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 a interactive system to simulate and visualise forms of data-flow analysis on simple assembly-like programs. Variations of the system are compared and contrasted by exploring user interaction and measuring the achievement of learning outcomes of students using this system.

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 of variables which have been assigned to user input.

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 data-flow 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. The system could potentially be extended to allow custom data-flows.

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 modules 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 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)* by Keith D. Cooper and Linda Torczon. The application could then be extended to cover said material in more depth using content from *Compilers: Principles, Techniques and Tools, 1st ed.* 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 to the simulation and stepping through a simulation. Elements of the simulation, such as the current state, 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.

Variations of the application would be tested on users and 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 play-back.
- 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 model.

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 (or *CFG*). 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 section 2.1.4.

2.1.1 Control-flow Graph

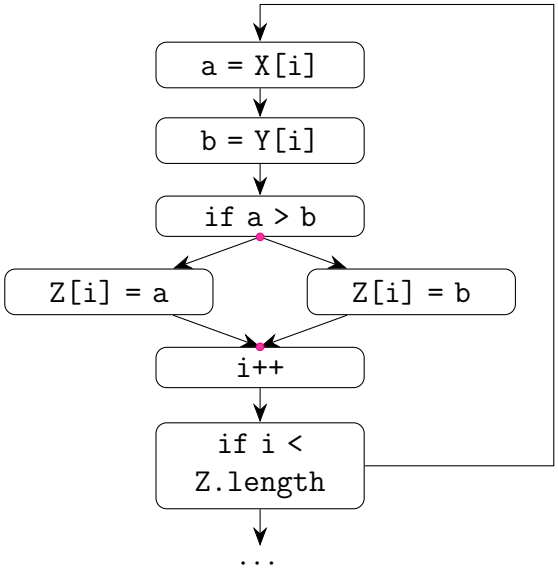

Figure 2.1: A control-flow graph.

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.

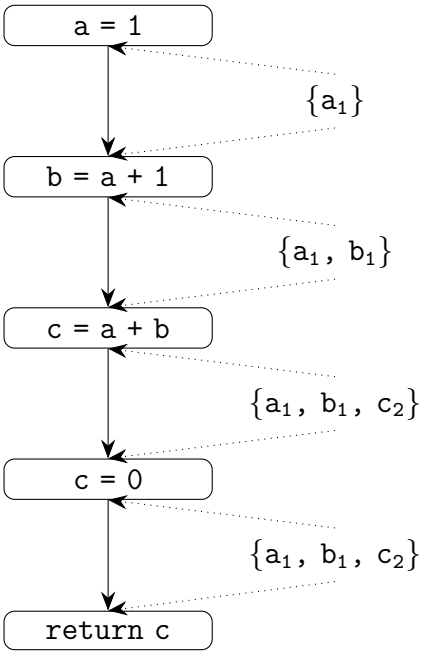

Figure 2.2: A control-flow graph.

Data-flows have *direction*. Reaching definitions is a *forward flow problem*; values flow from the entry node of the 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:

$$\mathrm{In}(n) = \bigcup_{p \in preds} \mathrm{Out}(n) \tag{2.1}$$

$$\mathrm{Out}(n) = \mathrm{DefGen}(n) \cup (\mathrm{In}(n) \setminus \mathrm{Out}(n)) \tag{2.2}$$

The equations for In(n) and Out(n) are often referred to as *meet* and *transfer* (or *join*) 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.

2.1.3 Lattices

Sets in a data-flow problem have a partial order. This can be expressed using a structure known as a *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:

$$a \geq b \text{ if and only if } a \wedge b = b \tag{2.3}$$
$$a \geq b \text{ if and only if } a \wedge b = b \text{ and } a \neq b \tag{2.4}$$

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

$$\forall a \in L, a \geq \perp \tag{2.5}$$
$$\forall a \in L, a \wedge \perp = \perp \tag{2.6}$$

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

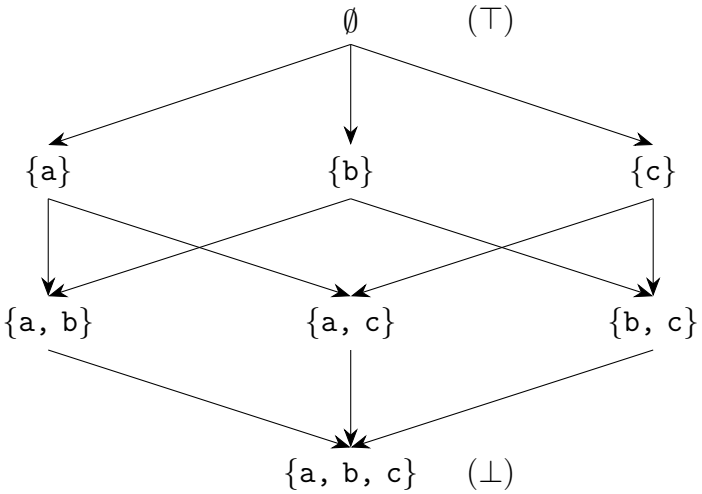

Figure 2.3: A hasse diagram for the meet function $a \wedge b = a \cup b$.

Some data-flows deal with sets of pairs of values. In constant propagation, we pair a variable with one of three elements: $undef(\top)$, $nonconst(\perp)$ and $const$. A variable is initially paired with $undef$. When it is assigned a constant value, we

$$nonconst \wedge c = nonconst \quad \text{for any constant } c \tag{2.7}$$
$$c \wedge d = nonconst \quad \text{for any constants } c \neq d \tag{2.8}$$
$$c \wedge undef = c \quad \text{for any constant } c \tag{2.9}$$
$$nonconst \wedge undef = nonconst \tag{2.10}$$
$$x \wedge x = x \quad \text{for any value } x \tag{2.11}$$

Figure 2.4: Equations describing the constant propagation meet function.

assign it that value. If it is later assigned another value, we assign it *nonconst*. This can be expressed as a meet function, seen in fig. 2.4.

These values also form a semi-lattice, seen in fig. 2.5.

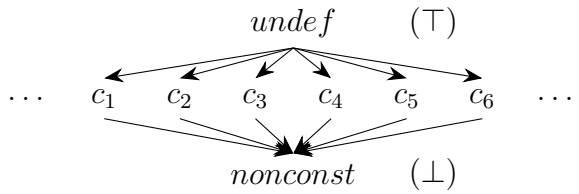

Figure 2.5: A hasse diagram for the meet function described in fig 2.4.

2.1.4 Types of Data-Flow Analysis

The table in fig. 2.6 covers a selection of data-flow analyses and lists some of their applications.

2.2 A General Framework

Blah Blah

2.3 Algorithms for Analysis

Blah Blah

2.3.1 Iterative Algorithm

Blah Blah

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.	Preventing security vulnerabilities such as SQL injection.

Figure 2.6: Types of data-flow analysis.

2.4 Intermediate Language for Optimising Compilers

Blah Blah

Chapter 3

Related Work

Blah Blah

3.1 Compiler Simulation

Blah Blah

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

4.2.2 Structural 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

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.