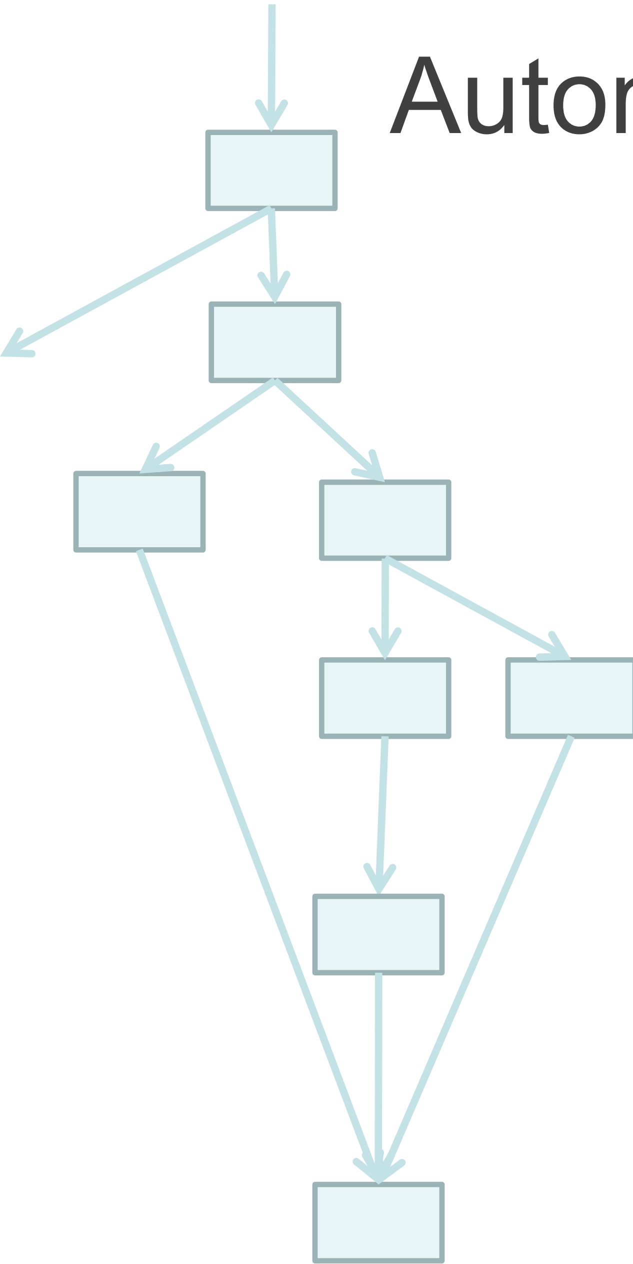


Automatic Determination of May/ Must Set Usage in Data-Flow Analysis

The DFAGEN Tool

Andrew Stone,

Colorado State University

M.S. (Computer Science) Final Examination
May 6th, 2009

Outline of talk

DFAGen: Data-flow Analysis Generator

Data-flow analysis

The problem

The DFAGen Tool

Two novel features
- May/must analysis
- Retargeting

Evaluation

Conclusions

What is data-flow analysis?

What is program analysis?

A technique to:
    determine program behavior

Static analysis:
    program analysis without running the program

Useful for:
    Optimization
    Debugging
    Verification
    Automatic Parallelization

A technique for static program-analysis:
    Data-flow analysis

Data-flow analyses

Some example analyses

- Reaching definitions
- Liveness
- Definite Assignment
- Activity

How analysis results are used

- Constant Propagation
- Dead code elimination
- Register Allocation
- Loop invariant code motion
- ....

Example: Reaching Definitions

```
S1   x = 1
S2   y = q * r
S3   x = 3
S4   if(cond) {
S5     x = 5
     } else {
S6     q = 6 * x
     }
S7   print x
```

At each statement, answer the question:

what definitions may have previously occurred, and not been overwritten.

Useful for:

simple constant propagation

Dataflow Equations

$$\text{in}[s] = \bigcup_{p \in \text{pred}[s]} \text{out}[p]$$

$$\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$$

$$\text{gen}[s] = s \text{ , if } \text{defs}[s] \neq \emptyset$$

$$\text{kill}[s] = \text{all } t \text{ such that } \text{defs}[t] \subseteq \text{defs}[s]$$

solving equations solves data-flow problem

note: gen and kill reference set of vars defined

Dataflow Equations Applied

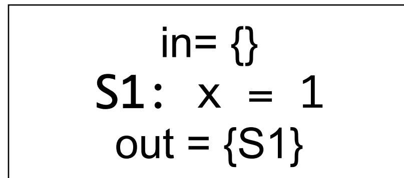
Next to each node: defs[s]

$$in[s] = \bigcup_{p \in pred[s]} out[p]$$

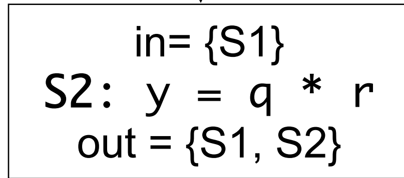
$$out[s] = gen[s] \cup (in[s] - kill[s])$$

$$gen[s] = s \text{ , if } defs[s] \neq \emptyset$$

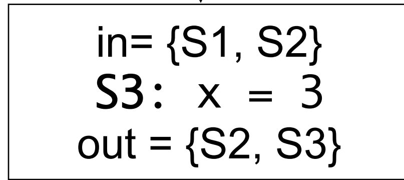
$$kill[s] = \text{all } t \text{ such that } defs[t] \subseteq defs[s]$$



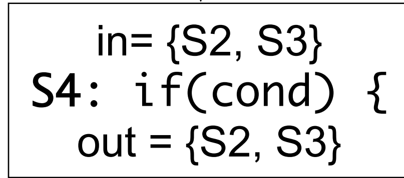
{x}



{y}



{x}

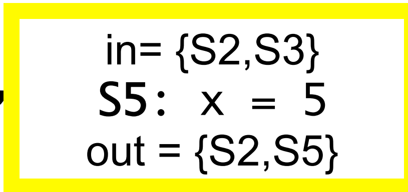


{}

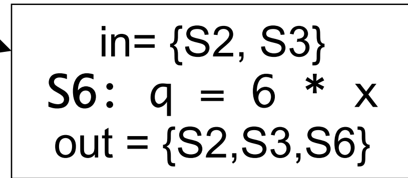
gen[s] = {S5}

kill[s] = {S1,S3,S4,S5,S7}

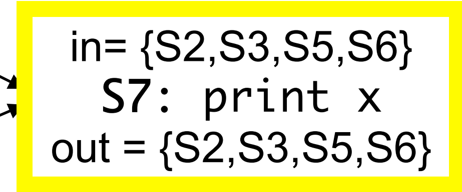
out[s] = {S5} U ({S2,S3} - {S1,S3,S4,S5,S7})



{x}



{q}



{}

meet operation
{S2,S5} U {S2,S3,S6}

The Problem

What do these equations not tell you?

$$\text{in}[s] = \bigcup_{p \in \text{pred}[s]} \text{out}[p]$$

$$\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$$

$$\text{gen}[s] = s \text{ , if } \text{defs}[s] \neq \emptyset$$

$$\text{kill}[s] = \text{all } t \text{ such that } \text{defs}[t] \subseteq \text{defs}[s]$$

Wait what? The variables that **may** be defined
or the variables that **must** be defined?

Amazing stick figure from XKCD (http://www.xkcd.com)

The Problem of May/Must

```
int x, y, z;
int *p, *q;

x = rand() % 1000;
y = rand() % 1000;
z = rand() % 1000;

p = &x;
q = &y;

if(*p < *q) {
    q = &z;
}

*q = 500;
*p = 400;
```

Must use: {x, y}
May use: {x, y}

Must def: {}
May def: {y, z}

Must def: {x}
May def: {x}

Why May/Must Happens

Language features such as:

Pointer aliasing

Aggregate structures

Side effects

For the curious:
Check out the thesis.

The DFAGen Tool

DFAGen Tool

Data-flow analysis generator tool

with declarative data-flow analysis language

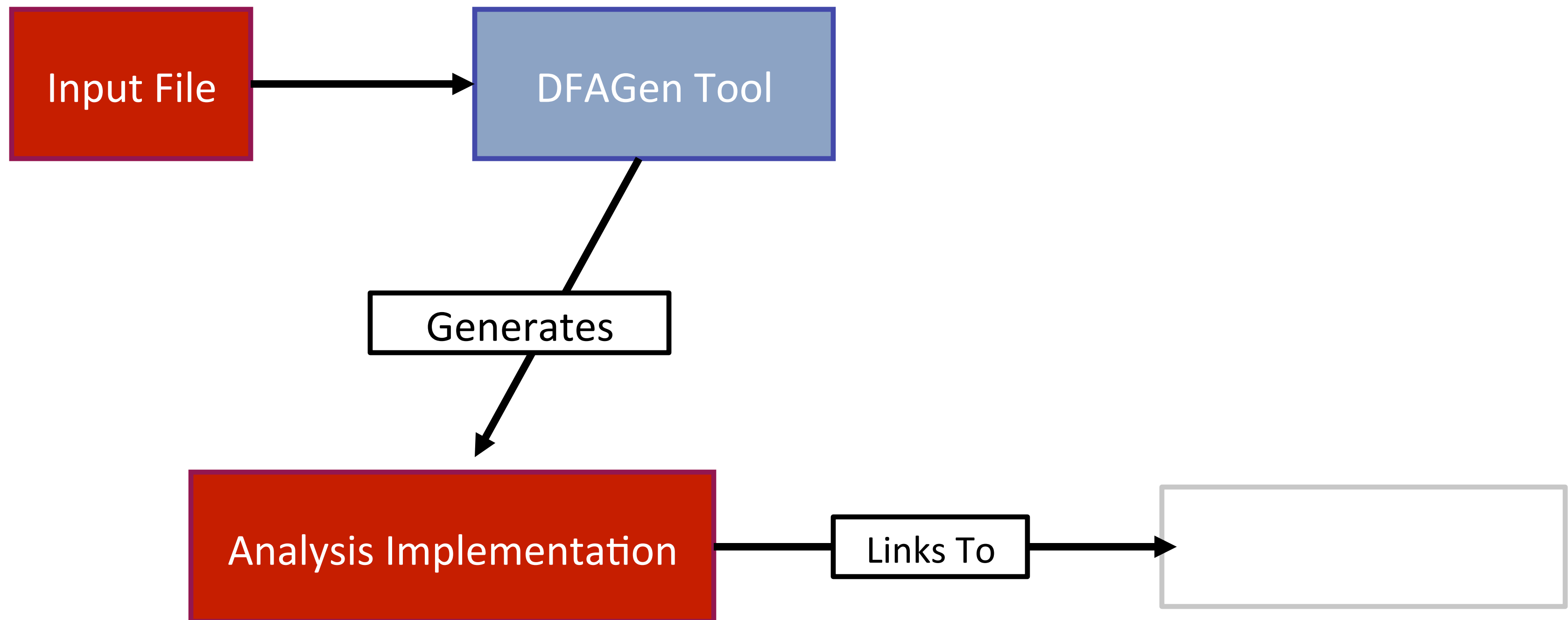


Fig. 3.1 in thesis shows a more detailed view of this

DFAGen Input

Separates analysis specification from definitions that are compiler and language specific.

3 Types of entities in input language

- Analysis specifications
- Predefined set definitions
- Type Mappings

Analysis Specifications

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

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

DFAGen data-flow equations

DFAGen analyzers solve one of:

Forward analyses:

$$\text{in}[s] = \bigwedge_{x \in \text{pred}[x]} \text{out}[p]$$

$$\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$$

Backward analyses:

$$\text{in}[s] = \text{gen}[s] \cup (\text{out}[s] - \text{kill}[s])$$

$$\text{out}[s] = \bigwedge_{x \in \text{succ}[x]} \text{in}[x]$$

gen and kill are supplied by user

$\bigwedge$ is $\bigcup$ or $\bigcap$

Analysis Specifications

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

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

Analysis Style

May Analysis

Results at each statement are set of all data-flow facts that **may** be true

Examples: Reaching definitions, Liveness

Must Analysis

Results at each statement must be true for all possible executions

Examples: Available expressions

Two novel features in the DFAGen Tool

May/Must Analysis

The Goal

Predefined sets have two variants

- May sets
- Must sets

The goal: to determine which variant to use

reachingdefs.dfa

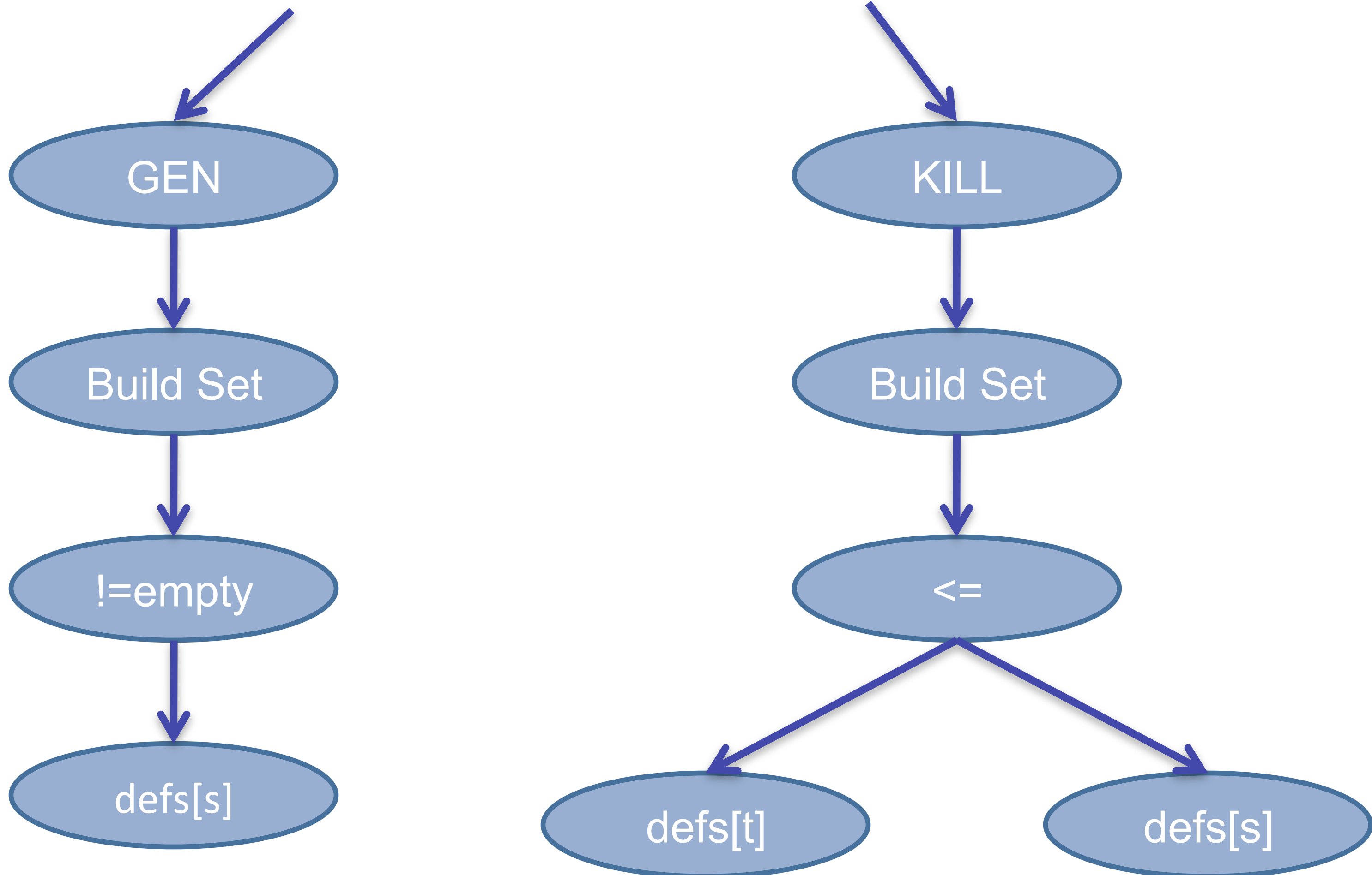
```
analysis: ReachingDefs
  meet: union
  direction: forward
  flowtype: stmt
  style: may

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

May

Must

Gen and Kill ASTs



The Analysis

Meet	Type	Gen	Kill
union	may	Upper	Lower
intersect	must	Lower	Upper

Operator	UB	UB	LB	LB
	lhs	rhs	lhs	rhs
<=	lower	upper	upper	lower
>=	upper	lower	lower	upper
<	lower	upper	upper	lower
>	upper	lower	lower	upper
union	upper	upper	lower	lower
intersect	upper	upper	lower	lower
!=empty	upper	-	lower	-

- Parse GEN and KILL expressions into an AST.
- Analyze in a top down fashion, determine whether we want an "upper bound" or "lower bound" value at each node.
- Use tables to left to determine these values.

The Analysis in Action

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

Meet	Style	Gen	
union	may	Upper	Lo
intersect	must	Lower	U

Operator	UB	UB	LB
	lhs	rhs	lhs
<=	lower	upper	uppe
!=empty	upper	-	lowe

Meet	Style	Gen	Kill
union	may	Upper	Lower
intersect	must	Lower	Upper

Operator	UB	UB	LB	LB
	lhs	rhs	lhs	rhs
<=	lower	upper	upper	lower
!=empty	upper	-	lower	-

Defs[s] **UB**

(may)

defs[t] **UB**

(may)

defs[s] **LB**

(must)

Second novel feature in the DFAGen Tool

Retargeting

Retargeting

Currently output:

C++ code for OpenAnalysis toolkit

To change this:

Modify template files

To change what an analysis is targeted for:

Modify its predefined sets and type mappings

Predefined Set Definitions

```
predefined: defs[s]
   description: Set of variables defined at a 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
```

Type Mappings

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

Include directive

```
include openanalysis.dfa

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

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

Code generator

Iterates through template files

For each template file an output source file is created

The code generator outputs the template, replacing macros with appropriate segments of code

Example macros: NAME FLOWTYPE
MEET GENSETCODE

Template File Format

```
template: ".NAME.".cpp
directory: ".NAME."
begin

// source code with
// macros
```

Source code

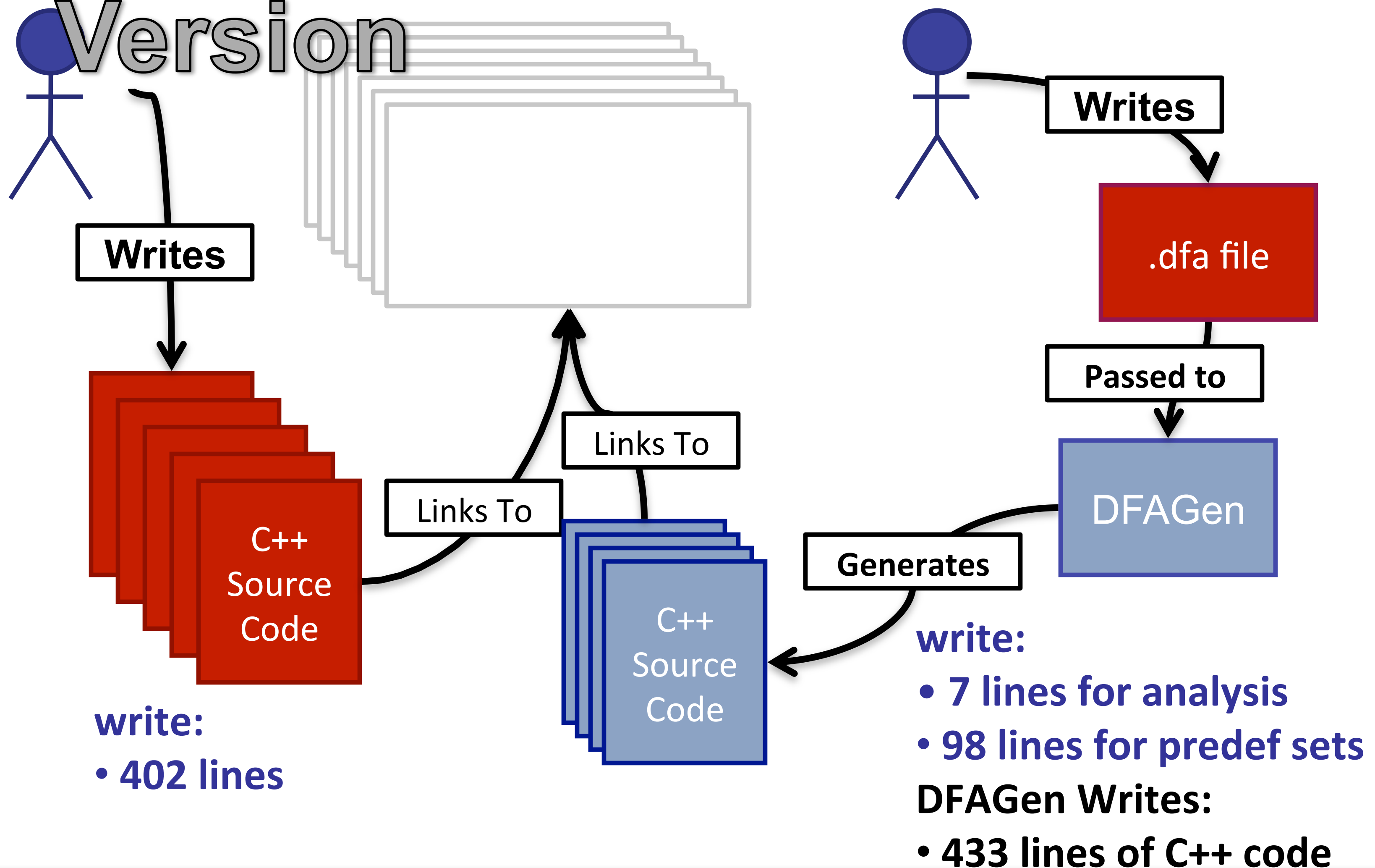
Using an analysis in another compiler

Changing templates to retarget the code generator

Writing required predefined sets and type mappings for this other compiler

Evaluation

Ease of Analysis Specification



Performance Evaluation

Table 5.2: Evaluations with SPEC C benchmarks.

Benchmark	SLOC	Liveness time automatic	Liveness time manual	Liveness time ratio	Reaching defs time automatic	Reaching defs time manual	Reaching defs time 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

Why the slow down?

Code generator constructs a number of unnecessary temporary sets

By hand optimizing generated code, to remove these sets I’ve been able to bring analysis time for reaching defs to within 5% of manual time

Conclusions

In the Thesis

More examples of may/must

Non locally separable analyses

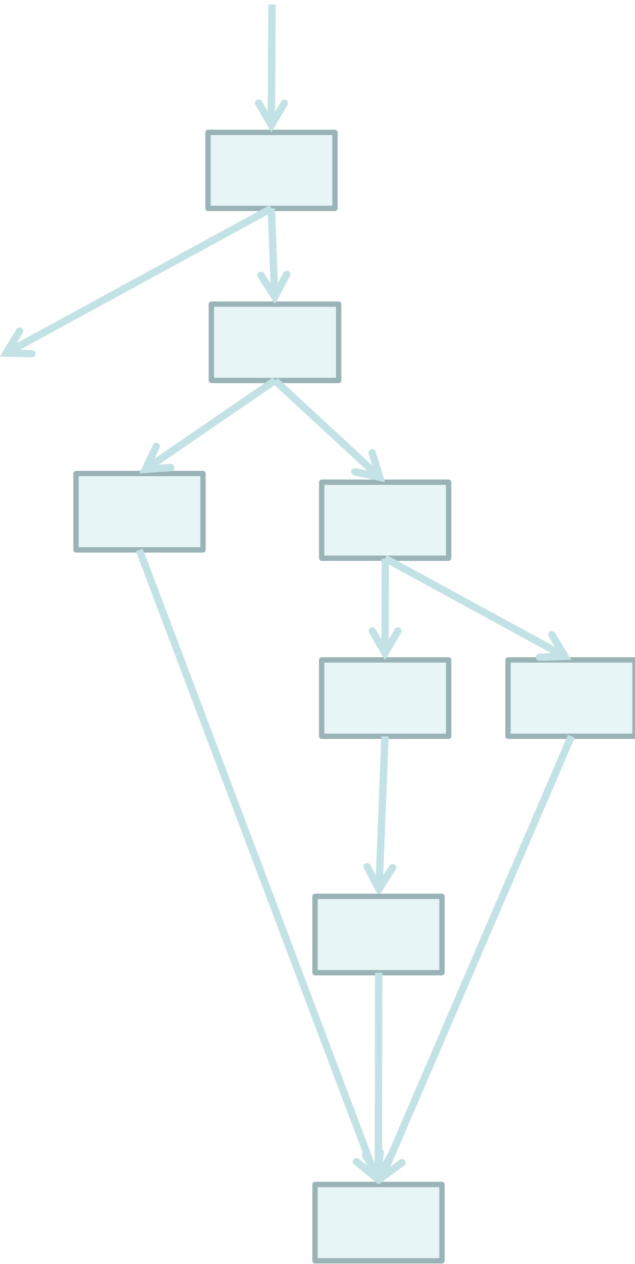
Explanation on how to derive upper/lower values in tables

Limitations and potential future work

Imported Predefined Sets

Type inference and checking

Related work



Conclusions

DFAGen makes writing analyses easier.

It generates the analysis from succinct specifications.

Its input language separates analysis details from language specific and compiler specific details

DFAGen deals with the may/must issue of aliasing automatically.