THESIS

AUTOMATIC DETERMINATION OF MAY/MUST SET USAGE IN DATA-FLOW ANALYSIS

Submitted by

Andrew Stone

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2009

COLORADO STATE UNIVERSITY

May 6, 2009

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY ANDREW STONE ENTITLED AUTOMATIC DETERMINATION OF MAY/MUST SET USAGE IN DATA-FLOW ANALYSIS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate Work

Committee Member (Sanjay Rajopadhye)

Committee Member (Jiangguo Liu)

Adviser (Michelle Strout)

Department Chair (Darrell Whitley)

ABSTRACT OF THESIS

AUTOMATIC DETERMINATION OF MAY/MUST SET USAGE IN DATA-FLOW ANALYSIS

Data-flow analysis is a common technique for gathering program information for use in performance improving transformations such as register allocation, dead-code elimination, common subexpression elimination, and scheduling. Current tools for generating data-flow analysis implementations enable analysis details to be specified orthogonally to the solution algorithm, but still require implementation details regarding the may and must use and definition sets that occur due to the effects of pointers, side effects, arrays, and user-defined structures. This thesis presents the Data-Flow Analysis Generator tool (DFAGen), which enables analysis writers to generate pointer, aggregate, and side-effect cognizant analyzers for separable and nonseparable data-flow analyses, from a specification that assumes only scalars. By hiding the compiler-specific details behind predefined set definitions, the analysis specifications for the DFAGen tool are typically less than ten lines long and similar to those in standard compiler textbooks. The two main contributions of this work are the automatic determination of when to use the may or must variant of a predefined set reference in the analysis specification, and the design of the analysis specification language so that data-flow problem and compiler framework implementation details are specified orthogonally.

Andrew Stone
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Summer 2009

ACKNOWLEDGEMENTS

Foremost, I would like to thank my advisor, Prof. Michelle Strout. During the past four years, as both an undergraduate and a graduate student, I have learned a lot from Michelle. Her guidance, advice, support, and encouragement have helped me conduct research, complete this thesis, and gain a better understanding of what research, academics, and computer science is.

I would also like to thank Shweta Behere for her research contributions with DFAGen, and Lisa Knebl for her editorial suggestions while I was writing a related journal paper.

Thanks also go to my family, friends, and fellow graduate students – they have certainly made the past few years pleasant, fun, humorous, and memorable.

TABLE OF CONTENTS

Chapter 1

Introduction

Compile-time program analysis is the process of gathering information about programs to effectively derive a static approximation of their runtime behavior. This information can be used to optimize programs, aid debugging, verify behavior, and detect potential parallelism. Data-flow analysis is a commonly used technique to perform compile-time analysis.

1.1 The Problem

A number of tools have been introduced that ease the process of implementing data-flow analyzers. These tools enable an orthogonality between analysis specification and the algorithm used to determine a solution [10, 12, 14, 19, 20, 22, 28, 38, 39, 42, 43]. However, they still require implementation details regarding when to use the may versus the must variants of variable-definition and variable-use sets. May and must variants occur due to the effects of pointers, side effects, arrays, and user-defined structures. Such details make analysis specifications more verbose and complex than what is typically seen in compilers textbooks [9, 11, 13]. Definitions of analyses in these textbooks are often written assuming that analyzed programs consist only of scalars, and have no pointers. The scalar assumption eliminates the requirement to determine when to use may versus must information. However,

- $\text{in}[s] = \bigcup_{p \in \text{pred}[s]} \text{out}[p]$
- $\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$
- $\text{gen}[s] = s\ ,\ \text{if}\ \ \text{defs}[s] \neq \emptyset$
- $\text{kill}[s] = \text{all}\ t\ \text{such that}\ \text{defs}[t] \subseteq \text{defs}[s]$

Figure 1.1: Data-flow equations for reaching definitions, where s, p, and t are program statements, $in[s]$ is the data-flow set of definition statements that reach statement s, $pred[s]$ is the set of statements that immediately proceed s in the control-flow graph, and $defs[s]$ is the set of variables assigned at statement s.

most real world programs consist of more than just scalars, which the analysis implementation must handle for correctness.

1.2 Introduction to Data-Flow Analysis

Gary Kildall introduced the technique of data-flow analysis in 1973 [25]. This technique computes sets of facts, at each program point, that are guaranteed to be true for all possible executions of the program. Compiler textbooks usually describe data-flow analysis in terms of data-flow equations [9, 11, 13], such as those in Figure 1.1.

Solving a data-flow problem is done by determining a solution such that all data-flow equations are satisfied. Figure 1.1 shows a specification of reaching definitions using such equations. Reaching definitions is a compile-time program analysis, which determines, at each program point, the set of variable definitions that may have occurred without any intervening writes. For each statement `s`, in the analyzed program, there is an associated `in` and `out` data-flow set (for reaching definitions these sets contain statements). A solution to this data-flow analysis problem is an assignment of data-flow values to all `in` and `out` sets, such that they satisfy the equations. Figure 1.2 shows a control-flow graph for an example

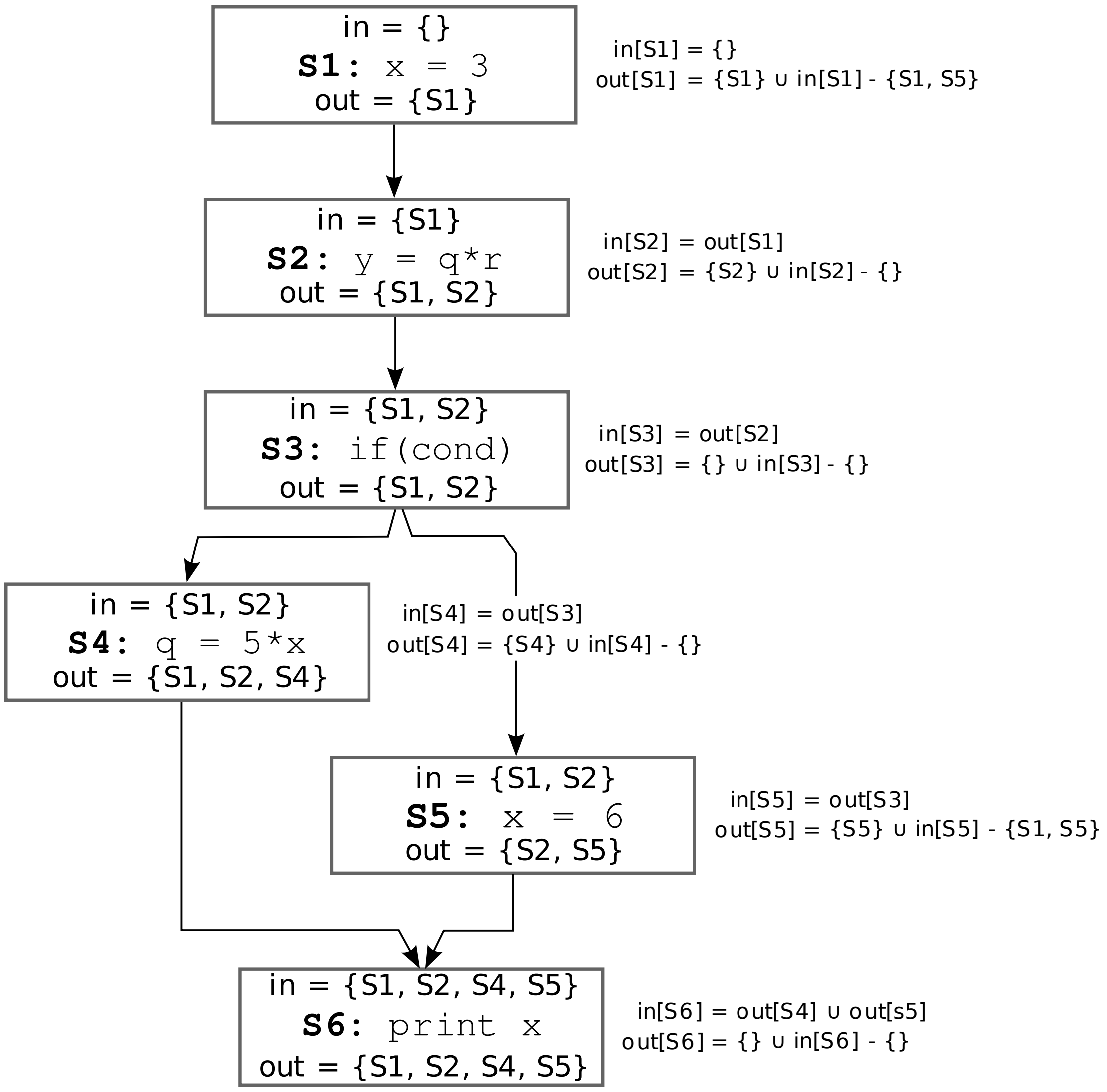

Figure 1.2: Solutions to `in` and `out` data-flow equations for reaching definitions.

program, and what `in` and `out` data-flow equations evaluate to when reaching definitions is applied (the equations in Figure 1.1).

Reaching definitions results are useful for determining when simple constant propagation transformations can safely be applied. For example, in Figure 1.2, an optimizer will be unable to replace the use of variable `x` at statement `S6` with its definition since multiple definitions of the variable reach the statement. However, in contrast, the use of variable `x` at statement `S4` could be replaced by its definition

```
     int a, b;
     int *p;
S1   if(input() > 100) {
S2      p = &a;
S3   } else {
S4      p = &b;
     }
S5   *p = b * 2;
```

Figure 1.3: We can only say what may be defined at statement S5, but we can state what must be used.

to `3` in `S1`. Further optimization would be able to collapse this expression (`5*3`) into `15`, thus evaluating the expression at compile time rather than runtime.

Another common data-flow analysis is Liveness, which determines the set of variables at each program point that have previously been defined and may be used in the future. Liveness is useful for detecting uninitialized variables and generating program dependence graphs [16]. It is also used for dead code elimination and register allocation. Other program optimizations that use data-flow analysis results include busy-code motion, loop invariant code-motion, partial dead-code elimination, assignment motion, and strength reduction [9]. In addition to optimization, data-flow analyses are used in program slicers and debugging tools [40].

1.3 The Data-Flow Analysis Generator Tool and May/Must Analysis

This work describes a tool we designed and implemented, DFAGen - the Data-flow Analysis Generator, that is able to generate data-flow analysis implementations from succinct descriptions written in a declarative domain-specific data-flow analysis language. The DFAGen analysis specification language is able to maintain an “analysis for scalars only" abstraction, white still generating analyzers that are cognizant of the may and must effects of pointers, aggregates, and side-effects.

This is possible due to the may/must analysis algorithm DFAGen uses to automatically resolves when to use may versus must variable-define and variable-use information. Specification of analyses are separated from may and must details by hiding compiler-specific details behind predefined set definitions and type mappings. These techniques enable DFAGen analysis specifications to be less than ten lines long and similar to those in standard compiler textbooks.

The issue of when to use may versus must information arises, in part, due to statements containing dereferences to pointers, function calls, and/or the use of aggregate data structures such as arrays or user-defined types. Such language features result in there being two variants of the statement-specific `def` and `use` sets, one for may definitions or uses and another for must definitions or uses. For example, in Figure 1.3, the `maydef` set for statement `S5` is {`a, b`}, the `mustdef` set is the empty set, the `mayuse` set is {`b`}, and the `mustuse` set is {`b`}. The difference between the `maydef` and `mustdef` set occurs because of multiple possible paths of control flow and pointer dereferencing.

Figure 1.4 shows a specification of reaching definitions that incorporates may and must information. Compiler textbooks do not typically present specifications with information incorporated like this. Since existing data-flow analysis implementation tools do not resolve this issue automatically users of such tools are responsible for determining when the may and must variants should be used in the analysis specification. Chapter 2.2 discusses the issue of may/must sets in more detail.

The specific contributions of this work are as follows:

- DFAGen automatically generates unidirectional, intraprocedural data-flow analysis implementations from succinct descriptions. These descriptions do not indicate whether sets refer to their may or must variant, and thus main-

- $\text{in}[s] = \bigcup_{p \in \text{pred}[s]} \text{out}[p]$
- $\text{out}[s] = \text{maygen}[s] \cup (\text{in}[s] - \text{mustkill}[s])$
- $\text{maygen}[s] = s$, if $\text{maydef}[s] \neq \emptyset$
- $\text{mustkill}[s] =$ all t such that $\text{maydef}[t] \subseteq \text{mustdef}[s]$

Figure 1.4: Data-flow equations for reaching definitions that are cognizant of may and must definitions due to aliasing, side-effects, and/or aggregates.

tain a "data-flow analysis for scalars" abstraction. DFAGen is able to automatically determine which variant to use by performing an analysis called may/must analysis. We will explain how we derived this analysis by examining how operators affect when may versus must information is required. We also discuss how our current implementation can be extended for new operators.

- The DFAGen specification language was designed so that data-flow problem specification and compiler-framework implementation details are specified orthogonally. Due to the hiding of compiler infrastructure details in the predefined set definitions, type mappings, and implementation template files, a single analysis specification could be used to generate an analysis across a wide variety of compilers.

1.4 Thesis Organization

The remaining chapters give background material, document the DFAGen tool and its contributions, and evaluate it. Specifically, this thesis is organized as follows:

- Chapter 2 discusses background material related to data-flow analysis, which will be useful in understanding the rest of the thesis and its contributions.

It reviews the concept of a data-flow framework and how such a framework enables the specification of an analysis problem and its solution. It also gives several examples of may/must issues that arise in modern languages, and how these issues complicate implementing data-flow analyses.

- Chapter 3 documents the DFAGen tool's architecture and specification language. It describes how this language enables a specification where analysis, compiler, and language specific concerns are specified orthogonally. This chapter also describes how DFAGen can be retargeted to output analyzers for various compiler infrastructures. Finally, the chapter describes how the tool is invoked from the command line, and how currently generated analyzers incorporate into the OpenAnalysis framework.

- Chapter 4 describes in detail the phases DFAGen undergoes to compile a data-flow analyzer from a specification. It also describes the algorithm DFAGen uses to determine may/must set usage and describes how this algorithm is derived.

- Chapter 5 evaluates a prototype implementation of DFAGen by comparing the size and performance of DFAGen generated analyses against handwritten versions.

- Chapter 6 describes other data-flow analysis frameworks and generator tools and qualitatively compares them to the DFAGen tool.

- Chapter 7 discusses the limitations of our current implementation of DFAGen, proposes methods of overcoming these limitations, and ends with some concluding remarks.

Chapter 2

Background

The DFAGen analysis specification language is based on lattice theoretic frameworks. Specifying analysis in terms of such a framework is useful because it ensures that an answer will be converged upon. This chapter reviews these frameworks and examines may/must issues, which can complicate implementing analyzers from the formalizations that these frameworks impose.

2.1 Data-Flow Frameworks

One advantage of lattice theoretic frameworks is that they enable a separation of concerns between the logic for a specific analysis and the solution algorithm and proof of convergence. The analysis is specified as a mathematical structure. Generic solution algorithms exist that can solve any analysis defined by such a structure. In a lattice theoretic framework an analysis is defined as a set of transfer functions, a set of initial values, a direction (either forward or backward), and a lattice of flow values [9, 25, 30].

A lattice is a set of values and a meet operator. The meet operator is a binary relation over the values in that lattice, and satisfies the closure, commutativity, and associativity properties. Lattices defined a partial ordering among the flow-values.

Transfer functions are used to compute sets of data-flow values at each state-

- $\text{in}[s] = \bigcup_{p \in \text{pred}[s]} \text{out}[p]$
- $\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$
- $\text{gen}[s] = s$, if $\text{defs}[s] \neq \emptyset$
- $\text{kill}[s] =$ all t such that $\text{defs}[t] \subseteq \text{defs}[s]$

Figure 2.1: Data-flow equations for reaching definition. This figure is the same as Figure 1.1.

ment. Analysis results are the resulting sets these functions produce. In a forward analysis these sets are called `out` sets, in a backward analysis these are called `in` sets. For many analyses the transfer function computes `in` or `out` sets using statement-specific `gen` and `kill` information, and a meet set. Statement specific `gen` and `kill` information is computed using `gen` and `kill` functions, which are parameterized with the statement, and sometimes the meet set at that statement. In a forward analysis meet sets are `in` sets, in a backward analysis they are `out` sets. Meet sets are computed by meet operators, which combine the data-flow sets of the previous or successive nodes (depending on whether the analysis is forward or backward).

The example in Figure 2.1 shows a lattice theoretic definition of reaching definitions analysis. The analysis direction is forward and the meet operator is union (as can be seen in the definition of `in`). If the transfer function can be cleanly broken into statement-specific sets, such as `def`, then most of the implementation work is focused in writing code that generates those sets for each statement type.

The lattice theoretic formalization has been leveraged by a number of tools, which ease the implementation of data-flow analyses [10, 12, 14, 19, 20, 22, 28, 38, 39, 42, 43]. Chapter 6 describes many of these tools in more detail.

```
     int a, b, c;
     int *pointsToOne;
     int *pointsToTwo;
S1   a = ...
S2   b = ...
S3   pointsToOne = &a;
S4   if(a < b) {
S5      pointsToTwo = &a;
     } else {
S6      pointsToTwo = &b;
     }
S7   *pointsToOne = ...;
S8   *pointsToTwo = ...;
```

Figure 2.2: May/must issues that arise because of pointer aliasing.

2.2 May/Must Issues

Analysis implementation can be complicated, even when lattice theoretic definitions of the analysis exist. Lattice theoretic definitions do not always explicitly specify how may and must variable-definition and variable-use information should be used in transfer functions. One of the important contributions of this work is the automatic determination of when to use such information. To aid in understanding why may and must information arises, this chapter describes three examples that demonstrate may/must behavior due to pointers, side effects, and aggregates.

Figure 2.2 is a C program that contains aliasing due to pointer variables. We can mentally analyze this program and claim that the definition at statement `S7` must be to the variable `a`, since at statement `S3`, the variable `pointsToOne` is assigned the address of `a`, and this assignment is not later overwritten. We can also assert a slightly weaker claim that the definition at statement `S8` may either be to variable `a` or to variable `b`. This claim is weaker because control-flow ambiguity forces us to consider the possibility of either definition of pointer variable

```
     int a;
     int *passedToFunc;
S1   a = 1;
S2   passedToFunc = &a;
S3   foo(passedToFunc);
S4   if(*passedToFunc) {
S5      ...
     } else {
S6      ...
    }
```

Figure 2.3: May/must issues that arise because of side-effects.

`pointsToTwo` (at statements `S5` and `S6`). For any statement that defines or uses variables we can ask two questions: 1) what variables *must* be defined (or used) when executing this statement, and 2) what variables *may* be defined (or used) when executing this statement. Many data-flow analyses will require answers to one or both of these questions at all program points. For example, to determine what definitions to kill reaching definitions requires the *must* definitions at each statement. A reaching definitions analyzer determining what to kill at statement `S8` will be unable to kill the definitions from statements `S1` and `S2` since it has no must definition to the variables defined at these statements. On the other hand, since the pointer variable `pointsToOne` must point at variable `a`, statement `S7` will be able to kill the definition of variable `a` (in statement `S1`).

Figure 2.3 shows how may/must issues can arise due to side-effects. It may be the case that the value of the variable `a` is modified by the call to the function `foo` at `S3`, since its address is passed. Since the value of `a` *may* be changed, a conservative assumption is that the variable used at statement `S4` may be `a`. On the other hand, if a particularly good side-effect analysis were run, it might recognize that under no execution of the function `foo` will the value being pointed at by its argument change. In which case the only definition of the variable `a` to reach `S4` would be

```
     struct tuple {int val1, int val2};
     int *tuplePtr1, *tuplePtr2,
         *tuplePtrWhole;
S1   tuple pairA(10, 20);
S2   tuple pairB(10, 20);
S3   tuplePtr1 = &pairA.val1;
S4   if(rand() > .5) {
S5      tuplePtr2 = &pairA.val2;
     } else {
S6      tuplePtr2 = &pairB.val2;
     }
S7   tuplePtrWhole = &tuple;
S8  *tuplePtr1 = ...;
S9  *tuplePtr2 = ...;
S10 *tuplePtrWhole = ...;
```

Figure 2.4: May/must issues that arise because of aggregates.

from S1. Given this fact, and that the assignment at S1 sets a to 1, an optimizer could safely remove the false branch of the if statement.

Figure 2.4 illustrates may/must behavior that arises due to aggregates. The variables tuplePtr1 and tuplePtr2 point to individual elements of a larger tuple structure. At statement S8 the variable that must be defined is the individual element pairA.val1. However, at statement S9 the variable that is defined may be one of {pairA.val1, pairB.val2}. On statement S7 the variables that must be defined are all the elements in pair and thus the must set is {pairA.val1, pairA.val2}.

May and must behavior is not limited to may and must variable use and definition. Unresolved control flow within a single statement can bring about may/must behavior for expression generation. For example in the C statement: a = ((b == test) ? c : d), c and d are may expressions while the assignment is a must expression. Aliasing due to pointers can also affect expression generation. The may/must sets of expressions generated by the syntax: `*x + *y`, is dependent on what *x and *y may and must point to. Analyses such as available expressions require information about what expressions may and must be generated at a statement.

An early edition of the dragon book [8], which is a compiler textbook, has a section describing how data-flow analyses can be implemented so as to make use of pointer information. However, this book does not describe how to automatically dertermine when may or must variants should be used within the implementation of transfer functions. The goal of may/must analysis is to do this.

Chapter 3

Using the DFAGen Tool

Algorithmically determining when to use may versus must information in a transfer function necessitates a formal specification of the transfer functions. Transfer functions in DFAGen are defined using DFAGen's domain specific data-flow analysis language. This chapter describes this language and the tool which uses it. This chapter also describes how the DFAGen tool can be targeted to generate code for various compiler infrastructures.

Specifically, this chapter 1) describes the DFAGen tool's architecture, 2) elaborates on the class of data-flow problems expressible within DFAGen, 3) presents the analysis specification language, 4) illustrates how predefined sets enable extensibility and reuse between analysis specifications, 5) describes type mappings, 6) discusses how the DFAGen tool is targeted for use within a compiler infrastructure, and 7) describes how the tool is invoked from the command line.

3.1 Architecture

Figure 3.1 illustrates DFAGen's input, output, and phases. The DFAGen tool is passed a set of input files that contain analysis specifications, predefined set definitions, and type mappings. The code generation component uses template files to guide code generation. The generated code is a set of source files intended

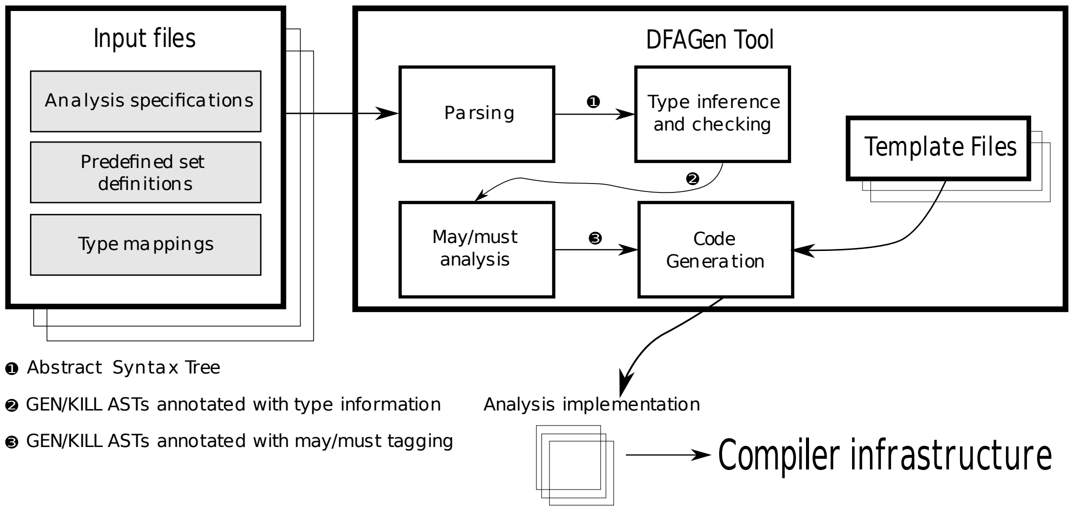

Figure 3.1: Architecture of DFAGen: Input files are passed to the tool, the tool undergoes a series of phases, transforming an abstraction of the analysis (labeled on the edges), to eventually output a series of source files that can be linked against a compiler infrastructure to include the analysis. The code generation phase uses template files to direct its output.

$$\begin{array}{rcl} \textit{Specification} & \Rightarrow & \textit{Structure}^{*} \\ \textit{Structure} & \Rightarrow & \textit{AnalysisSpec} \\ & & \textit{PredefinedSetDef} \\ & & \textit{TypeMapping} \end{array}$$

Figure 3.2: Grammar for input files. The grammars for the AnalysisSpec, PredefinedSetDef, and TypeMapping nonterminals are illustrated in Figures 3.3, 3.6, and 3.9.

to be linked with a compiler infrastructure.

The specification for each of these entities: analysis specifications, predefined set definitions, and type mappings, are represented separately entities in the DFAGen input language. Figure 3.2 shows the grammar for this language. Different users will be concerned with different structures. We envision three types of DFAGen tool users:

1. Analysis writers will want to use DFAGen to specify data-flow analyses. DFAGen is structured so that the analysis specification is not tied to a

particular compiler infrastructure. Users who write analyses need to know DFAGen's analysis specification language, outlined in Chapter 3.3, but do not necessarily need to know the details regarding type mappings or how predefined sets are defined, provided these structures have previously been defined.

2. Compiler writers will want to retarget DFAGen so that it is able to generate data-flow analyses for use within their compiler infrastructure. Currently, we target the tool to the OpenAnalysis toolkit [37]; however, by changing template files as outlined in Chapter 3.6, the tool can be retargeted to work with other compiler infrastructures.

3. Some users may already have DFAGen targeted to generate code for use with their compiler, but will need to create new predefined set definitions and type mappings to specify new analyses. Chapter 3.4 describes predefined sets in more detail; Chapter 3.5 describes type mappings.

3.2 The Class of Expressible Analyses

Currently, the DFAGen tool generates unidirectional, intraprocedural data-flow analyzers for analyses that satisfy certain constraints. These constraints are that the data-flow value lattice is of finite height (although infinite width is allowed), the domain of the data-flow values must contain sets of atomic data-flow facts, the meet operation is restricted to union or intersection, and the transfer function is in one of the following formats:

- $out[s] = f(s, in) = gen[s] \cup (in - kill[s])$ for forward, locally separable analyses
- $in[s] = f(s, out) = gen[s] \cup (out - kill[s])$ for backward, locally separable analyses

- $out = f(s, in) = gen[s, in] \cup (in - kill[s, in])$ for forward, nonlocally separable analyses

- $in = f(s, out) = gen[s, out] \cup (out - kill[s, out])$ for backward, nonlocally separable analyses

where the `gen` and `kill` sets can be expressed as a set expression consisting of predefined sets, set operations, sets defined with set builder notation, and the `in` or `out` set.

Atomic data-flow facts are facts that do not intersect with any other data-flow facts. For example, when the universal set of data-flow facts is the domain of variables, there can be no variable that represents the aggregate of several other variables in that domain. To represent an aggregate structure, a data-flow set must either consist of several elements that represent disjoint substructures, or contain a single element representing the whole aggregate structure. This condition is required to enable the use of set operations in the meet and transfer functions. This condition has an impact on what pointer analysis, or alias analysis algorithms, can be used to create the may and must variants of predefined sets. For example, pointer analysis algorithms that result in the mapping of memory references to possibly overlapping location abstractions [41] do not satisfy the condition.

The assumed transfer function formats enable the specification of both separable [32] and nonseparable [34] analyses. Separable analyses are also called independent attribute analyses [29]. Nonseparable analyses are those that have `gen` and `kill` sets defined in terms of the `in` or `out` parameter passed to f.

Common examples of locally separable problems are liveness, reaching definitions, and available expressions. Examples of nonseparable analyses are constant propagation and vary and useful analysis [21, 26]. Vary and useful analysis are used by activity analysis, an analysis used by automatic differentiation software

$$
\begin{array}{rcl}
\mathit{AnalysisSpec} & \Rightarrow & \textbf{Analysis} : \mathit{id} \\
 & & \quad \textbf{meet} : \\
 & & \qquad (\textbf{union} \mid \textbf{intersection}) \\
 & & \quad \textbf{flowtype} : \\
 & & \qquad (\mathit{id} \mid \mathit{id}\ \textbf{isbounded}) \\
 & & \quad \textbf{direction} : \\
 & & \qquad (\textbf{forward} \mid \textbf{backward}) \\
 & & \quad \textbf{style} : (\textbf{may} \mid \textbf{must}) \\
 & & \quad (\textbf{gen}[\,\mathit{id}\,] : \mid \textbf{gen}[\,\mathit{id},\ \mathit{id}] :)\ \mathit{Set} \\
 & & \quad (\textbf{kill}[\,\mathit{id}\,] : \mid \textbf{kill}[\,\mathit{id},\ \mathit{id}] :)\ \mathit{Set} \\
 & & \quad \textbf{initial} : \mathit{Set} \\
\mathit{Set} & \Rightarrow & \mathit{id}[\mathit{id}] \mid \mathit{BuildSet} \mid \mathit{Expr} \mid \textbf{emptySet} \\
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \ \mathit{Op}\ \ \mathit{Expr} \mid \mathit{Set} \\
\mathit{Cond} & \Rightarrow & \mathit{Expr}\ \ \mathit{CondOp}\ \ \mathit{Cond} \mid \mathit{Expr} \\
\mathit{Op} & \Rightarrow & \textbf{union} \mid \textbf{intersection} \mid \textbf{difference} \mid \\
\mathit{CondOp} & \Rightarrow & \textbf{and} \mid \textbf{or} \mid \textbf{subset} \mid \textbf{superset} \mid \\
 & & \textbf{equal} \mid \textbf{not equal} \mid \textbf{proper subset} \mid \\
 & & \textbf{proper superset} \\
\mathit{BuildSet} & \Rightarrow & \{\mathit{id} :\ \mathit{Cond}\}
\end{array}
$$

Figure 3.3: Grammar for analysis, gen, and kill set definition.

to determine what variables contribute to the evaluation of a dependent variable, given a set of independent variables. Constant propagation is an example of a nonseparable analysis, but it is an analysis that the specification language in the DFAGen tool is unable to express. Constant propagation is not expressible because its transfer function specification requires the evaluation of an expression based on the incoming data-flow set.

3.3 DFAGen Analysis Specification Language

When the DFAGen tool is invoked, it is passed one or more files. Each file contains one or more analysis specification(s), predefined set definitions, and type mappings. This section presents the analysis specifications.

Figure 3.3 shows the grammar for analysis specification. The analysis spec-

```
Analysis: ReachingDefinitions
  meet: union
  flowvalue: stmt
  direction: forward
  style: may
  gen{s]:
    {s | defs[s] != empty}
  kill[s]:
    {t | defs[t] <= defs{s]}
```

Figure 3.4: DFAGen specification for reaching definitions. Note that `<=` is interpreted as a subset operator.

ification includes a set of properties, input values, and transfer functions. The properties include the meet operation, data-flow value element type, analysis direction, and analysis style (may or must), and optionally whether there is a bound on the number of possible data-flow values. If there is such a bound, then for analysis efficiency, generated implementations will use bit-vector sets to implement the data-flow sets.

The initial predefined set indicates how to populate the `out`/`in` set for the entry/exit node in a forward/backward analysis, which is required for many non-separable analyses. If no initial value is specified then the empty set is used as a default.

Transfer functions are specified by assigning the `gen` and `kill` properties to set expressions consisting of predefined set references and set operations. Set operations include `union`, `intersection`, `difference`, and set constructors that build sets consisting of all elements where a conditional expression holds. Conditional expressions are specified in terms of conditional operations such as `subset`, `properSubset`, `==`, and logical operators such as `and`, `or`, and `not`.

Figure 3.4 shows an example specification for reaching definitions. Note how similar this specification is to those seen in compiler textbooks. Each property

```
Analysis: Vary
  meet: union
  flowvalue: variable
  direction: forward
  style: may
  gen[s, IN]:
    {x | (x in defs[s]) and
         (IN intersect uses[s]) != empty}
  kill[s]:
    defs[s]
  initial: independents
```

Figure 3.5: Vary analysis, a nonlocally separable analysis.

is specified with a simple keyword, for example, the meet operation for reaching definitions is specified with the `union` keyword. In the example, the `gen[s]` and `kill[s]` expressions reference the predefined set `defs[s]`, which is the set of definitions generated at statement `s`.

Figure 3.5 shows an example specification for vary analysis. Vary analysis is nonlocally separable and as such the `gen` equation is parameterized by the incoming set (i.e. `in` set for this analysis). Note that due to the use of the `initial` property in the specification, the `out` set for the entry node in the control-flow graph will be set to the predefined set `independents`. The independents set is the set of input variables that the vary analysis should use when determining transitive dependence.

3.4 Predefined Set Definitions

Predefined sets map program entities such as statements, expressions, or variables to may and must sets of other program entities that are atomic. The may and must sets for a predefined-set are called its variants. These sets are predefined in the sense that they are computed before applying the iterative solver on the data-flow

$$\textit{PredefinedSetDef} \Rightarrow \quad \begin{array}{l} \textbf{predefined} : id[\,id\,] \\ \quad \textbf{description} : line \\ \quad \textbf{argument} : id\; id \\ \quad \textit{CalculatedSet} \,|\, \textit{ImportedSet} \end{array}$$

$$\textit{CalculatedSet} \Rightarrow \quad \begin{array}{l} \textbf{calculates} : \\ \quad (id \,|\, \textbf{set of}\; id)\;,\; id,\; id \\ \textbf{maycode} : \\ \quad code \\ \textbf{end} \\ \textbf{mustcode} : \\ \quad code \\ \textbf{end} \end{array}$$

$$\textit{ImportedSet} \Rightarrow \quad \begin{array}{l} \textbf{imports} : \\ \quad (id \,|\, \textbf{set of}\; id)\;,\; (id \,|\, \textbf{none}),\; (id \,|\, \textbf{none}) \end{array}$$

Figure 3.6: Grammar for predefined set definition. The first `id` in the argument property specifies the type of element, the second specifies the identifier of a variable used to index variables in the set. The first and second `id`'s in the calculates and imports properties specify a data-flow value type, the third and fourth are identifiers for variables in the implementation where the may and must variants should be stored. For the calculates property these may be set to `none` which specifies that there is not a may or not a must variant. The code sections under the `maycode`/`mustcode` properties assign values to these variables respectively. The non-terminal value `line` (in the description property) is any text up to a newline.

```
predefined: vary[s]
    description: Results from vary analysis
    argument: stmt s
    imports: setof variable, mayVary, none
```

Figure 3.7: Predefined set definition to import results from vary analysis

analysis equations. When a predefined set is referenced in a data-flow equation, DFAGen is able to determine whether to use the may or must variant in the generated code by performing may/must analysis. Predefined sets are used to abstract compiler infrastructure specific details away from the compiler-agnostic analysis specification. Figure 3.6 shows the grammar for how users define predefined sets in DFAGen.

There are two types of predefined sets: imported sets and calculated sets. Imported sets are passed to the analysis before it is invoked. When an analysis makes use of an imported set, it is the responsibility of the user invoking the analysis to construct and pass the set in.

Imported sets are useful for passing the results of one analysis (including a DFAGen generated analysis) to another. For example activity analysis makes use of the results of vary analysis and useful analysis. Figure 3.7 shows a predefined set definition for the imported set `vary`. This definition does not supply an identifier for the must variant of the set, since this is the case the set `vary` will not have a `must` variant.

```
predefined: defs[s]
    description: Set of variables defined at a given statement.
    argument:    stmt s
    calculates:  set of var, mStmt2MayDefMap, mStmt2MustDefMap
    maycode:
        /* C++ code that generates a map (mStmt2MayDefMap) of
           statements to may definitions */

    mustcode:
        /* C++ code that generates a map (mStmt2MustDefMap) of
           statements to must definitions */
    end
```

Figure 3.8: Predefined set definition for `defs[s]`.

Constructed sets, for a particular specification, are computed by the generated analyzer. The analyzer uses the code specified in the `maycode` and `mustcode` properties of the predefined set definition. The code in these properties are compiler specific and have access to the alias and side-effect analysis results that will be passed to the analyzer. The C code commented out in Figure 3.8 uses this in-

```
TypeMapping ⇒ type : id
              impl_type : line
              dumpcode :
                  code
              end
```

Figure 3.9: Grammar for type mappings.

formation to generate may and must `def` and `use` sets for all statements in the program. Specifically, the code uses must point-to and may point-to information from the alias analysis results to build the may and must sets.

Common predefined sets include "variables defined at statement" (`defs[s]`), "variables used at statement" (`uses[s]`), and "expressions generated in a statement" (`exprs[s]`).

3.5 Type Mappings

Type mappings map the types in the analysis specification language to implementation types in the compiler infrastructure. Specification types are used to specify the `flowvalue` property in analysis specifications, the type of the `argument` for predefined sets, and the type of the predefined set itself, which is specified as the `calculates` property in a predefined set definition. Implementation types are the types used in generated code. For example, a specification type such as `variable` would map to an implementation type that is the class or structure the targeted infrastructure uses to represent variables.

The following example shows a type mapping for variables in our current prototype of the DFAGen tool:

```
type: var
    impl_type:  Alias::AliasTag
    dumpcode:
        iter->current().dump(os, *mIR, aliasResults);
    end
```

Table 3.1: Macros recognized by DFAGen code generator. Language specific macros currently output C++ code. Targeting these macros to a different language requires modifying the code-generator.

Language independent macros	
Macro	Description
NAME	name of the analysis
SMALL	name of the analysis in lower-case letters
MEET	meet operator (union or intersect)
FLOWTYPE	flow-type of the analysis
DIRECTION	direction of the analysis (forward/backward)
STYLE	style of the analysis (may/must)

Language specific macros	
Macro	Description
GENSETCODE	code to calculate the gen set for a given statement
KILLSETCODE	code to calculate the kill set for a given statement
PREDEF_SET_DECLS	code to declare variables that will contain predefined sets
INPUT_SET_PARAMS	code that lists the input sets that are passed into the analysis as parameters
PREDEF_SET_CODE	code to calculate the values included in a predefined set
DUMPCODE	code to output the current state of the analysis
CONTAINER	type of container to store data-flow values in
ITERATOR	type of iterator object to traverse objects in a container of data-flow values
ACCESS	returns '.' (quotes not included) if the data-flow type is not of a pointer type otherwise returns `->` (C++ arrow token)

The grammar for type mappings is quite simple and is given in Figure 3.9. The `dumpcode` property specifies compiler specific code for outputting an instance of the implementation type.

3.6 Targeting DFAGen for use in a Compiler Infrastructure

Our prototype of the DFAGen tool currently generates source files to be integrated with the OpenAnalysis framework – a toolkit for writing representation-independent analyses [37]. Analyses generated by DFAGen can be used within the Open64 or ROSE [31] compiler frameworks. However, DFAGen offers a mechanism for retargeting generated analyzers so that the operate within other compiler infrastructures. Retargeting involves modifying the code snippets within predefined set definitions, type mappings, and the code generation phase of the DFAGen tool. All other phases in DFAGen (parsing, type checking, may/must analysis) are independent and can be directly reused with other compiler infrastructures.

To make updating the code generation phase of the DFAGen tool easier, the tool has been designed so that the infrastructure-specific pieces are factored out into external template files. Retargeting is then possible by modifying these easily identifiable components.

Template files are text files that direct the code generation process. The template files are written in the same language as the generated analyzers, except they include a header and contain macros that indicate where analysis-specific sections of code should be inserted.

Since DFAGen currently outputs analyzers for integration with C++, it expects template files to have an extension of: {.c, .cpp, .h, .hpp, .C, .H, .cc, .hh, .cxx, .hxx}, additional extensions can be added by modifying a variable in the code generator. For each template file, the code generator will output a source file.

The header of a template file is in the format:

template: id
directory: id
begin

where id is a string of text, specifying the value of the property, terminated by a new-line character. The **template:** property specifies the name of the associated file to generate. The **directory:** property specifies what directory the generate file should be output to. This directory will be relative to the path that DFAGen is invoked from.

After the `begin` token, the remainder of the file consists of source code. The code generator will output a copy of this code but find and replace special sections of text, called template macros.

Template macros are always formatted as a keyword in all capital letters, prefixed by a double quote and period and suffixed by a period and double quote [1]. For example: ".NAME.", is a macro that the code generator recognizes and will replace with the name of the analysis. Macros can be used anywhere in the template file, including its header. Table 3.1 shows the macros that DFAGen recognizes.

The `GENSETCODE` and `KILLSETCODE` macros are replaced with code that calculates the set of generated and set of killed data-flow values for a statement, respectively. DFAGen does not currently provide a way for users to write their own macros, because the actions performed to replace macros are written directly into DFAGen's code generator. Users can change or add macros by modifying DFAGen's source code. This will likely be necessary if the output analyzer is to be in a language other than C++.

In summary, DFAGen can be retargeted for use with different compiler infrastructures through clearly identified code modifications in the predefined set

[1]Double quotes are used because most IDEs and source-code editors for code will syntax highlight quoted text making it more apparent.

```
include: basic.dfa

analysis: ReachingDefs
    meet: union
    direction: forward
    flowtype: stmt
    style: may

    gen[s]:  { s | defs[s] !=empty }
    kill[s]: { t | defs[t] <= defs[s] }
```

Figure 3.10: DFAGen specification file for reaching definitions. The include directive at the top of the file refers to a file (included with DFAGen) where the `def` predefined set and `stmt` type mapping are defined.

definitions, type mappings, and code generation template files.

3.7 Invocation and Use

This section describes how the DFAGen tool is executed from the command line and overviews how generated analyzers integrate with the OpenAnalysis toolkit.

The current prototype of the tool is invoked on a command line as follows:

```
dfagen.py <filename>
```

where filename is some specification file (typically ending in .dfa). Figure 3.10 gives an example of such a file. The tool parses and analyzes specifications and if there are no errors outputs source files containing the generated analyzer.

When errors do occur an appropriate error message is output to `stderr`. Errors fall into four categories: 1) syntax errors, 2) specification errors, 3) typing error, and 4) may/must errors. Syntax errors occur when input files do not follow the grammars in Figures 3.2, 3.3, 3.6, or 3.9. Specifications errors occur when a required property of an analysis specification, predefined set definition, or type mapping is missing or duplicated, for example, if the user specifies an analysis and forgets to supply a direction. Typing errors occur when the left and right operand types of an operation do not agree. May/must errors occur when may/must analysis determines that the variant required for a set reference is not one that is supplied. For example, in a non-locally separable analysis the set `x` for the `gen[s, x]` and `kill[s, x]` equations is always a may set if the style of the analysis is may and a must set if the style of the analysis is must. If may/must analysis determines that a reference to `x` is a reference to a variant that does not match the analysis's style then this is an error and is reported as such.

When the provided specification file contains no errors, generated analyzer source files will be output to the directory the tool was invoked from. In order for these files to be of any use they must be integrated with the compiler for which DFAGen was targeted. This typically involves adding these files to the compiler's build system and recompiling it. This is the case for our current targeting to the OpenAnalysis toolkit.

Our targeting has generated analyzers follow the design philosophy of OpenAnalysis (OA). Like other OA analyses DFAGen generated analyses consist of 1) a manager class that performs the analysis, 2) a results class that contains the results of the analysis, 3) and IR interface class that contains queries a compiler infrastructure dependent implementation must satisfy. Generated manager classes have a method that when called performs the analysis. This method is passed a program's control flow graph, alias analysis results, and interprocedural side-effect analysis results.

OpenAnalysis uses analysis-specific IR interface classes to ensure that analyses are representation independent. That is the analysis does not directly examine or manipulate a program's intermediate representation (IR). An intermediate representation is a data-structures that a compiler constructs to internally represent a program.

When the manager classes requires information from an intermediate representation it makes calls to the methods of an IR interface implementation object. IR interface implementations are classes that derive from IR interfaces and fill in the behavior of the functions that IR interfaces declare but do not define. The OpenAnalysis toolkit does not supply IR interface implementations, rather it is the responsibility of a compiler writer who wishes to use an OA analysis to write these. Currently there are two projects that have such classes to interface compilers to OpenAnalysis: UseOA-Rose, which integrates OpenAnalysis with the ROSE compiler, and UseOA-Open64 which likewise integrates OpenAnalysis with the Open64 compiler. We have only used DFAGen generated analyzers with the UseOA-ROSE package.

More detail about OpenAnalysis and UseOA-Rose can be found by looking at their documentation and websites [37, 3]. The DFAGen website [1] includes links to these projects. A README supplied with the DFAGen tool describes, in detail, how to compile a DFAGen generated analysis with OpenAnalysis and how to use this analysis within the UseOA-Rose package.

Chapter 4

The DFAGen Tool Implementation

The previous chapter presented how to use the DFAGEn tool in terms of its input and output, as well as how the output can be targeted to work with various compiler infrastructures. This chapter elaborates on the internals of the tool as illustrated in the four phases in Figure 3.1. We summarize these four phases as follows:

- Parsing: DFAGen constructs an abstract syntax tree containing the analysis specifications, predefined set definitions, and type mappings.

- Type inference and checking: Based on the declared data-flow set types for the predefined sets, DFAGen infers the type of the `gen` and `kill` set specifications and ensures that the inferred type matches the declared type for the analysis. The type information is also used to determine the domain of possible values in a set builder.

- May/must analysis: DFAGen automatically determines may/must predefined set usage in the `gen` and `kill` equations. The inference of may/must is possible due to DFAGen's declarative, set-based specification language, and its simple semantics.

- Code generation: DFAGen generates the data-flow analysis implementation for use in the target infrastructure. For the current prototype this infrastructure is OpenAnalysis [37] combined with ROSE [4].

The parsing stage is straightforward. The following sections describe the type inference and checking phase, the may/must analysis phase, and the code generation phase in detail.

4.1 Type Inference and Checking

The type inference and checking phase determines the domain of values to iterate over when constructing a set specified with set builder notation and ensures that the specified data-flow equations use the specification language types consistently. The current DFAGen specification language prototype includes the following types: statements, expressions, variables, and sets of these types. The possible types can be extended by passing new type implementation mappings to DFAGen (see Chapter 3.5). The specification language currently assumes that only one type of data-flow information is being propagated and that type is declared in the specification with the `flowvalue` label. The parsing phase of DFAGen generates an Abstract Syntax Tree (AST) for the whole analysis specification including the `gen` and `kill` equations. All leaf nodes in the AST are guaranteed to be references to either predefined sets or the empty set. We can directly infer the types for predefined set reference nodes from their definitions, and the empty set is assumed to have the same type as any set for which it is involved in an operation. The type for the `gen` and `kill` sets are inferred with a bottom-up pass on the abstract syntax tree representation of the data-flow analysis and checked against the specified flow-value type. Type checks are also performed on the operands to all of the set and Boolean operations. Figure 4.1 shows the results of applying type inference on the example in Figure 4.2.

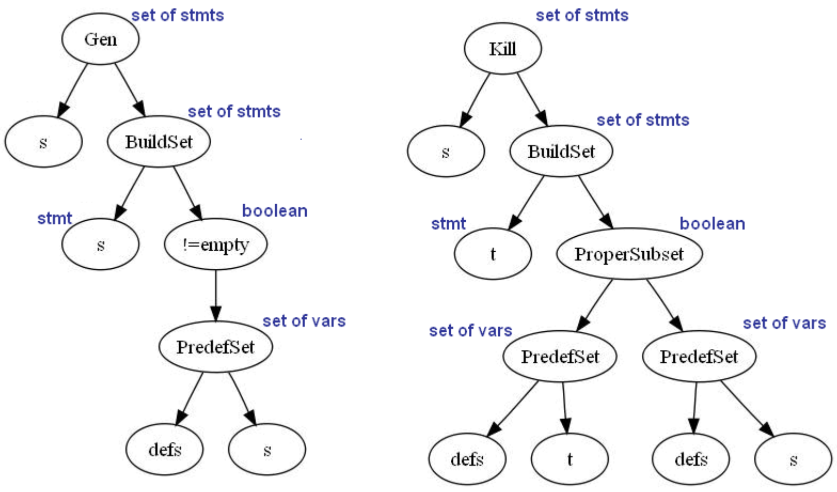

Figure 4.1: Set element type checking for reaching definitions. The type checker propagates type information from the leaves up the tree.

```
include: basic.dfa

analysis: ReachingDefs
    meet: union
    direction: forward
    flowtype: stmt
    style: may

    gen[s]:  { s | defs[s] !=empty }
    kill[s]: { t | defs[t] <= defs[s] }
```

Figure 4.2: DFAGen specification file for reaching definitions. The include directive at the top of the file refers to a file (included with DFAGen) where the `def` predefined set and `stmt` type mapping are defined.

Another important motivation for type inference is to determine the domain of values on which to check the set builder notation condition. Figure 4.3 shows an example specification where DFAGen must determine the domain of values the variable `x` should take when testing the condition `(x in def[s]) and (IN & uses[s]) != empty`. The general approach is to determine the type of the set-builder index and also determine whether the set-builder index is bound to the

```
Analysis: Vary
  meet: union
  flowvalue: variable
  direction: forward
  style: may
  gen[s, IN]:
    {x | (x in defs[s]) and
         (IN intersect uses[s]) != empty}
  kill[s]:
    defs[s]
  initial: independents
```

Figure 4.3: Vary analysis, a nonlocally separable analysis.

context of the specification or is a free variable. The set-builder index could play three possible roles. The the following examples provide examples of each role:

1. `gen[s] = {s | defs[s] != empty}`

2. `gen[s] = {x | x in defs[s] and ...}`

3. `kill[s] = {t | defs[t] <= defs[s]}`

In the first example, the set-builder index `s` represents the statement itself, which is implied by the use of `s` as the parameter to the `gen` set. If the condition (e.g., `defs[s] != empty`) evaluates to true then `gen[s]` will be assigned to a set consisting only of the statement `s`, otherwise it will be assigned to the empty set. In the second example, the domain of the variable `x` is inferred to be the set `defs[s]` due to the `in` expression. In the third example, the set builder index `t` is not bound to the current statement or to a specific set with the use of the `in` operation and, therefore, the set builder index is a free variable. In this case the domain of `t` can be assumed to be the set of all statements. However, since the current DFAGen implementation uses a transfer function, where the `kill[s]` set items are removed from the set of incoming values, the code generator only needs to iterate over the incoming values.

4.2 May/Must Analysis

Once the type checking phase is finished, may/must analysis occurs. May/must analysis determines whether the may or must variant of a predefined set reference should be used. May/must analysis is one of the main contributions of this research.

May/must analysis traverses the `gen` and `kill` equation abstract syntax trees in a top-down manner tagging nodes as either upper or lower bounded. A node tagged as `upper` / `lower` requires its child nodes be tagged in a manner such that the generated code will produce the largest/smallest possible value upon completion of the operation. The largest and smallest possible values depend on the partial ordering induced by the lattice for the operators type. For example, if the operator returns a Boolean type, then `false` is partially ordered before, or smaller, than `true`. This is because a set constructor will return a larger set if its condition conservatively favors true. For operations that return sets, may/must analysis uses the subset equal operator to induce a partial ordering (i.e., a lower bound indicates the smallest possible set and an upper bound indicates the largest possible set). A reference to a predefined set tagged as `upper`/`lower` indicates that the may/must implementation variant should be used in the generated implementation.

The may/must analysis tags the root nodes in `gen` and `kill` equation ASTs based on the style of the specified data-flow analysis (may or must) and the meet operator as shown in Table 4.1. The may/must data-flow analysis assumes that the transfer function should return as conservatively large/small a set as possible, thus the node for the `gen` equation is tagged `upper`/`lower`, and the node for the kill equation is tagged `lower`/`upper`. Given this initial assignment of upper and

```
Algorithm MayMust(n, s, eqtn)
  Input:  n - Root node of gen/kill equation AST
          s - Specifies whether the analysis is
              ‘may’ or ‘must’
          m - Specifies the meet operator of the
              analysis
  Postcondition:  All set reference nodes are
                  tagged ‘may’ or ‘must’

  MayMustRecur(n, I[s, m, type(n)])


Algorithm MayMustRecur(n)
  Input: n - Subtree node

  Let b be the bound on this node (upper of lower)

  if n is a set reference node then
     tag the reference ‘may’ if b is ‘upper’
     tag the reference ‘must’ if b is ‘lower’
  else
    if n is an operator node then
        tag children according to values in P[n, b]
    else
        tag children as b
    endif

    recursively call MayMustRecur on children
  endif
}
```

Figure 4.4: Psuedocode for the may/must analysis algorithm. I is Table 4.1, which specifies the initial bound for the analysis. P is Table 4.2, which specifies how to propagate upper/lower tags. Table I is indexed by an analysis style and meet operator. Table P is indexed by a node type and whether the node is lower or upper.

Table 4.1: In our current implementation of DFAGen the root nodes of the `gen` and `kill` equation ASTs are assigned values from this table.

Meet	Style	gen	kill
union	may	upper	lower
intersection	must	upper	lower

Table 4.2: May/must analysis tagging values. Each row shows an operator and based on that operator's tag, how the operands are tagged during may/must analysis. The operator's tag is shown in the two main columns.

	Upper bound		Lower bound	
	lhs	rhs	lhs	rhs
difference	upper	lower	lower	upper
union	upper	upper	lower	lower
intersection	upper	upper	lower	lower
subset	lower	upper	upper	lower
superset	upper	lower	lower	upper
proper subset	lower	upper	upper	lower
proper superset	upper	lower	lower	upper
not equal to empty set	upper	-	lower	-
and	upper	upper	lower	lower
or	upper	upper	lower	lower
not	lower	-	upper	-

lower tags to the root nodes of the `gen[s]` and `kill[s]` ASTs, the remainder of the may/must analysis can be implemented using a recursive algorithm that visits the `gen` and `kill` tree nodes in a pre-order traversal and tags nodes by looking up values in a table. While at a given node, the determination of tags for the child nodes is based on the current node's tag and the operation the current node represents. Figure 4.4 shows this algorithm. Table 4.2 shows how upper and lower bound tags are propagated to left and right children for various set operations (i.e., rows) based on how the node for that set operation is tagged (i.e., columns).

To derive the contents of Table 4.2, we show how a partial ordering can be determined for the output of most operators in the DFAGen specification language given all possible assignments of `upper` and `lower` to its operands. When a partial ordering of operator output does not result in a single minimal and single maximal tagging, then it is necessary to replace the subtree for that operator with a equivalent expression that includes operators where such an ordering is possible. If users would like to add operators to the specification language, a similar determination of how to tag that operator's children would be necessary.

We classify the operators in the DFAGen specification language into three categories:

1. Set expression operators: $set \times set \rightarrow set$
2. Set conditional operators: $set \times set \rightarrow bool$
3. Boolean conditional operators: $bool \times bool \rightarrow bool$

The set expression operators are those in the Op production of the grammar in Figure 3.3, the conditional and Boolean conditional operators are in the CondOp production. The next three sections establish partial orderings for the output of these operators.

4.2.1 Establishing Ordering of Set Expression Operators

Set expression operators have sets or set expressions as both their left and right operand. May/must analysis tags these operands as either `upper` or `lower`. There are four permutations of upper/lower tags that can be assigned to a binary operator's operands. We establish partial orderings of these permutations and organize them into one lattice per operator. These lattices have unique top and bottom permutations, which when applied to the operator node's children will generate the upper and lower bound sets respectively.

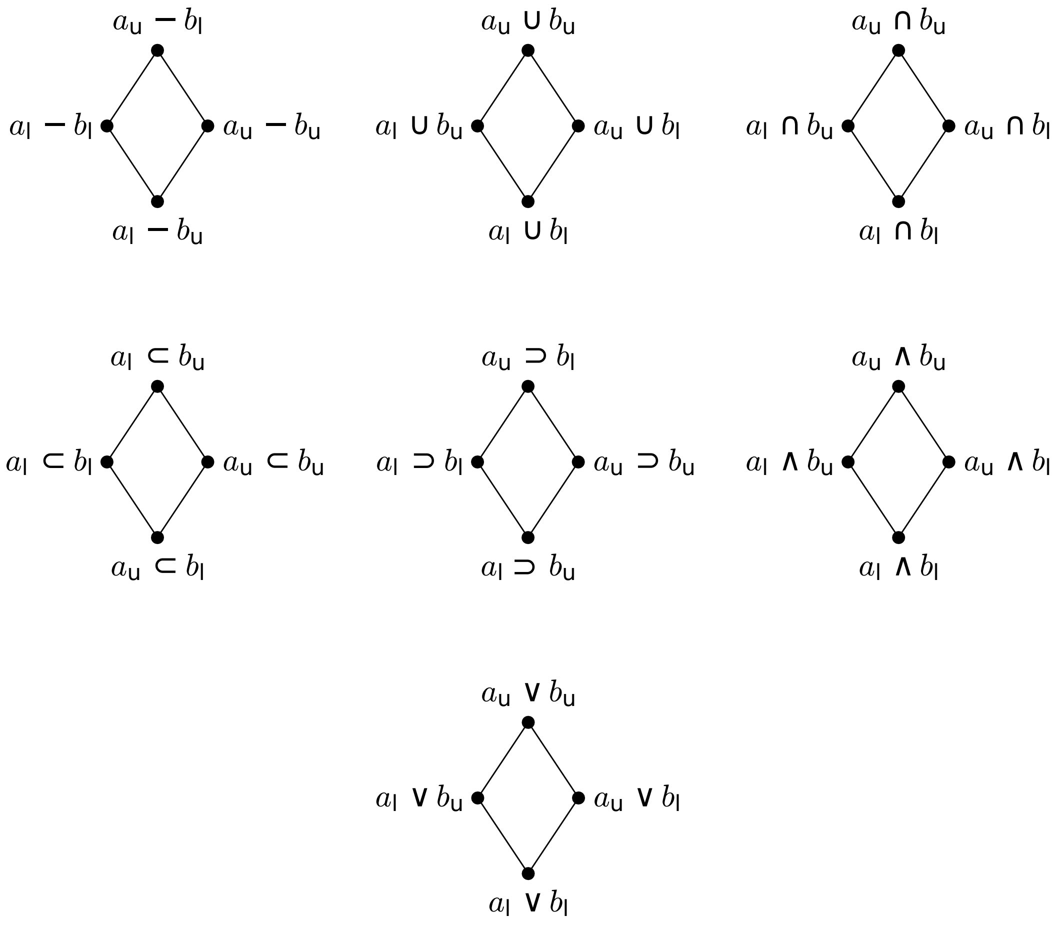

Figure 4.5: Lattices ordering how children of DFAGen specification language operators are tagged. Each lattice corresponds to an operator in the specification language, a and b represent left and right operands for each operator, and the subscripts l and u correspond to whether the operand is tagged as `lower` or `upper`.

We use the notation that the left side of an operator is either some lower bound set a_l, or some upper bound set a_u, and that the right side is either some lower bound set b_l or some upper bound set b_u. We establish lattices for the difference, union, and intersection operators. The lattices are shown graphically in Figure 4.5. In the following proofs the partial ordering operator (represented as $\leq$) is subset equals.

First we examine difference. Given two sets u and l, where u is an upper bound set and l is a lower bound set such that $l \leq u$, we know the following relationships hold for any set x:

$$x - u \leq x - l \tag{4.1}$$

$$l - x \leq u - x \tag{4.2}$$

The left child operand for the difference operator can be either a_l or a_u, where $a_l \leq a_u$. A similar relationship holds for the right child operand, $b_l \leq b_u$. Based on those relationships and Equations 4.1 and 4.2, the partial ordering in Equations 4.3 and 4.4 holds between the four possible operand variants for the difference operator.

$$a_l - b_u \leq a_u - b_u \leq a_u - b_l \tag{4.3}$$

$$a_l - b_u \leq a_l - b_l \leq a_u - b_l \tag{4.4}$$

Now we will establish an ordering on union and intersection. Given two sets u and l where u is an upper bound set and l is a lower bound set such that $l \leq u$, we know that given any set x:

$$(x \cup l) \leq (x \cup u) \tag{4.5}$$

$$(l \cup x) \leq (u \cup x) \tag{4.6}$$

The same holds true for intersection:

$$(x \cap l) \leq (x \cap u) \tag{4.7}$$

$$(l \cap x) \leq (u \cap x) \tag{4.8}$$

Similar to difference we establish a partial ordering for union and intersection. The ordering for union is:

$$(a_l \cup b_l) \leq (a_l \cup b_u) \leq (a_u \cup b_u) \tag{4.9}$$

$$(a_l \cup b_l) \leq (a_u \cup b_l) \leq (a_u \cup b_u) \tag{4.10}$$

The ordering for intersection is the same.

4.2.2 Establishing Ordering of Set Conditional Operators

Conditional operators are used within the context of set-builder expressions. The upper bound of a set-builder expression occurs when the condition is evaluated `true` as many times as possible, the lower-bound occurs when the condition is evaluated as `false` as many times as possible. The set conditional operators include `subset`, `superset`, `proper subset`, and `proper superset`, and are shown in the `CondOp` production of Figure 3.3.

Similar to the set operators, we establish a partial ordering on all possible `lower`/`upper` permutations for the left and right operands for conditional operators. The result of a set conditional operator is a Boolean value. We order these values as $false \leq true$.

To show the lattice for the subset operator requires showing that the following hold:

$$(a_u \subseteq b_l) \leq (a_l \subseteq b_l) \leq (a_l \subseteq b_u) \tag{4.11}$$

$$(a_u \subseteq b_l) \leq (a_u \subseteq b_u) \leq (a_l \subseteq b_u) \tag{4.12}$$

To see that these equations do indeed hold note that since $a_l \subseteq a_u$, we know that given some set x:

$$(a_u \subseteq x) \Rightarrow (a_l \subseteq x) \tag{4.13}$$

$$(x \subseteq b_l) \Rightarrow (x \subseteq b_u) \tag{4.14}$$

It is the case that $(a_u \subseteq x) \leq (a_l \subseteq x)$. This is the case because the only possible way for it not to hold would be if $(a_u \subseteq x) = true$ and $(a_l \subseteq x) = false$, which would contradict Equation 4.13.

It is also the case that $(x \subseteq b_l) \leq (x \subseteq b_u)$. The only way for this not to hold would be if $(x \subseteq b_l) = true$ and $(x \subseteq b_u) = false$, which would contradict Equation 4.14.

Given these facts its simple to see that Equations 4.11 and 4.12 hold.

Similar proofs can be developed for the superset, proper subset, proper superset operators.

4.2.3 Establishing Ordering of Boolean Operators

Boolean operators are those whose left and right operands are of type bool and whose resulting value is a bool. In DFAGen the Boolean operators are `and`, `or`, and `not`. Similar to set conditional operators they are found within set-builder AST nodes.

Let l be the result of a conditional expression tagged `lower` and u be the result of the same expression tagged `upper`. Note that if l is true, then u must also be true.

We assume the following orderings: $false \leq true$ and $l \leq u$, and that:

$$(x \text{ and } l) \leq (x \text{ and } u) \tag{4.15}$$

$$(l \text{ and } x) \leq (u \text{ and } x) \tag{4.16}$$

The same holds true for the `or` operator:

$$(x \text{ or } l) \leq (x \text{ or } u) \tag{4.17}$$

$$(l \text{ or } x) \leq (u \text{ or } x) \tag{4.18}$$

Note the similarly of these facts to those used to prove the lattices for set union and intersection. A similar process is used to prove the Figure 4.5 lattices for the `and` and `or` operators.

4.2.4 Normalization Pass: Handling the Equality Operator

Not all operators in Figure 3.3 are analyzable for `lower` / `upper` tagging. However, they can be normalized into equivalent expressions that may be analyzed. The set equality and set inequality conditional operators are such operators. Prior to running may/must analysis a normalization pass of the AST occurs where all instances of the expression ($x == y$) are translated into the equivalent expression: (`x <= y and y <= x`). Similarly all instances of the expression ($x! = y$) are translated into equivalent expression: (`not (x <= y and y <= x)`).

4.2.5 Non Locally-Separable Analyses and May/Must

May/must variants are calculated for all predefined sets, however, predefined sets are not the only set structures that may appear in a `gen` or `kill` equation. Non locally-separable analyses have gen or kill equations that are parameterized by an incoming set. Whether the incoming set is a set of data-flow values that must be true or may be true is determined by the style of the analysis. It is an error when may/must analysis tags an incoming set with a value opposite that of the analysis's style. For, example, in a may data-flow analysis the following definition would be illegal:

```
gen[s, IN] = defs[s] - IN
```

since the set `IN` would be tagged must and a may analysis propagates may sets.

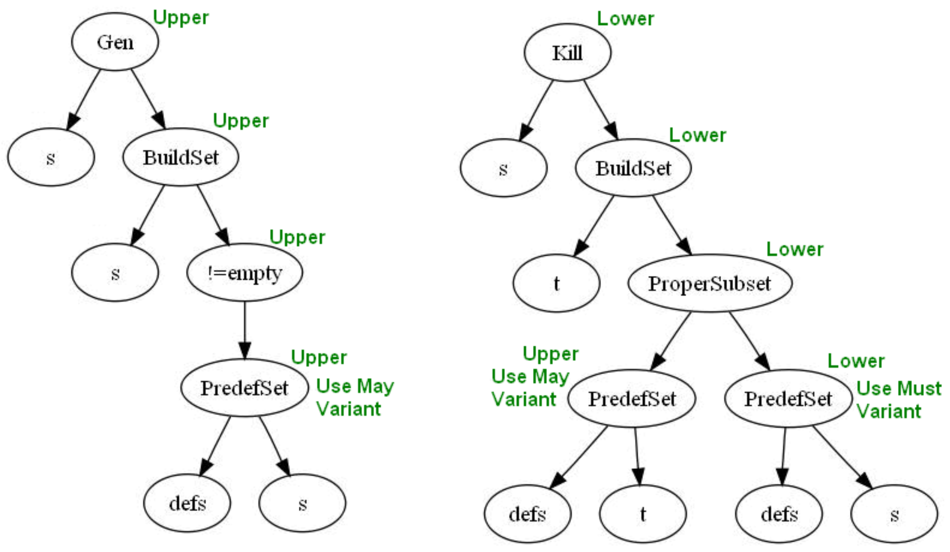

Figure 4.6: Typing predefined sets as may or must for reaching definitions. May/must analysis propagates information from the root down.

4.2.6 Examples of May/Must Analysis

In this subsection we illustrate and describe the results of may/must analysis on the transfer functions for two analyses: reaching definitions analysis and vary analysis.

Figure 4.6 illustrates how may/must analysis occurs for reaching definitions (specified in Figure 3.4) using the algorithm in Figure 4.4. The algorithm `MayMust` is invoked on the `gen` and `kill` nodes and is passed the analysis style and the meet operator. For reaching definitions the analysis style is may, and the meet operator is union. The `MayMust` algorithm refers to the the values in Table 4.1 to determine what to set the `gen` or `kill` nodes as based on these parameters. In this example the `gen` set node is tagged as `upper` and the `kill` set node is tagged as `lower`. The `MayMust` algorithm then applies the `MayMustRecur` algorithm on the `gen` or `kill` child node. Algorithm `MayMustRecur` recursively applies itself in order to traverse the nodes of the AST in a top-down fashion. Children of `gen`, `kill`, and `buildset` nodes directly inherent the tagging value of their parents. Thus

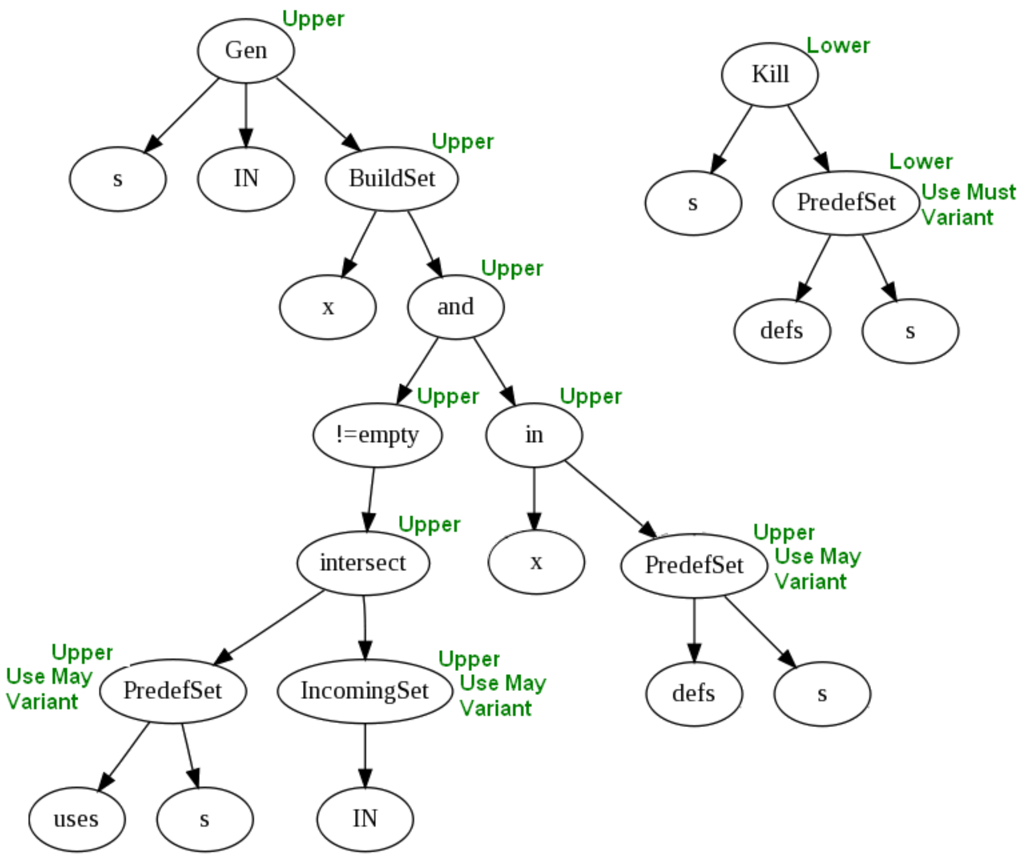

Figure 4.7: Typing predefined sets as may or must for vary analysis.

the `upper` tagging of `gen` propagates to the `buildset` node and to the `!=empty` operator node. Table 4.2 dictates how children of operator nodes are tagged based off of what the operator node is tagged as. Thus the `PredefSet` node in the `gen` AST is tagged as `upper`. When set reference nodes are reached we can determine whether the reference is to a may or must variant based on how its tagged. The set has type may if it has an upper bound and has a type must if it has a lower bound. Thus, the `defs[s]` predefined set reference for the `gen` AST has type may.

Figure 4.7 illustrates the results of may/must analysis when applied to the transfer functions of vary analysis, which was specified in Figure 3.5. Like in the previous example the meet operator is `union` and the analysis style is `may`. A major difference between this example and the previous is that the `gen` equation refers to the incoming set `IN`. There is only one variant for incoming sets: the style of the analysis. May/must analysis concludes that the references to the incoming sets are references to the may variant, so the analysis is legal. Had the analysis concluded otherwise there would have been an error in the specification.

4.3 Code Generation

The final task the DFAGen tool must perform is code generation. As previously described in Chapter 3.6 the code generator is directed by template files. The generator reads these files, then outputs their contents to a generated source file, replacing the macros as needed. Properties specified in the header of the template file determine the filename for the outputted source file as well as what directory it will be stored in.

The `PREDEF_SET_CODE` macro specifies where the code generator will insert code to calculate predefined sets. The inserted code is supplied by the user, from the values of the `maycode` and `mustcode` properties of predefined set definitions. The `GENSETCODE` and `KILLSETCODE` macros specify where the `gen` and `kill` sets are to be calculated. When one of these macros is encountered the code generator traverses the appropriate AST in a top down fashion outputting lines of code for each node. For every node that represents an operation (both set and boolean operations), a temporary is instantiated, and the results of performing the operation are stored in this temporary. Build sets iterate over a series of values, evaluate a condition, and store the iterated value into a temporary set when the condition holds. The values to iterate over is determined by the type inference phase.

With the current template files including with DFAGen, constructed data-flow analyzers follow an iterative approach to solving data-flow equations. That is the program's control-flow graph is traversed, and the data-flow equations for each visited node are evaluated, iteratively, until an answer is converged upon. The iterative solution algorithm is part of the OpenAnalysis toolkit for which DFAGen is currently targeted [37]. The generated analyzer takes previously generated alias analysis results, and a control-flow graph, as parameters. If the data-flow type was specified as bounded, a size bound is also passed in.

The files output by DFAGen are the analyzer's source code files, and are meant to be incorporated into the some compiler or compiler infrastructure (currently the OpenAnalysis toolkit).

Chapter 5

Evaluation

The automatic generation of data-flow analysis implementations entails trade-offs between 1) the ease of analysis specification, 2) the expressibility of the specification language, and 3) the performance of the generated implementation. The DFAGen tool emphasizes the ease of analysis specification. The ease of analysis comes at the cost of reduced analysis expressibility. We qualitatively and experimentally evaluate the DFAGen tool with respect to these three criteria.

The two experimental measures we use are source lines of code for analysis specifications and execution time for the application of some data-flow analyses to benchmarks. We compare the lines of source code necessary to specify an analysis with DFAGen versus the number of lines of source code (SLOC) in a previously written, and equivalent, analysis that was created without using DFAGen. The correlation between source code size and ease of implementation is imperfect, but we combine the SLOC results with qualitative discussions about ease of use. Another measurement compares the running times of previously written analyses against the DFAGen generated analyzers. This measurement aims to support the claim that DFAGen need not sacrifice performance for ease of implementation.

Table 5.1: Lines of code in manual and DFAGen generated analyses.

Analysis	Manual SLOC	Automatic SLOC	Spec SLOC	Predef set SLOC	Ratio of manual:spec
Liveness	394	798	7	98	56
Reaching definitions	402	433	7	98	57
Vary	-	482	8	106	-
Useful	-	482	8	106	-

5.1 Ease of Analysis Specification

We assume that there is a rough correlation between ease of use and the number of source lines of code (SLOC) required to write a data-flow analysis specification. Our hypothesis is that the SLOC required to specify a data-flow analysis to DFAGen is an order of magnitude smaller than the SLOC required to implement the data-flow analysis in the OpenAnalysis data-flow analysis framework.

Table 5.1 presents the results of our measurements. The "Manual SLOC" column shows the SLOC for pre-existing, hand-written implementations of the liveness and reaching definitions analyses. The "Automatic SLOC" column shows the SLOC in the data-flow analysis implementations generated by the DFAGen tool. We show the lines-of-code for DFAGen generated vary and useful analyses, but since there are no previously written versions of these analyses for us to compare against, we do not give manual SLOC numbers for these.

The implementations generated by the tool are not meant to be further modified by the user, therefore their SLOC is not relevant to ease of use. The column "Predefined set SLOC" refers to how many lines of C++ code are used to specify the *must-def*, *may-def*, *must-use*, and *may-use* predefined set structures. Since many analyses will only use the predefined sets included with DFAGen, and since predefined sets can be shared across multiple analyses, we hypothesize that predefined set SLOC will not play a large role in most analysis specifications. For completeness, the SLOC for the predefined sets are included in the comparison of the DFAGen tool versus a hand-coded implementation. The “Specification SLOC” column shows the SLOC in DFAGen specification file for each analysis. It is possible that a user would only need to write these seven or so lines of code to specify a data-flow analysis.

These results support our ease of use hypothesis because the values in the “Specification SLOC” column are more than an order of magnitude smaller than the values in the “Manual SLOC” column. We explicitly give the ratio of the manual SLOC to the specification SLOC in the “Ratio of manual SLOC and specification SLOC” column. If we include the predefined set SLOC in the SLOC for DFAGen, the ratio decreases to between 3 and 4, which indicates only a three-fold reduction in SLOC.

Qualitatively, the strictness of the specification language in DFAGen enables users to specify data-flow analyses using well-known set building semantics and with the assumption that the language being analyzed contains only scalar variables. The scalar variable assumption in the specification language semantics is supported by the may/must analysis that automatically determines whether the must or may variant of a predefined set should be used at each of the instantiations of that predefined set in the *gen* and *kill* set specifications. The simple semantics of the DFAGen specification language provide support for the claim that a significant reduction in SLOC aids ease of use.

One of the motivations for creating the DFAGen tool was to enable developers of the OpenAD automatic differentiation tool [2], to use it to generate data-flow analyses for automatic differentiation (such as vary and useful analysis). The developers of this tool should not have to worry about the details of analysis. Rather, they should focus on expressing analyses to derive the necessary information in a

Table 5.2: Evaluations with SPEC C benchmarks.

		Liveness time			Reaching defs time		
Benchmark	SLOC	automatic	manual	ratio	automatic	manual	ratio
470.lbm	904	0.37	0.28	1.32	0.48	0.30	1.60
429.mcf	1,574	0.71	0.57	1.25	0.90	0.58	1.55
462.libquantum	2,605	1.21	0.99	1.22	1.14	0.73	1.56
401.bzip2	5,731	12.51	11.95	1.05	52.07	43.01	1.21
458.sjeng	10,544	9.32	8.60	1.08	16.46	11.28	1.46
456.hmmer	20,658	18.52	15.43	1.20	24.58	16.53	1.49

function to compute its derivative. DFAGen aims to make this possible.

Our threats to validity with respect to evaluating the ease of analysis specification include the use of only one data-flow analysis framework and the specification of only four analyses. It is possible that other data-flow analysis frameworks would require fewer lines of code to specify the code snippets in the predefined set definitions and type mappings. It is also possible that handwritten implementations in other frameworks could be significantly shorter.

A second hypothesis is that the execution time of an automatically generated data-flow analysis implementation is comparable with the hand-written data-flow analysis. We experimentally compare the execution time of the automatically generated analysis implementations with handwritten implementations to examine the validity of these hypotheses.

Table 5.2 shows the time to execute the manually implemented liveness and reaching definitions analyzers on a number of benchmarks coming from the 2006 SPEC suite, and the time to analyze these benchmarks with DFAGen generated analyzers. Currently, generated analyses take about 50% longer to execute on these benchmarks than manual implementations.

5.2 Performance Evaluation

We believe that the 50% performance difference is due to implementation issues that can be solved in future versions of the tool, and not due to an inherent overhead due to the extra level of abstraction. By incorporating some simple optimizations into the code generation phase of the DFAGen tool matching the performance of the hand-written code is possible. For example, when the `def` and `use` predefined sets are calculated, the generated code iterates over the analyzed procedure twice: once for the `def` set and once for the `use` set. This could be optimized by collapsing both of these calculations into a single loop.

Another inefficiency is due to a number of temporary sets being generated to store intermediate results. For example, in our current implementation each time a transfer function is applied it constructs a `gen` set for the current statement then copies that `gen` set into the return set. Time could be saved by storing the generated values directly into the return set. We have done some preliminary experimentation with hand optimizations to the generated transfer functions, and have found we can match the performance of hand-written analyses within 5% for some example benchmarks. These optimizations could be easily automated by leveraging the structure of the transfer functions.

Our threats to validity with respect to evaluating the performance of the automatically generated analyses are that 1) we only evaluate the performance of liveness and reaching definitions analysis, 2) the manual versions of these analyses run faster, and 3) there are a number of optimizations such as interval analysis [43], which have not been applied to either the hand-written or DFAGen generated analyses.

The performance evaluations were done on an Intel(R) Pentium(R) Core Duo 2 CPU with 2.83 GHz processors. We used the March 2009 alpha release of the

DFAGen tool [1] with OpenAnalysis subversion revision 904 [3], UseOA-ROSE subversion revision 354 [7], the compiler infrastructure ROSE version 0.9.4a [4], and the 2006 SPEC benchmarks [6]. The source lines of code metric was determined using the SLOCCount tool [5].

The alias analysis used to determine may/must set variants was FIAlias [33], which is a flow-insensitive, context-insensitive, unification-based analysis similar to Steensgaard [35]. The analysis is field sensitive, but arrays are treated as single entities. Precision of alias analysis has a direct effect on the precision of analysis results. DFAGen is able to operate with the results of any alias analysis, provided these results can be encoded into predefined sets. Thus, a change in alias analysis may require changes in the defs[s] and uses[s] predefined set definitions, but not a change in the analysis specification.

Chapter 6

Related Work

This section describes related tools, which like DFAGen, attempt to make it easier for developers to create efficient data-flow analyzers. We classify these tools into two major categories: frameworks and generator tools. The next two sections review these categories, and list examples of tools in these categories, respectively.

6.1 Software Frameworks

The rationale for creating software frameworks is to to provide a re-usable method for creating software subsystems. The frameworks described in this section are software libraries, which include algorithms to solve data-flow problems. The frameworks leverage lattice theoretic frameworks.

Dwyer and Clark [14] describe a framework that enables the user to build data-flow analyzers where the solution method (e.g. iterative, interval, etc.) is expressed orthogonally to the data-flow problem. DFAGen also enables users to specify analyses orthogonally to the solution method. In DFAGen the solution method is embedded within template files, changing the method is a matter of changing these files. Our current protoype has only been used to generated analyzers that use an iterative solution method. Dwyer and Clark's framework was also designed to ease the composition of analysis. Composition of analyses is combining two or more analyses into a single analysis. Although DFAGen provides a method for integrating the results of one analysis into another, it provides no mechanism to combine analyses.

Another framework is included in the SUIF compiler system [22]. In this framework data-flow problems are defined by constructing a class that derives from a bit-vector data-flow problem class. Properties of the analysis are described by defining variables and overloading various methods of the base class. The properties that must be defined include analysis direction, analysis confluence rule, and functions that determine `gen` and `kill` sets. Dwyer and Clark's framework also requires a transfer function defined in terms of `gen` and `kill`, as does DFAGen. Dwyer and Clark's confluence rule property is analogous to DFAGen's meet property, and like DFAGen it can either correspond to the set operation union or intersection.

Chapter 3.7 described how DFAGen is targeted to work with the OpenAnalysis toolkit. One advantage of this targeting is the OpenAnalysis includes a framework for writing data-flow analyses [37]. OpenAnalysis differs from Dwyer and Clark's and the SUIF frameworks in that it does not enforce a transfer function defined in terms of `gen` and `kill` equations. In OpenAnalysis's framework the user supplies a transfer function that determines the out set for a node when given the node and its `in` set. However, DFAGen itself requires users to supply transfer functions in terms of `gen` and `kill`.

Although DFAGen currently leverages OpenAnalysis's data-flow analysis framework, it is not necessary for DFAGen to make use of an existing framework. Template files could be used to direct the code-generator to construct analyzers from scratch. However, in the interest of following the frequently quoted software maxim of not reinventing the wheel, we have chosen to leverage existing work.

Doing so enables us to inherit the relative advantages or disadvantages of the leveraged framework.

Many other data-flow analysis frameworks exist, including one written for Vortex compiler [12], FIAT [20, 19], and Wizard++ [38]. Common themes across frameworks are that they require analysis directions, transfer functions, and meet operators. These common characteristics are due to the lattice theoretic basis of these tools [9, 25, 30].

6.2 Generator Tools

Another type of tool that makes creating data-flow analyzers easier is generators. The approach of using generators has had a lot of success in easing the process of producing compiler frontends [23, 15]. Tools like YACC [23] enable developers to construct parsers from specifications of context free grammars. These specifications allow developers to disregard the specifics of the generated parsing algorithm, and thus eases the process of parser implementation. The success of generator tools with compiler frontends has helped to motivate using a similar approach to building other phases of the compilation process. With program analyzers generator tools are not passed context free grammars but rather analysis descriptions based on the mathematically defined lattice theoretic framework.

One tool for generating analyzers from specification files is Sharlit [39]. Sharlit was designed with extensibility and modularity in mind. In particular, it aims to reduce the complexity of analyzing on two-level control flow graphs. Analysis on such graphs complicates the nature of implementing data-flow analyzers, because they require implementors to understand the analysis and its data structures on two levels: statements and basic blocks. The motivation for using two level control-flow graphs is that they often improve analysis speed and memory requirements. Sharlit obtains greater analysis speed and improved memory requirements by performing interval analysis, which collapses control flow graphs while updating data-flow equations so that solving them will still lead to a conservatively correct analysis result. DFAGen does not perform interval analysis, although it may be possible to extend DFAGen to do so by modifying its template files. However, unlike DFAGen, Sharlit does not specify transfer functions declaratively, and is unable to automatically determine how to use may/must information in transfer functions.

Another generator tool is AG (Analyzer Generator), by Zeng *et al.* [43]. This tool synthesizes data-flow analysis phases for Microsoft's Phoenix compiler framework. Like DFAGen, the authors focus on intra-procedural analysis. The unique feature of AG is that it includes a method that allows developers to incorporate useful sets of information into objects representing program instructions by extending these objects classes without directly modifying them. These sets of information serve a similar purpose as DFAGen's predefined sets. Unlike DFAGen, AG provides no mechanism for determining when to use may versus must information.

Similar to DFAGen, the Program Analyzer Generator (PAG) [10, 27] separates specification into a few specification sub-languages. One such language is DATLA, which is used to specify data-flow value lattices. The possible data types for the data-flow set is much more expansive than what can be expressed in this initial prototype of DFAgen. PAG users express transfer functions with a fully functional language called FULA. This approach provides more flexibility in terms of specifying the transfer function when compared to the limited set builder notation provided by DFAgen. The main difference however is that in PAG a user must determine how transfer functions will be affected by pointer aliasing and side-effects. DFAgen on the other hand seeks to automate this difficulty.

Guyer and Lin [17, 18] present a data-flow analysis generator and annotation language for specifying domain-specific analyses that can accurately summarize the effect of library function calls with the help of library writer annotations. Their system defines a set of data-flow types including container types such as `set-of<T>`. Their system also includes a declarative language for specifying the domain-specific transfer functions and side-effect information for calls to library routines. They enable pessimistic versus optimistic descriptions of data-flow set types, but that only determines the meet operator as being intersection or union. Appropriate usage of may versus must information in the transfer function appears to still be the responsibility of the tool user.

Z1 [42] is a tool for constructing analyzers that are based on abstract interpretation. Z1 requires descriptions of the lattice of data-flow values. The descriptions enable a parameterization of lattice height, which enables generated analyses to use smaller of taller lattices as the user sees fit. Taller lattices lead to more accurate results, smaller lattices lead to faster convergence. Thus, the user is able to strike a balance between accuracy and speed. DFAGen does not have a mechanism for users to describe the lattice of data-flow values used. Rather it is assumed that sets of data-flow values are partially ordered by the subset/superset relation (depending on direction).

A recurring theme of these tools is that they do not automatically determine when to use may versus must information in transfer functions. In this respect DFAGen is unique. There are many techniques used by these other tools that DFAGen does not leverage but would be useful additions, such as interval analysis, or a language for describing data-flow value lattices.

Chapter 7

Future Work and Conclusions

This chapter describes the applicability and limitations of may/must analysis beyond the examples used to evaluate the DFAGen tool. It also discusses possibilities for extending the prototype DFAGen implementation. Finally, it ends with some concluding remarks.

7.1 Limitations and Possible Future Work

One limitation used in the current may/must analysis presentation is the requirement that the transfer functions be of a specific form that uses `gen` and `kill` sets. This is a limitation that was useful for the initial implementation and does provide the opportunity for some simple optimizations for the generated transfer function code, but the limitation is not strictly necessary. For example, the determination of whether the `gen` and `kill` sets should be tagged as upper bound or lower bound (see Table 4.1) can be derived by applying the may/must algorithm to the full transfer function, $gen \cup (X - kill)$.

An apparent limitation to the may/must analysis is the inability to handle the data-flow analysis constant propagation. It is unclear how to expose the evaluation of expressions in a statement as a predefined set for the statement.

The current DFAGen prototype does not enable sets of tuples, but that is only a limitation of the implementation.

In terms of the iterative data-flow solving algorithm, there are a number of ways our current prototype of the DFAGen tool could be extended. For one, the current framework focuses on unidirectional analyses. Previous work has indicated that not all analyses can be translated from a bidirectional analyses to a set of unidirectional analyses [24]. Analyses more recent than PRE have been formulated as bidirectional data-flow analyses [36]. The specification language can be extended to enable bidirectional analyses as long as the data-flow analysis framework that DFAGen targets is capable of solving such analyses. As with transfer functions that reference intermediate data-flow results, may/must analysis will still be relevant in the bidirectional analysis context but more from the standpoint of checking for appropriate usage of interdependent analysis results.

A second limitation to the iterative solving algorithm is that DFAGen currently targets the intraprocedural (also known as global analyses) data-flow analysis framework in OpenAnalysis. Interprocedural analyses propagate data-flow facts across procedure call parameter bindings, possibly leading to more precise results. To change our prototype to handle interprocedural analyses, the generated data-flow analysis algorithm would have to change, however we hypothesize that the data-flow specification language would not need to be extended and may/must analysis would still apply.

A third way the DFAGen prototype could be improved is to use a data-flow framework with a worklist based iterative algorithm. The current iterative solving algorithm visits all nodes in the control-flow graph until reaching convergence.

One of the novelties of DFAGen is that due to its declarative specification of data-flow analyses it is able to easily analyze these specifications. We have leveraged this ability in order to automatically determine when to use may versus must information in transfer functions, but analysis of data-flow specifications may be useful in other contexts. For example, it might be useful to aid in composing or optimizing transfer functions.

In conclusion, must/may analysis is actually quite capable of being applicable to analyses beyond the relatively limited set that the DFAGen prototype can handle. The theoretical limitations of must/may analysis are due to the reliance on the transfer function being expressible in terms of predefined sets, intermediate data-flow analysis results, and the `in` or `out` sets. This limitation affects our ability to express constant propagation that uses the propagated constant to variable assignments to evaluate expressions.

7.2 Concluding Remarks

Implementing data-flow analyzers, even within the context of a data-flow analysis frameworks or generator tool, is complicated by the need for the transfer function to handle the may and must variable definition and use information that arises due to pointers, side-effects, and aggregates. May/must analysis, which has been prototyped in the DFAGen tool, enables the automatic generation of data-flow analysis transfer functions that are cognizant of these language features while hiding the complexity from the user who is able to write an analysis specification that assumes only scalars. May/must analysis is made possible by constraints placed on the transfer function specifications, such as the use of predefined sets with atomic elements. These constraints prevent expressing the data-flow analysis constant propagation with full constant folding, but do allow other locally nonseparable analyses such as constant propagation with no constant folding, the domain-specific analyses vary and useful (used within the context of automatic differentiation tools), and any of the locally separable analyses. Experimental results with the DFAGen tool prototype indicate that the source lines of code required for specifying the analysis are an order of magnitude less than writing the analysis using an example data-flow analysis framework. Performance results of the implemented analyses indicate that the code currently being generated is about 50% slower than hand-written code, but more efficient code generation is possible.

REFERENCES

[1] Dfagen website.
`http://www.cs.colostate.edu/~stonea/dfagen/`.

[2] Openad website.
`http://www.mcs.anl.gov/OpenAD/`.

[3] Openanalysis website.
`http://developer.berlios.de/projects/openanalysis/`.

[4] Rose compiler website.
`http://rosecompiler.org/`.

[5] Sloccount tool website.
`http://www.dwheeler.com/sloccount/`.

[6] Spec benchmarks website.
`http://www.spec.org/`.

[7] Useoa-rose website.
`http://developer.berlios.de/projects/useoa-rose/`.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools. [First Edition].* Addison Wesley, 1986.

[9] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, second edition.* Pearson Addison Wesley, 2007.

[10] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symposium*, pages 33–50, 1995.

[11] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, second edition.* Cambridge University Press, 2002.

[12] Craig Chambers, Jeffrey Dean, David Grove, and David Grove. Frameworks for intra- and interprocedural dataflow analysis, 1996.

[13] Keith D. Cooper and Linda Torczon. *Engineering a Compiler.* Elsevier, 2004.

[14] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *Proceedings of the 18th International Conference on Software Engineering*, pages 554–564. IEEE Computer Society Press, 1996.

[15] Étienne Gagnon. Sablecc, an object-oriented compiler framework. Master's thesis, McGill University, Montreal, 1998.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[17] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *2nd Conference on Domain Specific Languages*, October 1999.

[18] Samuel Z. Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *In Languages and Compilers for Parallel Computing*, volume LNCS 2017, 2000.

[19] Mary Hall, John Mellor-crummey, Rene Rodriguez, Mary W. Hall, John M. Mellor-crummey, Alan Carle, and Alan Carle. Fiat: a framework for interprocedural analysis and transformation. In *In Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 522–545. Springer-Verlag, 1993.

[20] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and Rene G. Rodriguez. Fiat: A framework for interprocedural analysis and transformation. Technical report, Rice University CRPC-TR95522-S, 1995.

[21] Laurent Hascoet, Uwe Naumann, and Valerie Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.

[22] G. Holloway and A. Dimock. The machine-suif bit-vector data-flow analysis library, 1998.

[23] S. C. Johnson. Yacc - yet another compiler compiler. Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[24] Uday P. Khedker and Dhananjay M. Dhamdhere. Bidirectional data flow analysis: myths and reality. *ACM SIGPLAN Notices*, 34:47–57, 1999.

[25] G. A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[26] Barbara Kreaseck, Luis Ramos, Scott Easterday, Michelle Strout, and Paul Hovland. Hybrid static/dynamic activity analysis. In *In Proceedings of the 3rd International Workshop on Automatic Differentiation Tools and Applications (ADTA'04)*, May 2006.

[27] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[28] Leon Moonen. A generic architecture for data flow analysis to support reverse engineering. In *the 2nd International Workshop on the theory and Practice of Algebraic Specifications (ASF+SDF'97)*, 1997.

[29] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[30] Flemming Nielson, Hanne Riis Nielson, and Chris Hanken. *Principles of Program Analysis*, chapter 2. Springer, 2005.

[31] Daniel Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *the 16th International Parallel and Distributed Processing Symposium (IPDS, IPPS, SPDP)*, pages 105–104, 1997.

[32] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM Press.

[33] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[34] Y.N. Srikant, Priti Shankar, and Uday P. Khedker. *The Compiler Design Handbook*, chapter 2. CRC Press, 2003.

[35] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[36] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 108–120, 2000.

[37] Michelle Mills Strout, John Mellor-Crummey, and Paul Hovland. Representation-independent program analysis. In *Proceedings of The Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, September 5-6 2005.

[38] Peiyi Tang and John N. Zigman. Data-flow analysis framework in wizard++. Technical Report SC-MC-9605, University of Southern Queensland, 1996.

[39] Steven W.K. Tjiang and John L. Hennessy. Sharlit–a tool for building optimizers. In *The ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1992.

[40] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering, SE-10, (4)*, pages 352–357, 1984.

[41] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 1–12. ACM Press, 1995.

[42] Kwangkeun Yi and Luddy Harrison. Z1: A data flow analyzer generator. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[43] Jia Zeng, Chuck Mitchell, and Stephen A. Edwards. A domain-specific language for generating dataflow analyzers. *Electronic Notes in Theoretical Computer Science*, 164(2):103–119, 2006.