup\{(\ell', \ell)|\ell' \in final(S)\} \\
flow([\{]^{\ell_b}\ D\ S\ [\}]^{\ell_e}) & = & \{(\ell_b, init(D))\} \\
 & & \cup\{(\ell, init(S))|\ell \in final(D)\} \\
 & & \cup flow(D) \cup flow(S) \\
 & & \cup\{(\ell, \ell_e) \mid \ell \in final(S)\}
\end{array}
$$

Thus the *control flow graph* of a (sequence of) statement(s) $S$, CFG(S), can be defined as:

$$CFG(S) = (flow(S), labels(S)) = (V, A)$$

where $V$ is the set of vertices, and $A \subseteq V \times V$ is the set of arcs. $CFG(S)$ has a unique start vertex, which has no predecessors.

Clearly, $init(S)$ is the (unique) entry vertice for the control flow graph with arcs $A = labels(S)$ and vertices $V = flow(S)$. Also

$$A = \{init(S)\} \cup \{\ell|(\ell, \ell') \in V\} \cup \{\ell'|(\ell, \ell') \in V\}$$

and for composite statements (meaning those not simply of the form $[B]^{\ell}$) the equation remains true when removing the $\{init(S)\}$ component.

To simplify the formulation of the analysis method, it will be stated for a single fixed (but arbitrary) program.

### 2.1.4 The Single Fixed Program

We will use the notation $P_\star$ to represent the program we are analyzing, $\mathbf{Lab}_\star$ to represent the labels ($labels(P_\star)$) appearing in $P_\star$, $\mathbf{Var}_\star$ to represent the elementary blocks ($elementary(P_\star)$) occuring in $P_\star$.

Next consider the program $P_\star$ of the form $C_\star$ $B_\star$. The class declarations being of interest for intraprocedural analysis do not contribute to the control flow because default values of instance variables are defined to be initialized when a $newC$ operation is performed. This is integrated in the transfer functions for $[x = newC]^\ell$ which we shall introduce in section 2.4.3. Hence, we can define the flow for a program as the flow of the block $B_\star$ and for the entire program $P_\star$ we set:

$$\begin{array}{rcl} init_\star & = & init(B_\star) \\ final_\star & = & final(B_\star) \\ elemenary_\star & = & elementary(B_\star) \\ labels_\star & = & labels(B_\star) \\ flow_\star & = & flow(B_\star) \end{array}$$

As well as $\mathbf{L}\text{ab}_\star = labels_\star$.

We will redefine this setting for interprocedural analysis in section 3.

For the example program in figure 2.1 we get:

$init_\star = \{1\}$

$final_\star = \{9\}$

$elementary_\star$=$\{$[{]$^1$, [A a]$^2$, [int i]$^3$, [i = ...]$^4$, [i>0]$^5$, [a = new A]$^6$, [a = null]$^7$, [skip]$^8$, [}]$^9\}$

$labels_\star$=$\{$1,2,3,4,5,6,7,8,9$\}$

$flow_\star$=$\{$ (1,2), (2,3), (3,4), (4,5), (5,6), (5,7), (6,8), (7,8), (8,9) $\}$

The Program $P$ has isolated entries because we defined a program to consist at least of one single block. Since every block has a single entry (and single

```
class A extends O {
        A next;
        int v;
}
[{]^1;
        [A a]^2;
        [int i]^3;
        [i = ...]^4;
        if ([i>0]^5) {
              [a = new A]^6;
        } else {
              [a = null]^7;
        }
        [skip]^8;
[}]^9;
```

Figure 2.1: Program: If-Then-Else

exit) we follow the traditions of the literature that a program $P_\star$ has *isolated entries*; this means that:

$$\forall \ell \in \text{Lab} : (\ell, init(P_\star)) \notin flow(P_\star)$$

Similarly it holds that the program $P_\star$ has isolated exits; this means that

$$\forall \ell_1 \in final(P_\star) \forall \ell_2 \in \text{Lab} : (\ell_1, \ell_2) \notin flow(P_\star)$$

When we extend the language by methods and method calls in section 3 we shall ensure that these properties still hold.

A program, $P$, is *label consistent* if and only if:

$$[\alpha_1]^\ell, [\alpha_2]^\ell \in elementary(P) \rightarrow \alpha_1 = \alpha_2$$

Clearly, if all elementary blocks in $P$ are uniquely labeled (meaning that each label occurs only once), then $P$ is label consistent. When $P$ is label consistent the clause "where $[\alpha]^\ell \in elementary(P)$" is unambiguous in defining a partial function from labels to elementary blocks; we shall then say that $\ell$ *labels* the elementary block $\alpha$. We shall exploit this when defining the analysis below.

## 2.2 Access Paths and Node Naming Scheme

Due to the fact that data structures in the heap may become arbitrary large we have to deal with arbitrary long access paths. An access path always starts at some local variable which refers to an object and possibly continues on links between objects represented by values of instance variables.

A node in a flow-shape graph is denoted by a set of variables. We include a variable name $x$ in a node naming set $n$ if a variable $x$ directly *or* indirectly refers to the object that is represented by the node in the graph; i.e. an edge labeled with $x$ points to $n$ or to one predecessor node of node $n$ in the graph.

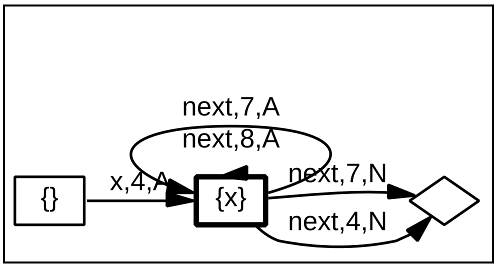



Figure 2.2: List of objects

**Example**. Let us assume a variable $x$ refers to a list of unknown length, each object refering to the next object by an instance variable next. The objects can be accessed using the access path expressions $x(.next)^*$; path expressions of unknown length. The objects in the flow-shape graph are denoted by {x}, with an edge refering to itself and representing a so called summary node (see fig. 2.2). The graph represents all accesspaths $x(.next)^*$. The accesspaths are represented by paths in the graph, each edge labeled with a local variable name or an instance variable name (additionally the edges are labeld with definition information and type information which we shall introduce in the next section).

Each node in the graph is denoted by a set of variables. In the above example each set consists of only one variable but this is not the general case. A node naming set may consist of up to $|LVar|$ variables.

**Definition 2.1** *A node $n$ in a flow-shape graph is denoted by a set of variable names formally $n \subseteq LVar$.*

Although sets of (local) variables are used to denote nodes in a flow-shape graph we are able to represent access paths longer than two. To represent access paths of arbitrary length, we use so called summary nodes.

When a flow-shape graph is altered due to adding or removing information to or from the graph it is necessary to “propagate” information through the graph because we repesent transitive dependence of variables on their successor nodes.

We define an auxiliary function *app* which shall be used frequently in transfer functions when renaming flow-shape graphs:

$$app(n, v) = \begin{cases} \{v\} & : \quad n = \{\} \\ n & : \quad \text{otherwise} \end{cases}$$

where $n$ is a node naming set denoting a node and $v$ is the name of a local variable or an instance variable. If $n$ is the root node $\{\}$ we include $v$ in the node naming set.

For the graph in figure 1.3 it holds that each node is denoted by a set of local variable names that can be obtained by using *app*. For example, the node $\{x\}$ in figure 1.3 has as in-going edge $[\{\}, (x, 4, A), \{x\}]$. It holds that $\{x\} = app(\{\}, x)$, where $x$ is the variable attribute of the label of the in-going edge of $\{x\}$. And it holds that $\{x\} = app(\{x\}, next)$. We will use this auxiliary function to keep the formalism compact and utilize the uniform representation of var-edges and sel-edges in the set of edges.

Alternatively, the notion of reachability can be used as well: for every node in the graph it holds that a node-set consists of *all* those variable names $v$, for which an access paths starting with $v$ reaches this node.

We shall see in the next section that this is a general property of flow-shape graphs.

## 2.3 Flow-Shape Graphs

We use graphs to represent the shape of data structures that may exist at a program point. We need to represent references and scalar values. The null reference will be represented similar to a scalar value utilizing the special node $\diamond$ (see section 2.3.1). All information regarding variable names, reaching definitions, and type information is represented as labels of edges.

**Remark**: We will systematically use the terms "nodes" and "edges" when refering to elements of flow-shape graphs, and "vertices" and "arcs" when refering to elements of control-flow graphs.

An essential property of a flow-shape graph is its node naming scheme and that the transfer functions are designed such that they maintain this scheme.

We first develop a terminology for kinds of edges and kinds of nodes in a flow-shape graph and proceed to define well named nodes and well named graphs based on this terminology.

### 2.3.1 Kinds of Edges

Edges in a flow-shape graph represent variable values or links in data structures in the heap. Data structures may have cycles and so may have flow-shape graphs. Several kinds of edges are necessary to represent and separate finite, possibly infinite, or cyclic data structures. The terminology for edges shall be used extensively when defining kinds of nodes based on their position in the graph.

We first define an edge and its labeling:

**Definition 2.2** *An edge is denoted by a triple of the form* $[s, a, e]$*, where* $s$ *and* $e$ *are nodes and* $a$ *is a triple of the form* $(v, d, t)$*, where* $v$ *is a name of a variable,* $d$ *is a label* $\ell$ *of a statement in the program where the value represented by the edge may be defined (assigned), and* $t$ *is a type.*

When we need to refer to the attributes of the edge label $a$, we write $a_v$ for the variable attribute, $a_t$ for the type attribute, and $a_d$ for the definition attribute respectively.

We use the following terminology for edges $[s, a, e]$ in a flow-shape graph:

| Name | Meaning |
|---|---|
| in-edge of node $n$ | $[s, a, e] \in Es \wedge e = n$ |
| out-edge of node $n$ | $[s, a, e] \in Es \wedge s = n$ |
| self-edge of node $n$ | $[s, a, e] \in Es \wedge s = e$ |
| var-edge in Es | $[s, a, e] \in Es \wedge a_v \in \text{LVar}$ |
| sel-edge in Es | $[s, a, e] \in Es \wedge a_v \in \text{IVar}$ |

In Figure 2.2 the out-going edge of node $\{\}$ is a var-edge. All nodes with attribute $a_v = next$ are $sel$-edges.

Given a set of edges $Es$ we define the set of nodes $N(Es)$ as follows:

$$N(Es) = \{s : [s, a, e] \in Es\} \cup \{e : [s, a, e] \in Es\}$$

We shall make frequent use of the function $succ^*(n, Es)$ to access the set of successor nodes of the node $n$ in the set of edges $Es$ ($succ^*(n, Es)$ always includes node $n$).

Based on the terminology for edges we define kinds of nodes in the next section.

### 2.3.2 Kinds of Nodes

We use four different kinds of nodes in a flow-shape graph. A root node which does not represent any object and is used as source for local variable edges. To represent single objects we use so called single nodes, and summary nodes to represent data structures of unknown size. To integrate scalar values in the graph we use the special node $\diamond$ which serves as a target for edges in the graph that do not represent references to objects. We represent scalar values and the null-reference by edges refering to node $\diamond$.

Which variable names have to be elements of a node-set depends on the edges refering to it. This can be understood as information flow from one node to all its successor nodes or each node representing a set of local variables of access paths reaching it. It is obvious that altering the graph requires to rename the nodes to reflect the adding or removing of information (edges and nodes). Informally, we say that a node is *well named* if its node-set bears all the approximated information to represent all paths leading to it.

Objects are represented by single nodes or summary nodes in the graph, values are represented by edges. We have two special nodes, the root node $\{\}$ and the non-object node $\diamond$ that do not represent objects.

### Root node $\{\}$

An approach that has been widely used in the literature is to use two kinds of nodes, variable and shape-nodes - and two kinds of edges, variable edges and selector edges. In [SRW96, SRW98] two separate sets of edges are used to deal with these two different kinds of edges, in [AZ97, AZ98] one set of edges but (implicitly) two sets of nodes are used to distinguish these two kinds of nodes. A shape node represents an object or a set of objects whereas a variable node represents a variable.

We use a slightly different formal representation. There is exactly one *root node* that does not represent any object nor does it represent a variable. Its purpose is to serve as source for variable edges. Edges emanating from the root node represent values stored in local variables.

### Non-Object node $\diamond$

The node $\diamond$ is a special node whose naming is independent from predecessor nodes and it does not have any successor nodes. The node naming set of the non-object node $\diamond$ consists of exactly one name that is distinct from all other variable names. It serves as artificial target for edges representing scalar values (which are not references to objects but are represented as edges) or as target for edges representing the null-value.

For reasons of better presentation of the alias information we visualize the node $\diamond$ several times in a graph although there is only *one* non-object node.

Edges representing scalar values are distinguished from edges representing null-values by its type attribute.

### Summary Nodes

If a node in a graph may represent more than one object of a data structure it "summarizes" the data structure and we call it a summary node. The idea is to collapse the data structure into one single node and to represent all links between the collapsed objects by self-edges of the summary node.

In [SRW98] a single summary node is used to collapse all nodes that are not directly refered to by a variable into one summary node. We use *multiple* summary nodes and collapse only those objects into the same summary node that are indirectly refered to by the *same set* of variables. The use of multiple summary nodes is vital for a seamless integration of reaching definition information and type information in our approach.

**Definition 2.3** *A node n in a flow-shape graph* $(Es, is)$ *is a summary node if it has a self-edge; formally, it must hold that:* $\exists[n, a, n] \in Es$

We shall use a predicate $is\_summary\_node_{Es}(n)$ that holds if node $n$ is a summary node in $Es$ for a given flow shape graph $G = (Es, is)$. A summary node may represent a non-cyclic data structure or a cyclic data structure. In both cases the self-edge of the summary node represents an unknown number of access paths but it depends on the additional explicit sharing information $is$ whether a node represents a non-cyclic or a cyclic data structure. If a node n is marked to be induced shared, $n \in is$ (is induced shared), then more than one $sel - edge$ may refer to at least one of the objects represented by the summary node. We shall discuss this in detail in section 2.3.3. In figure 1.3 the node $\{x\}$ is a summary node that represents a list of unknown length with the property that the *next* field does not create any sharing.

### Single nodes

A single node represents one single object in the heap.

**Definition 2.4** *A node n in a flow-shape graph* $(Es, is)$ *is a single node if it is refered to by at least one edge and it is not a summary node and it is different to the non-object node* $\diamond$*; formally* $\exists[s, a, n] \in Es \wedge \neg is\_summary\_node_{Es}(n) \wedge n \neq \diamond$.

The node $\diamond$ is excluded from the set of single nodes in a flow-shape graph because it does not represent an object; similar for the root node $\{\}$ which does not represent an object.

If a single node $n$ is induced shared, $n \in is$, more than one $sel$-edge may refer to node $n$, but it does not represent a cycle.

Using this terminology we can say that an object is represented either by a single node or a summary node and that values are represented by edges in a flow-shape graph.

### 2.3.3 Kinds of Sharing

An object in the heap may either be shared by local variables or by instance variables (or by a combination of both). Nodes which represent objects that may be shared by instance variables are marked. Formally, we use an additional component to the set of edges, a set of nodes $is$, which consists of all marked nodes. Hence, a flow-shape graph $G$ is denoted by a pair of edges $Es$ and a set of marked nodes $is$, $G = (Es, is)$.

Only those nodes may be marked that are refered to by at least two $sel$-edges or by a $sel$-edge that may represent more than one reference *and* these sel-edges represent references that may share the object (or set of objects, if the node is a summary node). We shall introduce an auxiliary function $iis$ that allows to determine when nodes have to be marked to be induced shared.

The advantage of this explicit representation of sharing is that if we have not marked nodes to be shared with more than two sel-edges refering to it, we can treat the edges refering to the node as references to *different* objects. This allows us to represent lists, trees, and graphs without the need of using different types of nodes.

In the example graph in figure 1.3 the set $is$ is empty. No node is marked to be shared by sel-edges. Therefore the node $\{x\}$ represents different objects and the edge $[\{x\}, (next, 7, A), \{x\}]$ represents references to different objects but never to itself. Hence, it does not represent a cyclic data structure. Because these edges represent references that do never refer to the same object and because there is only one self-edge we can conclude that the data structure represented by the graph is a list.

This is in contrast to use different types of nodes to represent different kinds of data structures. In [GH96b] different types of data structures (trees,dags,arbitrary graphs) are considered explicitely. Our approach is similar to that used in [SRW98]; we use an explicit representation of sharing. In contrast to [SRW98] we use multiple summary nodes and integrate type information for each edge in the graph. Using the simple fact that values of different types (declared in $P_\star$) that are not in a subtype relationship cannot be aliased, can be used to reason further on the shape of data structures represented by a flow-shape graph. We use the type information for method invocation resolution which we shall develop in section 3.

### 2.3.4 Property Well Named

Using the terminology presented above we can define properties that must hold on well named nodes.

**Definition 2.5** *A node $n$ in a FSG $G = (Es, is)$ is* well named *if $n \in succ^*(\{\}, Es)$ holds and it has one of the following properties:*

1. $n \in \{\{\}, \diamond\}$
2. $n \in is \wedge n \notin \{\{\}, \diamond\} \wedge n = \bigcup\{app(s, a_v) | [s, a, e] \in Es \wedge e = n\})$
3. $$\begin{array}{l} n \notin \{\{\}, \diamond\} \cup is \\ \wedge\ \{a_v | [\{\}, a, e] \in Es \wedge e = n\} \subseteq n \\ \wedge\ (\forall [s', a', e'] \in \{[s, a, e] : [s, a, e] \in Es | e = n \wedge s \neq \{\}\} \\ \quad : \ n \ = \ app(s', a'_v) \cup \{a''_v | [\{\}, a'', n] \in Es\}) \\ \wedge\ (n - \bigcup\{app(s, a_v) | [s, a, n] \in Es\} = \emptyset) \end{array}$$

Denoting $app(s, a_v)$, in the definition of a well named node above, as "information of an edge" we can summarize the three aspects of the definition informally as follows:

1. The root node $\{\}$ and the non-object node $\diamond$ are well named nodes.

2. If a node $n$ is marked to be shared, $n \in is$, then $n$ must represent the unioned information of all in-going edges (a self-edge is also an in-going edge).

3. If a node $n$ is not marked to be shared, $n \notin is$, the unioned information of all in-going var-edges must be a subset of $n$. All variables that refer directly to the object represented by $n$ must be included in the node naming set $n$. For each in-going sel-edge of $n$ it must hold that the information of the in-going edge equals the information of the node; and no other information is represented by the node.

Now we can define well named flow-shape graphs.

**Definition 2.6** *A graph $FSG = (Es, is)$ is a flow-shape graph if for all nodes $n \in Es$ holds that $n$ is a well named node, formally $(\forall n \in N(Es) : n\ is\ wellnamed)$.*

The class of flow-shape graphs is denoted by $\mathcal{FSG}$. All transfer functions are designed such that they operate on well named flow-shape graphs. If the input graph is well named then the output graph is also well named. All graphs created by our analysis are well named flow-shape graphs.

### 2.3.5 Equivalence of Nodes

Two single nodes represent different objects if they have different node-sets. Summary nodes in a flow-shape graph may represent more than one object. Therefore one summary node *may* represent two different objects. Formally

| | |
|---|---|
| $n_1 \equiv_{Es} n_2$ | $n_1 = n_2$ |
| $n_1 \not\equiv_{Es} n_2$ | $n_1 \neq n_2 \vee (n_1 = n_2 \wedge is\_summary\_node_{Es}(n_1))$ |

If the node naming sets are equal both nodes may represent the same objects. Whether two nodes $n_1$ and $n_2$ may represent different objects, $n_1 \not\equiv_{Es} n_2$, depends on whether a node is a single node or a summary node. If $n_1 \neq n_2$ the objects represented by the nodes may be different. If the nodes are equal, $n_1 = n_2$, they may represent different objects if the node is a summary node because a summary node represents more than one object.

### 2.3.6 Compatibility of Nodes

Compatible nodes represent objects that may exist at the same time during the execution of a program. Edges need only to be established between nodes representing objects that may exist at the same time during execution of the program being analyzed. We shall use the predicate *compatible* to determine whether we have to consider nodes to be candidates for establishing edges between them in a FSG. Edges

We shall use a predicate $con(n_1, n_2, Es)$ that holds if a path exists from $n_1$ to $n_2$ or from $n_2$ to $n_1$ in a given graph Es.

Let $Es$ be the set of edges of a FSG and $n_1$ to $n_k$ be nodes of the FSG. $compatible_{Es}(n_1, \ldots, n_k)$ holds if

$$\begin{aligned} \forall_{i,j} \quad : \quad & n_i = n_j \\ & \vee \{n_i, n_j\} \cap \{\{\}, \diamond\} \neq \emptyset \\ & \vee ((n_i \cap n_j) \neq \emptyset) \rightarrow con(n_i, n_j, Es)) \end{aligned}$$

Two nodes are compatible if they are equal or if they do not share any names. If they do share names then they are compatible if exists a path between them in $Es$.

The predicate *compatible* holds if two nodes $n_1$ and $n_2$ represent objects that may exist at the same time during execution of a program. We shall use this predicate to determine whether an edge can be established between to nodes to represent a new alias in a graph.

Example: In fig. 2.3 position 4, the nodes $\{x, y\}$ and $\{x\}$ are not compatible; $compatible_{Es}$({x},{x,y}) does not hold because the node naming sets are not disjoint but are not in a predecessor or successor relation (*con*) in the graph. The variable $x$ may only hold one reference when the program is executed. It either refers to an object that is refered to by $y$ as well or it refers to an object that is not refered to by $y$, but not both. Therefore these two nodes are not compatible. The nodes $\{x\}$ and $\{y\}$ are compatible because the node naming sets are disjoint.

### 2.3.7 Induced Shared

In section 2.3.3 we discussed the two kinds of sharing that are kept implicitly and explicitely in a FSG. We refine the explicite representation of sharing in this section by using equivalence, compatibility, and connectivity in a FSG to determine whether a node has to be marked. Every transfer function that removes or adds edges (and/or nodes) to or from a FSG must update the *is* status of a FSG. In most cases this is done by using the *iis* function which we shall introduce now.

The induced-sharing status of a node $n$ only concerns the sharing via sel-edges (edges that represent values of instance variables; edges not starting at the root node). Edges representing variable values (var-edges), do *not* contribute to the induced-shared status of a node. Hence, nodes that are refered to by more than one var-edge are shared but this information is not represented in *is*. Instead, it is implicitly represented in the graph. A node that is refered to by more than one sel-edge may be induced-shared or not.

If data structures are created and objects are pointed to by var-edges and connected by sel-edges such that not more than one sel-edge refers to an object and no cycle is created - then nullifying the var-edges may create a graph with more than two sel-edges refering to a node $n$ but having $n$ not marked as shared.

We set the induced-shared status of a node by adding the node to the *is* set of a FSG. To determine the status we use the predicate *iis* that holds if there exist in-going sel-edges of $n$ either emanating from inequivalent nodes or with different variable attributes $a_v$ of the edge labels.

| | |
|---|---|
| $iis_{Es}(Z)$ | $Z \neq \diamond \wedge \exists [s_1, a, e] \in Es, \exists [s_2, a', e] \in Es$<br>$\lvert s_1 \neq \{\} \wedge s_2 \neq \{\}$<br>$\wedge\ compatible_{Es}(s_1, s_2, e)$<br>$\wedge\ (s_1 \not\equiv_{Es} s_2 \ \vee a_v \neq a'_v)$ |

Please note that this includes the situation that only one edge exists but emanates from a summary node and hence may *represent* more than one edge that emanate from nodes that *may* represent different objects. I.e., an out-edge of a summary node with a node $n$ as target node represents the information that from any node represented by the summary node an edge may refer to the node $n$.

### 2.3.8 The Complete Lattice of Flow-Shape Graphs

In the following definition, we impose an order on FSGs where $FSG_1 \sqsubseteq FSG_2$:

**Definition 2.7** *Let $FSG_1 = (Es_1, is_1)$ and $FSG_2 = (Es_2, is_2)$. We define the following ordering on $\mathcal{FSG}$: $FSG_1 \sqsubseteq FSG_2$ if and only if all of the following conditions hold:*

1. $E_1 \subseteq E_2$
2. $is_1 \subseteq is_2$

The domain $\mathcal{FSG}$ is a complete join semi-lattice with a join operator $\sqcup$ defined by

$$FSG_1 \sqcup FSG_2 = (Es_1 \cup Es_2, is_1 \cup is_2).$$

The binary join operation, $\sqcup$, is idempotent, commutative and associative, i.e. $G \sqcup G = G$, $G_1 \sqcup G_2 = G_2 \sqcup G_1$, and $(G_1 \sqcup G_2) \sqcup G_3 = G_1 \sqcup (G_2 \sqcup G_3)$.

Let $G$ be a FSG with $G = (Es, is)$. Then an equivalent FSG $G'$ can be defined such that the set of all definition-type pairs, $(d, t)$, which are attached to edges between two nodes $s_1$ and $e_1$ with the same v-attribute can be attached to a single edge in G', formally

$$\begin{aligned} G' \;=\; & (\{[s, (a_v, \{(a'_d, a'_t) | [s', a', e'] \in Es \wedge s' = s \wedge e' = e \wedge a'_v = a_v\}), e] \\ & | [s, a, e] \in Es\}, is) \end{aligned}$$

We map sets of edges $[s, a, e]$ in $G$ to single edges in $G'$ such that edges $[s, a, e]$ in $G$ with the same var-attribute $a_v$ are mapped to the same edge $[s, (a_v, DT), e]$ in $G'$ with $DT$ equal to the set of all definition-type pairs that are the $a_d$ and $a_t$ attributes of the corresponding edges in $G$.

For example in the flow-shape graph in figure 1.1 we have two edges representing a value of variable $x$ from the root node {} to node {p,q}. These two edges, $[\{\},(q,14,T),\{p,q\}]$ and $[\{\},(q,21,T),\{p,q\}]$ are mapped to the edge $[\{\},(q,\{(14,T),(21,T)\}),\{p,q\}]$.

We shall make use of this mapping for statistical evaluation of our method in section 2.5.

## 2.4 The Analysis

The analysis will be specified as an instance of a Monotone Framework with the complete lattice of properties being $\mathcal{P}(FSG)$. For each label consistent program $S_\star$ with isolated entries we obtain a set of equations of the form

$$\begin{array}{rcl} Analysis_\circ(\ell) & = & \bigsqcup\{Analysis_\bullet(\ell')|(\ell',\ell)\in F\}\sqcup \iota_E^\ell \\ & & \text{where } \iota_E^\ell = \begin{cases} \iota & : \text{ if } \ell \in E \\ \perp & : \text{ if } \ell \notin E \end{cases} \\ Analysis_\bullet(\ell) & = & f_\ell(Analysis_\circ(\ell)) \end{array}$$

where $\iota \in \mathcal{P}(FSG)$ is the initial analysis information and $\perp$ satisfies $G \sqcup \perp = G$; $F$ is the finite flow $flow_\star$ and $E = init_\star$. $f_\ell$ is the transfer function associated with a statement. The transfer functions are developed below. The space of functions $\mathcal{F}$ over $\mathcal{FSG}$ contains all the transfer functions $f_\ell$ in question, the identity function $id$, and $\mathcal{F}$ is closed under composition of functions. The mapping $f.$ from the labels $\text{Lab}_\star$ of $F$ and $E$ to transfer functions in $\mathcal{F}$ associates a transfer function $f$ with a label $\ell \in \text{Lab}_\star$.

The analysis is a forward analysis since it is defined in terms of the set $flow_\star$ and it is a may analysis due to the definition of the join operator $\sqcup$. However, there are also aspects of must-alias analysis because each individual shape graph must not contain any superfluous information. This will be useful for achieving strong update and strong nullification; here strong updates means that an update or nullification of a pointer expression allows one to remove the existing binding before adding a new one. This in turn leads to a very powerful analysis.

### 2.4.1 Transfer Function for $[b]^\ell$ and $[skip]^\ell$

The boolean tests do not create any definitions nor do they modify the heap. We take

$$f_\ell((E, is)) = (E, is)$$

so that the transfer function $f_\ell$ will be the identity function. Similarly for the skip-statement.

### 2.4.2 Transfer Function for $[x = null]^\ell$

The effect of this assignment will be to remove the binding to $x$, if the variable $x$ is a reference variable but not a recursion variable and to rename all nodes so that they do not include any $x$ in their node naming set.

When $x$ is a local variable of a function that may be called recursively, we call such a variable a *recursion variable*. Whether a function is recursive is determined by the ICFG defined in section 2.1.3 and section 3.2. We do not perform a strong update on recursion variables. Finally we present the transfer function for the case that $x$ is a scalar variable.

A node that is refered to by a var-edge labeled with $a_v = x$ includes $x$ in its node naming set. Every successor node of such a node has $x$ in its node naming set representing a transitive dependence. The assignment $x = a$ nullifies the variable and hence, kills all direct or indirect dependencies. We express this operation by renaming all nodes such that no $x$ remains in any node naming set of the entire graph. One edge, representing the null value itself, must be present in the graph to represent the null-value. This edge is added to the graph after having removed all the edges represent the value $x$. This is in contrast to approaches that do not explicitely integrate a null-reference as value in shape graphs [AZ97, AZ98, SRW96, SRW98, NNH99].

The renaming of a node $n$ is specified by the function

$$k(x, n) = n - \{x\}$$

and we then take

$$\begin{aligned} f_\ell((Es, is)) &= (Es'', is'') \\ (Es', is') &= kill_x(Es, is) \\ (Es'', is'') &= (Es' + \{[\{\}, (x, \ell, N), \diamond]\}, is') \end{aligned}$$

where $kill_x((E, is)) = (Es', is')$ is given by

$$\begin{aligned} Es' &= \{[k(x,s), a, k(x,e)] | [s,a,e] \in Es \\ &\quad \wedge a_v \neq x \wedge (k(x,s) \neq \emptyset \vee s = \{\}) \wedge k(x,e) \neq \emptyset\} \\ is' &= \{k(x,n) | n \in is \wedge iis(Es', k(x,n))\} \end{aligned}$$

Variable $x$ must not be a recursion variable, so that we obtain strong nullification. If $x$ is refering to an object the reference is removed by $kill_x$. The function $kill_x$ defines how a flow-shape graph is mapped such that the renaming of the nodes represents the removing of all edges that have $x$ as edge label $a_v$.

In the case that $x$ is a recursion variable we do not perform a strong nullification and $kill_x$ is the identity function.

We treat a null value similar to a scalar value and do not create any aliases. Hence, we only need to add edges to the graph to represent the new definition information created by the assignment itself ($[\{\}, (x, \ell, N), \diamond]$). We use the Null type (denoted $N$) as type information which is assumed to be the type of the null-value (like in Java [GJSB00]).

**Theorem 2.1** *The function $kill_x$ is monotone.*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds and let $[s, a, e]$ be any edge with $[s, a, e] \in E_1 \cup E_2$.

1. case $s = r, a_v = x$: Since only edges with $a_v \neq x$ are mapped by $kill_x$, never edges with $a_v = x$ are elements of $E_1'$ or $E_2'$ (strong nullification).

2. case $s = r, a_v \neq x, x \in e$: Every edge $[s, a, e]$ is mapped to $[\{\}, a, e - \{x\}]$. Since $a_v \neq x$ it holds that $e - \{x\} \neq \emptyset$ and therefore $[r, a, e - \{x\}] \in Es'_1$ and $[r, a, e - \{x\}] \in Es'_2$.

3. case $s \neq r$: $s \neq r$ implies that $a_v \neq x$ ($[s, a, e]$ is a sel-edge). Every edge $[s, a, e]$ with $s \neq r$ is mapped to an edge $[s - \{x\}, a, e - \{x\}]$. Hence $[s - \{x\}, a, e - \{x\}] \in E'_1$ and $[s - \{x\}, a, e - \{x\}] \in E'_2$.

All marked nodes $n \in is_1 \cap is_2$ are mapped to nodes $n - \{x\}$. Hence $n - \{x\} \in is'_1$ and $n - \{x\} \in is'_2$.

It follows that $Es'_1 \subseteq Es'_2$ and $is'_1 \subseteq is'_2$ and it holds for $kill_x$ that $(Es'_1, is'_1) = kill_x((Es_1, is_1)) \sqsubseteq kill_x((Es_2, is_2)) = (Es'_2, is'_2)$.

Therefore, $G_1 \sqsubseteq G_2 \rightarrow kill_x(G_1) \sqsubseteq kill_x(G_2)$. Function $kill_x$ is a monotone function. $\Box$

In the case that $x$ is a recursion variable we do not perform a strong nullification and $kill_x$ is the identity function, and therefore monotone.

**Theorem 2.2** *The function $f_\ell$ is a monotone function*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds. According to theorem 2.1 the function $kill_x$ is monotone and for $f_\ell$ we have $(Es', is') = kill_x(Es, is)$. For any two graphs $Es'_i, i \in \{1, 2\}$ the graph $Es''_i = Es'_i \cup \{[r, (x, \ell, t), \diamond]\}$. Hence $Es''_1 \subseteq Es''_2$; and the sharing information is not effected, $is''_i = is'_i$.

Therefore, $G_1 \sqsubseteq G_2 \rightarrow f_\ell(G_1) \sqsubseteq f_\ell(G_2)$. Function $f_\ell$ is a monotone function. $\Box$

The renaming of the flow-shape graph is particularly interesting when $[\{\}, x, \{x\}] \in E$. The node $\{x\}$ is exclusively pointed to by an edge labeled with $a_v = x$. The nullification of $x$ will cause the node $\{x\}$ to be *removed* from the graph. This is due to the fact that we perform a garbage collection. Because the nullification of a variable can cut off a whole data structure represented by parts of the FSG we need to update the sharing information $is$ as well.

Two cases are possible why sharing may be reduced by $iis$: Firstly, we must remove the sharing information of nodes that are garbage collected because if nodes with the same name are introduced again they would be considered to be shared.

Secondly, the sharing of nodes that are refered to from nodes that are garbage collected may be reduced to one sel-edge. In this case it is safe to set the sharing information to non-shared and remove this node from the set $is$. In general, $x = null$ can never increase the sharing in a flow-shape graph.

### 2.4.3 Transfer Function for $[x = new\ C]^\ell$

In our programming model we assume that instance variables of an object are initialized "when" an object is created and therefore we use as labeling of all default values of instance variables the label $\ell$ of the new-statement. We could also use a label $\ell'$ different from all other labels in the program. This would model one single value representing an uninitialized value. Due to strong update an edge representing the default value of an instance variable is removed whenever a value is assigned to the instance variable. The interesting case is when we infer that a default value may not have been updated but used. For example, in Java local variables have to be initialized before being used. We may use the definition information to extend this check to instance variables and give informations to the user that an instance variable that is used may not have been initialized if a definition of an instance variable of a new-statement reaches a use in an other statement.

To model the semantics of the new statement we have to remove the binding to $x$ and then introduce a new (unshared) object pointed to by $x$.

Thus we define

$$f_\ell((E, is)) = g_{new}(\ell, C, (E, is))$$

and

$$g_{new}(\ell, C, (Es, is)) = (E' + \{[\{\}, [x, \ell, C], \{[x]\}]\} + DV, is')$$

where $(E', is') = kill_x(x, (E, is))$ and we set definition information and type information for default values $DV$ to be

$$DV = \{[\{[x]\}, [y, \ell, t'], \diamond] | [t, y] \in instancevars(C)\}$$

with $t' = t$ if $t$ is the type of a scalar variable and $t' = N$ (Null Type) if $t$ is a reference type.

The integration of a default value of an instance variable is formally similar to the integration of a declaration of variable (see section 2.4.10). For declared local variables we use $a_v = \ell_0$, whereas for default values of instance variables of created objects we use $a_v = \ell$ and *assume* the default value being assigned when the object is created. In the case of declared local variables $\ell_0$ marks a value to be uninitialized.

We add an edge to the graph emanating from $x$ to a node representing the new object. The instance variables of the object are initialized with default values by creating the set $DV$ of edges emanating from node $\{[x]\}$. Constructor methods are not considered (or are assumed to be empty methods and are modeled by the identity function). Interprocedural aspects are discussed in section 3.

**Theorem 2.3** *The transfer function $f_\ell$ is monotone.*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds. According to theorem 2.1 the function $kill_x$ is monotone and for $f_\ell$ we have $(Es', is') = kill_x(Es, is)$. In any case we add edges $DV$ to $Es'$. Hence, $Es'_1 \subseteq Es'_2$ holds. The sharing information is not effected by adding the edges $DV$ and therefore $G_1 \sqsubseteq G_2 \rightarrow f_\ell(G_1) \sqsubseteq f_\ell(G_2)$. Function $g_{new}$ is monotone; function $f_\ell$ is a monotone function. □

### 2.4.4 Transfer Function for $[x = y]^\ell$

First suppose that $x \neq y$. The first effect of the assignment is to remove the old bindings to $x$; for this we use the $kill_x$ operation introduced above. Then the new binding to $x$ is recorded; this includes renaming the node that includes $y$ in its name set to also include $x$ in its name set. The renaming of the nodes is specified by the function:

$$gc_{x,y}(n) = \begin{cases} n \cup \{x\} & : \quad y \in n \\ n & : \quad \text{otherwise} \end{cases}$$

We shall then take

$$f_\ell((Es, is)) = g_{11}(\ell, x, y, (Es, is)) = (Es'', is'')$$

where

$$\begin{array}{rcl} (Es', is') & = & kill_x((Es, is)) \\ Es'' & = & \{[gc_{x,y}(s), a, gc_{x,y}(e)] | [s, a, e] \in Es'\} \\ & \cup & \{[s, (x, a_d, a_t), gc_{x,y}(e)] | [s, a, e] \in Es' | a_v = y\} \\ & \cup & \{[s, (x, \ell, a_t), gc_{x,y}(e)] | [s, a, e] \in Es' | a_v = y\} \\ is'' & = & \{gc_{x,y}(n) | n \in is'\} \end{array}$$

so that we obtain strong update (if $x$ is not a recursion variable as above). Here the second clause in the formula for $Es''$ adds the new binding to $x$ and creates edges representing the reaching definitions from $y$ and the newly created definitions $\ell$.

Note that the nodes pointed by $x$ inherit the sharing information of the appropriate nodes pointed to by $y$ although both $x$ and $y$ will point to the same node. Although this adds a pointer to the same node in the graph the sharing information is *not* effected because edges that are labeled by a variable do not contribute to the shared status of a node.

Next suppose $x = y$. We take

$$\begin{array}{rcl} Es'' & = & \{[s, (x, \ell, a_t), e] | [s, a, e] \in Es \wedge a_v = x\} \\ is'' & = & is \end{array}$$

For an assignment $[x = x]^\ell$ we only need to add the definition information for the label $\ell$ and the sharing information is not effected as for the case $x \neq y$.

**Theorem 2.4** *The transfer function $f_\ell$ is monotone.*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds.

1. case $x \neq y$: According to theorem 2.1 the function $kill_x$ is monotone and for $f_\ell$ we have $(Es', is') = kill_x(Es, is)$. Next the graph is renamed using the auxiliary function gc. We identify three cases how edges are mapped. Let edge $[s, a, e] \in Es'_1 \cap Es'_2$.

   (a) case $s = \{\}$,$a_v = y$: $a_v = y$ implies that $y \in e$. Every edge $[\{\}, a, s]$ with $a_v = y$ and $y \in e$ is mapped to an edge $[\{\}, a, s \cup \{x\}]$. Therefore, $[\{\}, a, s \cup \{x\}] \in Es''_1$ and $[\{\}, a, s \cup \{x\}] \in Es''_2$.

   (b) case $y \notin s$, $a_v \neq y$, $y \in e$: Every edge $[s, a, e]$ is mapped to $[s, a, e \cup \{x\}]$ and therefore $[s, a, e \cup \{x\}] \in Es''_1$ and $[s, a, e \cup \{x\}] \in Es''_2$.

   (c) case $y \in s$: $y \in s$ implies that $s \neq \{\}$ and $y \in e$. Hence, every edge with $y \in s$ is mapped to $[s \cup \{x\}, a, e \cup \{x\}]$ and therefore $[s \cup \{x\}, a, e \cup \{x\}] \in Es''_1$ and $[s \cup \{x\}, a, e \cup \{x\}] \in Es''_2$.

   Next, if exists an edge $[r, a, e] \in Es'_i$, $i \in \{1, 2\}$, with $y \in e$, edges $[r, a', e']$ and $[r, a'', e']$ are established such that $e$ is mapped to $e'$ as in case 1a and $a' = (x, a_d, a_t)$ with $a_d$ being the corresponding reaching definition of the edge representing the existing values of $y$ and $a_t$ the type attribute, respectively. With $a'' = (x, \ell, a_t)$ the new definitions are introduced. Therefore, for the established edges $[r, a', e']$ and $[r, a'', e']$ it holds that these edges are in $Es''_1$ and in $Es''_2$. $is''$ is renamed as nodes in cases 1a-1c (by auxiliary function gc). Hence, it holds that $Es'_1 \subseteq Es'_2$ and $is'_1 \subseteq is'_2$; $G'_1 = f_\ell(G_1) \sqsubseteq G'_2 = f_\ell(G_2)$.

2. case $x = y$: For each edge $[s, a, e] \in Es_1 \cap Es_2$ with $a_v = x$ an edge $[s, (x, \ell, a_t), e]$ is added to the graph. Hence, $[s, (x, \ell, a_t), e] \in Es''_1$ and $[s, (x, \ell, a_t), e] \in Es''_2$. The sharing information is not effected, $is'' = is$.

Therefore, $G_1 \sqsubseteq G_2 \rightarrow f_\ell(G_1) \sqsubseteq f_\ell(G_2)$. Function $g_{11}$ is monotone; function $f_\ell$ is a monotone function. $\square$

### 2.4.5 Transfer Function for $[x = y.sel]^\ell$

This FSG transformer is one of the most elaborate operations. The reason for this is that the FSG transformer has to create an FSG that conservatively covers all the possible new configurations of variable sets whose members all may point to the same object after the assignment.

First we introduce three auxiliary functions that shall be used to specify the semantics of the transfer function for $x = y.sel$.

For renaming nodes we take

$$hd(n, set, N_s) = \begin{cases} n \cup set & : \quad n \in N_s \wedge n \neq \diamond \\ n & : \quad \text{otherwise} \end{cases}$$

The function *insert* will need to rename all nodes in a flow-shape graph that are successor nodes of the target node of an inserted edge and will use the auxiliary function *hd*.

The auxiliary function *hd* renames the node naming set $n$ to include the set *set* of variable names if $n$ is a member of a set of nodes $N_s$ and $n$ is different to the non-object node $\diamond$. The function *insert* will define the set $N_s$ to be the set of successor nodes of the target node of an inserted edge.

The function *insert* adds an edge to a given flow-shape graph $(Es, is)$, renames the nodes to represent the added information of the inserted edge, and updates the sharing status *is*.

We take

$$insert(s, a^1, a^2, e, (Es, is)) \quad = \quad (Es', is')$$

where

$$\begin{array}{rcl} N_s & = & succ^*(e, Es \cup \{[s, a^1, e], [s, a^2, e]\}) \\ Es' & = & \{[hd(s', app(s, a^1_v), N_s), a', hd(e', app(s, a^1_v), N_s)] \\ & & \quad \mid [s', a', e'] \in Es \cup \{[s, a^1, e], [s, a^2, e]\}\} \\ is' & = & \{hd(n, app(s, a^1_v), N_s) | n \in is'\} \\ & \cup & \begin{cases} \{hd(e, app(s, a^1_v), N_s)\} & : \quad s \neq \{\} \wedge iis(Es'', hd(e, app(s, a^1_v), N_s)) \\ \emptyset & : \quad \text{otherwise} \end{cases} \end{array}$$

The major task of *insert* is to rename the graph such that the information of the source-node of the edge is propagated to its successor nodes taking the information of the inserted edge into account. In fact, the information $a_v$ of the inserted edge is only considered when the source-node of the edge is the root node $\{\}$. The information of the source-node is propagated in the graph. Secondly, two edges, $a^1$ and $a^2$ are inserted. We need one edge, $a^1$, representing the reaching definition of the aliased edge and $a^2$ for representing the created definition information of the inserted edge. Clearly, two edges are added to the graph, to represent two different definition informations but both "belong" to the same nodes (that is, with the same source and target nodes).

A summary node which is marked to be shared may represent a cyclic data structure. If a summary node is not marked to be shared it never represents a cyclic data structure. A single node can be marked to be shared (by more than one sel-edge). This does not represent a cycle. In a well named flow-shape graph cycles are always represented by solitary summary nodes.

Having introduced the auxiliary functions we can now define the transfer function and take

$$f_\ell((Es, is)) = g_{12}(\ell, x, y, (Es, is))$$

First assume that $x \neq y$ and let (Es,is) be a FSG before the analysis of the statement and we set

$$g_{12}(\ell, x, y, (Es, is)) = (Es'', is'')$$

As for the previous transfer function the first step will be to remove the old binding for $x$ and again we use the auxiliary function $kill_x$:

$$(Es', is') = kill_x((Es, is))$$

The next step will be to rename the nodes corresponding to all nodes directly reachable by $x$ to include $x$ in their name and establish a binding of $x$ to those nodes. Furthermore we need to rename all nodes that are successors of those nodes by including $x$ in their name. This task is complicated when multiple summary nodes exist in a graph that require that multiple summary nodes are to be renamed.

$$\begin{array}{rcl}
(Es', is') & = & kill_x(x, (Es, is)) \\
(Es'', is'') & = & \bigsqcup\{insert(\{\}, (x, a''_d, a''_t), (x, \ell, a''_t), e'', (Es', is')) \\
& & \quad |[\{\}, a', e'] \in Es', [e', a'', e''] \in Es' \\
& & \quad \wedge y \neq \diamond \wedge a'_v = y \wedge a''_v = sel\} \\
& \cup & (Es', is')
\end{array}$$

We can now identify two possibilities:

1. There exists an edge $[\{\}, [y, d, t], \diamond]$ in the graph representing a null value. Definition $d$ and and type $t$ are irrelevant then; in this case we need to model the fact that a null value is dereferenced. Since there is no value that can be assigned, no value can be copied and assigned to $x$. Therefore it is safe not to create a new edge because never a value can be assigned. If there is at least one execution path then a value is assigned and a definition $\ell$ created which is covered by case two.

2. It exists a node $e'$, such that $[s', a', e'] \in Es'$ with $a'_v = y$ and a node $e''$, such that $[e', a'', e''] \in Es'$ with $a''_v = sel$. Hence, a path $y.sel$ exists in the graph which needs to by aliased by $x$. The function $insert$ handles various possibilities. The node $e'$ can be a single node or a summary node. Also the node $e''$ can be either a summary node or a single node. Bear in mind that

   - $y.sel$ may have selector-edges to many shape-nodes and
   - $y.sel$ may have selector-edges to many summary-nodes, which in general represents many concrete objects connected via references that are represented by back edges of the summary-nodes.

   It is necessary to create all possible alias configurations that may be created by an assignment $x = y.sel$. Two kinds of definition information need to be added to the graph for each $x$-edge that is to be inserted: the reaching definition information of the $sel$ edge ($(x, a''_d, a''_t)$) and the definition information that is created by the statement itself $((x, \ell, a''_t))$. Both have the same source-node, $\{\}$, and the same target-node, $e''$.

Both cases mentioned above have to be considered because both can exist in the same FSG. As a first step we nullify variable x and obtain $(Es', is')$. $kill_x$ also models the kill aspects of reaching definitions but does not remove any information if $x$ is a recursion variable.

For the case $x = y$ ($x = x.sel$) we set

$$\begin{array}{rcl} f_\ell(G) & = & G''' \\ G' & = & g_{12}(\ell, h, y, sel, G) \\ G'' & = & g_{11}(\ell, x, h, G') \\ G''' & = & kill_h(G'') \end{array}$$

where $h$ is a fresh variable. All functions have been introduced before. When $x = y$ holds we need to "save" the values of $y.sel$ (which are in this case $x.sel$ values as well) because these values are removed when $x$ is nullified by $kill_x$ in $g_{12}$. Therefore we introduce an auxiliary variable, $h$, and assign $y.sel$ to $h$. Next we model $h$ being assigned to $x$ by applying $g_{11}$ and finally remove variable $h$ from the graph by $kill_x$.

**Theorem 2.5** *The transfer function $f_\ell$ is monotone.*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds.

1. case $x \neq y$: $kill_x$ is a monotone function, and $(Es'_i, is'_i) = kill_x((Es_i, is_i))$, $i \in \{1, 2\}$. Hence, $G'_1 \sqsubseteq G'_2 holds$. $G'_i$ is combined by $\sqcup$ with the graph obtained by applying *insert* to $G'_i$ and therefore $Es'_i \subseteq Es''_i$ and $is'_i \subseteq is''_i$ holds. Therefore, for an edge representing $y = null$ no edges are added to the graph. For all combinations of edges representing values of $x$ and edges representing values of $y \neq null$ a renamed graph is created by *insert* including the two edges being inserted. For each edge $[\{\}, a', e'] \in Es'_1 \cap Es'_2$ and $[e', a'', e''] \in Es'_1 \cap Es'_2$ the same two edges are added to the graph which is renamed by insert.

Since this is done edge by edge and the graphs are unioned (by $\bigsqcup$) it holds that $G_1'' \sqsubseteq G_2''$.

Therefore, for the case $x \neq y$, $G_1 \sqsubseteq G_2 \rightarrow g_{12}(G_1) \sqsubseteq g_{12}(G_2)$.

2. case $x = y$: According to case 1 the function $g_{12}$ is monotone, according to theorem 2.4 the function $g_{11}$ is monotone, according to theorem 2.1 the function $kill_x$ is monotone. The functions operate on FSGs and are closed under composition.

Therefore, $G_1 \sqsubseteq G_2 \rightarrow f_\ell(G_1) \sqsubseteq f_\ell(G_2)$; Function $f_\ell$ is monotone. $\square$

### 2.4.6 Transfer Function for $[x.sel = null]^\ell$

First assume that $e = \diamond$ such that $[\{\}, a, \diamond] \in Es$, with $a_v = x$; then $x$ will not point to an object in the heap and the statement will have no effect on the shape of the heap so the transfer function $f_\ell$ is just the identity.

Next assume that there is a node $e \neq \diamond$ such that $[s, a, e] \in Es$ with $a_v = x$ and there is a node $e'$ such that $[e, sel, e'] \in Es$.

The renaming of nodes $n$ in the flow-shape graph $(Es, is)$ is specified by the function:

We shall then take

$$f_\ell((Es, is)) = (Es'', is'')$$

where $(Es', is') = kill_{x,sel}((Es, is))$

and $kill_{x,sel}$ is defined as

$$\begin{array}{rcl} kill_{x,sel}((Es, is)) & = & (Es', is') \\ Es' & = & Es - \{[e, a', e'] | [s, a, e] \in Es \wedge [e, a', e'] \in Es \\ & & \quad \wedge\ a_v = x\ \wedge\ a'_v = sel \wedge e' = \diamond \\ & & \quad \wedge \neg is\_summary\_node_{Es}(e)\} \\ is' & = & is \end{array}$$

In the case that an edge with $a_v = x$ refers to a node $n$ and there exists an edge $[s, a, n] \in Es$ and $s$ being a recursion variable then we do not perform any nullification and $kill_{x,sel}$ is the identity function, and therefore monotone.

Otherwise, similar to $kill_x$, $kill_{x,sel}$ only removes edges from the graph but unlike $kill_x$ it only removes edges refering to node $\diamond$ that are not out-edges of summary nodes. Hence, we only perform a conditional update on $sel$ edges. We remove some but not all edges and therefore we do not perform a strong update on $sel$-edges. We remove edges representing scalar values and edges representing null-values but we do not remove any out-edge of summary nodes because an out-edge of a summary node may represent more than one reference value or scalar value.

**Theorem 2.6** *The function $kill_{x,sel}$ is monotone.*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds. Only edges $[e, a', \diamond]$ are removed from $Es_i$, $i \in \{1, 2\}$ if edges $[s, a, e]$ with $a_v = x$ exist in $Es_i$. Hence, edges $[e, a', \diamond] \in E_1 \cap E_2$ are removed from $E_1$ and from $E_2$ as well and $E_1' \subseteq E_2'$ holds. The sharing information $is_i$ is not effected. Therefore, $G_1 \sqsubseteq G_2 \rightarrow kill_{x,sel}(G_1) \sqsubseteq kill_{x,sel}(G_2)$. Function $kill_{x,sel}$ is a monotone function. $\Box$

It remains to define the edges that represent the new null-values:

$$\begin{array}{rcl} N_x & = & \{e|[s, a, e] \in Es'|a_v = x \wedge e \neq \diamond\} \\ Es'' & = & Es' \cup \{[e, (sel, d, N), \diamond] : e \in N_x\} \\ is'' & = & is' \end{array}$$

We treat a null value similar to a scalar value and do not create any aliases. Hence, we only need to add edges to the graph to represent the new definition information created by the assignment itself. Again, like for $x = null$ we use the Null type as type information which is assumed to be the type of the null-value (Java [GJSB00]).

### 2.4.7 Transfer Function for $[x.sel = new\ C]^\ell$

The transfer function for the statement $[x.sel = new\ C]^\ell$ is set to be

$$
\begin{array}{rcl}
f_\ell(G) & = G^{\text{iv}} & \\
G' & = & kill_{x,sel}(G) \\
G'' & = & g_{new}(\ell, C, h, G') \\
G''' & = & g_{21}(\ell, x, sel, h, G'') \\
G^{\text{iv}} & = & kill_h(G''')
\end{array}
$$

where $h$ is a fresh variable. All functions have been introduced before. First, we nullify the $x.sel$ edges in the graph by setting $G' = kill_{x,sel}(G)$. The auxiliary function $g_{new}$, creates a new node in the graph representing a created object in the heap and establishes an edge representing a reference from the fresh variable $h$ to the new object, as we did for $[x = new\ C]^\ell$. It remains to create a $sel$-edge from nodes reachable by $x$ to $h$ by $g_{12}$ and finally to remove edges representing values of the auxiliary variable $h$. Again we use the function $kill_x$ the remove var-edges from the graph.

The functions $kill_{x,sel}$, $g_{new}$, $g_{21}$, and $kill_x$ have been introduced before, so this completes the specification of the transfer function.

**Theorem 2.7** *The transfer function $f_\ell$ is monotone.*

**Proof**: According to theorem 2.6, case 1 in theorem 2.5, theorem 2.3, and theorem 2.1 the functions $kill_{x,sel}$, $g_{new}$, $g_{21}$, and $kill_x$ are monotone. The composition of monotone functions operating on FSGs is monotone. Therefore $f_\ell$ is monotone. □

### 2.4.8 Transfer Function for $[x.sel = y]^\ell$

This FSG transformer is another elaborate operation. The reason for this is that the FSG transformer has to create an FSG that conservatively covers all the possible new configurations of variable sets that are determined by the transitive dependence of variables and successor nodes that are created by adding $sel$-edges to the graph.

The transfer function follows the same pattern as the transfer function for $x = y.sel$ in some ways: For adding edges to the graph and for renaming of the graph the auxiliary function *insert* is used and the resulting graphs are combined by the join operator $\bigsqcup$.

In contrast to the transfer function for $x = y.sel$ we need to consider other edges to be aliased. The edges of interest are those labeled with $a_v = x$ followed by edges labeled with $a_v = sel$, and edges labeled with $a_v = y$. The $sel$-edges need to be established if $x$-edges exist in the graph.

Having introduced the auxiliary functions in section 2.4.5 we can now define the transfer function for $[x.sel = y]^\ell$ and take

$$f_\ell((Es, is)) = g_{21}(\ell, x, y, (Es, is))$$

First assume that $x \neq y$ and let (Es,is) be a FSG before the analysis of the statement and we set

$$g_{21}(\ell, x, y, (Es, is)) = (Es'', is'').$$

The first step will be to remove the old binding for $x.sel$ if possible and we remove edges representing such a binding by the auxiliary function $kill_{x,sel}$:

$$(Es', is') = kill_{x,sel}((Es, is))$$

The next step will be to rename the nodes corresponding to all nodes directly reachable by $x.sel$ to include $x$ in their name and establish a binding of $sel$ to those nodes. Furthermore we need to rename all nodes that are successors of those nodes by including $x$ in their name. This task is complicated when multiple summary nodes exist in a graph that require that multiple summary nodes are to be renamed.

$$\begin{array}{rcl}
(Es', is') & = & kill_{x,sel}((Es, is)) \\
(Es'', is'') & = & \bigsqcup\{insert(e', (sel, a''_d, a''_t), (sel, \ell, a''_t), e'', (Es', is')) \\
 & & \quad |[s', a', e'] \in Es', [s'', a'', e''] \in Es' \\
 & & \quad \wedge a'_v = x \wedge a''_v = y \wedge e' \neq \diamond \wedge compatible_{Es}(e', e'')\} \\
 & \cup & (Es', is')
\end{array}$$

We can now identify two possibilities:

1. There exists an edge $[\{\}, (x, d, t), \diamond]$ in the graph representing a null value. Definition $d$ and and type $t$ are irrelevant then; in this case we need to model the fact that a null value is dereferenced. Since there is no value that can be assigned, no value can be copied and assigned to $x.sel$. Therefore it is safe not to create a new edge because never a value can be assigned. If there is at least one execution path then a value is assigned and a definition $\ell$ created, which is covered by case two.

2. It exists a node $e'$, such that $[s', a', e'] \in Es'$ with $a'_v = x$ and a node $e''$, such that $[s'', a'', e''] \in Es'$ with $a''_v = y$. Hence, a reference $y$ exists in the graph which needs to by aliased by $x.sel$. The function *insert* handles various possibilities. The node $e'$ can be a single node or a summary node. Also the node $e''$ can be either a summary node or a single node. Bear in mind that

   - $x.sel$ may be established to refer to many shape-nodes and
   - $x.sel$ may be established to refer to many summary-nodes, which in general represents many concrete objects connected via references that are represented by back edges of the summary-nodes.

   It is necessary to create all possible alias configurations that may be created by an assignment $x.sel = y$. Two kinds of definition information need to be added to the graph for each $sel$-edge that is to be inserted: the reaching definition information of the $y$ edge $((sel, a''_d, a''_t))$ and the definition information that is created by the statement itself $((sel, \ell, a''_t))$. Both have the same source-node, $\{\}$, and the same target-node, $e''$.

Both cases mentioned above have to be considered because both can exist in the same FSG. As a first step we (possibly) nullify $x.sel$ and obtain $(Es', is')$. $kill_x$ also models the kill aspects of reaching definitions but does not remove any information if $y$ is a recursion variable or not the node $\diamond$.

For the case $x = y$ $(x.sel = x)$ we set

$$\begin{aligned} f_\ell(G) &= G''' \\ G' &= g_{11}(\ell, h, y, G) \\ G'' &= g_{21}(\ell, x, sel, h, G'') \\ G''' &= kill_h(G'') \end{aligned}$$

where $h$ is a fresh variable. All functions have been introduced above. When $x = y$ we need to "save" the values of $x$ because these values may be effected when $x.sel$ is nullified by $kill_{x,sel}$ in $g_{21}$. Therefore we introduce an auxiliary variable, $h$, and assign $x$ to $h$. Next we model $h$ being assigned to $x$ by applying $g_{21}$ and finally remove variable $h$ from the graph by $kill_x$.

**Theorem 2.8** *The transfer function $f_\ell$ is monotone.*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds.

1. case $x \neq y$: $kill_x$ is a monotone function, and $(Es'_i, is'_i) = kill_x((Es_i, is_i))$, $i \in \{1, 2\}$. Hence, $G'_1 \sqsubseteq G'_2 holds.$ $G'_i$ is combined by $\sqcup$ with the graph obtained by applying *insert* to $G'_i$ and therefore $Es'_i \subseteq Es''_i$ and $is'_i \subseteq is''_i$ holds. For an edge representing $x = null$ no edges are added to the graph. For all combinations of edges representing values of $x.sel$ and edges representing values of $y$ a renamed graph is created by *insert* including the two edges being inserted. For each edge $[s', a', e'] \in Es'_1 \cap Es'_2$ and $[s'', a'', e''] \in Es'_1 \cap Es'_2$ the same two edges are added to the graph which is renamed by insert. Since this is done edge by edge and the graphs are unioned by $\bigsqcup$ it holds that $G''_1 \sqsubseteq G''_2$.

   Therefore, for the case $x \neq y$, $G_1 \sqsubseteq G_2 \rightarrow g_{21}(G_1) \sqsubseteq g_{21}(G_2)$.

2. case $x = y$: According to case 1 the function $g_{21}$ is monotone, according to theorem 2.4 the function $g_{11}$ is monotone, according to theorem 2.1 the function $kill_x$ is monotone. The functions operate on FSGs and are closed under composition.

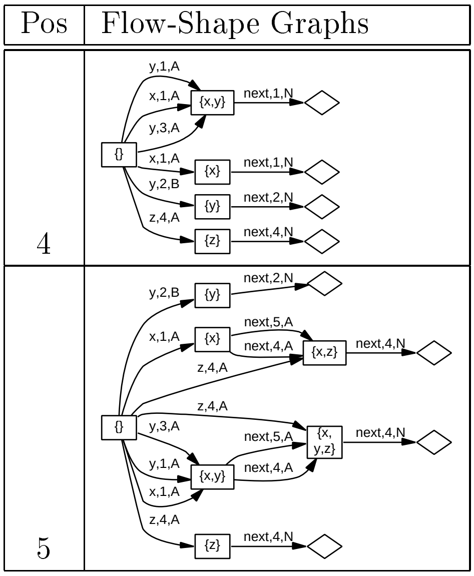



Figure 2.3: Duplication of node $\{z\}$ by $[x.next = z]^5$

Therefore, $G_1 \sqsubseteq G_2 \rightarrow f_\ell(G_1) \sqsubseteq f_\ell(G_2)$; Function $f_\ell$ is monotone. $\square$

In fig. 2.3 an interesting case of node duplication is shown. An edge labeled with $a_v = x$ refers to two different nodes. To ensure that the graph remains to be well named it is necessary to create for each inserted edge, representing the value of *next*, a subgraph of successor nodes that are renamed such that all successor nodes include the information $\{x, y\}$ or $\{x\}$ in the node naming set. Therefore, two graphs are created and unioned by $\bigsqcup$ by the transfer function for $[x.next = z]^5$. $\{z\} \cup \{x, y\} = \{x, y, z\}$ becomes a node of one subgraph and $\{z\} \cup \{x\} = \{x, z\}$ a node of the other subgraph. The node representing the object refered to by $z$ is duplicated and two nodes, $\{x, y, z\}$ and $\{x, z\}$ are elements of the graph $f_\ell(G)$ at position 5 with $G$ being the graph at position 4 in fig. 2.3.

### 2.4.9 Transfer Function for $[x.sel = y.sel]^\ell$

The transfer function for this statement can be formulated using already defined functions. We take

$$\begin{array}{rcl} f_\ell(G) & = & G''' \\ G' & = & g_{12}(\ell, h, y, sel, G) \\ G'' & = & g_{21}(\ell, x, sel, h, G') \\ G''' & = & kill_h(G'') \end{array}$$

The functions $g_{12}$, $g_{21}$, and $kill_x$ all have been introduced above and proved to be monotone. Therefore $f_\ell$ is a monotone function and operates on flow-shape graphs.

### 2.4.10 Transfer Function for $[t\ x]^\ell$

When a local variable is declared no memory is allocated and no dynamic properties need to be analyzed. Our analysis integrates type information and definition information and in the case of uninitialized variables in general two solutions integrate well in our framework:

- Only include edges in the graph if a value may exist.
- Integrate a special value representing uninitialized values.

We use the approach of integrating a special value. Since values are represented by edges we establish an edge to the non-object node $\diamond$ and use as definition information a special definition $\ell_0$, $\ell_0 \notin \mathrm{Lab}_\star$, to represent an uninitialized value by an edge with $a_d = \ell_0$. The type information for scalar values is the declared type of the variable. For reference variables we use the type Null, denoted $N$.

$$\begin{array}{lcl} f_\ell((Es, is)) & = & (Es', is') \\ Es' & = & Es + \{[\{\}, [x, \ell_0, t'], \diamond]\} \\ is' & = & is \end{array}$$

with $t' = t$ if $t$ is the type of a scalar variable and $t' = N$ (Null Type) if $t$ is a reference type.

Please note the type of values never is altered by the analysis. This allows to track the type of an object and that is why shadowed variables need to be renamed to be distinct to avoid any name conflict. To use the Null type for null-values is convenient because a null-value is assumed to be of type Null (written 'N'), similar to the typing system of Java and Modula-3.

The definition information, $a_d = \ell_0$, can be used to determine that a (possibly) uninitialized variable is used in an expression.

**Theorem 2.9** *The transfer function $f_\ell$ is monotone.*

**Proof**: Let $G_1 = (Es_1, is_1)$ and $G_2 = (Es_2, is_2)$ be any two well named FSGs. Suppose that $G_1 \sqsubseteq G_2$ holds. In any case an edge $\{[\{\}, [x, \ell_0, t'], \diamond]\}$ is added to $Es_1$ as well as to $E_2$ and no other edges are added to the graphs. The sharing information $is_1$ as well as $is_2$ remains unchanged. Therefore, $G_1 \sqsubseteq G_2 \rightarrow f_\ell(G_1) \sqsubseteq f_\ell(G_2)$; Function $f_\ell$ is monotone. □

### 2.4.11 Transfer Function for Block Entry $[\{]^{\ell_b}$

Declarations of variables are analyzed by the transfer function for $[t\ x]^\ell$. We do not need to take any actions at block entry (begin) and therefore the transfer function for block entry is the identity.

### 2.4.12 Transfer function for Block Exit $[\}]^{\ell_e}$

We assume that the functions $decl_{\circ(\ell)}$ and $decl_\bullet(\ell)$ give the set of declared variables before and after a label $\ell$ for a given program.

$$\begin{aligned} \text{let } \{v_1, \ldots, v_n\} &= decl_\bullet(\ell) - decl_\circ(\ell) \\ f_\ell &= kill_{v_n} \circ \ldots \circ kill(v_1) \end{aligned}$$

We kill (un-declare) all variables that are declared before the statement but are "undeclared" by leaving the scope. These variables need to be killed, i.e. removed from the graph G.

This completes the specification of the transfer functions for the intraprocedural analysis.

### 2.4.13 Scalar Values

The transfer functions are designed such that they can handle scalar values as well because the *null* value is treated like a scalar value. Actually, the defined functions specify the computation of reaching definitions for programs with scalar expressions with not more than one scalar variable on the right hand side of an expression but the extention to deal with arbitrary expressions is straightforward.

Let $use(a)$ denote the set of path expressions of expression $a$. For an assignment of a scalar value, $[x = a]^\ell$ or $[x.sel = a]^\ell$, with $|use(a)| = 0$ the transfer functions for $x = null$ and $x.sel = null$ need only to be modified to use the type of $a$, $type(a)$, instead of the type $N$ in the respective transfer function when creating edges to represent the value of a variable. For example $[i = 5]^\ell$ can be analyzed using the transfer function for $[x = null]^\ell$ with $x = i$ and substituting type $N$ by $type(5)$.

For assignments $[x = a]^\ell$ or $[x.sel = a]^\ell$, with $|use(a)| = 1$, the transfer functions for assignments as defined in sections 2.4.4 and 2.4.8 can be applied without any modifications. For example, $[i = i + 1]^\ell$, with $i$ being an integer variable, can be analyzed using the transfer function for $[x = y]^\ell$, with $x = i$ and $y = i$. In the example program in figure 2.4 this case is represented by $[i = i + 1]^{10}$. Similar for cases like $[i = x.val]^\ell$ or $[x.val = y.val']^\ell$, where $i$, $val$, and $val'$ are scalar variables. The type information is represented in the graphs due to the declaration of the variables.

For more complex right hand sides the use set of *path expressions* $x$ and $x.sel$, $|use(a)| > 1$, of the right hand side needs to be considered. It suffices to use the corresponding transfer function for *each* element of the use set. For example, a statement $[i = x.val + y.val']^\ell$ can be analyzed applying the transfer functions for $[i = x.val]^\ell$ *and* $[i = y.val']^\ell$ as above.

The seamless integration of the computation of reaching definition information of scalar variables is a consequence of our uniform treatment of null-values, scalar values, and reference values as edges in a flow-shape graph.

In the next section we discuss some interesting aspects of our analysis by example and present experimental results.

## 2.5 Basic Operations on Data Structures

In this section we demonstrate the accuracy of our analysis by discussing operations on data structures. We will discuss some basic programs for creating, accessing, altering, and destroying structural information.

As data structures we investigate lists, cyclic lists, trees, and arbitrary graphs. We will demonstrate how the reaching definition information is organized such that we are able to track the alias configurations and reaching definition information in the case of creation and (possibly partial) destruction of data structures.

In the visualization we use bold lines whenever a node is a summary node (i.e., has a self-edge) and use double boxes to visualize a shared status (i.e, the node is element of $is$).

### 2.5.1 List

A list is a basic data structure that serves well if one is interested in properties that include answers to questions such as: Which element is first in the list? Which is last? Which elements precede and follow a given one? Is it a shared list? How many variables refer to the head of the list? How many variables refer to any element of the list?

We demonstrate that our algorithm allows to answer these questions for dynamically allocated lists with a high amount of precision and computes dependence information for the approximated representation of a list.

#### Creation of a Single Linked List

We start with a simple program that generates a list. This program has been already mentioned in the introduction and we illustrate now how the knowledge about possibly created data structures is created in each iteration of the fix-point search.

For the given program of list creation we only need to deal with five different types of assignments but not with the form $x = y.sel$ introduced in section

```
class A extends O {
A next;
...;
}
{
[A x,y,z]^1;
[int i]^2;
[i = ...]^3;
[x = new A]^4;
[y = x]^5;
while ([i > 0]^6) {
        [z = new A]^7;
        [y.next = z]^8;
        [y = z]^9;
        [i = i−1]^10;
}
[y = null]^11;
}
```

Figure 2.4: Program: creation of a list

2.4.5. The variable $x$ points to the start of the list, variable $y$ serves as pointer to the advancing end of the list and variable $z$ is an auxiliary variable which is not necessary but helps to demonstrate the effects of assignments and how we deal with destructive updates. Variable $z$ either points to the end of the list or to a new element that is be appended to the end of the list. The pointers $y$ and $z$ always point to the same object at positions 6 (except when we enter the loop) 9 and 10 but refer to different objects at positions 7 and 8. We do not consider the number of iterations performed when the program is executed because boolean conditions are not considered by the analysis.

We discuss now the effects of assignments during each iteration in the analysis and why it is iteration 3 that a summary node is created in the flow-shape graph. We need four iterations to find a fix-point which means that in iterations 3 and 4 exactly the same graphs are computed at all program positions. Thus, we show the flow-shape graphs of iterations 3 and 4 in the same figures (fig. 2.8 and fig. 2.9).

In the first iteration we compute all aliases that may be created by the program in its first loop iteration. At program position 5 the variables $x$ and $y$ are aliased and refer to the same object. The assignment $[y.next = z]^8$ aliases $z$ and $y.next$. Hence, at position 8 the object created at position 7 can be accessed using three different path expressions, namely $x.next$, $y.next$, and $z$. In the flow-shape graph for position 8 we also have the reaching definitions 4,5 for $y$ and 4 for $x$. And for both, $x.next$ and $y.next$, the reaching definitions 7 and 8. Because we assume that with a new-operation the instance variables of an object are initialized with default values, $z.next$ is represented by a null-value (reference of type $N$). Note that $z.next$ is an alias of $x.next.next$ and $y.next.next$ which is represented in the graph as well (see fig. 2.5, position 8). The edge refering to node $\{z\}$ represents a may-alias of the above mentioned references.

The sharing information for variables that is evident in the graph bears also aspects of must-alias information: if $x$ refers to an object then must also $y$ refer to this object because there are no other edges for $x$ and $y$. For $z$ we gather the information that it may either refer to an object to which $x.next$ and $y.next$ refer to as well or it is not aliased. May-alias information always fully includes must-alias information. Our analysis is a may-alias analysis and only some aspects of must-alias analysis can be understood from the graphs. If an edge exists in a graph it represents a value that may exist. In some cases, like here for the variables $x$ and $y$, we can reason from the naming scheme of the nodes that both, $x$ and $y$ may exist. This reasoning can be refined when taking into account the reaching definition information and that we create for each declared variable an edge in the graph. If no edge of a declaration exists (an edge with $a_d = 0$) then the variable does have been initialized. For example, variable $x$ is declared at position 1, but it is initialized at position 4. If both, a “declaration”-edge ($a_d = 0$) and an edge representing a different reaching definition exist, then this variable *may* have been initialized. Similar for path expressions. Therefore, our analysis can also be used to analyze programs for uninitialized *values*.

In iteration 1 a list consisting of two objects is created. At position 8 variables $x$ and $y$ refer to the first element, variable $z$ to the second element. At position 9 the pointer $y$ is advanced to refer to the second element of the list.

In iteration 2, see figures 2.6 and 2.7, at position 8 the two elements are linked via $next$ with the new element refered to by $z$ and a list of three elements is represented in the flow-shape graph at position 8.

When pointer $y$ is advanced from the second element to the third element at position 9 we deliberately drop some information and represent the first and second element by a summary node and the third element by a single node. The summary node represents a list of objects of unknown length. We drop the information that there are only two elements.

The explicite sharing information $is$ comes into play here. Without $is$ we could not know that only one sel-edge refers to node $\{x, y, z\}$. The summary node represents an unknown number of objects and an out-edge of summary node may represent an unknown number of edges. But in this particular case - because $\{x, y, z\}$ is not shared - we know that only one sel-edge can refer to $\{x, y, z\}$. This kind of knowledge only is created when data structures are created. When $sel$-edges out of summary nodes created the object that is pointed to by this edge becomes shared because we cannot know how many edges may refer to this object - it may be more than one.

In iteration 3, see figures 2.8 and 2.9, we again have the case of creating a list of two elements but we already have the knowledge that these two elements are preceeded by an unknown number of elements (represented by the summary node). That is how we approximate data structures that are created in loops.

Iteration 4 cannot add any information to the flow-shape graphs and the algorithm terminates by finding a fix-point.

Finally, we present statistics for the given program. In figure 2.10 in rows one to four statistics for each iteration are shown:

**max nodes:** The maximum number of nodes created in all flow-shape graphs in the respective iteration. Please note that the non-object node $\diamond$ is *displayed* multiple times (to represent unshared null-values or unshared scalar values of *different* paths). The maximum number of nodes in a flow-shape graph is $2^n + 1$ where $n$ is the number of local reference variables and values of scalar variables and the null-value are represented by edges to the node $\diamond$. Instance variables do not contribute to the worst case of the number of nodes in a flow-shape graph.

**max edges:** The maximum number of edges with *different* start and end nodes. We represent each reaching definition-type pair by one edge in the graph. This matches the formalism we use to specify the transfer function. In fact, the reaching definition information that "belongs" to an $a_v$ attribute of an edge is the *set* of rd-type pairs. Refer to section 2.3.8 for the precise definition of this relation.

**max rt-sets:** Similar to max-edges; we evaluate the maximum size of all rt-sets of all flow-shape graphs in one iteration.

**diff graphs:** The number of flow-shape graphs of an iteration being different to the respective flow-shape graphs of the previous iteration. When the number of flow-shape graphs that "do not change" is zero we have found a fix-point. Therefore the last number in the column is always zero.

**ALL:** The last row shows above four statistics for *all* iterations. We count the number of different nodes and edges in all graphs, the maximum size of all rt-sets in all graphs and how many different graphs are generated by the analysis in all iterations. For example, in the analysis of the create list program in each iteration 11 different graphs are computed but only 19 different graphs in all iterations. The reason is that for the part of the program where we do not have to analyze a loop, the "final" result is obtained after the second iteration. This is why work list algorithms can significantly improve the efficiency of fixed points algorithms, if they only apply transfer functions at program points with new information. For the efficient computation of fix-points see [CHY95].

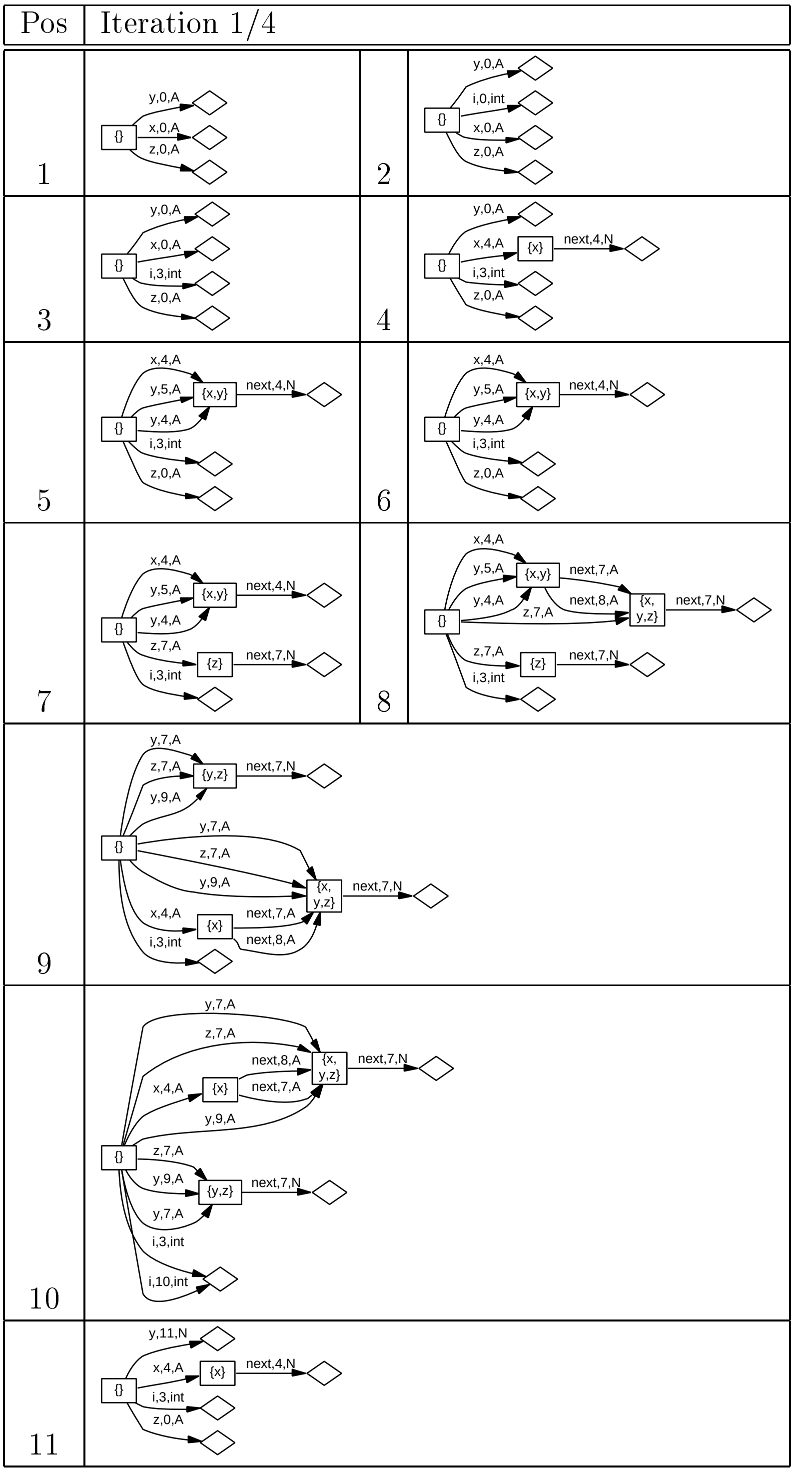



Figure 2.5: Creation of a list, iteration 1 of 4

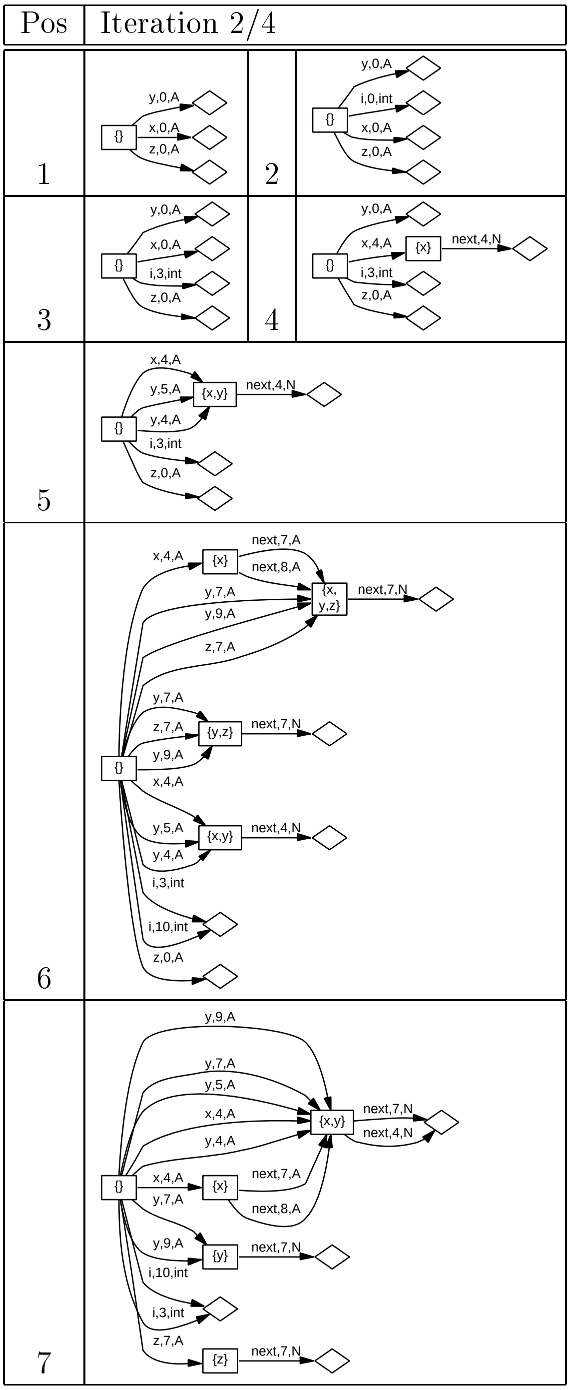



Figure 2.6: Creation of a list, iteration 2 of 4, positions 1-7

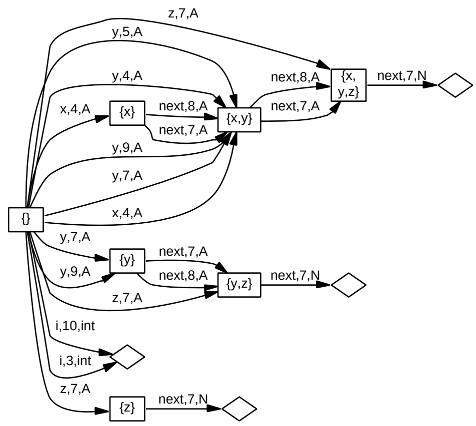
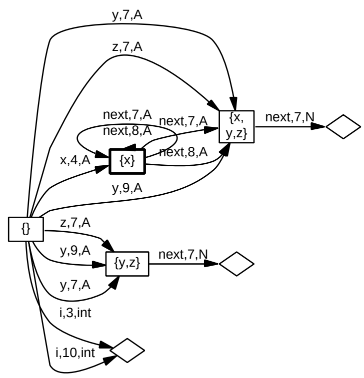
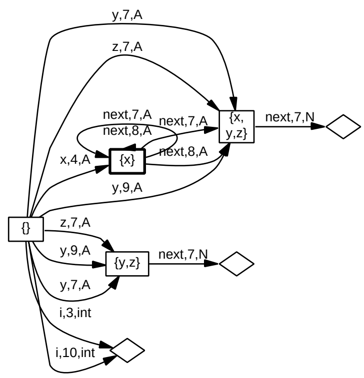
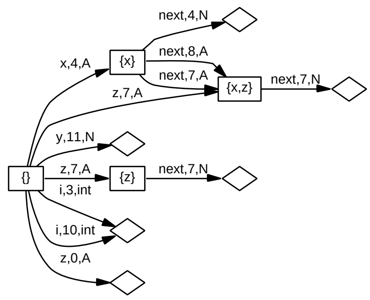
<table>
<tr><th>Pos</th><th colspan="3">Iteration 2/4</th></tr>
<tr><td>8</td><td colspan="3">
</td></tr>
<tr><td>9</td><td colspan="3">
</td></tr>
<tr><td>10</td><td>
</td><td>11</td><td>
</td></tr>
</table>

Figure 2.7: Creation of a list, iteration 2 of 4, positions 8-11.

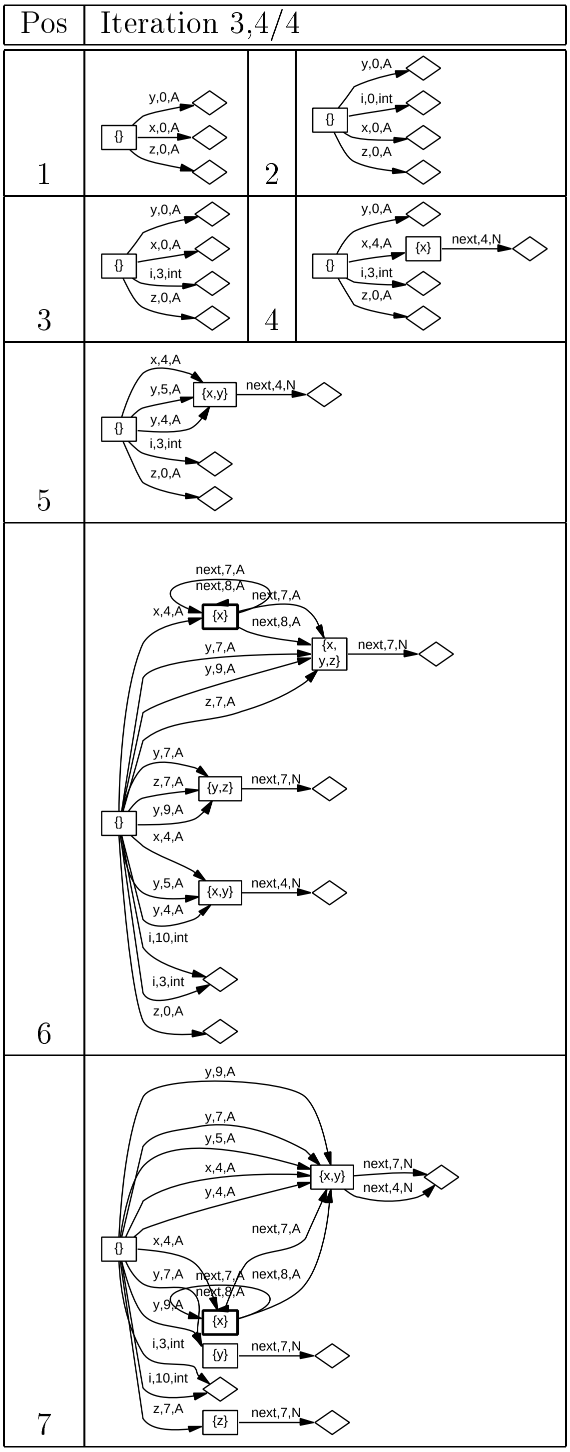



Figure 2.8: Creation of a list, iteration 3 and 4 of 4, positions 1-7.

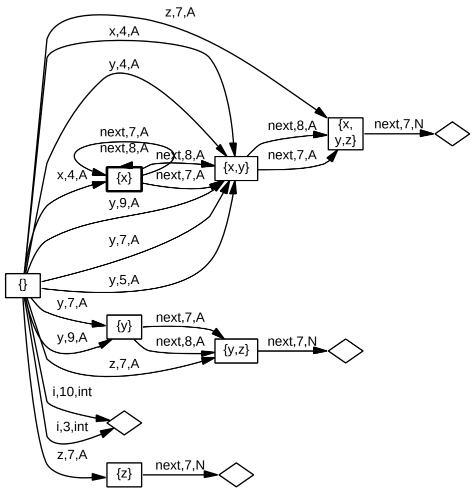
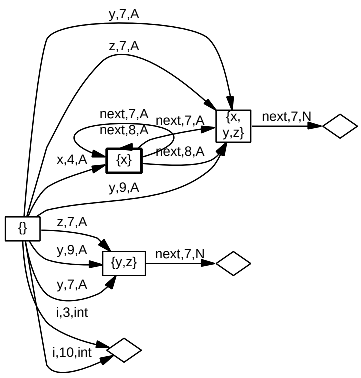
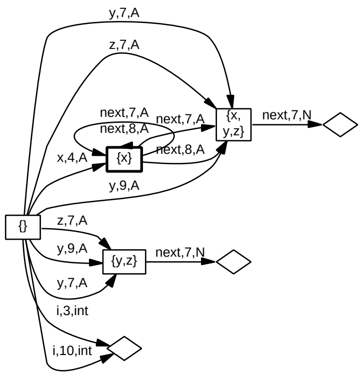
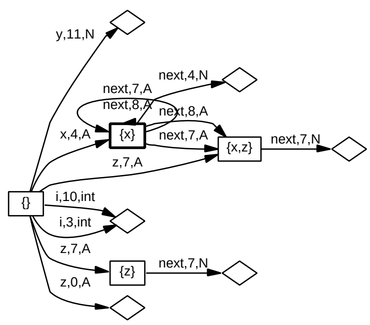
<table>
<tr><th>Pos</th><th colspan="3">Iteration 3,4/4</th></tr>
<tr><td>8</td><td colspan="3">
</td></tr>
<tr><td>9</td><td colspan="3">
</td></tr>
<tr><td>10</td><td>
</td><td>11</td><td>
</td></tr>
</table>

Figure 2.9: Creation of a list, iteration 3 and 4 of 4, positions 8-11.

| Iter | max nodes | max edges | max rt-sets | diff graphs |
|---|---|---|---|---|
| 1 | 5 | 9 | 2 | 11 |
| 2 | 8 | 14 | 4 | 6 |
| 3 | 8 | 15 | 4 | 4 |
| 4 | 8 | 15 | 4 | 0 |
| ALL | 9 | 27 | 4 | 19 |

Figure 2.10: Statistics for create list

### Accessing information

An interesting case is when a list is traversed and how we deal with the created aliases and whether we are able to keep the reaching definition information apart from dependencies created in other data structures. Our algorithm keeps dependence information created for different data structures apart by using multiple summary nodes.

When ever nodes are collapsed to one node because a cyclic data structure is identified, dependence information is unioned as well.

Let $x$ refer to a list as created by the program in figure 2.4 at program position 21. The variable $z$ is assumed to be of a different scope than $x$ and $y$ here.

```
[skip]^21;
[y = x]^22;
while ([y != null && ...]^23) {
      [y = y.next]^24;
}
[skip]^25;
```

Figure 2.11: Program: searching for an element in a list

The variable $y$ is used to traverse the list. The condition is not considered by our analysis. The interesting aspect, we are able to analyze precisely, is that the non-cyclic list remains to be a non-cyclic list. The reaching definition information of summary nodes is represented by its self-edges. Please note that we represent access paths of infinite length by self-edges.

It takes three iterations until we eventually find a fix-point. The reaching definition information added to that of the list creation programs comes from the pointer $y$ advancing the list.

The non-aliased list - only refered to by $x$ - represents the fact that variable $y$ is advanced pass the last element of the list and assigned *null*. In this case the variables $x$ and $y$ are not aliased.

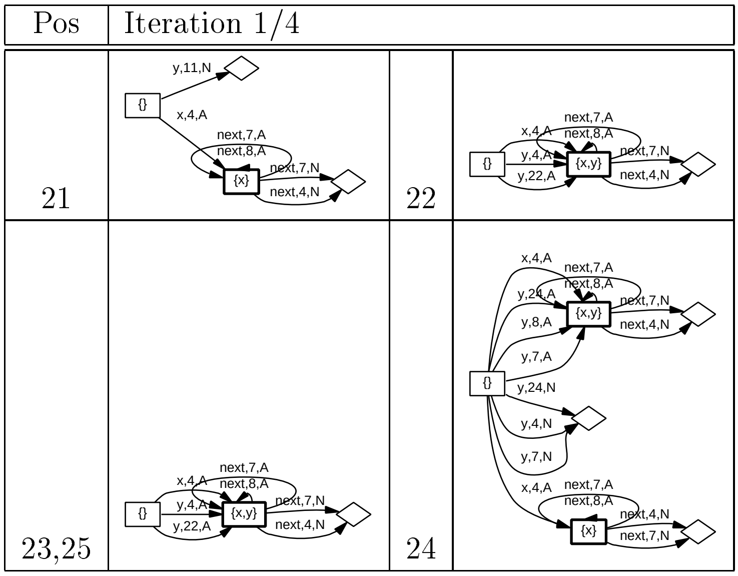



Figure 2.12: Searching for an element in a list, iteration 1 of 4

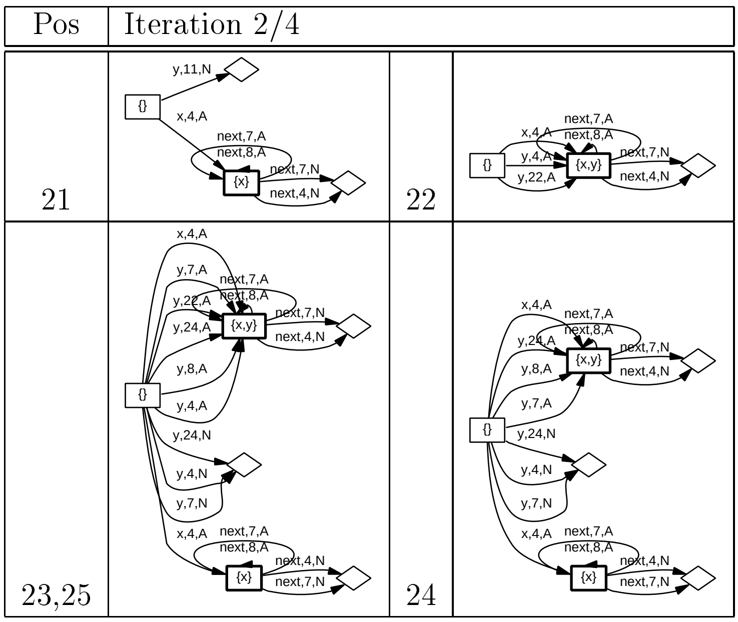



Figure 2.13: Searching for an element in a list, iteration 2 of 4

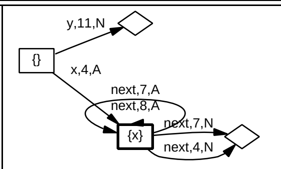
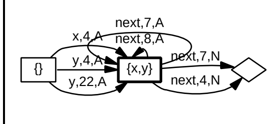
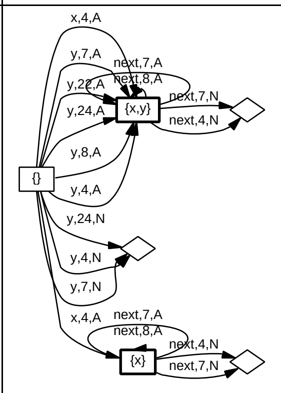
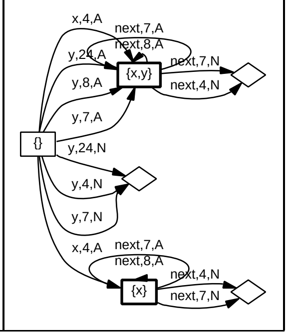
| Pos | Iteration 3,4/4 | | |
|---|---|---|---|
| 21 |  | 22 |  |
| 23,25 |  | 24 |  |

Figure 2.14: Searching for an element in a list, iterations 3,4 of 4

| Iter | max nodes | max edges | max rt-sets | diff graphs |
|---|---|---|---|---|
| 1 | 4 | 8 | 3 | 4 |
| 2 | 4 | 8 | 5 | 2 |
| 3 | 4 | 8 | 5 | 0 |
| ALL | 4 | 8 | 5 | 4 |

Figure 2.15: Statistics for search list

### Cyclic List

Our analysis always uses one node to represent a cyclic data structure. This is in contrast to [SRW98] where cycles can be represented by various nodes in a static shape graph. When cycles are created, only little information can be extracted. A high number of aliases needs to be created to cope with all possible alias configurations. Our naming scheme, to use only variable names, and to "propagate" this information to all successor nodes of a node in a graph "leads" to collapsing of nodes of a cycle to one summary node that represents the cycle. In the visualization we use bold lines whenever a node is a summary node (i.e., has a self-edge) and use double boxes to visualize a shared status (i.e, the node is element of $is$). Hence, the nodes of interest in this section are bold double boxed nodes.

The program is similar to the program create list (see fig. 2.4) but extended by a statement $[z.next = x]^{12}$. Variable $z$ may refer to the last element of a list and variable $x$ may refer to the first element of the list. The assignment $[z.next = x]^{12}$ creates a cyclic list. We add nodes to the graph where all successor nodes of $x$ are successor nodes of $z.next$ as well. This models the fact the $z.next$ and $x$ have been aliased and that all objects accessible by $x(.next)^*$ are accessible by $z(.next)^*$ as well.

In figure 2.17 at program position 11 we have a list whose last element is pointed to by $z$ and $x$ refers to the first element of the list. The assignment $[z.next = x]^{12}$ creates a cyclic list. To all successor nodes of $z.next$ all variable names of node $\{x, z\}$ are added. Hence both nodes, being members of the cyclic list, are mapped to the same name {x,z}. This node is induced shared the out-going edge of the summary node (precisely, by the number of edges that are represented by an out-going edge of a summary node) as well as by the new edge $z.next$. The node {x,z} is marked to be shared and added to $is$.

The unshared list "remains" in the graph (it is unioned with the renamed copies of the nodes). One of the reason is to ensure the monotonicity of the transfer function for $x.sel = y$. The transfer function $f_\ell$ for $x.sel = y$ is monotone. Hence, it must hold that $G_1 \sqsubseteq G_2 \rightarrow f_\ell(G_1) \sqsubseteq f_\ell(G_2)$. Let the graph at position 11 be $G_2$. A subgraph $G_1$ of $G_2$ consists of the summary node $\{x\}$ with its self-edge, the edge refering to the head of the list labeled with $x$, and the edge $[\{\}, (z, 0, A), \diamond]$. For this subgraph, $G_1$, the assignment

```
class A extends O{
        A next;
        ...;
}
{
        [A x,y,z]^1;
        [int i]^2;
        [i = ...]^3;
        [x = new A]^4;
        [y = x]^5;
        while ([i > 0]^6) {
              [z = new A]^7;
              [y.next = z]^8;
              [y = z]^9;
              [i = i-1]^10;
        }
        [y = null]^11;
        [z.next = x]^12;
}
```

Figure 2.16: Program: creation of a cyclic list

$[z.next = x]$ is without effect because $z$ cannot be dereferenced and $f_\ell(G_1) = G_1$. But it must hold that $f_\ell(G_1) \sqsubseteq f_\ell(G_2)$ and therefore the unshared list refered to by $x$ must be included in the flow-shape graph.

In figure 2.15 a table with the statistics for create cyclic list compared to create list shows that the maximum number of nodes is the same. This is due to our naming scheme. The shared node (the cyclic node) is represented by the same name set as the single node that represented the last element of the list.

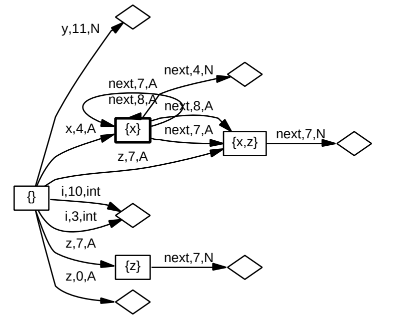
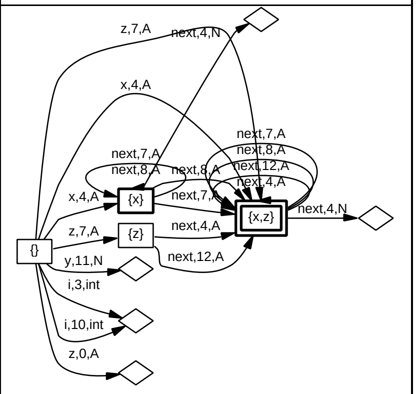
<table>
<tr><th>Pos</th><th>Graphs</th></tr>
<tr><td>11</td><td><br></td></tr>
<tr><td>12</td><td><br></td></tr>
</table>

Figure 2.17: Creating a cyclic list by assignment. Collapsed nodes.

| Iter | max nodes | max edges | max rt-sets | diff graphs |
|---|---|---|---|---|
| 1 | 5 | 9 | 2 | 12 |
| 2 | 8 | 14 | 4 | 7 |
| 3 | 8 | 15 | 4 | 5 |
| 4 | 8 | 15 | 4 | 0 |
| ALL | 9 | 30 | 4 | 21 |

Figure 2.18: Statistics for create cyclic list

### 2.5.2 Tree

An alias analysis being capable of representing tree structures can be utilized for parallizing algorithms and for static garbage collection. Whenever we are able to identify data structures to be trees, computations on the nodes of the tree can be parallized. The reaching definition information is the bases for the computation of program dependencies and is computed for each link in the tree that is represented in the graph. The tree can be represented by one summary node and two self-edges, one representing the left-reference ($a_v$ =l) and the other representing the right-reference ($a_v$ =r).

```
class T extends O{
        T l;
        T r;
        ...;
}
{
              [T p,q,a,b]^1;
              [p = new T]^2;
              [a = new T]^3;
              [b = new T]^4;
              [p.l = a]^5;
              [p.r = b]^6;
              [a = null]^7;
              [b = null]^8;
}
```

Figure 2.19: Program: explicite creation of a tree

The program creates a tree with $p$ refering to the root object and $a$ and $b$ being the left and right child. In figure 2.20 the tree refered to by $p$ is shown. When $a$ is nullified the left child is removed from the graph and the edge label with l become a self-edge of $p$. Similar at position 8 with the right child $b$, figure 2.21 . In figure 2.22 the collapsed tree is shown with two self-edges, for left and right childs. Actually a data structure Tree is represented, a tree of unkown size. Similar to the list create program, we drop information about the size of the tree.

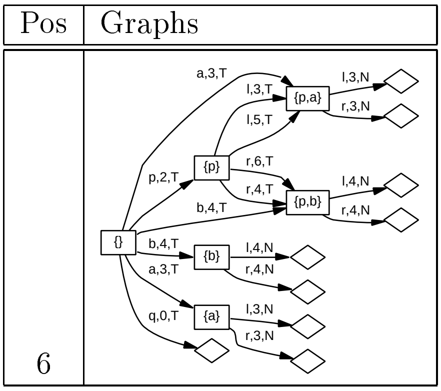



Figure 2.20: References to second level binary tree nodes

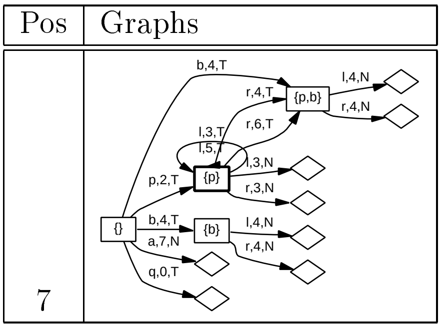



Figure 2.21: “Collapsing” the binary tree: $[a = null]^7$

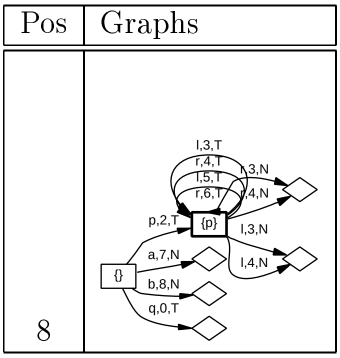



Figure 2.22: The “collapsed” binary tree: $[b = null]^8$

| Iter | max nodes | max edges | max rt-sets | diff graphs |
|---|---|---|---|---|
| 1 | 7 | 16 | 2 | 8 |
| 2 | 7 | 16 | 2 | 0 |
| ALL | 7 | 23 | 2 | 8 |

Figure 2.23: Statistics for create tree

### 2.5.3 Arbitrary Graph

Finally we discuss an example for accessing an arbitrary graph. An arbitrary graph is represented by a shared summary node (it may be cyclic) and with at least one self-edge. This may represent a DAG or a cyclic graph.

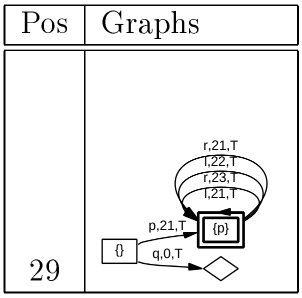



Figure 2.24: An arbitrary graph

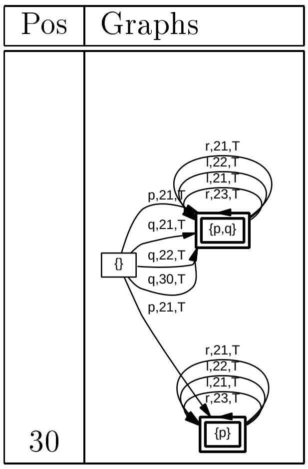



Figure 2.25: Establishing a reference to a graph node by [q = p.l]$^{30}$;

We establish a reference to an element of the arbitrary graph represented by the flow-shape graph at position 29 by the assignment $[q = p.l]^{30}$. The path $p.l$ is aliased by $q$ and thus, $q$ may refer to any object in the graph that is refered to by $p.l$. Therefore both paths $q$ and $p.l$ are aliased. This case represents all possible aliases of $p$, $p.l$, and $q$. Please note that any path in

| Iter | max nodes | max edges | max rt-sets | diff graphs |
|---|---|---|---|---|
| 1 | 3 | 7 | 3 | 1 |
| 2 | 3 | 7 | 3 | 0 |
| ALL | 4 | 8 | 3 | 2 |

Figure 2.26: Statistics: Establishing a reference to a graph node

the data structure (as well as any alias) consisting of $l$-links and $r$-links is represented by the shared (cyclic) summary node and its self-edges (see fig. 2.24)

In figure 2.25 the reference to an object refered to by $q$ is established. The reaching definition information of $p.l$ reaches $q$ and therefore two definitions, $\{21{,}22\}$ reach $q$. The variable $q$ is initialized and the information stemming from the declaration of $q$ is killed.

In the statistics the max-rt set value of 3 is generated by the $q$-edges in the flow-shape graph at position 30.

All data structures discussed in this section are represented by sets of edges and some explicit sharing information $is$. Whether a data structure is a list, a tree, or an arbitrary graph, can be concluded from the graph and the $sel$-edges emanating from a node. Data structures of unknown size are represented by summary nodes.

In the next section we extend the intraprocedural framework to an interprocedural framework.

# Chapter 3

# Interprocedural Analysis

The program analysis presented in the previous section was an intraprocedural analysis because we did not deal with method calls. We did allow objects to have instance variables and presented an analysis that computed a safe approximation of data structures that may be created, traversed, or destroyed by a given program $P_\star$. We extend the intraprocedural analysis presented in the previous section by transfer functions to an interprocedural analysis that integrates method invocation resolution. We shall make use of the precise type and aliasing information included in FSGs and show how this information can be integrated in an interprocedural frame work.

## 3.1 Language Definition Extention

We extend our language defined in section 2.1 by method declarations and a method call statement.

$$m \quad \in \quad \textbf{Meth} \quad \text{methods}$$

We extend the class declaration $C$ to allow methods to be declared:

$$C \quad ::= \quad \ldots \mid \text{class } c_1 \text{ extends } c_0 \ \{ \ D \ M \ \} \mid C \ C$$

where $c_1$ and $c_2$ are class names, $D$ is a sequence of variable declarations. $M$ is a list of method declarations of the form:

$$M \quad ::= \quad t_0\ m(t_1\ a_1, \ldots, t_u\ a_u)[\{]^{\ell_n}\ D\ S\ [\}]^{\ell_x}|M\ M$$

The methods may have multiple call-by-value parameters and we define the return value to be of type $t_0$. Method names (denoted $m$) are syntactically distinct from variables. The label $\ell_n$ marks the entry to the method body and the label $\ell_x$ marks the exit from the method body. The syntax of statements is extended with method calls:

$$S \quad ::= \quad \ldots |[x = y.m(a_1, \ldots, a_u)]^{\ell_c}_{\ell_r}$$

Method calls are statements (but not expressions). The variable $x$ shall be assigned the return value of the method. Different return values can be defined since multiple *return* statements are allowed. The transfer function for the return statements shall be designed such that it equals an assignment of the return expression to a special variable ret which serves as result-variable and is assigned to variable $x$. This is ensured by the transfer functions for return, method exit, and call return. We shall discuss each transfer function in detail in the next sections.

The method call statement $[x = y.m(\ldots)]$ has two labels: $\ell_c$ will be used for the call of the method and $\ell_r$ will be used for the associated return; the actual parameters are $a_1, \ldots, a_u$. The language is statically scoped, the parameter mechanism is call-by-value for the parameters 1 to $u$ and an assignment of the return value to the variable $x$. The methods may be mutually recursive. We *assume* throughout that the program is uniquely labeled and hence label consistent (see section 2.1.4); also we *assume* that only methods that have been declared in $M$ in the respective class declaration are ever called.

The signature of a method consists of the name of the method and the number and types of formal parameters of the method. A class may not declare two methods with the same signature. A method is always invoked with respect to an object, which becomes the current object to which the keywords this refers during execution of the method body. An instance method $m_2$ declared in a class $B$ overrides another method with the same signature, $m_1$, declared in class $A$ if $B$ is a subclass of $A$.

We *assume* that all variables in a given program are distinct. The this-variable of a class is also assumed to be distinct from the this-variable of a different class. To emphasize this we write $this_C$, indicating that the this-variable is indexed by its class. A similar approach has been taken in [SA00].

## 3.2 Interprocedural Flow graphs

The next step is to extend all definitions necessary to cope with method calls. We define an interprocedural control flow graph and shall define appropriate contexts to keep information of different calling contexts apart. Based on these definitions we introduce transfer functions for method invocation resolution.

### 3.2.1 Statements

We need to extend the definitions of the functions $init, elementary, labels$, and $flow$ to specify the flow graphs also for our extended language. For the new statement call we take:

$$
\begin{array}{lcl}
init([x = y.m(a_1, \ldots, a_u)]^{\ell_c}_{\ell_r} & = & \ell_c \\
final([x = y.m(a_1, \ldots, a_u)]^{\ell_c}_{\ell_r} & = & \{\ell_r\} \\
elementary([x = y.m(a_1, \ldots, a_u)]^{\ell_c}_{\ell_r} & = & [x = y.m(a_1, \ldots, a_u)]^{\ell_c}_{\ell_r} \\
flow([x = y.m(a_1, \ldots, a_u)]^{\ell_c}_{\ell_r} & = & \bigcup\{\{(\ell_c, \ell_n), (\ell_x, \ell_r)\} \mid \\
 & & \quad t_0\ m(t_1\ p_1, \ldots, t_u\ p_u)[\{]^{\ell_n}\ D\ S\ [\}]^{\ell_x} \\
 & & \quad \text{is in} M_\star\}
\end{array}
$$

Here $(\ell_c, \ell_n)$ and $(\ell_x, \ell_r)$ are new kinds of flows:

- $(\ell_c, \ell_n)$ is the flow corresponding to *calling* a method at $\ell_c$ and with $\ell_n$ being the entry point for a method body. Please note that in the case of overridden methods we create multiple flows, from the label $\ell_c$ to all methods of subclasses that may be called. It is subject to the analysis to reduce the set of possibly called methods.

- $(\ell_x, \ell_r)$ is the flow corresponding to *exiting* a method body at a label $\ell_x$ of the set of possibly called methods and *returning* to the call at $\ell_r$.

The definition of $flow([x = y.m(a_1, \ldots, a_u)]^{\ell_c}_{\ell_r})$ exploits the fact that the syntax of method calls only allows us the name of a method defined in the program. We declared the set of method names to be disjunctive with the set of variable names. Hence we do not allow methods to be assigned to variables. But, since our analysis integrates the computation of types for references that may refer to an object it is straightforward to integrate method invocation resolution and to deal with inheritance and overriding methods.

Our analysis integrates method invocation resolution but we do not modify the control flow graph at any stage in the analysis. It shall be the transfer functions for method calls that perform method invocation resolution. We will discuss this in detail when introducing the transfer functions for method entry in section 3.3.2.

### 3.2.2 Programs

Next consider the program $P_\star$ of the form $C_\star\ B_\star$. For each method declaration $t_0\ m(t_1\ p_1, \ldots, t_u\ p_u)[\{]^{\ell_n}\ D\ S\ [\}]^{\ell_x}$ we set

$$
\begin{array}{rcl}
init(m) & = & \ell_n \\
final(m) & = & \{\ell_x\} \\
elemenary(m) & = & \{[\{]^{\ell_n}, [\}]^{\ell_x}\} \cup elementary(D) \cup elementary(S) \\
labels(m) & = & \{\ell_n, \ell_x\} \cup labels(D) \cup labels(S) \\
flow(m) & = & \{(\ell_n, init(D))\} \cup \{(\ell, init(S)) | \ell \in final(D)\} \\
& & \cup flow(D) \cup flow(S) \\
& & \cup \{(\ell, \ell_x) \mid \ell \in final(S)\} \\
& & \cup \{(\ell, \ell_x) \mid \ell \in retlabs(S)\}
\end{array}
$$

and for the entire program $P_\star$ we set

$$
\begin{array}{rcl}
init_\star & = & init(B_\star) \\
final_\star & = & final(B_\star) \\
elemenary_\star & = & elementary(B_\star)\cup \\
 & & \bigcup\{elemtary(m) \mid t_0\ m(t_1\ p_1,\ldots,t_u\ p_u)[\{]^{\ell_n} D\ S[\}]^{\ell_x} \\
 & & \text{is in } M_\star\} \\
labels_\star & = & labels(B_\star)\cup \\
 & & \bigcup\{labels(m) \mid t_0\ m(t_1\ p_1,\ldots,t_u\ p_u)[\{]^{\ell_n} D\ S[\}]^{\ell_x} \\
 & & \text{is in } M_\star\} \\
flow_\star & = & flow(B_\star)\cup \\
 & & \bigcup\{flow(m) \mid t_0\ m(t_1\ p_1,\ldots,t_u\ p_u)[\{]^{\ell_n} D\ S[\}]^{\ell_x} \\
 & & \text{is in } M_\star\}
\end{array}
$$

as well as $\mathbf{Lab}_\star = labels_\star$.

We use $F = flow_\star$ and $E = \{init_\star\}$ much as before.

Having added the interprocedural flow between method calls and methods the CFG defined in section 2.1.3 has been extended to an interprocedural control flow graph (ICFG).

## 3.3 Interprocedural Monotone Framework

For a precise interprocedural analysis we need to take context information into account. The context information is necessary to restrict the attention to paths in the program that have a proper nesting of method calls and returns.

Not considering a proper nesting of method calls we would spread alias, type, and reaching definition information of a method to all vertices $\ell_r$ of all call sites. By taking context information into account we keep the information of different calls separated.

An obvious approach is to encode information about the paths taken into the data flow properties themselves; to this end we introduce *context information*:

$$\delta \in \Delta$$

Since our main interest is in method calls we record only flows $(\ell_c, \ell_n)$ corresponding to a method call. Because methods may be recursive we need to restrict the number of recorded calls to some fixed number $k$ with the idea to record only the latest $k$ calls, i.e. labels $\ell_c$.

Formally we take

$$\Delta \quad = \quad \mathbf{Lab}^{\leq k}$$

Elements of $\Delta$ are called *call strings* of bounded length $k$ whereas the most recent label being recorded is at the right end. The empty call string is denoted $\Lambda$ and has length 0.

To keep call strings to a maximum length $k$ we possibly need to truncate them. Let $[\delta, \ell_c]$ denote a string of some arbitrary length, where $\delta$ is of arbitrary length and $\ell_c$ is the last label being recorded. We truncate a call string by omitting elements on the *left* and write $\lceil \delta, \ell_c]_k$ for a possibly truncated call string of maximum length $k$.

Consider the instance $(\mathcal{FSG}, \mathcal{F}, F, E, \iota, f.)$ of the monotone framework of our intraprocedural analysis. We shall now construct an instance

$$(\widehat{\mathcal{FSG}}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \widehat{f}.)$$

of an *embellished Monotone Framework* [NNH99] that takes context into account:

- $\widehat{\mathcal{FSG}} = \Delta \to \mathcal{FSG}$
- for each intraprocedural transfer function $f_\ell$ the interprocedural transfer function $\widehat{f_\ell}$ is given by

$$\widehat{f}_\ell(\widehat{G}) \quad = \quad \{(\delta, f_\ell(G)) \mid (\delta, G) \in \widehat{G}\}$$

In other words, the new instance applies the transfer functions of the intraprocedural instance for each context. We take the extremal value $\widehat{\iota}$ to be defined by the formula

$$\widehat{\iota}(\delta) = \begin{cases} \iota & \text{if } \delta = \Lambda \\ \bot & \text{otherwise} \end{cases}$$

Note that in the case $k = 0$ we have $\Delta = \{\Lambda\}$ which is equivalent to having no context information. In this case the analysis is called context insensitive analysis; for the case $k > 0$ we call it a context-sensitive analysis, where in general, the greater $k$ is the more precise is the analysis and the better it is able to separate different calling contexts.

We have extended $\mathcal{FSG}$ with context information and have to ensure that only FSGs of the same context are unioned as before whereas FSGs of different contexts are kept apart.

Let $\widehat{G_1}, \widehat{G_2} \in \widehat{\mathcal{FSG}}$ and $\widehat{G}^\delta = \{\delta | (\delta, G) \in \widehat{G}\}$ denote the set of all contexts of $\widehat{G}$ then the join operator $\widehat{\sqcup}$ can be defined as

$$G_1 \widehat{\sqcup} G_2 = \{(\delta', \bigsqcup\{G | (\delta', G) \in \widehat{G_1} \cup \widehat{G_2}\}) \mid \delta' \in (\widehat{G_1}^\delta \cup \widehat{G_2}^\delta)\}$$

to ensure that information of the same context is combined by $\sqcup$ when different paths in the ICFG meet.

We shall now define the transfer functions for method calls and returns.

### 3.3.1 Transfer Functions for $[x = y.m(a_1, \ldots, a_u)]_{\ell_r}^{\ell_c}$

For a method call $[x = y.m(a_1, \ldots, a_u)]_{\ell_r}^{\ell_c}$ we define the transfer function $\widehat{f_{\ell_c}}$ to take context into account where $\lceil \delta, \ell_c \rceil_k$ denotes the path obtained by appending $\ell_c$ to $\delta$ (so as to reflect that we enter the body of the method) and possibly truncating this path to a maximum length $k$. Thus we define

$$\begin{aligned} \widehat{f_{\ell_c}}(\widehat{G}) \quad = \quad & \widehat{\bigsqcup}\{(\delta', f_{\ell_c}(G)) \\ & \quad \mid (\delta, G) \in \widehat{G} \wedge \delta' = \lceil \delta, \ell_c \rceil_k\} \end{aligned}$$

The transfer function $f_{\ell_c} : \mathcal{FSG} \rightarrow \mathcal{FSG}$ describes how a FSG is modified and we take

$$\begin{array}{rcl} f_{\ell_c}(G) & = & G^{u+1} \\ G' & = & g_{11}(\ell_c, h_0, y, G) \\ G'' & = & g_{11}(\ell_c, h_1, a_1, G') \\ \dots & & \\ G^{u+1} & = & g_{11}(\ell_c, h_u, a_u, G^u) \end{array}$$

where $h_i$, $0 \leq i \leq u$, are auxiliary variables. Variable $h_0$ serves to pass on the this-reference to the called method whereas $h_1 \dots h_u$ pass on the call-by-value parameters of the call to the transfer function $f_{\ell_n}$ for method entry.

In $\widehat{f_{\ell_c}}$ the FSG $G$ is "marked" with its respective calling context $\delta'$ which has $\ell_c$ appended to its right (for $k > 0$). The variable this shall be initialized by $h_0$ which holds the reference to the object $y$.

Note that if call strings are truncated such that from different call sites the information is merged at $\ell_n$, the information remains correct because the graphs FSG are designed such that they represent information of different execution paths in the program. Hence, the constant $k$ also affects the size of FSGs.

**Theorem 3.1** *The transfer function $f_{\ell_c}$ is monotone.*

**Proof**: According to theorem 2.4 the function $g_{11}$ is monotone. The transfer function $f_{\ell_c}$ is set to be the composition of functions $g_1 1$. The composition of monotone functions $\mathcal{FSG} \to \mathcal{FSG}$ is monotone. Therefore $f_{\ell_c}$ is monotone. $\Box$

When returning from a method call we take:

$$\begin{array}{rcl} \widehat{f_{\ell_r}}(\widehat{G}) & = & \widehat{\bigsqcup}\{(\delta, f_{\ell_r}(G)) \\ & & \mid (\delta, G) \in \widehat{G} \wedge \delta = \lceil \delta', \ell_c \rceil_k\} \end{array}$$

The transfer function $f_{\ell_r}$ needs to assign the value stored in the unique auxiliary variable ret to $x$ (ret $\notin Var_\star$) and un-declare the variable ret. We use the variable ret to return the values that may have been returned by a return expression. Clearly, we assign the return expressions to an auxiliary variable ret and pass it back to the call site. At the call site this value (or values) is assigned to the variable $x$. This *models* a method call and the handling of parameters and return values.

We set the transfer function $f_{\ell_r}$ to be a composition of assigning ret to $x$ and killing (removing) variable ret.

We take

$$\begin{array}{rcl} f_{\ell_r}(G) & = & G'' \\ G' & = & g_{11}(\ell_r, x, \mathrm{ret}, G) \\ G'' & = & kill_{\mathrm{ret}}(G') \end{array}$$

The variable ret only "exists" when information is propagated from nodes labeled with $\ell_x$ to $\ell_r$. Again, similar to variables $a_i$ from above, graphs $G$, propagated from different overridden methods of different subtypes, are unioned but since the graphs are designed to hold information of different execution paths the analysis remains correct. The *precision* is increased by the use of call strings because the graphs representing information of different calling contexts are kept separate.

**Theorem 3.2** *The transfer function $f_{\ell_r}$ is monotone.*

**Proof**: According to the theorems 2.4 and 2.1 the transfer functions $g_{11}$ and $kill_{ret}$ are monotone. The composition of monotone functions $\mathcal{FSG} \to \mathcal{FSG}$ is monotone. Therefore $f_{\ell_r}$ is a monotone function. $\Box$

### 3.3.2 Transfer Functions for Method Entry and Exit

It remains to formulate the transfer functions corresponding to methods. For a method $t_0\ m(t_1\ p_1, \ldots, t_u\ p_u)[\{]^{\ell_n} D\ S\ [\}]^{\ell_x}$ we have two transfer functions. One for method entry and one for method exit. The transfer function for method entry needs to initialize the this-reference and the formal parameters according to the values that are passed on by a method call via the auxiliary variables $h_i$.

Thus we take

$$
\begin{array}{rcl}
f_{\ell_n}(G) & = & G^{2u+1} \\
G^0 & = & g_{11}(\ell_n, this_C, h_0, G) \\
G^1 & = & g_{11}(\ell_n, p_1, h_1, G^0) \\
\dots & & \\
G^u & = & g_{11}(\ell_n, p_u, h_u, G^{u-1}) \\
G^{u+1} & = & kill_{h_0}(G^u) \\
\dots & & \\
G^{2u+1} & = & kill_{h_u}(G^{2u})
\end{array}
$$

where $this_C$ is the *this*-variable of class $C$ and $p_1, \dots, p_u$ are the parameters. Please note, that we assume that all variables in a program have distinct names. Hence, parameters of different methods have different names. The *this*-variable is assumed to be renamed such that it is unique in each class $C$. This can easily be accomplished by extending the *this*-variable name by some representation of the class name (we signify this by writing $this_C$).

This is necessary to ensure that in the case of *nested* method calls the two different formal parameters of the calls $call_1$ and a nested call $call_2$ can be represented in the *same* FSG. This is why all side effects of methods can be handled with a high amount of precision.

These variables are undeclared, i.e. removed from the FSG , when information is passed back to the caller at the exit of the method. Please note that we have single exit points, $\ell_x$, of methods.

**Theorem 3.3** *The transfer function $f_{\ell_n}$ is monotone.*

**Proof**: According to theorem 2.4 the transfer function $g_{11}$ is monotone. $f_{\ell_n}$ is set to be the composition of functions $g_{11}$. Therefore $f_{\ell_n}$ is a monotone function. □

We shall now integrate method invocation resolution in our analysis by defining $\widehat{f_{\ell_n}}$ such that paths in the CFG that represent a method that can *never* be called to be *treated* like paths that are not in the actual context.

$$
\begin{array}{rcl}
\widehat{f_{\ell_n}}(\widehat{G}) & = & \{(\delta', f_{\ell_n}((Es, is))) \mid (\delta', (Es, is)) \in \widehat{G} \\
& & \wedge \exists [s, a, e] \in Es : a_v = h_0 \wedge a_t \in types(m)\}
\end{array}
$$

We apply the transfer function $f_{\ell_n}$ on each FSG with a context $\delta'$ if the method being analyzed is a method of a type that may be the type of a value stored in $h_0$ $(y)$.

Clearly, $m$ is the method with method entry $\ell_n$. $x = y.m(\ldots)$ requires to take the method $m$ into account if it is declared for a type of those objects that may be refered to by $y$. This set of types is represented by the edges $[s, a, e] \in Es : a_v = h_0$ because $y$ is assigned to $h_0$ by the transfer function for $[x = y.m(a_1, \ldots, a_u)]_{\ell_r}^{\ell_c}$. Hence, only edges with $a_v = h_0$ are of interest. To perform method invocation resolution we only take the method into account if it is declared for a type $a_t$ of an existing edge $[s, a, e]$. If for a type $T$ in $types(m)$ no edge with $a_t{=}T$ exists no information is propagated in the ICFG representing the method $m$. Therefore no reaching definitions are generated for methods that are definitely not called. Since our alias analysis is a may-alias analysis we may include methods that are actually not called by the program.

At returns the right hand side is assigned to a special variable ret. This variable serves as auxiliary variable for return values; i.e. we "simulate" a call-by-result value.

Thus, for $[return\ y]^\ell$ we take

$$f_\ell(G) \quad = \quad g_{11}(\ell, \mathrm{ret}, y, G)$$

Hence, the return statement is analyzed like a single assignment and the general case of $[return\ a]^\ell$ is handled as described in section 2.4.13 for scalar values, in section 2.4.2 for the case $a = null$ and in section 2.4.5 for the case $a = y.sel$.

It remains to define the transfer function for method exit. The variable ret is treated like any other variable at method exit (but different at call return). We only need to un-declare the auxiliary variables that are declared at the corresponding method entry. Thus, we take

$$\begin{aligned} \widehat{f_{\ell_x}}(\widehat{G}) &= \{(\delta, f_{\ell_x}(G)) | (\delta, G) \in \widehat{G}\} \\ f_{\ell_x} &= kill_{p_u} \circ \ldots kill_{p_1} \circ kill_{this_C} \end{aligned}$$

The transfer function $f_{\ell_x}$ is set to be the composition of functions $kill_x$, each removing one of the variables that are created in the corresponding method entry.

**Theorem 3.4** *The transfer function $f_{\ell_x}$ is monotone.*

**Proof**: According to theorem 2.1 the function $kill_x$ is monotone. The transfer function $f_{\ell_x}$ is set to be the composition of functions $kill_x$. Therefore $f_{\ell_x}$ is monotone. $\square$

This completes the specification of the transfer functions for the analysis.

In this section we have extended the intraprocedural analysis by methods and method calls and developed the transfer functions for method call and return, method entry and exit and the return statement. The interprocedural framework used $k$-bounded call strings to separate different calling contexts. The type information present in FSGs was used to integrate method invocation resolution.

In the next section we compare our analysis with existing approaches to alias analysis and dependence analysis.

# Chapter 4

# Related Work and Comparison

Our approach basically differs from other related work in three aspects: (i) computation of shapes of data structures, reaching definition information, and type information in the same pass. (ii) Use of multiple summary nodes to separate different regions of the heap by direct and transitive reachability from sets of variables, and (iii) integration of method invocation resolution for object-oriented programs based on the type information in a FSG .

Our analysis can be classified as an alias analysis that integrates the computation of reaching definition information and type information. The alias and type information is essential for method invocation resolution which is integrated in the inter-procedural frame work. The precision of computed reaching definitions mostly relies on the precision of the alias analysis. Thus, we shall compare different approaches to alias analysis with our approach. Also we shall discuss properties of our alias analysis that allow to integrate type and definition information in the same pass.

There is an extensive literature on the analysis of alias problems for languages with pointers. Following [GH96b] we can distinguish between analyses of pointers to (1) *statically* allocated data (typically on the stack) and (2) *dynamically* allocated data (typically on the heap). The analysis of pointers to statically allocated is the simplest: typically the data will have compile time names and the analysis result can be presented as a set of points-to pairs of the form $(p, x)$ meaning that the pointer $p$ points to the data $x$ or as alias pairs of the form $(*p, x)$ meaning that $*p$ and $x$ are aliased.

An analysis of dynamically allocated data is more complicated since the objects of interest are inherently anonymous. There are two main sources of imprecision: firstly, the naming scheme for anonymous objects in the heap may not be capable to give all nodes that may be created in the heap different names and hence, maps different concrete objects - or dependencies in the case of conflict and connection analysis - to the same abstract representation. Secondly, since data structures in the heap are potentially infinite, a finite representation must be capable to represent data structures of unknown size. The better this representation allows to describe different kinds of structures the higher is the precision of the analysis.

In graph based approaches different kinds of nodes serve to represent different kinds of data structures. The maximum length of a path in such graphs can be limited by a fixed constant ($k$-limiting) [JM79, JM81, AZ97, AZ98] or induced by the naming scheme of nodes. For example, in [SRW96, SRW98] the maximum length of a (non-cyclic) path in the graph is limited by the number of variables being used for determining the name of a node. In [LH88] a variation of $k$-limiting is used to limit the number of outgoing edges of a node and also the name of a node, here a label, is limited to some fixed constant. An other approach is to represent the heap structure as path relationships between pointer variables (handles) [HN90].

An other powerful model proposed is to represent an infinite number of access paths by a pair of symbolic access paths qualified by an equation [Deu92, Deu94]. An equation is used to express properties such as "the $i$th element of list X is aliased to the $2i + 1$th element of list Y". This allows to describe infinite structures by a finite representation.

Most of the approaches that we are going to discuss in detail are combinations of several different methods to name anonymous heap objects, limit the size of the representation, and to reason on properties a data structure or object of interest may have.

## 4.1 Statically allocated data

The analysis of pointers to statically allocated data is the simplest: typically the data will have compile time names. The main focus of the analyses presented in this section is analysis of static data with extentions that allow some simple analysis of the heap. Some of the approaches have a heap analysis included that represents all heap objects by one single node in a bounded graph. Others give different names to heap objects either by naming the heap objects by the allocation site where they are created or by call strings.

Landi and Ryder [LR92] use a representation that explicitly describes all alias relations and combine allocation-site naming with k-limiting of object names. They collect alias information in the form of pairs of *object names*. An *object name* consists of a variable and a (possibly empty) sequence of dereferences and field accesses. In the presence of recursive data structures, the number of object names would be infinite and therefore they $k$-limit object names to limit the number of object names. For example, a pair of object names is: (x→next,y→next→next). For $k = 1$, y→next→next is represented by y→next. They also name heap objects according to the allocation site. This naming scheme can sometimes provide better information than connection analysis. The concept is based on the fact that objects allocated at different allocation sites are different objects and that access paths to those different objects cannot be aliases. The major draw back of this method is that alias information is only killed when the left hand side of an assignment does not contain a pointer dereference.

In our approach the allocation site is not relevant to the shape of the data structure. It is only relevant to which variable the reference to a new object is assigned. For example, given an if statement with $x = newA$ in both branches creates one node named $\{x\}$ and the reaching definition information is unioned when both branches meet. If the references to the new objects are assigned to different variables (possibly of the same subtype) we keep the reaching definition information (which includes the label of the new-assignment) from the if-branches apart.

Their method effectively resolves stack-based analyses. It is not designed to accurately handle heap-allocated recursive data structures. In contrast to our analysis they name heap objects to the allocation site. We encode the allocation site into the naming scheme of the this-variable but all other aspects of the naming scheme for nodes in the graph are independent from the allocation site. In contrast to our method they use assumed alias sets as calling contexts whereas we use call strings. An introduction to both approaches can be found in [NNH99].

Emami, Ghiya, and Hendren [EGH94] present a context-sensitive approach that generates a graph representing all invocation paths (in the absence of recursion) and precisely handle indirect calls through functions pointers in C. They focus on analysis of stack-directed pointers, and collect alias information in the form of points-to relationships. A points-to relationship is denoted by a triple ($px, x$,d), which indicates that pointer $px$ definitely (=d) points to the location named $x$, or by a triple ($px, x$,p), which indicates that pointer $px$ possibly (=p) points to the location named $x$. This is similar to the notion of may and must alias analysis. They decouple stack and heap analysis and represent objects in the heap by one abstract location named *heap*. All heap-directed pointers are reported to be pointing to this location. They claim through empirical evidence that exponential behavior is not seen in practice and suggest the use of a memoization scheme to avoid redundant analyses.

Our analysis integrates the analysis of scalar values by the use of the non-object node $\diamond$. Local variables are typically allocated on the stack. The references stored in local variables are represented by edges emanating from the root node $\{\}$. Thus, we integrate pointers from the stack but not to the stack. For languages like Java this is not necessary. We do not integrate arbitrary function pointers but method invocation resolution.

Choi, Burke and Carini [CBC93] compute aliases of pairs of *access paths* [LH88]. Their access paths are similar to object names [LR92]. An access path is constructed from variables, pointer indirection operators, and structure field select operators. However, they do not use access paths to name heap objects. They use a naming scheme based on the position (label $\ell$) in the program, where an anonymous heap object is created, to name it, as in [CWZ90, MCL$^+$93], combined with the $k$-limited approach. They allow up to $k$ instances of an object to have distinct names for each statement. To avoid giving the same name to heap objects created at the same statement, they qualify these named objects with an additional name string that captures call path information. They handle call chains with cycles (due to recursive calls) by not allowing a name string to have duplicate call graph edges in it and hence, the number and length of name strings are bounded by the program size.

In contrast to [CBC93] we use sets of variable names to name the nodes in a FSG . We do not avoid to give the same name to objects created at the same statement but separate objects created in different *iterations* at the same statement. This is achieved if different alias configurations are created in an iteration which is the case when a pointer is used to traverse a data structure.

In general, our analysis focuses on the analysis of dynamic data structures and the data dependencies that may be created by assignments. However, since our analysis is name based it can well be used for analysis of static data as well because every static object has a name and can be represented in our analysis by its name.

## 4.2 Dynamically allocated data

Our analysis focuses on the analysis of dynamic data structures. We use transitive information of variables to separate anonymous objects on the heap by the set of variables that directly or indirectly refer to them. The naming scheme, to use sets of variables, is also used in [SRW96, SRW98, NNH99]. The concept of "propagating" information through nodes in a graph can also be found in [AZ97, AZ98] but our approach can analyze programs with (possibly) cyclic data structures as well whereas in [AZ97, AZ98] it is assumed that programs operate on non-cyclic data structures. On the other hand [AZ97, AZ98] integrates the use of multiple summary nodes whereas [SRW96, SRW98, NNH99] use one single summary node. The advantage of this approach is that they are able to integrate a powerful method named materialization which in consequence allows to perform a strong update on var-edges and sel-edges in any case. Our approach performs a strong update only on *var*-edges and a conditional update on *sel*-edges. The analysis presented in [SRW96, SRW98] is a pure alias analysis. This approach for shape analysis has been continued to a parametric shape analysis via 3-valued logic [SRW99]. In [AZ97, AZ98] reaching definition information, attached to nodes, is computed - but no type information. We have extended this approach to deal with type information as well in [SA00, ASRB99].

### 4.2.1 Connectivity

Ghiya and Hendren presented a practical heap analysis, connection analysis, for C [GH96a]. It is not sensitive to the location of allocation sites in the program. In contrast to other approaches that solve the stack-directed and heap-directed pointer problems simultaneously, they separate the two problems. The stack-based problem is solved with points-to analysis. A hierarchical approach to the heap-directed pointer problem is used. For programs with few heap references, points-to analysis can be used directly (level-0 heap analysis). Points-to analysis gives a very conservative answer by treating the entire heap as one named location. For programs that allocate many disjoint structures, such as scientific programs with dynamically-allocated arrays, connection analysis (level-1 heap analysis) provides useful information about the disjointness of the heap-allocated structures.

They implemented the method in the McCAT compiler, and provide experimental results to show that connection analysis gives a substantially more precise answer than points-to analysis. For applications that use structures which are heavily linked, connection analysis is not effective, and the approach has been extended to provide a more expensive direction/interference/shape analysis that allows to estimate the shape (tree/dag/graph) of each data structure allocated in the heap [GH96b] and disambiguate accesses to disjoint parts of the same data structure.

It simply makes use of the fact that an allocation routine returns a new node not pointed to by any pointer in the program. Further if the call to an allocation routine is embedded inside a user-defined function, the analysis does not lose any precision because of its inter-procedural nature.

In contrast to [GH96b] we do not explicitly discriminate different kinds of data structures. In our terminology we would need to integrate different kinds of nodes to represent lists, trees and dags as nodes. We only represent objects by nodes in a graph and the data structure (which may be represented by one node but this is not the general case) is represented by a graph. We only have an explicit representation of sharing and reason on the kind of a data structure by its shape in the flow-shape graph. Whether a shape represents a tree or an arbitrary graph can be understood from the shape and the declared type of variables in the program. Unlike [GH96b] we do not have a good abstraction for directed graphs because those are subsumed by arbitrary graphs. In general, conflict analysis is less precise than graph oriented approaches that precisely reason on the possible shape of a data structure.

### 4.2.2 $k$-Limited

Jones and Muchnick [JM79, JM81] introduced $k$-limited graphs and analyzed LISP-like structures for a simple language without procedures. For each program point they approximate the structure of the heap in the form of a set of directed graphs. Nodes in the graph represent objects in the heap, while edges represent the links between these objects. To $k$-limit the graph all nodes in the heap accessible from a variable after traversing $k$ or more links, are coalesced into one summary node. Most of the structural information about structures extending beyond the $k$-limit are lost due to the summary process. In [AZ97, AZ98] this concept is extended to use multiple summary nodes. Nodes bound to variables are labeled by variable names which is similar to our approach and the naming scheme for nodes directly pointed to by var-edges. But [AZ97, AZ98] do not handle cyclic data structures and do not have an explicit representation of sharing as [SRW96, SRW98] and as we use in our analysis (see section 2.3.3).

In our approach nodes which can possibly be shared (have more than one parent in the flow-shape graph) or become part of a cycle, are marked as shared if the node is a summary node.

The goal of the analysis in [JM79, JM81] is to optimize storage allocation. The k-limited graphs at all program points are analyzed to classify variables into three categories:

1. Variables which cannot access any shared or cyclic nodes at any program point.
2. Variables which can access shared nodes but no cyclic nodes.
3. Variables which can access cyclic nodes.

All three categories can be obtained from flow-shape graphs as well. Summary nodes which are marked to be shared have to be assumed to be representing cyclic data structures. Single nodes which are shared are not cyclic.

The first variety of variables is represented by edges refering to non-shared single nodes or summary nodes (the sharing of var-edges must be considered here). The second variety of variables is represented by edges refering to shared single nodes (either by var-edges or marked to be induced shared); and the third by edges refering to shared summary nodes. This information is useful to garbage collectors. The reference count of heap cells accessible from the first variety of variables can never exceed one and can be deallocated as soon as pointers to them are destroyed. Those accessible from the second variety can be reference-counted while the rest need to be garbage-collected.

In FSGs these properties can also be determined for the more general case of paths $x.(sel)^*$. $k$-bounded approaches lose information when a data structure is traversed beyond the $k$-horizon. In some cases we are able to traverse a data structure without losing information about the shape of data structure as shown for the search list program in fig. 2.11. In contrast to [JM79, JM81] we use a single graph at each program point whereas Jones and Muchnick use sets of graphs which can become rather expensive for large programs.

The goal of Larus and Hilfinger [LH88] is to detect potential conflicts between heap accesses at different program statements. They use a variation of $k$-limited graphs called *alias graphs* for analyzing Lisp programs. A node in the graph corresponds to one or more structure instances. The edges correspond to pointers. A node label is a pair consisting of a node and a path expression. The label of a node is either the name of a variable, if the node represents a variable, or a path from a variable to a node.

A newly created node is labeled by an aggregate of the labels of the arguments to the allocation function (new). This proves to be more precise than labeling the node by the program point where it is allocated. Unlike [JM79], they maintain only one alias graph per program point instead of a set of graphs. They define a meet operator that combines two alias graphs into a new alias graph that contains all aliases in either graph. Because the result of this operation can be larger than either argument, the meet operator also limits the size of its result by summarizing groups of nodes by a single node with a compound label. It is ensured that no node has more than $s$ outgoing edges (except the element representing the bottom element), and no node has a label longer than $l$, called $s - l$ limiting.

Once an alias graph is computed for each program point, conflict detection is done. A potential conflict exists if access paths at the given statements can lead to a node with the same label in their respective alias graphs. This method is rendered expensive by its complex meet, node summary and node labeling operations.

Horwitz, Pfeiffer and Reps [HPR89] presented another variation on $k$-limited graphs, called storage graphs which abstract the dynamic store. They present a variety of ways to $k$-limit the storage graphs. The goal of their analysis is detection of dependencies between program statements. They label each node in the storage graph with the program point that last set its contents. Their notion of dependence analysis is more precise than conflict analysis of Larus and Hilfinger [LH88], because they take the intervening writes between statements into account. This information can be computed from the reaching definition information integrated in flow-shape graphs as well. In contrast to our approach of representing the alias information by one FSG at each program point they maintain a set of storage graphs at each program point. This makes their analysis more precise, but also more expensive. They use abstract interpretation [CC77] augmented with a fourth semantics called instrumented semantics, to prove the correctness of their technique.

Amme and Zehendner describe in [AZ97, AZ98] a store-less method to determine data dependences in programs with pointers. They use so-called A/D graphs as data flow information set and perform a single-pass data-dependence analysis by solving a monotone data flow system for a class of restricted imperative languages. Based on an intra-procedural analysis with A/D graphs, they develop a method to derive a safe approximation of the data dependences by employing $k$-bounded A/D graphs.

In [AZ97, AZ98] multiple summary nodes are used and reaching definition information is computed in the same pass. But the analysis cannot be applied to programs operating on cyclic data structures. Our analysis is designed to cope with cyclic data structures and integrates multiple summary nodes. In our terminology strong update is only performed when an edge is not an out-edge of a summary node. The strong update on sel-edges is "conditional" in [AZ97, AZ98] because var-edges eventually are pointing to summary nodes if a data structure of unknown size is traversed. In contrast to our analysis they cannot handle out-edges of summary nodes. The combination of cyclic and non-cyclic data structures, using a combination of the naming scheme in [SRW98] and [AZ97, AZ98], and multiple summary nodes aids to the powerfulness of our analysis.

### 4.2.3 Allocation Site

Another approach to abstracting the heap structure in the form of a bounded graph was given by Chase, Wegman and Zadeck [CWZ90]. They proposed the storage shape graph (SSG) in which each node corresponds to one or more objects in the heap and the edges correspond to references. An allocation-site node in the SSG represents all the heap objects that can be allocated at that allocation site. This abstraction can introduce cycles in otherwise un-aliased structures like lists and trees. They use the term “join” for an operation whose result of “joining” two shape-graphs depends on the program point at which the operation is applied. The join operation we use is independent to the program point.

This method gives precise results in some special cases but it can be overly conservative because of one SSG node abstracting several objects created at the same program position. In our analysis names given to an object created at the same position may be different due to different alias configurations in different passes of a loop. For example, [CWZ90] gives highly imprecise results, if the program uses a single routine for allocating nodes. In our analysis the actual shape of a data structure can be created and the alias configurations tracked (see example program create list in section 2.5.1).

Landi and Ryder [LR92] and Ruf [Ruf95] simply name heap objects based on their allocation site (identified by calls to library routines like malloc). With this approach, if the programmer uses his own routine with the malloc call embedded in it, all heap objects get the same name. To overcome this flaw, Wilson and Lam [WL95] summarize the effects of procedures using partial transfer functions (PTFs). A PTF describes the behavior of a procedure assuming that certain alias relationships hold when it is called. With this technique, heap objects allocated at the same allocation site but along different call-chains get distinct names. However this strategy sometimes results in a large set of names, which can slow down the analysis. Further this approach can still give the same name to completely unrelated heap objects.

Hind, Burki, Carini, and Choi give experimental results of comparing flow-sensitive and flow-insensitive flow analysis algorithms in [HBCC99] based on memory locations associated with names. The call graph is constructed while alias analysis is performed. Our approach assumes that a call graph is created before alias analysis is performed.

Pleviak, Chien and Karamcheti [PCK93] have improved the intra-procedural aspects of the analysis in [CWZ90] to handle regular cyclic structures like doubly linked lists and trees with parent pointers, more precisely. They introduce additional nodes called choice nodes, to represent that two in-going edges of a summary node would not exist at the same time. This is similar to unshared summary nodes in our analysis but they do not handle procedure calls.

### 4.2.4 Access Paths

Deutsch [Deu92, Deu94] presented an alias analysis for an imperative subset of ML. This abstraction is particularly suited to recursive data structure analysis. The model, also called a store-less model, uses pairs of symbolic *access paths* qualified by constraints that make them aliased. For certain programs that manipulate lists, it allows to represent the exact infinite set of may aliases in a compact way. Access paths are defined in terms of monomial relations (a kind of multi-variable polynomial expression with structure accessors as the variables). For example, a position dependent alias relationship of the form: "the $i$th element of list $x$ is aliased to the $2i+1$th element of list $y$", can be precisely expressed as $((x \to (l)^i, y \to (l)^j), j = 2i+1)$. How to handle procedure calls is described in general but they do not propose any particular strategy for inter-procedural analysis.

### 4.2.5 Shape Analysis

Ghiya and Hendren [GH96b] focus on identifying un-aliased tree-like and acyclic DAG-like data structures built by the program, and provide conservative estimates otherwise. Shape analysis is designed for programs that primarily use recursive data structures, or a combination of arrays and recursive data structures. Rather than look for a complex abstraction, their approach is to start with simple abstractions that can be implemented in real compilers and examine the usefulness of these simple abstractions with respect to a set of representative benchmark programs. Thus, the main contribution of their work is the design and implementation of practical abstractions to perform shape analysis for an important class of C programs.

Sagiv, Reps, and Wilhelm [SRW96, SRW98] have presented static shape graphs (SSGs) to accurately capture the heap structure in the presence of destructive updates. The use sets of variable names to denote nodes in a SSG. Their method is quite accurate and can also be applied to cyclic data structures. Manipulation of the representation is based on sets of access paths, distinguishing this work from previously published approaches. They use an elegant form of materialization. Therefore, when information appears that is not denoted by the graph, a new node from the summary node is materialized. In contrast to our technique, they only utilize one summary node in their data structure.

The algorithm handles all five of the basic list-manipulation operations with a remarkable degree of precision, as well as similar tree- and circular-list-manipulation operations. An additional component of SSGs distinguishes the abstractions of cyclic concrete structures from the abstractions of acyclic concrete structures. A summary node deliberately loses information about the length of a data structure. The main problem is to distinguish between abstractions of cycles and abstractions of non-cyclic infinite data structures.

Similar to [SRW96, SRW98] we use an explicit representation of sharing to cope precisely with cyclic data structures. The naming scheme in our analysis is similar to that in [SRW96, SRW98] because we also use a set of variable names to name nodes in the graph but we also "propagate" the variable names to all successor nodes in the graph which allows us to integrate multiple summary nodes exclusively based on the names of variables.

That variables indirectly refer to an object is not integrated in the naming scheme in [SRW96, SRW98]. They only use one summary node to represent all objects in the heap that are not directly refered to by a variable. If two variables are aliased and refer to a cyclic data structure all other variables that exist at this given point in the program analysis are also aliased. They have proposed several methods how to integrate more than one summary node into the analysis. They have also proposed several methods to integrate multiple summary nodes like indexing of nodes by allocation sites. Our approach for naming summary nodes is most similar to that presented in [AZ97, AZ98] but with a better worst-case in the maximum number of nodes in a graph. Our approach is not a $k$-limited approach because we do not limit the maximum length of a path in the graph to some fixed $k$. The maximum length of a path in a flow-shape graph is determined by the number of variables existing in a graph. With the number of variables also the maximum length of a path increases.

## 4.3 Type based abstractions

Currently there exist only few techniques that are useful for alias analysis of object-oriented programs.

Diwan, McKinley, and Moss evaluate three alias analysis algorithms based on programming language types. The most precise of these three is a flow-insensitive analysis that uses type compatibility and additional high-level information such as field names [DMM98]. They use redundant load elimination to demonstrate the effectiveness of the algorithms in terms of opportunities for optimization.

The analysis is less precise than our analysis because it is flow insensitive. But they do make use of field names and type compatibility to reduce the dependencies that may exist within a given program. Our approach is to reason on the types of values which is similar to how they organize the type information.

Pande and Ryder [PR94] present a polynomial time combined algorithm to perform program-point-specific, inter-procedural type determination and aliasing algorithm for C++. The algorithm is a work-list based, fixed point iteration method that reports points-to-type and may-hold alias information. In contrast to their approach we represent all information by one single alias graph during analysis. The method described in [PR94] is based on conditional aliasing. This technique analyzes the code of a procedure under certain input assumptions and then combines the results of this conditional analysis with the situations that actually may occur during program execution. Therefore, compared to our technique, this algorithm is more precise when recursive procedure calls are possible.

For programs working on data structures with more than one level of indirection, our technique outperforms the latter, because our algorithm precisely represents aliases of data structures of unknown size and allows strong updates on variables refering to those data structures.

In [DMM96] the effectiveness of four methods for detecting whether a method invocation is monomorphic (always calls the same method) are investigated. They apply type hierarchy analysis, type propagation, aggregate analysis and inter-procedural type propagation (only to scalars). In contrast to our approach they apply type propagation and aggregate analysis in two different passes.

We combine the computation of reaching definitions (which is more precise than aggregate analysis) with alias analysis in one single pass and represent the data in one uniform data structure.

Our representation integrates local variables and instance variables of objects as well as scalar variables. We have a similar worst-case for the maximum number of nodes in a graph ($2^n+1$) as in [SRW96, SRW98] but in our analysis nodes are named by a transitive node naming scheme that includes the name of a variable $x$ in all node naming sets that represents objects are directly or *indirectly* refered to by $x$ in the graph. A cyclic data structure is always represented by one summary node whereas non-cyclic data structures may be represented by various nodes and links between them.

# Chapter 5

# Conclusion

Without dependence analysis, compilers must make worst-case assumptions about pointers. These assumptions impede optimizations and can result in inefficient code and increased compilation time. Before optimizations can be performed the control flow, aliases, data flow, and dependences need to be determined.

In the previous section we discussed several approaches that have been proposed in the literature for computing the aliases generated by assignments of references to dynamic data structures. Based on this information, reaching definitions and in turn the data dependence graph can be determined. In object-oriented programs type information is necessary to determine which overridden methods can never be called by a given program to not create superfluous dependencies for such methods.

We presented a new algorithm for the combined computation of shapes of data structures, reaching definitions and type information of object-oriented programs in the same pass. The analysis was presented as an instance of a monotone framework and the transfer functions for elementary blocks of the program were proved to be monotone.

We introduced flow-shape graphs that are a finite approximated representation of data structures created, traversed, or altered by a program. All essential properties our algorithm is able to determine, are integrated in flow-shape graphs. The shape of a data structure is represented as a graph. The reaching definition information and the type information are edge labels in the graph.

The uniform representation of all different kinds of program information in one single graph for each statement, allowed us to provide an easy to understand visualization of the program information computed.

We implemented a prototype in SETL2 [Sny90] that produces as output a textual description of the generated flow-shape graphs. This description of the graphs was visualized using the tool dot [Kou96]. All flow-shape graphs presented in this thesis have been computed by our prototype implementation.

The purpose of the implementation was two fold. On the one hand it aided the development of the method due to the visualization of the computed program information and on the other hand, it served to verify systematically certain properties of flow-shape graphs developed in this thesis. For example, the property well named (see section 2.3.4) of flow-shape graphs was verified for all combinations of flow-shape graphs generated by various test programs for all combinations of variables for assignments. On the other hand, the properties developed formally, were used to verify the correctness of the implementation. Similarly to the property well named, the monotonicity of the functions was verified.

We used one single graph at each program point to represent all possible aliases of execution paths that meet at a program point. The intraprocedural analysis was developed in section 2. Most aspects regarding the precision of our analysis stem from the design of the transfer functions for the assignments in the intraprocedural framework. In section 2.5 we discussed our method by example for creating, traversing and altering different kinds of data structures. In section 3 we extended the intraprocedural analysis by methods and method calls. The interprocedural framwork used $k$-bounded call strings to separate different calling contexts. The type information present in FSGs was used to perform method invocation resolution.

Given reaching definition information, which is computed by our algorithm, a data dependence graph and a program dependence graph can be computed [ADS91]. Thus, slicing of programs with pointers, dead-code elimination or even dead-data elimintation can be performed.

The program information computed by the algorithm presented in this work can aid the developement of today's software systems by integrating a graphical visualzation of precise dependence information and give precise dependence information to optimzing and parallelizing compilers.

# Bibliography


[ADS91] H. Agrawal, R.A. Demillo, and E.H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proc. ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73, 1991.

[ASRB99] Wolfram Amme, Markus Schordan, Wilhelm Rossak, and Laszlo Boeszoermenyi. PAOLA-program analysis of object-oriented languages. In *ECOOP '99. Intercontinental Workshop on Aliasing in Object-Oriented Systems. In association with the 13th European Conference on Object-Oriented Programming*, pages 143–146, 1999.

[AZ97] Wolfram Amme and Eberhard Zehendner. Experiences in analyzing data dependences for programs with pointers and structures. In *Proceedings of the 3rd International Euro-Par Conference (Euro-Par'97)*, pages 342–346, Passau, Germany, September 1997. Springer Lecture Notes in Computer Science.

[AZ98] Wolfram Amme and Eberhard Zehendner. Data dependence analysis in programs with pointers. *Parallel Computing*, 24(3–4):505–525, May 1998.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 232–245, New York, NY, USA, 1993. ACM Press.

[CC77] P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[CHY95] Liling Chen, Luddy Harrison, and Kwangkeun Yi. Efficient computation of fixpoints that arise in complex program analysis. *Journal of Programming Languages*, 3(1):31–68, 1995.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the PLDI*, pages 296–310, White Plains, New York, 1990. ACM Press. copied.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, pages 2–13, San Francisco, April 1992. IEEE Press. copied.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond $k$-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, June 1994.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, pages 292–305, 1996.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 106–117, Montreal, Canada, 17–19 June 1998.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, June 1994.

[GH96a] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.

[GH96b] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15. ACM SIGACT and SIGPLAN, ACM Press, 1996.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[HBCC99] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 24(7):28–40, July 1989.

[JM79] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, San Antonio, Texas, January 29–31, 1979. ACM SIGACT-SIGPLAN.

[JM81] Neil D. Jones and Steven S. Muchnick. *chapter 4, Flow Analysis and Optimization of LISP-like Structures*. Prentice-Hall, 1981.

[Kou96] Eleftherios Koutsofios. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 1996. This report, and the program, is included in the `graphviz` package, available for non-commercial use at.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 21–34, Atlanta, GE, USA, June 1988. ACM Press. copied.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992.

[MCL+93] Thomas J. Marlowe, Jong-Deok Choi, William Landi, Michael G. Burke, Barbara G. Ryder, and Paul R. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[NNH99] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[PCK93] John Plevyak, Andrew A. Chien, and Vijay Karamcheti. Analysis of dynamic structures for efficient parallel execution. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 37–56, Portland, Oregon, August 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

[PR94] Hemant Pande and Barbara Ryder. Static type determination for C++. In USENIX Association, editor, *Proceedings of the 1994 USENIX C++ Conference: April 11–14, 1994, Cambridge, MA*, pages 85–97, Berkeley, CA, USA, April 1994. USENIX.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. *ACM SIGPLAN Notices*, 30(6):13–22, June 1995.

[SA00] Markus Schordan and Wolfram Amme. Virtual method resolution with typed alias graphs. In *CPC '00. 8th International Workshop on Compilers for Parallel Computers*, pages 151–162, 2000.

[SBF00] B. Scholz, J. Blieberger, and T. Fahringer. Symbolic Pointer Analysis for Detecting Memory Leaks. In *Proc. of the Workshop on*

*Partial Evaluation and Semantics-Based Program Manipulation*, pages 104–113, Boston, MA, USA, January 2000. ACM Press.

[Sny90] W. Kirk Snyder. The SETL2 programming language. Technical Report 490, New York University, Courant Institute of Mathematical Sciences, Computer Science Division, 251 Mercer St., New York, NY, January 1990.

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: The* 23$^{rd}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, 21–24 January 1996.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 105–118, New York, NY, USA, 1999. ACM Press.

[Ste98] Christoph Steindl. Intermodular slicing of object-oriented programs. In *Proc. Int. Conf. Compiler Construction, CC*, volume 1383 of *Lecture Notes in Computer Science, LNCS*, pages 264–278. Springer-Verlag, 1–3 April 1998.

[TS00] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36(12):927–982, 2000.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Notices*, 30(6):1–12, June 1995.