May/Must Analysis and the DFAGen Data-flow Analysis Generator

Andrew Stone[a], Michelle Strout[a], Shweta Behere[b]

[a] *Colorado State University*
[b] *Avaya Inc.*

Abstract

Data-flow analysis is a common technique for gathering program information for use in program 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 iterative analysis 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 paper presents the Data-Flow Analysis Generator tool (DFAGen), which enables analysis writers to generate analyses for separable and nonseparable data-flow analyses that are pointer, aggregate, and side-effect cognizant 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 main contribution of this work is the automatic determination of when to use the may or must variant of a predefined set usage in the analysis specification.

Key words: Data-flow analysis, may, must, static analysis, program analysis, compilers

1. Introduction

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

Data-flow analysis is a common technique for statically analyzing programs. Common examples of data-flow analysis include 1) reaching definitions, which propagates sets of variable definitions that reach a program point without any intervening writes, and 2) liveness, which determines the set of variables at each program point that have been previously defined and may be used in the future. Reaching definitions analysis is useful for transformations such as loop invariant code-motion and simple constant propagation. It is also useful for detecting uninitialized variables and generating program dependence graphs [1]. Liveness analysis results can be used for dead code elimination and register allocation. Data-flow analyses are also used in program slicers and debugging tools. Other optimizations that use data-flow analysis results include busy-code motion, partial dead-code elimination, assignment motion, and strength reduction [2].

Compiler textbooks [2, 3, 4] specify data-flow analyses with data-flow equations. Solving a data-flow problem is done by determining a solution such that all the equations are satisfied. One way data-flow problems are solved is by iteratively evaluating the equations until convergence to a solution.

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

Figure 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.

Figure 1 shows a specification of reaching definitions using such equations. Each statement `s` has an associated `in` and `out` data-flow set (for reaching definitions these sets contain statements). A solution to a data-flow analysis problem is an assignment of data-flow values in all `in` and `out` sets, such that they satisfy the equations.

Data-flow analysis problems can be formalized within lattice theoretic frameworks [2, 5, 6]. Lattice theoretic frameworks enable a separation of concerns between the logic for a specific data-flow analysis and the iterative algorithm and proof of convergence. Such frameworks specify analyses as a transfer function, meet operator, set of

Email addresses: stonea@cs.colostate.edu (Andrew Stone), mstrout@cs.colostate.edu (Michelle Strout), behere@avaya.com (Shweta Behere)

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

Figure 2: Data-flow equations for reaching definitions that are cognizant of may and must definitions due to aliasing, side-effects, and/or aggregates.

initial values, and analysis direction. In a forward analysis the transfer function computes `out` sets for each statement in the program using statement-specific `gen` and `kill` information and an `in` set. The `in` set is computed by using the meet operator to combine the `out` set of predecessor statements.

Research in data-flow analysis implementation has led to the development of a number of tools that leverage the lattice theoretic formalization in order to ease the implementation of data-flow analyses [7, 8, 9, 10, 11, 12, 13, 14, 15]. If the transfer function can be cleanly broken into statement-specific sets such as `def` in Figure 1, then most of the implementation work is focused in writing code that generates those sets for each statement type. This task is complicated by statements involving dereferences to pointer, 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 sets, which are shown in Figure 2 as the `mustdef` and `maydef` sets. The user of existing data-flow analysis implementation tools is responsible for determining when the may and must variants should be used in the implementation of a transfer function.

This paper presents a method for automatically determining the usage of may and must set variants within the implementation of the transfer function. The may/must analysis has been prototyped within the DFAGen tool. Another important feature of the DFAGen tool is that it has been designed so that the implementation language and data-flow analysis framework specific components can be targeted to different compiler infrastructures.

We first introduced the may/must analysis and the DFAGen tool in [16]. This paper advances the previous research by 1) extending the class of analyses specifiable in DFAGen to include nonseparable analyses, 2) introducing a method for retargeting DFAGen's output to different compiler infrastructures, and 3) describing how may/must analysis could be extended if new operators were added to the analysis specification language.

The specific contributions of this paper and the DFAGen tool are as follows:

- DFAGen automatically generates unidirectional, intraprocedural data-flow analysis implementations from succinct descriptions that do not indicate whether sets should be may or must, and that can maintain a "data-flow analysis for scalars" abstraction.
- We present an algorithm that determines whether to use the may or must variant of predefined sets in a specified transfer function. We show how may/must analysis can be extended for new operations including examples of how this is done for conditional operators, and showing how may/must analysis applies in a larger set of data-flow analysis types including some nonlocally separable analyses.
- 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 specific 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.

Section 2 shows language features such as pointers, side-effects, and aggregates can result in may and must variants of data-flow information. Section 3 introduces the DFAGen tool, its architecture, the DFAGen data-flow specification language, and how the DFAGen tool can be retargeted to other compiler infrastructures. Section 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. Section 5 evaluates a prototype implementation of DFAGen by comparing the size and performance of DFAGen generated analyses against handwritten versions. Section 6 discusses the limitations of our current implementation of DFAGen and proposes methods of overcoming these limitations. Section 7 discusses related work. Section 8 contains concluding remarks.

2. May and Must Predefined Sets

Determining whether the may or must version of a predefined set should be used in the transfer function is typically dealt with manually and on an analysis-by-analysis basis. This section gives examples for how various programming language features result in may and must sets of information. The examples in Figures 3, 4, and 5 demonstrate how may/must behavior is exhibited because of these features.

Figure 3 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

```
      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 3: May/must issues that arise because of pointer aliasing.

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

Figure 4: May/must issues that arise because of side-effects.

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 `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 both questions at all program points. For example, reaching definitions "kills" definitions, at statements, when one of the definitions reaching the statement may define the same variable as the statement. Thus, for a reaching definitions analyzer to determine what variables are killed at a given statement it requires an answer for the second question at that 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 4 shows how may/must issues can come about from 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

```
      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 5: May/must issues that arise because of aggregates.

be changed, we can only state 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 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 5 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 can also affect expression generation. The may/must sets of expressions generated for the statement: `*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 [17], which is a compiler textbook, has a section describing how data-flow analyses can be implemented so as to make use of pointer information. The goal of may/must analysis is to automatically determine when may or must variants should be used within the implementation of transfer functions.

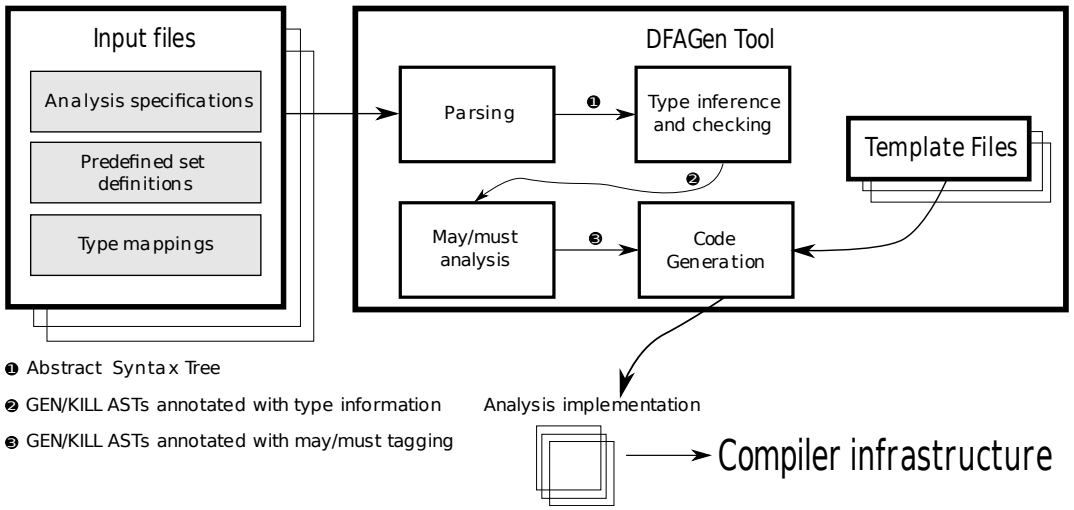

Figure 6: Architecture of DFAGen: Input files are passed to the tool, the tool undergoes a series of phases, transforming an abstraction of the analysis (labeled on the edges), to eventually output a series of source files that can be linked against a compiler infrastructure to include the analysis. 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 7: Grammar for input files. The grammars for the AnalysisSpec, PredefinedSetDef, and TypeMapping nonterminals are illustrated in Figures 8, 12, and 13.

3. Using the DFAGen Tool

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

3.1. Architecture

Figure 6 illustrates DFAGen's input, output, and components. The DFAGen tool is passed a set of input files that contains analysis specifications, predefined set definitions, and type mappings. The code generation component uses template files to guide how it generates code. The generated code is a set of source files intended to be linked into a compiler infrastructure.

The specification for each of these entities: analyses, predefined sets, and type mappings, are represented as separate structures in the DFAGen specification language. Figure 7 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 Section 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. 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. Section 3.4 describes predefined sets in more detail; Section 3.5 describes type mappings.
3. 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 framework [18]; however, with the changes outlined in Section 3.6, the tool can be retargeted to work with other compiler infrastructures.

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 must be 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 others 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 [19] do not satisfy the condition.

The assumed transfer function formats enable the specification of both separable [20] and nonseparable [21] analyses. Separable analyses are also called independent attribute analyses [22]. 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 [23, 24]. Vary and useful analysis are used by activity analysis, an analysis used by automatic differentiation software 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 discusses the analysis specifications.

Figure 8 shows the grammar for analysis specification. The analysis specification includes a set of properties, input values, and transfer functions. The properties include the meet operation, data-flow value type, analysis direction, and analysis style (may or must), and optionally whether there is a bound on the number of possible

AnalysisSpec	⇒	**Analysis** : *id* **meet** : (**union** | **intersection**) **flowtype** : (*id* | *id* **isbounded**) **direction** : (**forward** | **backward**) **style** : (**may** | **must**) (**gen**[*id*] : | **gen**[*id*, *id*] :) *Set* (**kill**[*id*] : | **kill**[*id*, *id*] :) *Set* **initial** : *Set*
Set	⇒	*id*[*id*] | *BuildSet* | *Expr* | **emptySet**
Expr	⇒	*Expr Op Expr* | *Set*
Cond	⇒	*Expr CondOp Cond* | *Expr*
Op	⇒	**union** | **intersection** | **difference** |
CondOp	⇒	**and** | **or** | **subset** | **superset** | **equal** | **not equal** | **proper subset** | **proper superset**
BuildSet	⇒	{*id* : *Cond*}

Figure 8: Grammar for analysis, gen, and kill set definition.

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

Figure 9: DFAGen specification for reaching definitions. Note that `<=` is interpreted as a subset operator.

data-flow values. If there is such a bound then 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 with 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 9 shows an example specification for reaching definitions. Note how similar this specification is to similar specifications in compiler textbooks. Each property is specified with a simple keyword, for example, the meet operation for reaching definitions is specified with the `union`

```
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 10: Predefined set definition for `defs[s]`.

```
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 11: Vary analysis, a nonlocally separable analysis.

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 11 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` or `out` set). 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 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 by using may/must analysis. Predefined sets are used to abstract compiler infrastructure specific details away from the compiler-agnostic analysis specification.

Common predefined sets include "variables defined at statement" (`defs[s]`), "variables used at statement"

PredefinedSetDef ⇒
predefined : *id*[*id*]
description : *line*
argument : *id id*
calculates :
(*id* | **set of** *id*) , *id*, *id*
maycode :
code
end
mustcode :
code
end

Figure 12: 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 property 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. 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.

(`uses[s]`), and "expressions generated in a statement" (`exprs[s]`). Figure 12 shows the grammar for how users define predefined sets in DFAGen. Figure 10 shows an example specification for the set of variables defined at a statement.

The predefined set generation code definitions (the `maycode` and `mustcode` properties) have access to the alias and side-effect analysis results that will be passed to the analyzer when it is called within the context of a specific compiler infrastructure. The C code commented out in Figure 10 uses this information 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.

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

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

Figure 13: Grammar for type mappings.

`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
```

The grammar for type mappings is quite simple and can be seen in Figure 13. The `dumpcode` property specifies code for outputting an instance of the implementation type.

3.6. Using DFAGen With a Compiler Infrastructure

Our prototype of the DFAGen tool currently generates source files to be integrated into the OpenAnalysis framework – a toolkit for writing representation independent analyses [18]. Analyses generated by DFAGen can be used within the Open64 or ROSE [25] compiler frameworks.

One new contribution in this paper is a method for retargeting generated analyzers for use with 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 are independent and can be directly reused with other compiler infrastructures.

To make updating the code generation phase of the DFAGen tool easier, the design and implementation of the tool has been modified 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, but include meta-data about naming files based on the template and macros that indicate where analysis-specific sections of code should be inserted within the template.

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. The macros that DFAGen recognizes are shown in Table 1.

Table 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
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 GENSETCODE and KILLSETCODE macros output code that calculates the set of generated and set of killed data-flow values for a statement, respectively. DFAGen does not 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

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

different compiler infrastructures through clearly identified code modifications in the predefined set definitions, type mappings, and code generation template files.

4. The DFAGen Tool Implementation

Once input files are passed to the DFAGen tool the structures defined in these files are parsed, verified, analyzed, and finally an implementation is generated. Figure 6 illustrates the four phases of the compilation process in the DFAGen tool, which we summarize 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 [18] combined with ROSE [26].

The parsing stage is straightforward. The following subsections 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.

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.

Another important motivation for type inference is to determine the domain of values to check the condition on when using the set builder notation. Figure 11 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 also determine whether the set-builder index is bound to the context of the specification or is a free variable. The set-builder index could play three possible roles, with the following examples providing 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 the `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. Propagating May and Must Information

Once the type checking phase is finished, may/must analysis occurs. May/must analysis determines if predefined set references in transfer functions should be to their may or must variants. May/must analysis is one of the main contributions of this research. Our prior paper [16] introduced the concept of may/must analysis for locally separable analysis using a strict transfer function format. We extend that previous work by presenting may/must analysis as a table driven algorithm, showing how may/must analysis can be extended for new operations including examples of how this is done for conditional operators. We also show how may/must analysis applies in a larger set of data-flow analysis types including some nonlocally separable analyses and bidirectional data-flow analyses

Table 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	-

Table 3: 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

May/must analysis traverses `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`. 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). 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 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 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 tags node and the operation the current node represents. Figure 14 shows this algorithm. Table 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).

```
Algorithm MayMust(n, s, eqtn)
  Input:  n - Root node of gen/kill equation AST
          s - Specifies whether the analysis is
              'may' or 'must'
          eqtn - Specifies whether
  Postcondition:  All predefined set 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 predefined set reference node then
     tag the reference 'may' if b is 'upper'
     tag the reference 'must' if b is 'lower'
  else if
    tag children according to values in P[n, b]
    recursively call MayMustRecur on children
    endif
  endif
}
```

Figure 14: Pseudocode for the may/must analysis algorithm. I is Table 3, which specifies the initial bound for the analysis. P is Table 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.

To derive the contents of Table 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 different expression that includes operators where such an ordering is possible. If users would like to add operators to the specification language, a determination of how to tag that operators 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: $bool \times bool \rightarrow bool$

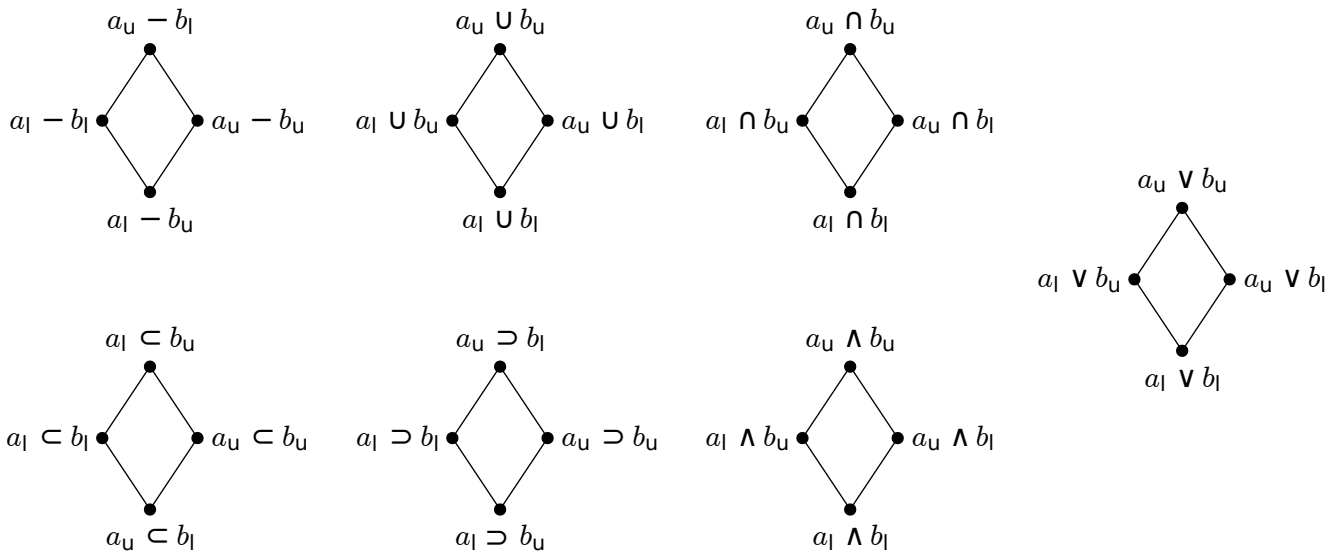

Figure 15: Lattices ordering how children of DFAGen specification language operators are tagged.

The set expression operators are those in the Op production of Figure 8, the conditional and Boolean conditional operators are in the CondOp production. The next three sections establish partial orderings for the output of these operators given the tagging of their subexpressions. Figure 15 shows the derived partial orderings with items higher in the lattice being partially ordered after items lower in the lattice.

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 lattices. These lattices have unique top and bottom `lower`/`upper` tags, which when applied to the operator node's children will generate the upper and lower bound sets respectively.

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 15. In the following proofs the partial ordering operator (represented as $\leq$) is subset equals.

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:

$$x - u \leq x - l \tag{1}$$
$$l - x \leq u - x \tag{2}$$

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 1 and 2, the following partial ordering holds between the four possible operand variants for the difference operator.

$$a_l - b_u \leq a_u - b_u \leq a_u - b_l \tag{3}$$
$$a_l - b_u \leq a_l - b_l \leq a_u - b_l \tag{4}$$

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:

$$x \cup l \leq x \cup u \tag{5}$$
$$l \cup x \leq u \cup x \tag{6}$$

The same holds true for intersection:

$$x \cap l \leq x \cap u \tag{7}$$
$$l \cap x \leq u \cap x \tag{8}$$

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

$$a_l \cup b_l \leq a_l \cup b_u \leq a_u \cup b_u \tag{9}$$
$$a_l \cup b_l \leq a_u \cup b_l \leq a_u \cup b_u \tag{10}$$

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 8.

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. It has been established [2] that in a partial ordering:

$$x \leq y \text{ if and only if } x \wedge y = x, \tag{11}$$

for some meet operator $\wedge$ that is idempotent, commutative, and associative. For this proof we use logical-and as the meet operator. Assuming that $a_l \subset a_u$, we know that given some set x:

$$a_u \subset x \Rightarrow a_l \subset x \tag{12}$$

$$x \subset b_l \Rightarrow x \subset b_u \tag{13}$$

It would be easy to establish the following lattice if these facts were orderings rather than implications. That is, if the following were true:

$$a_u \subset b_l \leq a_l \subset b_l \leq a_l \subset b_u \tag{14}$$

$$a_u \subset b_l \leq a_u \subset b_u \leq a_l \subset b_u \tag{15}$$

In fact the implications in Equations 12 and 13 could be substituted for orderings. To see why, note that if we assume $\wedge$ is the `and` operator that Equation 11 holds when $x \wedge y \Leftrightarrow x$. $x \wedge y \Rightarrow x$ is trivially sound. To show $x \Rightarrow x \wedge y$, it is sufficient to show $x \Rightarrow y$.

Using Equation 11 we develop four implications that prove the relations in the lattice:

$$(a_u \subset x) \wedge (a_l \subset x) \Longleftrightarrow (a_u \subset x) \tag{16}$$

$$(x \subset b_l) \wedge (x \subset b_u) \Longleftrightarrow (x \subset b_l) \tag{17}$$

Since it's the case that Equation 11 holds when $x \Rightarrow y$, it is also the case that:

- Equation 16 holds since Equation 12 holds.
- Equation 17 holds since Equation 13 holds.

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

4.2.3. Establishing Ordering of Boolean Conditional Operators

Boolean conditional operators are those whose left and right operands are of type bool and whose resulting value is a bool. In DFAGen the Boolean conditional operators are `and`, `or`, and `not`. Similar to set conditional operators they are found within set-builder AST nodes, and may/must analysis tags their operands so that their evaluation will either result in the set-builder returning an upper or lower bound set depending on how the operator itself is tagged.

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:

$$(x \text{ and } l) \leq (x \text{ and } u) \tag{18}$$

$$(l \text{ and } x) \leq (u \text{ and } x) \tag{19}$$

The same holds true for the `or` operator:

$$(x \text{ or } l) \leq (x \text{ or } u) \tag{20}$$

$$(l \text{ or } x) \leq (u \text{ or } x) \tag{21}$$

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 lattices for the `and` and `or` operators.

4.2.4. Normalization pass

Not all operators in Figure 8 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 expressions ($l == u$) are translated into equivalent expressions: (`l <= u and u <= l`). Similarly all expressions ($l! = u$) are translated into equivalent expressions: (`not (l <= u and u <= l)`).

4.2.5. Input Sets and Tagging

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 this 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 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.

4.2.6. Example: Reaching Definitions

Figure 16 illustrates how may and must propagation takes place for reaching definitions (specified in Figure 9). As the meet operator is `union` and the type is `may`, the `gen` set node is tagged as `upper` and the `kill` set node is tagged as `lower`. This information is propagated from the root of the syntax tree down to the leaves, which are predefined sets. As we reach the predefined sets, based on the bound information, we decide whether it is a may or must set. The set has type may if it has an upper bound and has a type must if it has a lower bound. For the set operator (! =empty), the lhs child takes the type may. Thus, the `defs[s]` for the `gen` set has type may. A similar process is followed to determine the type of the leaf nodes for the `kill` set.

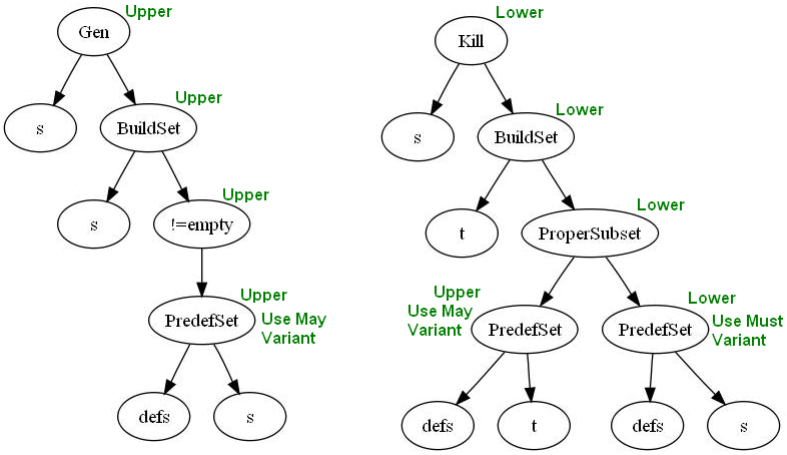

Figure 16: Typing predefined sets as may or must for reaching definitions. May/must analysis propagates information from the root down.

Table 4: Lines of code in manual and DFAGen generated analyses.

Analysis	Manual SLOC	Automatic SLOC	Specification SLOC	Predefined set SLOC	Ratio of manual SLOC and specification SLOC
Liveness	394	798	7	98	56
Reaching definitions	402	433	7	98	57
Vary	-	482	8	106	-
Useful	-	482	8	106	-

4.3. Code Generation

Generated analyses follow an iterative approach for data-flow analysis. The analysis takes previously generated alias analysis results, and a control-flow graph, as parameters. If the data-flow type was specified as bounded, a bound is also passed in. Nodes in the control flow graph are visited and the transfer statement is called for the node only if the `in` and `out` sets change for the node. Data-flow sets to contain the analysis's data-flow values are generated by our tool from C++ templates. Depending on the type of the analysis, we initialize the top and bottom of the lattice. Once we know the may/must information for the predefined sets, code for the predefined sets is directly obtained from their specification file. DFAGen generates code for the `gen` and `kill` sets corresponding to the analysis specification.

5. Evaluation

The automatic generation of data-flow analysis implementations entails trade-offs between ease of analysis specification, expressibility of the specification language, and performance of the generated implementation. The DFAGen tool emphasizes the ease of analysis specification while having a negligible affect on the analysis performance. The ease of analysis comes at the cost of reduced analysis expressibility.

We qualitatively and experimentally evaluate the DFAGen tool with respect to 1) ease of analysis specification, 2) expressibility of the specification language, and 3) the performance of generated analyses. 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.

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 4 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

Table 5: Evaluations with SPEC C benchmarks.

Benchmark	SLOC	Liveness time			Reaching defs time		
		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

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 allow developers of the OpenAD automatic differentiation tool [27], to use it to generate data-flow analyses for automatic differentiation (such as vary and useful analysis). The developer's 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 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.

5.2. Performance Evaluation

A second hypothesis is that the execution time of 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 hand-written implementations to examine the validity of these hypotheses.

Table 5 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.

We believe that this 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.

Also, while computing transfer function the generated implementation introduces a number of temporary sets to store intermediate results. For example, in our current

implementation every 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 1) that we only evaluate the performance of liveness and reaching definitions analysis, 2) that the manual versions of these analyses run faster, and 3) that there are a number of optimizations [8] such as interval analysis, 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 [28] with OpenAnalysis subversion revision 904 [29], UseOA-ROSE subversion revision 354 [30], the compiler infrastructure ROSE version 0.9.4a [26], and the 2006 SPEC benchmarks [31]. The source lines of code metric was determined using the SLOCCount tool [32].

The alias analysis used was FIAlias [33], which is a flow-insensitive, context-insensitive, unification-based analysis similar to Steensgaard [34]. 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 analysis specification.

6. Capabilities and Limitations

This section describes the applicability and limitations of may/must analysis beyond the examples used to evaluate the DFAGen tool. We also discuss possibilities for extending the prototype DFAGen implementation.

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 3) can be derived by applying the may/must algorithm to the full transfer function.

Another implementation limitation is that the current prototype DFAGen tool only enables the specification of transfer functions that depend on predefined sets and either the `in` or `out` data-flow set. This is an implementation limitation and not a limitation for may/must analysis. Various analyses such as partial redundancy elimination (PRE) [35, 2] and activity analysis [23] refer to the results of intermediate data-flow analyses in their transfer function specifications, and it is an important feature we plan to implement in the future. When the result of one analysis is used in the transfer function of another analysis, then may/must analysis would be able to determine if the previous results are being used appropriately. In other words, if may/must analysis attempts to tag a set resulting from a may analysis as lower bound, then there is an error in the transfer function specification.

An apparent limitation to the may/must analysis is the ability 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. We could theoretically represent the assignment of a variable to a constant in terms of a pair of statements associated with a constant to specify the separable version of constant propagation. 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 [36]. Analyses more recent than PRE have been formulated as bidirectional data-flow analyses [37]. 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 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.

In conclusion, must/may analysis is actually quite ca-

pable 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. Related Work

Guyer and Lin [38, 39] present a data-flow analysis framework 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.

The Program Analyzer Generator (PAG) [9, 40] provides a language for specifying general lattices and transfer functions. The possible data types for the data-flow set are 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.

Zeng *et al.* [8] have developed a domain-specific language for generating data-flow analyzers. The authors have developed a data-flow analyzer generator that synthesizes data-flow analysis phases for Microsoft's Phoenix compiler infrastructure. The authors focus on intraprocedural analysis, which is similar to DFAGen. Also similar to DFAGen the user can specify the code for the predefined sets. However, AG (Analyzer Generator) does not automatically determine may and must usage of those sets. Also, the analysis specification is still imperative versus being declarative.

In [10], Dwyer and Clark describe a software architecture for building data-flow analyses where the user can also orthogonally specify the solution method (e.g. iterative, interval, etc.). The framework is also designed to ease the composition of analyses. Their framework includes a gen-kill function space generator where the transfer functions are generated from the specifications of gen and kill. The DFAGen tool uses a similar approach for all of its transfer functions in that we assume the transfer function is $f(X) = gen \cup (X - kill)$ with *gen* and *kill* depending on the input data-flow set X for nonseparable problems.

8. Conclusions

Implementing data-flow analyses, even within the context of a data-flow analysis framework, 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 is 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 element. These constraints prevent expressing the data-flow analysis constant propagation with full constant folding, but other locally nonseparable analyses such as constant propagation with no constant folding, the domain-specific analyses vary and useful analysis used within the context of automatic differentiation tools, and any of the locally separable analyses, which are also called bit-vector 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.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under award #ER25724. We would like to thank Paul Hovland and Amer Diwan for their comments and suggestions with regard to this paper, and Lisa Knebl for her editorial suggestions. We would like to thank the reviewers for their extensive comments and suggestions on this paper and our prior conference paper.

References

[1] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.

[2] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, second edition, Pearson Addison Wesley, 2007.

[3] A. W. Appel, J. Palsberg, Modern Compiler Implementation in Java, second edition, Cambridge University Press, 2002.

[4] K. D. Cooper, L. Torczon, Engineering a Compiler, Elsevier, 2004.
[5] G. A. Kildall, A unified approach to global program optimization, in: ACM Symposium on Principles of Programming Languages, 1973, pp. 194–206.
[6] F. Nielson, H. R. Nielson, C. Hanken, Principles of Program Analysis, Springer, 2005, Ch. 2.
[7] S. W. Tjiang, J. L. Hennessy, Sharlit–a tool for building optimizers, in: The ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1992.
[8] J. Zeng, C. Mitchell, S. A. Edwards, A domain-specific language for generating dataflow analyzers., Electronic Notes in Theoretical Computer Science 164 (2) (2006) 103–119.
[9] M. Alt, F. Martin, Generation of efficient interprocedural analyzers with PAG, in: Static Analysis Symposium, 1995, pp. 33–50.
[10] M. B. Dwyer, L. A. Clarke, A flexible architecture for building data flow analyzers, in: Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, 1996, pp. 554–564.
[11] M. Hall, J. Mellor-crummey, R. Rodriguez, M. W. Hall, J. M. Mellor-crummey, A. Carle, A. Carle, Fiat: a framework for interprocedural analysis and transformation, in: In Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag, 1993, pp. 522–545.
[12] M. W. Hall, J. M. Mellor-Crummey, A. Carle, R. G. Rodriguez, Fiat: A framework for interprocedural analysis and transformation, Tech. rep., Rice University CRPC-TR95522-S (1995).
[13] L. 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.
[14] K. Yi, L. Harrison, Z1: A data flow analyzer generator, in: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1993.
[15] J. Knoop, B. Steffen, Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework, Tech. Rep. Bericht Nr. 9309, Institut fuer Informatik und Praktische Mathematik, Christian-Albrechts-Universitaet zu Kiel (April 1993).
[16] A. Stone, M. M. Strout, S. Behere, Automatic determination of may/must set usage in data-flow analysis, in: In Proceedings of the Eighth International Working Conference on Source Code Analysis and Manipulation, 2008.
[17] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools. [First Edition], Addison Wesley, 1986.
[18] M. M. Strout, J. Mellor-Crummey, P. Hovland, Representation-independent program analysis, in: Proceedings of The Sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2005.
[19] R. P. Wilson, M. S. Lam, Efficient context-sensitive pointer analysis for c programs, in: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, ACM Press, 1995, pp. 1–12.
[20] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, New York, NY, USA, 1995, pp. 49–61.
[21] Y. Srikant, P. Shankar, U. P. Khedker, The Compiler Design Handbook, CRC Press, 2003, Ch. 2.
[22] S. S. Muchnick, Advanced compiler design and implementation, Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.
[23] L. Hascoet, U. Naumann, V. Pascual, "to be recorded" analysis in reverse-mode automatic differentiation, Future Generation Computer Systems 21 (8).
[24] B. Kreaseck, L. Ramos, S. Easterday, M. Strout, P. Hovland, Hybrid static/dynamic activity analysis, in: In Proceedings of the 3rd International Workshop on Automatic Differentiation Tools and Applications (ADTA'04), 2006.
[25] D. Quinlan, B. Miller, B. Philip, M. Schordan, Treating a user-defined parallel library as a domain-specific language, in: the 16th International Parallel and Distributed Processing Symposium (IPDS, IPPS, SPDP), 1997, pp. 105–104.
[26] Rose compiler website, `http://rosecompiler.org/`.
[27] Openad website, `http://www.mcs.anl.gov/OpenAD/`.
[28] Dfagen website, `http://www.cs.colostate.edu/~stonea/dfagen/`.
[29] Openanalysis website, `http://developer.berlios.de/projects/openanalysis/`.
[30] Useoa-rose website, `http://developer.berlios.de/projects/useoa-rose/`.
[31] Spec benchmarks website, `http://www.spec.org/`.
[32] Sloccount tool website, `http://www.dwheeler.com/sloccount/`.
[33] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, R. Altucher, A schema for interprocedural modification side-effect analysis with pointer aliasing, ACM Trans. Program. Lang. Syst. 23 (2) (2001) 105–186.
[34] B. Steensgaard, Points-to analysis in almost linear time, in: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 1996, pp. 32–41.
[35] E. Morel, C. Renvoise, Global optimization by suppression of partial redundancies, Communications of the ACM 22 (2) (1979) 96–103.
[36] U. P. Khedker, D. M. Dhamdhere, Bidirectional data flow analysis: myths and reality, ACM SIGPLAN Notices 34 (1999) 47–57.
[37] M. Stephenson, J. Babb, S. Amarasinghe, Bitwidth analysis with application to silicon compilation, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000, pp. 108–120.
[38] S. Z. Guyer, C. Lin, An annotation language for optimizing software libraries, in: 2nd Conference on Domain Specific Languages, 1999.
[39] S. Z. Guyer, C. Lin, Optimizing the use of high performance software libraries, in: In Languages and Compilers for Parallel Computing, Vol. LNCS 2017, 2000.
[40] F. Martin, PAG – an efficient program analyzer generator, International Journal on Software Tools for Technology Transfer 2 (1) (1998) 46–67.