Data-Flow Analysis: Simulation and Visualisation

Ayrton Massey

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2016

Abstract

Data-flow analysis is one of the cornerstones of modern compiler optimisation. A thorough understanding of the processes involved is essential to further exploration of the subject. A tool which allows exploration of data-flow analysis in an interactive environment would prove invaluable to students encountering the topic for the first time.

This report describes the design and implementation of an interactive system to simulate and visualise forms of data-flow analysis on simple assembly-like programs. The system is evaluated by in terms of user experience and the achievement of learning outcomes, through self-assessment and by examining usage data collected during the evaluation period.

The software proved (successful/unsuccessful) in providing a tool for increasing the learning capacity of the subjects.

Acknowledgements

Acknowledgements go here.

Table of Contents

Chapter 1

Introduction

This chapter gives a short introduction to the topic of data-flow analysis, describes motivations and desired outcomes for the project and provides a brief summary of contributions.

1.1 Data-Flow Analysis

Data-flow analysis is a tool for analysing the flow of data through a program at various points in its execution. Analysis is performed over a control-flow graph, computing the properties of values flowing *in* and *out* of each node. Many forms of data-flow analysis exist to compute various properties, for example *liveness analysis* identifies variables which will be used in future instructions and *available expressions* identifies those expressions whose value has been previously computed at some point in the control-flow graph.

This analysis is used to inform optimisations which can be performed on a given program. Using the example of *liveness analysis*, the values computed can be used to optimise register allocation: a variable which is not live at a given point does not need to be allocated to a register, enabling more efficient use of available resources.

Data-flow analysis is not only useful in compiler optimisation. The information gathered can be used in other ways, such as identifying unsafe operations in PHP web applications[1] by monitoring sanitization[1] of variables which have been assigned to user input.

[1] To *sanitize* a user input is to remove any potentially dangerous elements from said input; for example, if a user input string is to be inserted into the HTML of a webpage it could be sanitized by replacing instances of < and > with `<` and `>`, respectively. This would prevent that input being misinterpreted as HTML and thus avoid malicious scripts contained within that input from being executed.

1.2 Motivations

This project was inspired by the project's supervisor, Hugh Leather. The original concept was an online tutor for data-flow analysis which would allow users to simulate an analysis on simple programs. The user could vary parameters, such as the data-flow in question or the order in which nodes are evaluated, and examine the resulting solution.

As lecturer of the Compiler Optimisations course at the University of Edinburgh, Hugh desired a system which could teach students the foundations of the course in a more interactive format than standard lectures. The system should be suitable for hosting on the course web page to make it accessible to all students.

My personal interest in this project stemmed from a desire to use my practical skills to increase my capacity for understanding theoretical content. As noted in our early discussions, many students find it difficult and time consuming to read and understand material from the course textbook. Presenting this information in such a way that it could be easily digested by even a novice to Computer Science provided an exciting challenge.

1.3 Objectives

The main aim of this project was to create an interactive system to teach students the basic principles of data-flow analysis in compilers.

This would take the form of a web application using visual components which could be combined in different ways, for example to present a series of tutorials on data-flow analysis or to provide a sandbox environment to explore. The content of the application would cover a range of topics from the basics of data-flow analysis to algorithms and frameworks for solving generic data-flow problems.

The application would be aimed at students of the Compiler Optimisations course and as such would be based on material from the course textbook *Engineering a Compiler (2nd Edition)*[2] by Keith D. Cooper and Linda Torczon. The application could then be extended to cover the topic in more depth using content from *Compilers: Principles, Techniques and Tools, 1st ed.*[3] by Alfred V. Aho, Ravi Sethi and Jeffery D. Ullman.

Users would be able to interact with the system by providing simple assembly-like programs, altering parameters of the analysis and stepping through a simulation. Elements of the simulation such as the current state and the control-flow graph of the program would be visualised on-screen and update as the simulation progressed. Each of these elements would be linked visually to show how the concepts relate.

The application would be tested on real users. It would be evaluated in terms of user experience by analysing interactions with the system and conducting a user experience survey. Achievement of learning outcomes would be assessed by examining responses to questions built into the software and self-assessment by the user.

1.4 Summary of Contributions

The final version of the software is capable of the following:

- Simulation of pre-defined data-flows using generic framework models. (p.)
- Simulation of user-defined programs using the ILOC[2, appx. A] language from *Engineering a Compiler*. (p.)
- Simulation using the round-robin iterative algorithm (p.)
- User-controlled simulation allowing step-by-step, instant or automated playback.
- Visualisation of the following simulation elements:
  - Control-flow graph (p.)
  - Simulator state incl. currently evaluated node, framework etc. (p.)
  - Table of results displaying data flowing in / out of each node.
  - Hasse diagram of meet semi-lattice (p.)
- Tutorials covering basics of the topic with interactive elements (p.)
- Interactive test to assess achievement of learning outcomes

In addition, I have produced an API to record user interaction events modeled on the Google Analytics event tracking system (p.).

Detailed explanations of the terminology mentioned above can be found in chapter **??**. A quick summary of terms can be found in the glossary in appendix A.

Chapter 2

Background

In this chapter I will briefly cover the necessary background information required to understand this project, both to inform the reader and to demonstrate my own understanding of the topic.

2.1 Introduction to Data-Flow Analysis

Data-flow analysis computes information about data flowing *in* and *out* of each node in a program's control-flow graph. Many types of analysis can be performed and the information gathered can be used to inform the decisions of optimising compilers. A brief list of analyses and their purposes can be found in appendix B.

2.1.1 Control-flow Graph

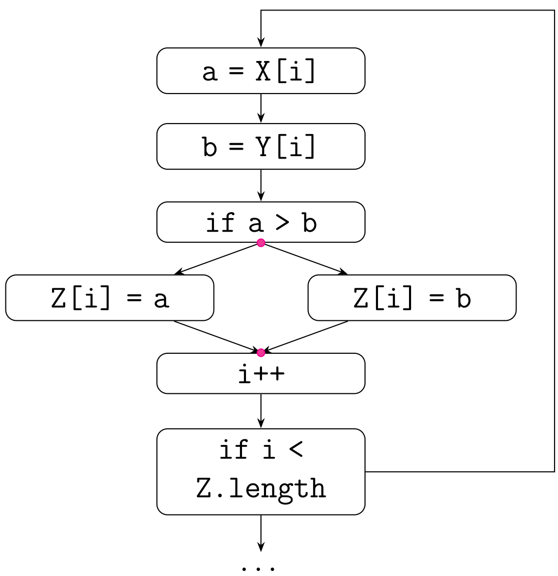

Figure 2.1: A control-flow graph.

A control-flow graph displays the possible execution paths for a given program. Each node in the graph represents an instruction or basic block, each edge represents an execution path leading from that instruction. A node can have multiple outward edges if it is a branching instruction. Branches may point backward in the control-flow. An example of a simple control-flow graph can be seen in fig. 2.1.

A *point* in the control-flow graph refers to some point along the edges of the graph. In data-flow analysis we usually deal with sets of values at the *in* and *out* points of each node, i.e. the point where the *in-edges* meet and the *out-edges* originate, respectively (shown in magenta on the left).

2.1.2 A Simple Example

An oft-used example of data-flow analysis is that of *reaching definitions*, which we will demonstrate here due to its simplicity. Reaching definitions computes the set of variable definitions which are are available at a given point. A definition is said to *reach* a point p if there is no intermediate assignment to the same variable along the path from the definition of the variable to the point p.

Let us take the example in fig. 2.1. The first node defines the variable `a`. We shall refer to this definition of `a` as $\mathtt{a}_1$. The definition of `a` reaches each point in the control-flow graph as it is never re-defined. The definition $\mathtt{c}_1$, however, does not reach the exit node as `c` is re-defined by $\mathtt{c}_2$. The graph in fig. 2.2 shows the set of reaching definitions at each point in the program. We have combined the *in* and *out* points to save space.

Data-flows have *direction*. Reaching definitions is a *forward flow problem*; values flow from the entry node of the control-flow graph (CFG) to the exit node.

The values at each point are determined using *data-flow equations*. For example, the equations for reaching definitions (defined in terms of *in* and *out*) at a given node n are:

$$\mathrm{In}(n) = \bigcup_{p \in preds} \mathrm{Out}(n)$$
$$\mathrm{Out}(n) = \mathrm{DefGen}(n) \cup (\mathrm{In}(n) \setminus \mathrm{Out}(n))$$

The equations for $\mathrm{In}(n)$ and $\mathrm{Out}(n)$ are often referred to as *meet* and *transfer* functions. In a forward flow problem the meet function combines the *out* sets of a node's predecessors to form its *in* set. The transfer function computes a node's *out* set from its *in* set and information obtained from the node itself, thereby *transferring* values through a node.

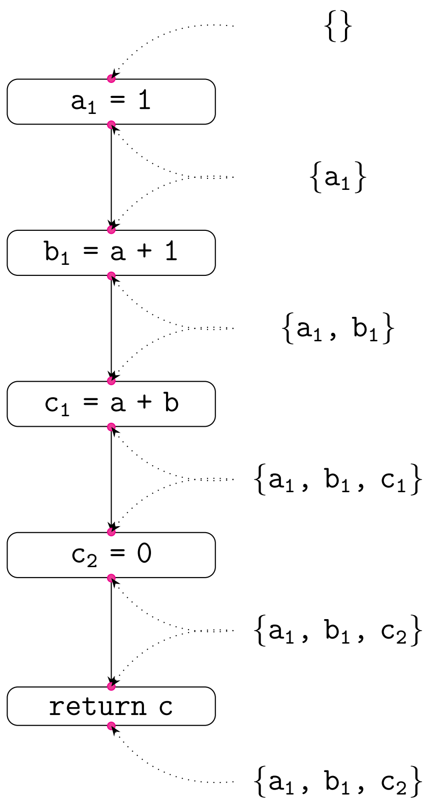

Figure 2.2: A control-flow graph.

2.1.3 Lattices

Sets in a data-flow problem have a partial order. This can be expressed using a structure known as a *meet semi-lattice*. A meet semi-lattice consists of a set of possible values L, the meet operator $\wedge$, and a *bottom element* $\perp$. The semi-lattice imposes an order on values in L such that:

$$a \geq b \text{ if and only if } a \wedge b = b$$
$$a \geq b \text{ if and only if } a \wedge b = b \text{ and } a \neq b$$

For the bottom element $\bot$, we have the following:

$$\forall a \in L,\ a \geq \bot$$
$$\forall a \in L,\ a \wedge \bot = \bot$$

We can express the semi-lattice using a *Hasse diagram*, shown in fig. 2.3.

Some data-flows deal with sets of pairs of values. In constant propagation, we pair a variable with one of three elements: $undef(\top)$, $nonconst(\bot)$ and $const$. A variable is initially paired with $undef$. When it is assigned a constant value, we assign it that value. If it is later assigned another value, we assign it *nonconst*. This can be expressed as the meet function, $\wedge$, seen in fig. 2.4.

The values form a semi-lattice, as seen in fig. 2.5.

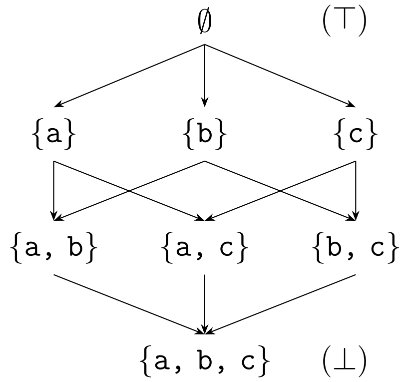

Figure 2.3: A Hasse diagram for the meet function $a \wedge b = a \cup b$.

$$\begin{aligned}
nonconst \wedge c &= nonconst && \text{for any constant } c \\
c \wedge d &= nonconst && \text{for any constants } c \neq d \\
c \wedge undef &= c && \text{for any constant } c \\
nonconst \wedge undef &= nonconst && \\
x \wedge x &= x && \text{for any value } x
\end{aligned}$$

Figure 2.4: Equations describing the constant propagation meet function.

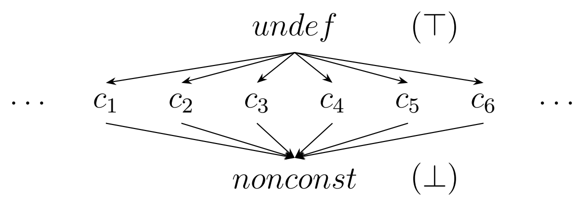

Figure 2.5: A Hasse diagram for the meet function described in fig 2.4.

2.2 Algorithms for Analysis

There exist a number of algorithms for performing data-flow analysis. In this section, I will give a brief overview of the methods proposed in the related reading[2] [3].

2.2.1 Round-Robin Iterative Algorithm

The simplest algorithm used to solve data-flow problems is the round-robin iterative method. We consider each node of the CFG in turn, calculating the In and Out sets using our data-flow equations. This is a fixed-point computation: we iterate until our value sets stop changing between iterations. Algorithm 2.1 shows the round-robin iterative method for solving reaching definitions.

Algorithm 2.1: Iterative Round-Robin Method for Reaching Definitions

```
for each (node n in the CFG)
    In(n) = ∅;
end

while (changes to any sets occur)
    for each (node n in the CFG)
        In(n)  = ∪_{predecessors p of n} Out(p);
        Out(n) = DefGen(n) ∪ (In(n) \ DefKill(n));
    end
end
```

2.2.2 Worklist Algorithm

The above solution holds value in its simplicity, but it is trivial to find a more efficient solution. A node's sets will only change if the Out sets of its predecessors change. Thus, we may use a worklist: starting with the initial node, we calculate the Out set for that node; if it changes we add the node's successors to the list. We continue this process until the worklist is empty. Algorithm 2.2 shows the worklist method for solving reaching definitions.

Algorithm 2.2: Worklist Method for Reaching Definitions

```
worklist = [ n0 ];

while (worklist is not empty)
    n = pop(worklist);
    In(n)  = ∪_{predecessors p of n} Out(p);
    Out(n) = DefGen(n) ∪ (In(n) \ DefKill(n));
    if (Out(n) has changed) append n to worklist;
end
```

2.2.3 Structural Algorithm

A third algorithm takes an entirely different approach. Instead of iterating over each node, we perform a number of simple transformations on the graph in order to reduce it to a single region. We then expand each node, calculating the In and Out sets of our reduced nodes as we expand them. A more detailed explanation of this concept may be found in the *dragon book*[3, p. 673].

2.3 Data-Flow Frameworks

It is possible to model data-flow problems using a generic framework. This allows us to use the same algorithm for multiple problems by specifying the following constraints[3, p. 680]:

- The *domain* of values on which to operate;
- The *direction* in which data flows;
- A set of *data-flow equations* including the *meet operator* $\wedge$ and the set of *transfer functions* F[1];
- The *boundary* value $v_{BOUNDARY}$ specifying the value at the entry or exit to the CFG; and
- The *initial value*, $\top$, at each point in the graph.

2.3.1 Algorithm for General Frameworks

Algorithm 2.3, adapted from the one in the *dragon book*[3, p. 691], computes the value sets at each node using the elements of our general framework.

Instead of referring to the value sets as In and Out as the *dragon book* does, we may call them Meet and Transfer. This allows us to generalise our algorithm to

Algorithm 2.3: Data-Flow Analysis of General Frameworks

```
Transfer_BOUNDARY = v_BOUNDARY;

for each (block B in the CFG)
    Transfer_B = ⊤;
end

while (changes to any Transfer occur)
    for each (block B in the CFG)
        Meet_B     = ∧_{priors P of B} Transfer_P;
        Transfer_B = F_B(In_B);
    end
end
```

Figure 2.6: Algorithm for Data-Flow Analysis of General Frameworks

[1] The function corresponding to a particular node/block B is denoted F_B

both forward and backward analyses; in the forward direction Meet is In whereas in the backward direction it is Out (and vice-versa for Transfer).

We first initialise the Transfer set using the boundary condition, then initialise each node's transfer set to our initial value $\top$.

Next, we perform a fixed-point computation on the CFG, evaluating each node's Meet and Transfer sets using our data-flow equations until the sets stop changing.

The meet is taken over a node's *priors*: in the forward direction, the node's predecessors; in the backward direction, the node's successors.

This algorithm can be applied to any framework. In fact, all of the data-flow problems in appendix B may be solved using this process.

2.3.2 Conditions for Termination

We must be careful when constructing our general frameworks. If our value sets continuously change we may never reach a fixed-point and thus our computation will never halt.

To avoid this, our frameworks must satisfy the following conditions[3, p. 684]:

- The set of transfer functions, F, contains the identity function[2];
- F is closed under composition: that is, for any two functions f and g, $f(g(x))$ is also in F;
- F is monotone; and
- The domain and the meet operator, $\wedge$, must form a meet semi-lattice.

These conditions ensure that during every iteration of the algorithm the values at each point will either become *smaller* (with respect to the partial ordering) or stay the same. Since F is monotone, all of the sets must eventually stop changing **or** reach the bottom element of the lattice, $\perp$, at which point they cannot change any further.

[2] The identity function maps its input to its output, i.e. $F(x) = x$

Chapter 3

Related Work

In this chapter I will discuss the merits and drawbacks of related work and identify aspects of said work which can be applied to this project.

3.1 Compiler Simulation

3.2 Compiler Visualisation

Blah Blah

3.3 Interactive Learning Environments

Blah Blah

Chapter 4

Data-Flow Analysis Simulation

Blah Blah

4.1 Data-Flow Framework

Blah Blah

4.2 Algorithms for Analysis

Blah Blah

4.2.1 Iterative Algorithm

Blah Blah

4.3 Intermediate Language for Optimising Compilers

Blah Blah

4.3.1 Parsing Expression Grammar

Blah Blah

4.3.2 Control-Flow Graphs

Blah Blah

Chapter 5

Data-Flow Analysis Visualisation

Blah Blah

5.1 Data-Flow Framework

Blah Blah

5.2 Simulator

Blah Blah

5.3 Control-Flow Graph

Blah Blah

5.4 Lattice

Blah Blah

Chapter 6

Interactive Learning

6.1 User-defined Programs

Blah Blah

6.2 Interactive Tutorials

Blah Blah

6.3 Experimental Sandbox

Blah Blah

Chapter 7

Evaluation

Blah Blah

Appendix A

Terminology

A.1 Glossary

available expression An expression is *available* if it has been computed along all paths leading to the current node. 1, 23

boundary The initial value at the starting point of an analysis, i.e. In($ENTRY$) or Out($EXIT$). 9

control-flow graph A graph representing the possible execution paths in a program. Nodes represent instructions, edges represent possible jumps between said instructions. 1–3, 5, 6, 21

data-flow A system of equations and conditions which constitute a data-flow problem, that is, a problem which may be solved through data-flow analysis. 2, 3, 6, 8–10, 21

data-flow analysis A technique for gathering information at various points in a control-flow graph. i, 1, 2, 5, 6, 8

data-flow equations A system of equations which determine how data flows through a CFG. 8–10

direction The direction in which data flows in a data-flow problem. Either forward (from the entry point of the CFG to the exit point) or backward (the opposite). 9

domain The domain of values considered in a data-flow problem, e.g. definitions or expressions. 9, 10

dominate In a CFG a node n_i is said to dominate a node n_j if every path from n_0 (the entry node) to n_j must go through n_i [2, p. 478]. 22, 23

fixed-point A computation in which the required process is repeated until the state stops changing. 8, 10

Hasse diagram A diagram used to represent partially ordered sets. Nodes represent elements of the sets, edges represent an ordering between a pair of elements. 3, 7

liveness analysis A variable is *live* if its current value will be used later in the program's execution. 1, 23

meet An equation (or set of equations) which determines how data flows *between* nodes in a CFG. 6, 9, 10

meet operator An operator which defines how the meet of two sets is obtained, such as $\cup$, $\cap$ or another operator entirely. 9, 10

meet semi-lattice A partially ordered set in which there exists a greatest lower bound (or meet) for any non-empty, finite subset. 3, 6, 10

reaching definition A definition of a variable *reaches* a block if there exists at least one path from its definition to the block along which it is not overwritten. 6, 8, 23

region A set of nodes in a graph which includes a *header* node which dominates all other nodes in the region[3, p. 669]. 9

transfer An equation (or set of equations) which determines how data flows *through* a node in a CFG. 6, 9, 10

A.2 Acronyms

CFG control-flow graph. 6, 8–10, 21, 22

Appendix B

Types of Data-Flow Analysis

Data-Flow	**Purpose**	**Applications**
Dominators	Computes the set of nodes which dominate the current node.	Computing SSA form.
Reaching definitions	Computes the set of variable definitions which are available at points in the CFG.	Generating def-use chains for other analyses.
Liveness analysis	Computes the set of variables whose current value will be used at a later point in the control flow graph.	Register allocation. Identifying useless store operations. Identifying uninitialised variables.
Available expressions	Identifies expressions which been computed at a previous point in the CFG.	Code motion.
Anticipable Expressions	Computes expressions which will be computed along all paths leading from the current point.	Code motion.
Constant Propagation	Computes the set of variables which have a constant value based on previous assignments.	Constant propagation. Dead code elimination.
Copy Propagation	Computes the set of variables whose values have been copied from another variable.	Dead code elimination. Code motion.
Tainted Flow Analysis[1]	Identifies unsafe operations which have been passed unsanitized *(tainted)* input as a parameter.	Preventing security vulnerabilities such as SQL injection and XSS attacks.

Table B.1: Types of data-flow analysis.

Bibliography

[1] A. Rimsa, M. d'Amorim, and F. Pereira. Tainted Flow Analysis on e-SSA-form Programs. *International Conference on Compiler Construction*, 2011.

[2] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, 2nd edition, 2011.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1st edition, 1986.