Workshop on Essential Abstractions in GCC

Introduction to Data Flow Analysis

GCC Resource Center

(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

1 July 2012

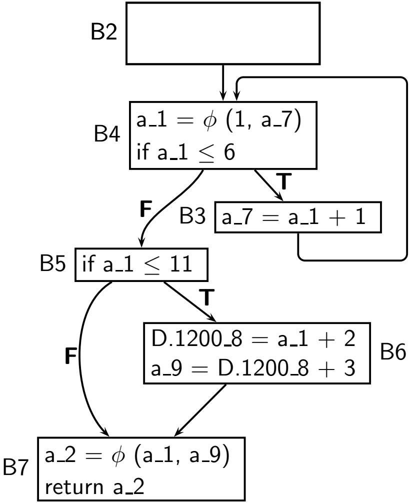
Outline

- Motivation
- Live Variables Analysis
- Available Expressions Analysis
- Pointer Analysis

Part 2

Motivation

Dead Code Elimination

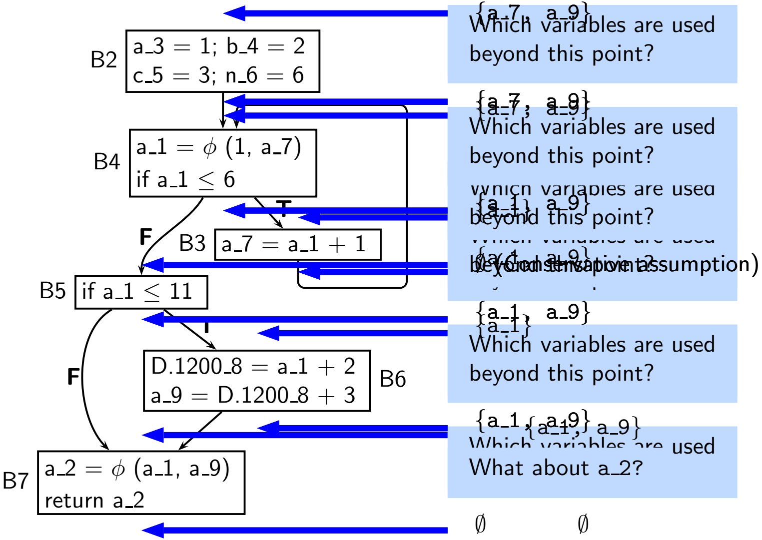


- No uses for variables a_3, b_4, c_5, and n_6
- Assignments to these variables can be deleted

How can we conclude this systematically?

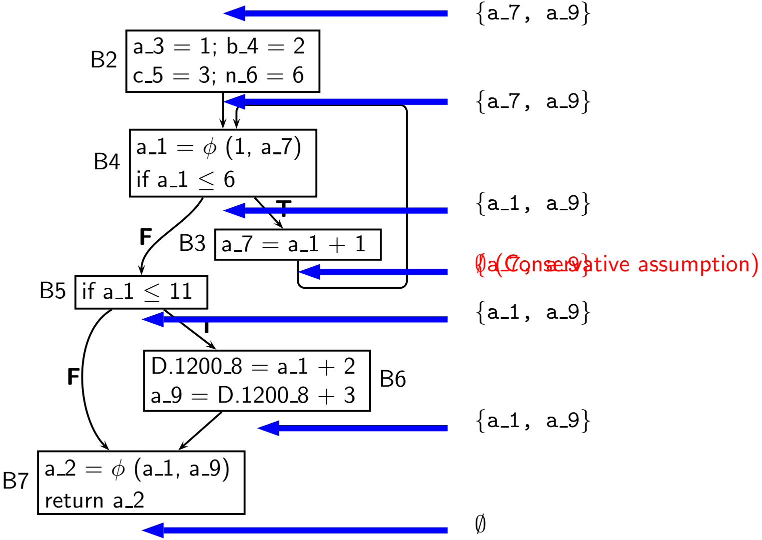
Liveness Analysis of Variables

Find out at each program point *p*, the variables that are used beyond *p*

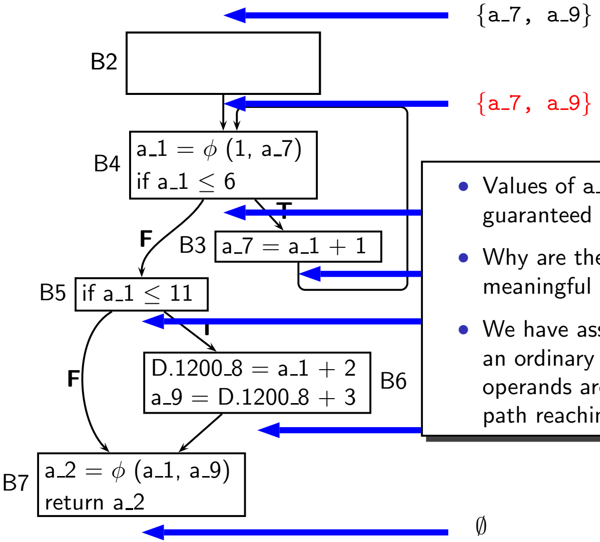


Liveness Analysis of Variables: Iteration 2

Find out at each program point p, the variables that are used beyond p



Using Liveness Analysis for Dead Code Elimination



- Values of a_3, a_4, c_5, and n_6 are guaranteed not to be used
- Why are the values of a_7 and a_9 meaningful at the exit of B2?
- We have assumed a ϕ function to be an ordinary expression in which operands are computed along every path reaching the computation

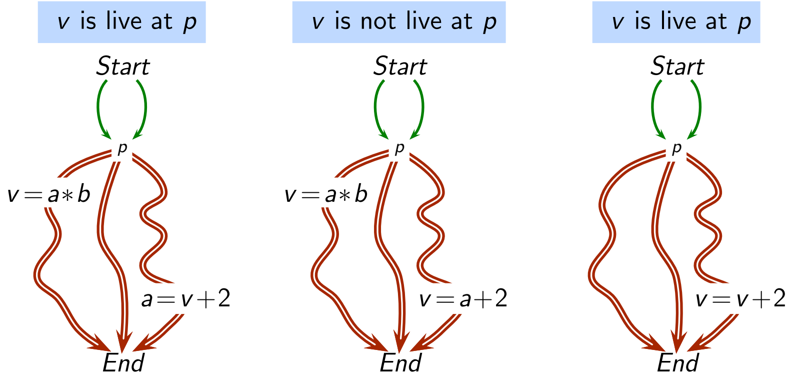
Part 3

Live Variables Analysis

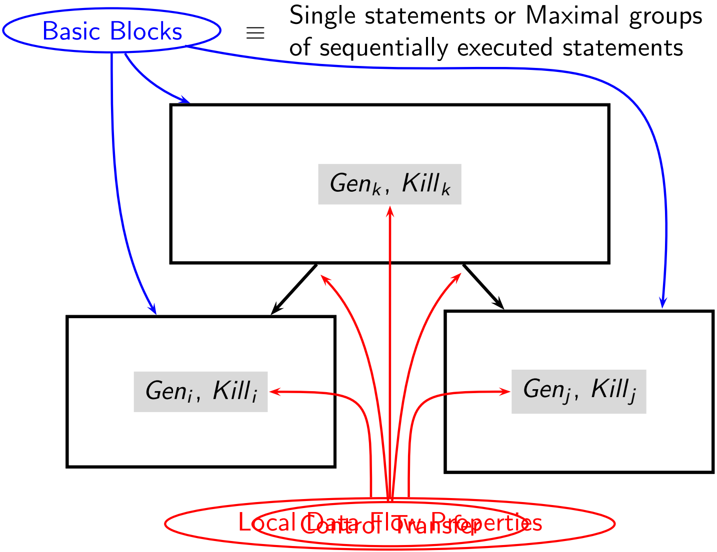
Defining Live Variables Analysis

A variable v is live at a program point p, if some path from p to program exit contains an r-value occurrence of v which is not preceded by an l-value occurrence of v.

Path based specification



Defining Data Flow Analysis for Live Variables Analysis



Local Data Flow Properties for Live Variables Analysis

r-value occurrence

Value is only read, e.g. x,y,z in

x.sum = y.data + z.data

l-value occurrence

Value is modified e.g. y in

y = x.lptr

$$Gen_n = \{ v \mid \text{variable } v \text{ is used in basic block } n \text{ and is not preceded by a definition of } v \}$$

$$Kill_n = \{ v \mid \text{basic block } n \text{ contains a definition of } v \}$$

within n

anywhere in n

Local Data Flow Properties for Live Variables Analysis

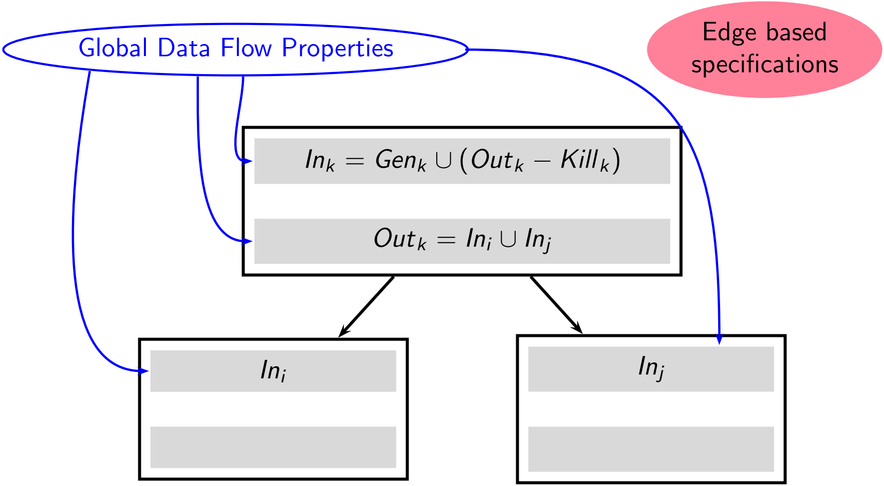
- Gen_n : Use not preceded by definition

  Upwards exposed use

- $Kill_n$: Definition anywhere in a block

  Stop the effect from being propagated across a block

Defining Data Flow Analysis for Live Variables Analysis



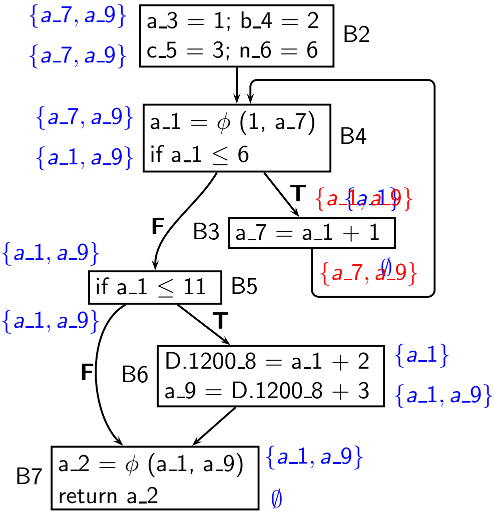
Data Flow Equations For Live Variables Analysis

$$In_n = (Out_n - Kill_n) \cup Gen_n$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcup\limits_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

In_n and Out_n are sets of variables.

Performing Live Variables Analysis



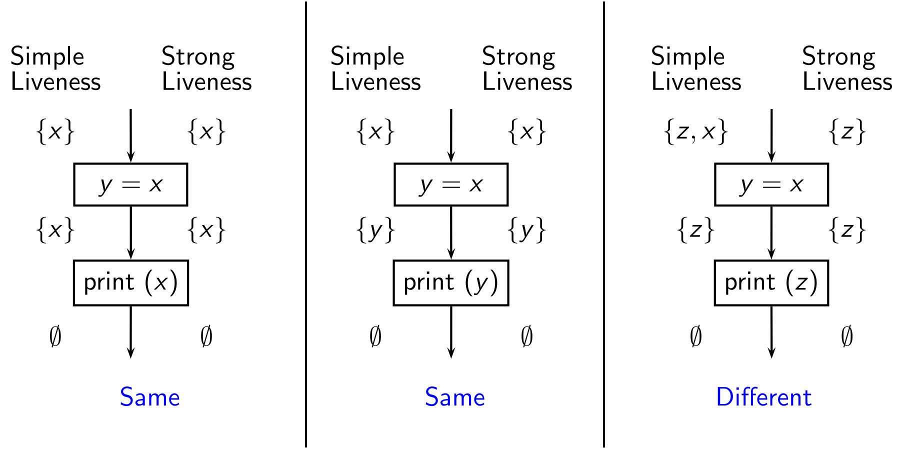
	Gen	*Kill*
B2	∅	{a_3, b_4, c_5, n_6}
B4	{a_7}	{a_1}
B3	{a_1}	{a_7}
B5	{a_1}	∅
B6	{a_1}	{a_9}
B7	{a_1, a_9}	{a_2}

Strongly Live Variables Analysis

A variable v is strongly live if it is used in

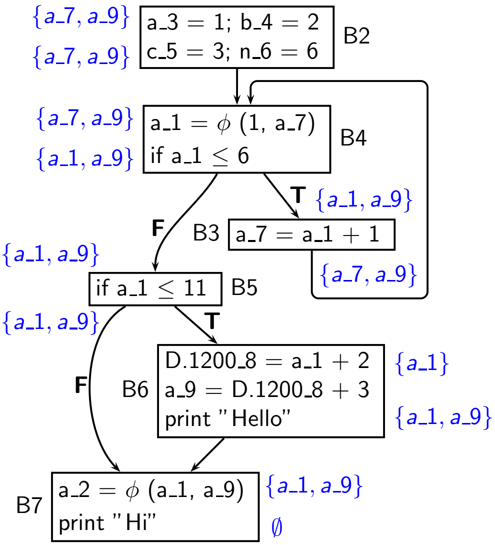
- in statement other than assignment statement, or
  (this case is same as simple liveness analysis)
- in defining other strongly live variables in an assignment statement
  (this case is different from simple liveness analysis)

Understanding Strong Liveness

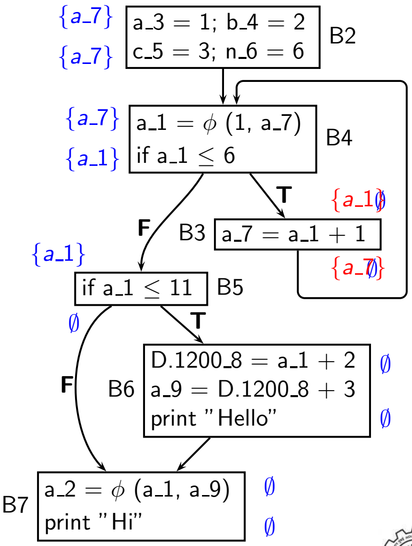


Comparision of Simple and Strong Liveness for our Example

Simple Liveness



Strong Liveness



Using Data Flow Information of Live Variables Analysis

- Used for register allocation.

  If variable x is live in a basic block b, it is a potential candidate for register allocation.

- Used for dead code elimination.

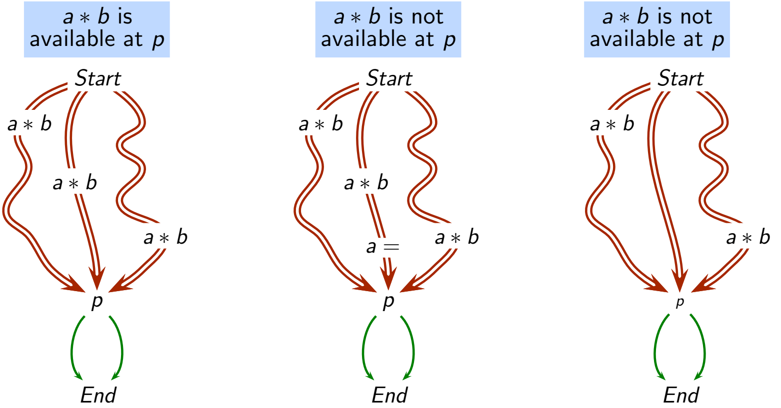
  If variable x is not live after an assignment $x = \ldots$, then the assginment is redundant and can be deleted as dead code.

Part 4

Available Expressions Analysis

Defining Available Expressions Analysis

An expression e is available at a program point p, if
every path from program entry to p contains an evaluation of e
which is not followed by a definition of any operand of e.



Local Data Flow Properties for Available Expressions Analysis

$$Gen_n = \{ e \mid \text{expression } e \text{ is evaluated in basic block } n \text{ and this evaluation is not followed by a definition of any operand of } e \}$$

$$Kill_n = \{ e \mid \text{basic block } n \text{ contains a definition of an operand of } e \}$$

	Entity	Manipulation	Exposition
Gen_n	Expression	Use	Downwards
$Kill_n$	Expression	Modification	Anywhere

Data Flow Equations For Available Expressions Analysis

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcap\limits_{p \in pred(n)} Out_p & \text{otherwise} \end{cases}$$

$$Out_n = Gen_n \cup (In_n - Kill_n)$$

Alternatively,

$$Out_n = f_n(In_n), \qquad \text{where}$$

$$f_n(X) = Gen_n \cup (X - Kill_n)$$

In_n and Out_n are sets of expressions.

Using Data Flow Information of Available Expressions Analysis

- Common subexpression elimination
  - If an expression is available at the entry of a block b **and**
  - a computation of the expression exists in b **such that**
  - it is not preceded by a definition of any of its operands

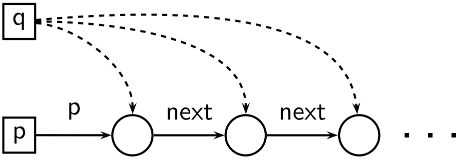
  Then the expression is redundant

- Redundant expression must be upwards exposed
- Expressions in Gen_n are downwards exposed

Part 5

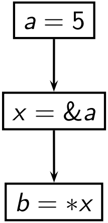
Introduction to Pointer Analysis

Code Optimization In Presence of Pointers

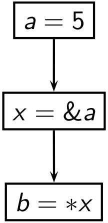
Program	*Memory graph at statement 5*
1. q = p; 2. while (. . .) {do { 3. q = q→next; 4. }while (. . .) 5. p→data = r1; 6. print (q→data); 7. p→data = r2;	

- Is p→data live at the exit of line 5? Can we delete line 5?
- No, if p and q can be possibly aliased
  (while loop or do-while loop with a circular list)
- Yes, if p and q are definitely not aliased
  (do-while loop without a circular list)

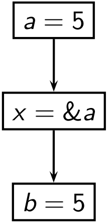
Code Optimization In Presence of Pointers



Original Program

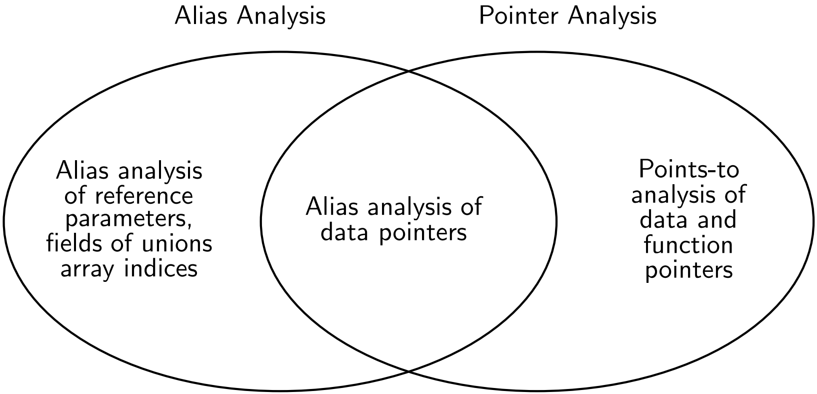


Constant Propagation without aliasing

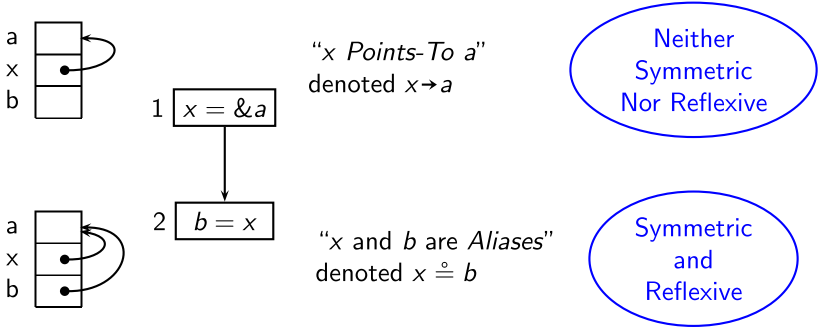


Constant Propagation with aliasing

The World of Pointer Analysis



Alias Information Vs. Points-To Information



- What about transitivity?
  - Points-To: No.
  - Alias: Depends.

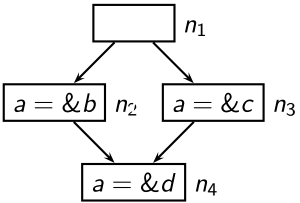
Introduction

Two important dimensions for precise pointer analysis are

- Flow Sensitivity
- Context Sensitivity

Flow Sensitive analysis

A flow-sensitive analysis computes the data flow information at each program point according to the control-flow of a program.

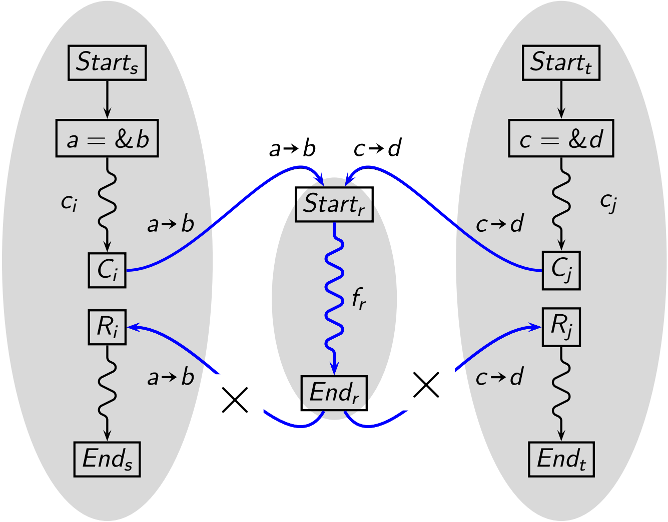


At the exit of node n_4

Flow insensitive information:
$\{a \rightarrow b, a \rightarrow c, a \rightarrow d\}$

Flow sensitive information:
$\{a \rightarrow d\}$

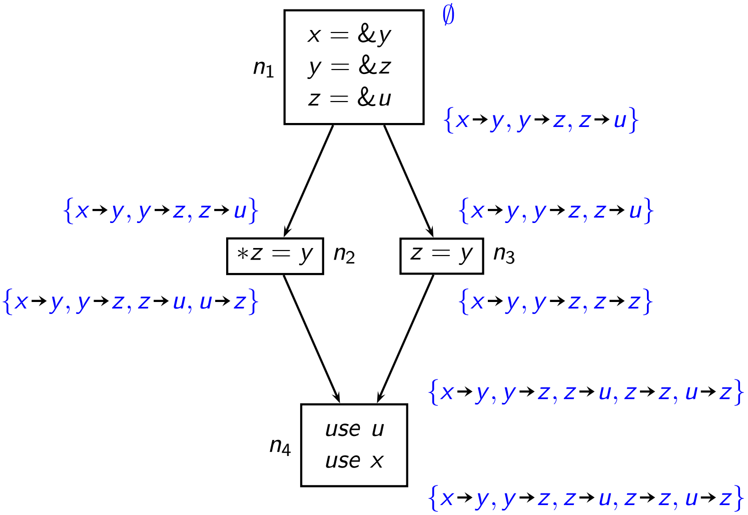
Context Sensitivity in Interprocedural Analysis



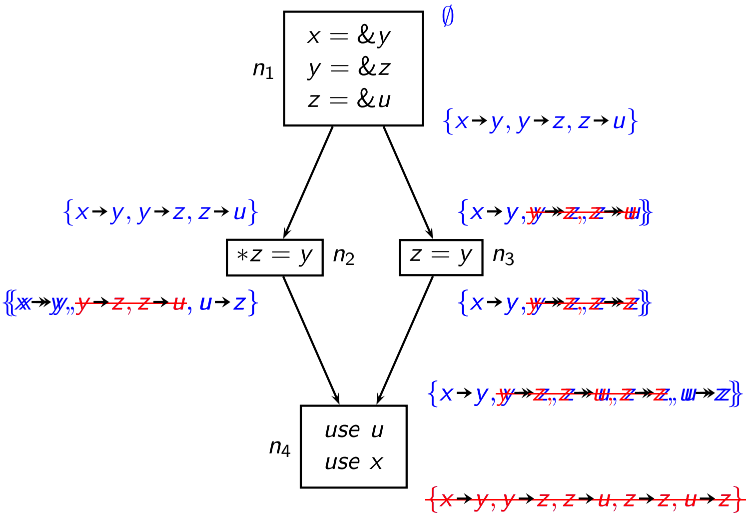
Issues with Pointer Analysis

- For precise pointer information, we require flow and context sensitive pointer analysis
- Flow and context sensitive pointer analysis computes a large size of information

Example of Points-to Analysis



Is All This Information Useful?



Improving pointer analysis

For a fast flow and context sensitive pointer analysis, we can reduce the number of computations done at a program point. This can be done in following ways :

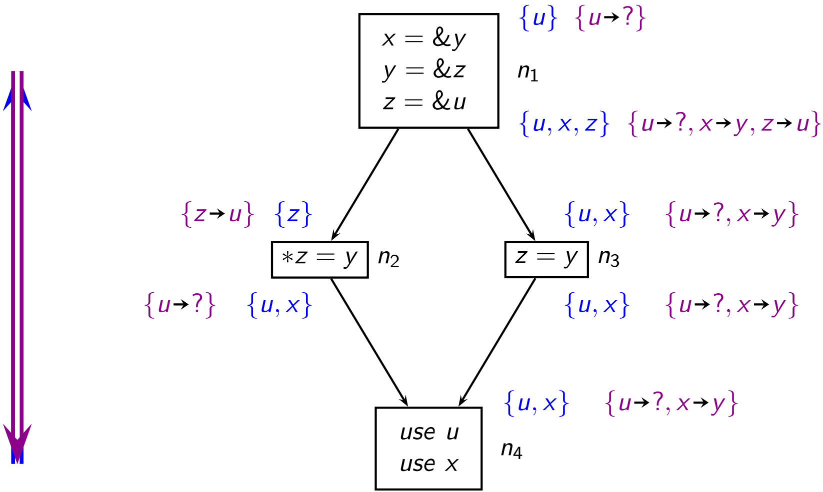
- Computing pointer information for only those variables that are being used at some later program point.
- Propagating only the new data flow values obtained in current iteration to the next iteration.

Liveness Based Pointer analysis(L-FCPA)

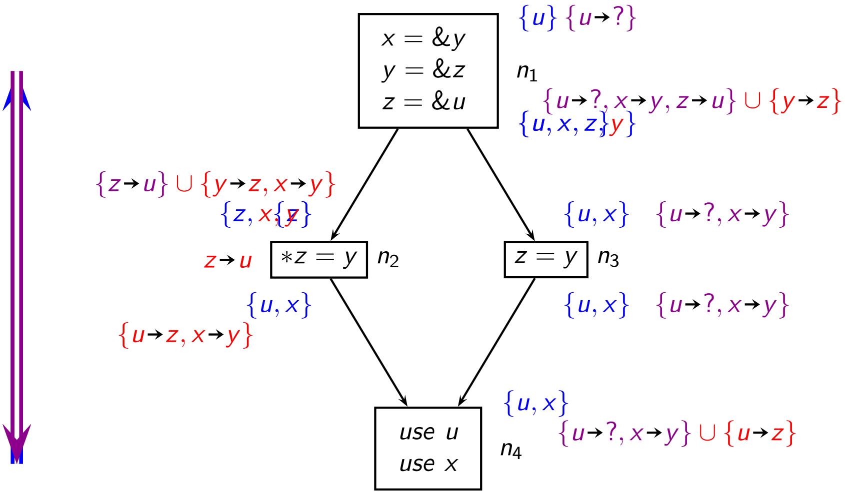
- A flow and context sensitive pointer analysis
- Pointer information is not computed unless a variable becomes live.
- Strong liveness is used for computing liveness information.

  If basic block contains statement like $x = y$, then y is said to be live, if x is live at the exit of basic block.
- Pointer information is propagated only in live range of the pointer

First Round of Liveness Analysis and Points-to Analysis



Second Round of Liveness Analysis and Points-to Analysis



Observation

- L-FCPA has 2 fixed point computations :
  - Strong Liveness analysis
  - Points-to analysis
- Liveness and Points-to passes are interdependent.
- Both the computations are done alternatively until final value converges.

Conclusions: New Insights in Pointer Analysis

- Usable pointer information is very small and sparse
- Earlier approaches reported inefficiency and non-scalability because they computed far more information than the actual usable information
- Triumph of *The Genius of AND over the Tyranny of OR*
- Future work
  - Redesign data structures by hiding them behind APIs
    Current version uses linked lists and linear search
  - Incremental version
  - Using precise pointer information in other passes in GCC

Precise Context Information is Small and Sparse

Our contributions: Value based termination, liveness

Program	Total no. of functions	No. and percentage of functions for call-string counts							
		0 call strings		1-4 call strings		5-8 call strings		9+ call strings	
		L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA
lbm	22	16 (72.7%)	3 (13.6%)	6 (27.3%)	19 (86.4%)	0	0	0	0
mcf	25	16 (64.0%)	3 (12.0%)	9 (36.0%)	22 (88.0%)	0	0	0	0
bzip2	100	88 (88.0%)	38 (38.0%)	12 (12.0%)	62 (62.0%)	0	0	0	0
libquantum	118	100 (84.7%)	56 (47.5%)	17 (14.4%)	62 (52.5%)	1 (0.8%)	0	0	0
sjeng	151	96 (63.6%)	37 (24.5%)	43 (28.5%)	45 (29.8%)	12 (7.9%)	15 (9.9%)	0	54 (35.8%)
hmmer	584	548 (93.8%)	330 (56.5%)	32 (5.5%)	175 (30.0%)	4 (0.7%)	26 (4.5%)	0	53 (9.1%)
parser	372	246 (66.1%)	76 (20.4%)	118 (31.7%)	135 (36.3%)	4 (1.1%)	63 (16.9%)	4 (1.1%)	98 (26.3%)
	9+ call strings in L-FCPA: Tot 4, Min 10, Max 52, Mean 32.5, Median 29, Mode 10								
h264ref	624	351 (56.2%)	?	240 (38.5%)	?	14 (2.2%)	?	19 (3.0%)	?
	9+ call strings in L-FCPA: Tot 14, Min 9, Max 56, Mean 27.9, Median 24, Mode 9								

Precise Usable Pointer Information is Small and Sparse

Our contribution: liveness

Program	Total no. of BBs	No. and percentage of basic blocks (BBs) for points-to (pt) pair counts							
		0 pt pairs		1-4 pt pairs		5-8 pt pairs		9+ pt pairs	
		L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA
lbm	252	229 (90.9%)	61 (24.2%)	23 (9.1%)	82 (32.5%)	0	66 (26.2%)	0	43 (17.1%)
mcf	472	356 (75.4%)	160 (33.9%)	116 (24.6%)	2 (0.4%)	0	1 (0.2%)	0	309 (65.5%)
libquantum	1642	1520 (92.6%)	793 (48.3%)	119 (7.2%)	796 (48.5%)	3 (0.2%)	46 (2.8%)	0	7 (0.4%)
bzip2	2746	2624 (95.6%)	1085 (39.5%)	118 (4.3%)	12 (0.4%)	3 (0.1%)	12 (0.4%)	1 (0.0%)	1637 (59.6%)
	9+ pt pairs in L-FCPA: Tot 1, Min 12, Max 12, Mean 12.0, Median 12, Mode 12								
sjeng	6000	4571 (76.2%)	3239 (54.0%)	1208 (20.1%)	12 (0.2%)	221 (3.7%)	41 (0.7%)	0	2708 (45.1%)
hmmer	14418	13483 (93.5%)	8357 (58.0%)	896 (6.2%)	21 (0.1%)	24 (0.2%)	91 (0.6%)	15 (0.1%)	5949 (41.3%)
	9+ pt pairs in L-FCPA: Tot 6, Min 10, Max 16, Mean 13.3, Median 13, Mode 10								
parser	6875	4823 (70.2%)	1821 (26.5%)	1591 (23.1%)	25 (0.4%)	252 (3.7%)	154 (2.2%)	209 (3.0%)	4875 (70.9%)
	9+ pt pairs in L-FCPA: Tot 13, Min 9, Max 53, Mean 27.9, Median 18, Mode 9								
h264ref	21315	13729 (64.4%)	?	4760 (22.3%)	?	2035 (9.5%)	?	791 (3.7%)	?
	9+ pt pairs in L-FCPA: Tot 44, Min 9, Max 98, Mean 36.3, Median 31, Mode 9								