An Interactive Learning Platform for Compiler Data-Flow Analysis

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 (COPT) course at the University of Edinburgh, Dr. Leather 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 COPT course and as such would be based on material from the course textbook *Engineering a Compiler, 2nd ed.* [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 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 tracking system (p.).

Detailed explanations of the terminology mentioned above can be found in chapter 2. A quick summary of terms can be found in the glossary in appendix A.

Chapter 2

Background

This chapter briefly covers the necessary background information required to understand this project, both to inform the reader and to demonstrate 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
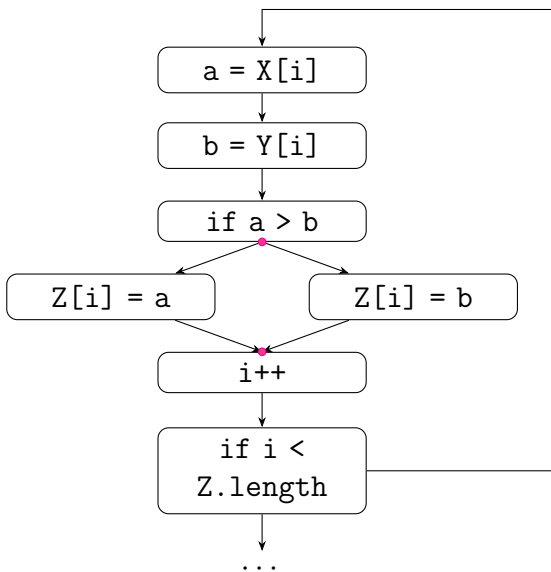
Fig. 2.1: A control-flow graph.

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 a_1. The definition of `a` reaches each point in the control-flow graph as it is never re-defined. The definition c_1, however, does not reach the exit node as `c` is re-defined by 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:

$$\text{In}(n) = \bigcup_{p \in preds} \text{Out}(n)$$

$$\text{Out}(n) = \text{DefGen}(n) \cup (\text{In}(n) \setminus \text{DefKill}(n))$$

The equations for $\text{In}(n)$ and $\text{Out}(n)$ are often referred to as *meet* and *transfer* functions. Other symbols such as $\text{DefGen}(n)$ and $\text{DefKill}(n)$ are referred to as local information, as they are constant values containing information about the current node. 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.
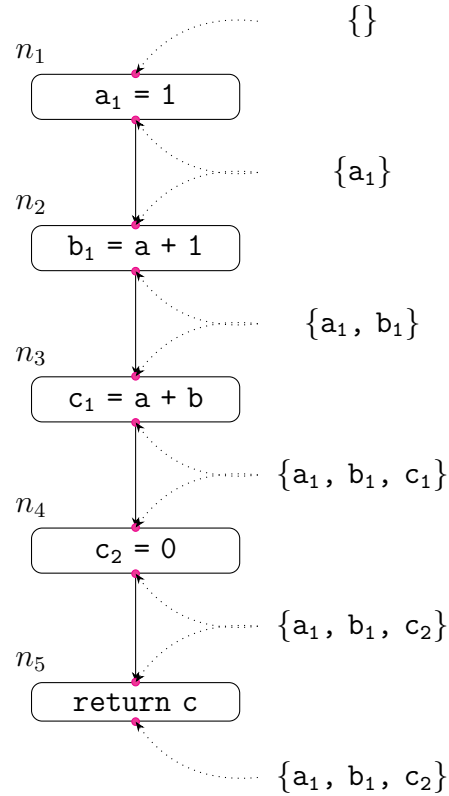

Fig. 2.2: A control-flow graph.

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 $\perp$, we have the following:

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

The semi-lattice can be expressed using a *Hasse diagram*, shown in fig. 2.3.
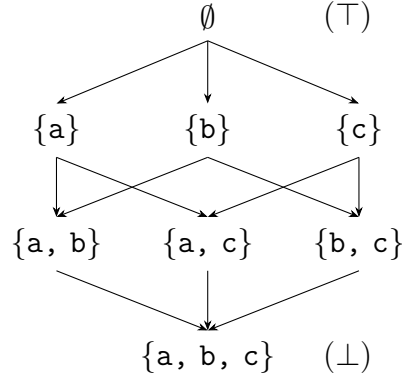
Fig. 2.3: A Hasse diagram for the meet function $a \wedge b = a \cup b$.

2.1.4 Bit-Vector Data-Flows

The data-flows present in the system described in this project are commonly known as bit-vector data-flows. In these problems values are single items, such as a variable or definition. A set of values can be represented as a vector of bits, each bit representing a particular value and a 0 or 1 indicating the presence or absence of that value. The reaching definitions example from the previous section is a bit-vector data-flow problem. Fig. 2.4 shows how this can be applied to the example in §2.1.2.

$$\text{Out}(n_5) \quad = \quad \{ \quad \begin{matrix} a_1 & b_1 & c_1 & c_2 \\ 1 & 1 & 0 & 1 \end{matrix} \quad \}$$

Fig. 2.4: A bit-vector for the CFG in fig. 2.2

Tuple-valued data-flows consider values as tuples instead of single items. One such example is constant propagation, in which variables are paired 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 assign it that particular value. If it is later assigned another value, we assign it *nonconst*. This can be expressed as the meet function, $\wedge$, seen in fig. 2.5. The values form the semi-lattice in fig. 2.6.

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

Fig. 2.5: Equations describing the constant propagation meet function.
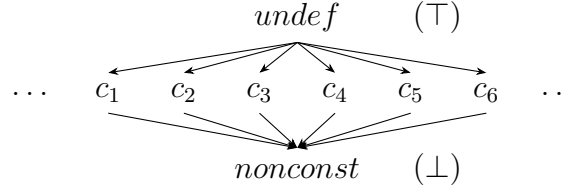
Fig. 2.6: A Hasse diagram for the meet function described in fig 2.5.

2.2   Algorithms for Analysis

There exist a number of algorithms for performing data-flow analysis. This section contains 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

```
1  for each (node n in the CFG)
2      In(n) = ∅;
3
4  while (changes to any sets occur)
5      for each (node n in the CFG)
6          In(n)  = ∪predecessors p of n Out(p);
7          Out(n) = DefGen(n) ∪ (In(n) \ DefKill(n));
```

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

```
1   worklist = [ n_0 ];
2
3   while (worklist is not empty)
4       n = pop(worklist);
5       In(n)  = ∪_predecessors p of n Out(p);
6       Out(n) = DefGen(n) ∪ (In(n) \ DefKill(n));
7       if (Out(n) has changed) append n to worklist;
```

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.

---

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

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.

Algorithm 2.3: Data-Flow Analysis of General Frameworks

```
1   Meet_BOUNDARY  = v_BOUNDARY ;
2
3   for  each ( block  B  in  the CFG)
4        Meet_B  = ⊤ ;
5
6   while ( changes  to  any  Transfer  occur )
7        for  each ( block  B  in  the CFG)
8             Meet_B        = ∧_priors P of B  Transfer_P ;
9             Transfer_B = F_B ( In_B ) ;
```

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 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.

―――――――――――――――――――――――

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

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, $\bot$, at which point they cannot change any further.

Chapter 3

Related Work

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

3.1 Introduction

Although there exists a wide range of literature dealing with data-flow analysis in compilers *independent* of interactive tutoring and vice-versa, the combination of the two has yet to be explored. Therefore, research has been widened to two main areas: the topic of data-flow analysis, and advancements in interactive tutoring software.

Chapter 2 provided a brief introduction to data-flow analysis with reference to two major textbooks; the first, *Engineering a Compiler, 2nd ed.*[2] by Keith D. Cooper and Linda Torczon, serves as an excellent introduction to the topic and an overview of some of the more complex elements. The second, *Compilers: Principles, Techniques and Tools, 1st ed.*[3] (often referred to as the *Dragon Book*) provides a more theoretical discussion of the material.

This chapter discusses the second area of research, advancements in interactive tutoring software.

3.2 Online Learning Platforms

In recent years there has been a surge in online learning platforms such as Khan Academy and Coursera which deliver a series of lectures or tutorials over the internet.

The videos are very similar in format to a traditional lecture, consisting of a voice-over accompanied by related imagery or text-based slides. However, the advantage over the traditional format is two-fold: the ability to view the content
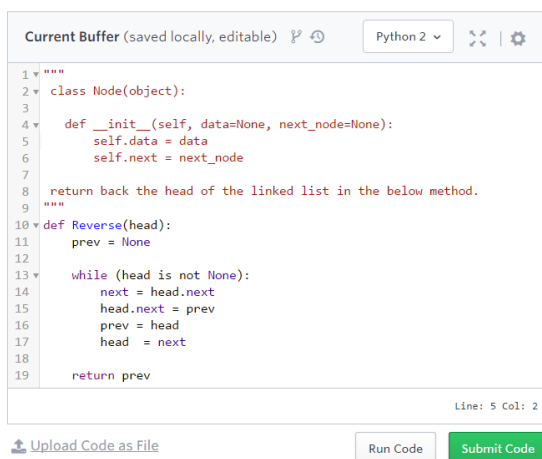
at the student's convenience, rather than in a specific location and time slot, allows students to organise their learning around their own schedule. Users are also able to replay portions of the class they have missed or struggled to understand. In this way, learning may be tailored to a student's specific needs.

These platforms often integrate an interactive or practical element into their teaching; whilst services such as Coursera offer feedback through the more traditional peer-assessed hand-in[4], others such as Khan Academy and HackerRank provide live demos and code interpreters to develop understanding.

For example, in its series on linked lists[5], HackerRank presents the user with a series of coding challenges. The user inputs code to solve a given problem (fig. 3.1), such as reversing a linked list, and the site verifies their solution. Khan Academy's introduction to binary search[6] demonstrates the increase in efficiency with a visual representation of both linear and binary search (fig. 3.2), allowing the user to step through each and compare the number of steps taken for themselves.

These demonstrations enable a different kind of learning than that which can be found in the classroom. Students are able to use their intuition to form their own understanding of the material and take a much more active role in their own development. This kind of learning, referred to as *kinaesthetic* learning, can be incredibly effective (see §3.4.2).

There are some drawbacks to this model: whilst students may learn at their own pace, the only information available is that on the page or in the video they are watching. If they have any questions or struggle to understand the material as presented, there is no lecturer or teaching assistant to provide alternate explanations or answer queries. Some platforms provide discussion forums for peer support, but this can result in "the blind leading the blind" – without an expert hand to guide them, students may cement incorrect knowledge, defeating the purpose of the learning platform.

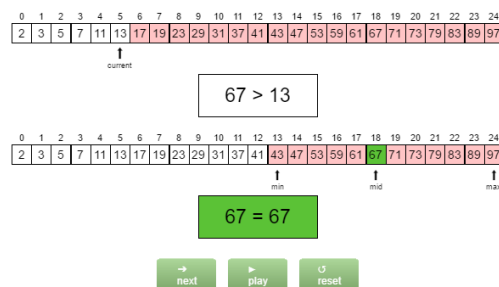Fig. 3.1: Code editor from HackerRank[5]

Fig. 3.2: Binary search visualisation from Khan Academy[6]

3.3   Simulators as a Teaching Aid

In his article *Evaluating A System Simulator For Computer Architecture Teaching And Learning Support*[7], Mustafa discusses the design & implementation of a system which integrates the simulation of a compiler, CPU and operating system to aid in the teaching of undergraduate computer architecture and operating systems modules.

Although brief, the article provides valuable insight into the design and evaluation of such teaching software. The primary source of feedback came from an opinion survey using a 5-point Likert scale for quantitative analysis and open-ended questions for qualitative feedback. In addition, students were administered a test to assess their knowledge pre- and post- use of the system.

The evaluation results highlight an important point to consider when designing teaching software; over 20% of respondents indicated that they spent more time learning how to use the software than they did completing the given exercises. An equal number of students reported that the simulator left them more confused than before. In addition, 7.1% of respondents said that the simulator was too complicated to use effectively in their tutorials.

However, the converse of these results gives a positive outlook for this project: 72.4% of respondents disagreed with the above statements and 79.3% believed that the system improved their understanding of the topics covered in lectures. Over 95% of responents agreed that the simulator was more useful than reading textbooks or searching the internet in helping them understand the material.

3.4   Learning and Teaching Styles

A 1988 report by Richard M. Felder and Linda K. Silverman, entitled *Learning and Teaching Styles in Engineering Education*[8], categorizes students' learning methods and describes ways in which professors may target specific categories in their teaching strategy.

Felder refers to the learning modalities (or VAK) model proposed by Walter Burke Barbe, which defines three modalities:

- **visual** – those who learn best by viewing images and diagrams;

- **auditory** – those who learn best by listening or speaking aloud; and

- **kinaesthetic** – those who learn best by actively experiencing things, learning by doing.

The report claims that most people of college age and above, the intended audience of this project, identify as *visual* learners – those who benefit from charts, diagrams and such. In contrast, the presentation of content in university courses

is primarily *auditory* (via lectures) or a visual representation of auditory information (i.e. words or mathematical formulae).

The survey results from Mustafa[7, p. 103] support this observation: although only a small sample (N=54) responded, 31% of respondents identified as visual learners, while a mere 6% identified as auditory learners. The majority of students (52%) identified as kinaesthetic learners, indicating that taking a hands-on approach is a valuable tool for learning.

It should be noted that Felder discounts kinaesthetic learning from his report as he considers the *learning by doing* to be a separate category, perceiving the remaining attributes of kinaesthetic learning to have little value in engineering. A later observation by Felder agrees with Mustafa, stating that engineers are "more likely to be active than reflective learners"[8, p. 678].

3.4.1   Ability to Identify Own Learning Style

In her PhD dissertation *"Individual Differences in Learning: Predicting One's More Effective Learning Modality"*[9], Beatrice J. Farr claims that students are able to accurately predict the learning style in which they perform best:

> "An experiment with 72 college students confirmed that individuals could accurately predict the modality in which they could demonstrate superior learning performance. The data also revealed that it is advantageous to learn and be tested in the same modality and that such an advantage is reduced when learning and testing are both conducted in an individual's preferred modality."[10, p. 242]

Coffield[11, p. 120] disagrees with this, based on an observation by Merrill[12] that "most students are unaware of their learning styles and so, if they are left to their own devices, they are most unlikely to start learning in new ways". This would indicate that basing teaching methods on a student's preferences is damaging to their education. This is, however, a misinterpretation; the actual text of Merrill reads:

> "... a student must engage in those activities ... that are required for them to acquire a particular kind of knowledge or skill ... Most students are unaware of these fundamental instructional (learning) strategies and hence left to their own are unlikely to engage in learning activities most appropriate for acquiring a particular kind of knowledge or skill."[12, p. 4]

Merrill later argues that the optimal strategy for teaching is decided first and foremost by the content being taught, then fine-tuned to the learner's preferred style[12, p. 4]. By not knowing the most effective strategy for learning the content they are studying, a student limits his or her potential. However, the appropriate strategy *should* in fact be tailored to the student's preferred learning style to obtain the best results.

3.4.2 Criticism of Learning Modalities

Some of the criticism levied at the concept of learning styles is that the idea of *matching* – exclusively teaching a student based on his or her preferred learning style – is harmful to a student's education. Although Coffield's[11, p. 120] reasoning is flawed, the conclusion drawn by him and Merrill[12, p. 4] is sound: by allowing students to exclusively use their preferred way of learning, students are at risk of missing out on more effective methods of study.

Felder's report mentions a study carried out by the Socony-Vacuum Oil Company, which concludes that:

> "...students retain 10 percent of what they read, 26 percent of what they hear, 30 percent of what they see, 50 percent of what they see and hear, 70 percent of what they say, and 90 percent of what they say as they do something."[8, p. 677]

This indicates that by relying solely upon auditory methods, professors can only hope to convey as little as 26% of the desired material to their students. Felder advocates using a mixture of teaching methods in order to appeal to all students' preferred learning styles.

3.5 Summary

While there may be some disagreement over the validity of learning styles or modalities, even its critics seem to agree that there is value in varying the methods of teaching in use. At present, the only available resources for learning data-flow analysis are lecture slides, videos and textbooks. These are mostly auditory and sometimes visual methods of teaching, with little kinaesthetic learning involved. There is a clear need for more active study; although 52% of students identified as learning best through kinaesthetic learning[7, p. 103], there is almost no support for this method of study.

Given the relative success of the examples discussed in this chapter and the apparent lack of any such resources for data-flow analysis, there is a strong precedent for this project and as such a system will be developed to fulfil this role.

However, it is important to note the mistakes made by the examples discussed here. The simulator designed by Mustafa was deemed too complex[7, p. 103] and a detriment to their learning by over 20% of respondents. This project will seek to ensure that that content is clear and concise on order to produce an effective learning resource.

To evaluate the success of the system this project will build upon the methods presented by Mustafa, aggregating opinion using a Likert scale and examining this data to judge overall satisfaction with the software and identify specific areas for improvement.

Chapter 4

Design

This chapter discusses the high-level design and architecture of the system, including motivations for design choices and solutions to conceptual problems.

4.1 Introduction

The original goal of this project was to produce an online simulation of data-flow analysis to show students how data-flow works. As discovered in the related reading, however, a large portion of users of a similar system found it to be too complex and claimed to have spent more time studying the software itself than the topic at hand. The literature also raised another key point; a wide variety of learning techniques are necessary to gain a true understanding of a topic.

For these reasons the proposal has been extended to create a comprehensive learning platform. In addition to the simulator the software will provide supporting lectures to explain the basic concepts and gradually introduce each element of the simulation. These lectures will include visual and interactive elements to engage the user through a range of learning styles. It is hoped that the system will prove a valuable tool for learning alongside existing resources such as textbooks and lectures.

4.2 Design Constraints

In this section, design constraints are identified and potential solutions are suggested.

4.2.1 Technical Constraints

As the intention isto make such a system available to COPT students via the University, the following technical considerations must be made:

- The system must be distributed to all students in some format;

- This format must be functional on and compatible with a wide range of devices owned by said students;

- The system must be secure and require little maintenance; and

- The system must be hosted on some platform available to the University.

In order to meet these criteria the system must rely on as little technology as possible. The easier the platform is to host and distribute, the more likely it is to be made available to students – a system which uses new technology and requires its own dedicated hosting would be more difficult to set up and maintain than one which can be deployed to existing hardware. It is necessary to ensure that the system is secure to avoid damaging University or student property.

The application could be developed using purely client-side technologies such as JavaScript, HTML and CSS. These static files may be distributed over HTTP using any standard web server such as the one already used to host the course webpage. Efforts must be made to keep the performance of the system consistent across a range of web browsers and devices; although all modern web browsers are capable of interpreting these types of content there are subtle differences which may break functionality on one platform but not others.

Use of popular libraries and frameworks such as jQuery and Bootstrap will be encouraged as this provides a number of benefits. The user will experience reduced load times since they have likely already downloaded the required files, and such libraries will provide security and robustness due to their wide use and active development. Likewise, the more popular a library is the more documentation and resources will be available to aid in developing the best possible system.

The specific technologies used in each are detailed in the implementation (chapters 5 & 6).

4.2.2   Content Constraints

To be viable as a learning platform, the system must:

- Appeal to a range of learning styles;

- Provide comprehensive coverage of the topic at hand;

- Ensure the content included is correct, clear and concise; and

- Maintain a shallow learning curve, gradually introducing students to each topic or element of the simulation.

Extending the proposed system into a comprehensive learning platform will provide a range of ways in which to study data-flow analysis. Students will have the choice to use the areas of the software which most appeal to them and content will be presented using a mixture of interactive, visual and textual formats.

As the aim is to assist students of the COPT course, coverage of topics will be prioritised based on their inclusion in the course syllabus and whether they are required to understand the simulation:

- Basic principles of data-flow analysis;

- A few simple data-flows, including:

  - Reaching Definitions

  - Liveness Analysis

  - Available Expressions

- Round-robin fixed-point algorithm;

- Effect of orderings on analysis efficiency;

- Generic frameworks for data-flow analysis;

- Hasse diagrams and lattice representation of values; and

- Conditions for analysis termination.

If the content delivered by the platform is incomplete, incorrect or too difficult to understand, the system will not present a viable alternative method of study. When evaluating the system it will be important to assess the clarity and presentation of the material, and the ease of use of the platform as a whole.

4.3 System Architecture

This system is divided into two main areas of development. The back-end of the system controls the logic and state of a data-flow analysis simulation. The front-end components will hook into and expose elements of the back-end, visualising the state of the simulation and allowing the user to interact with it. See fig. 4.1 for a visual representation.

Object-oriented design will be paramount, allowing functionality to be extended and shared between similar elements. In addition to speeding up development of the system by reducing the need to duplicate code, this is generally good design practice. Components will be written as re-usable modules which can be combined in different ways, for example to present the user with a series of lectures or an interactive simulation. See §4.4 for more details on the user interface.

Each component needs to be be self contained; it should be possible to instantiate more than one of each type of component at once, including the simulator itself. This would allow, say, displaying the annotated control-flow graphs of two simulations side-by-side. The simulation will also be independent of the user interface so that it may be adapted for re-use in other projects.
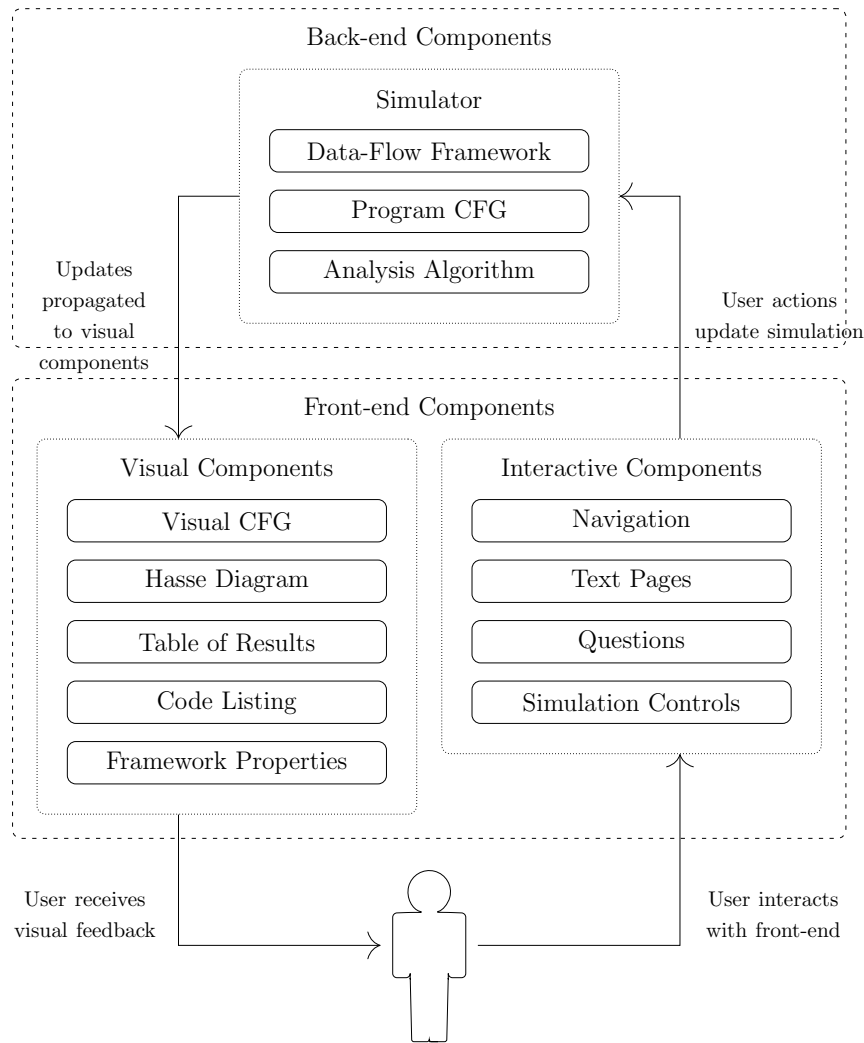
Fig. 4.1: An overview of the planned system architecture.

4.4   User Interface Design

4.5   Data Collection

To evaluate the system anonymous usage data will be collected. A variety of web platforms exist for this purpose including Google Analytics and Piwik. However, these platforms are intended for large-scale data collection and as such are not suited for the fine-grained analysis necessary in the evaluation of this project. As an example, it is not possible to link tracked events to specific users or view a complete list of triggered events using Google Analytics. Furthermore, extended use of these platforms often comes at significant financial cost. It is also important to consider the ethical issues that may arise from sharing usage data with companies like Google; using such a platform would require a much deeper discussion around user privacy and use of the data for financial gain.

Developing a custom event-tracking API would allow full control of the data

collected both in terms of accessing the raw data and maintaining user privacy.

The data model must be able to store the following on a per-user basis:

- Page views;

- Button clicks and interactions with components;

- Page load times; and

- Question attempts and overall scores.

All of these can be modeled using an API with a single point[1] based on the Google Analytics event model. Each `event` record has the following fields:

- A `tracking token` used to group events by user;

- A `type` such as page view or button click;

- The `datetime` at which the event occurred; and

- Four levels of categorization: `category`, `action`, `label`, and `value`. Each field can be used to group similar events and store values such as question scores or which element was clicked. The actual content of these fields does not matter as long as the grouping is consistent.

The tracking token will be generated when the user first visits the page and persist across visits to the site. Storing this token will enable analysis of how a user's actions affect their achievement of learning outcomes, such as whether completing a given tutorial or using a simulator component increases question scores in the related topic. It will be possible to determine which elements of the software users find difficult to understand based on their interactions with each component.

---

[1]An endpoint is a URL at which a particular API service may be accessed, e.g. `api.com/users/` may provide access to user data

Chapter 5

Back-End Implementation

This chapter discusses the implementation of the data-flow analysis simulation – the back-end elements which control the logic and state of the analysis, which is then presented by the user interface components described in chapter 6.

5.1 Overview

The implementation of the simulator is based upon the concept of a general framework as described in §2.3 of this report and §10.11 of the *Dragon Book*[3]. The simulator takes as input a program written in the ILOC language, a data-flow framework and an order in which to evaluate nodes. It parses the program into an abstract syntax tree (AST) and uses this to build a CFG, then performs an analysis on said graph using the round-robin fixed-point algorithm for general frameworks (algorithm 2.3).

Playback of the simulation can be controlled through the simulator's API. Function calls which update the state of the simulation trigger an event handling system to update the visual components. However, the simulation is entirely independent of the rest of the system; re-using it in another project is as simple as copying the directory containing the simulator code and including the required libraries.

The simulator is implemented entirely in JavaScript. Snippets of code taken from external sources are explicitly commented as such in the source code and will be mentioned here.

5.2 Value Sets

JavaScript includes a native `Set` data structure for representing collections of unique objects. Unfortunately, two objects are only seen as equal if they refer to the exact same instance. For the purposes of this simulation it would be useful to

compare objects which share some attributes but refer to different instances; for example, when operating on sets of operands it would be useful to consider two operands of different instructions as the same if they refer to the same variable.

For this reason the simulator uses a new `ValueSet` data structure, backed by a native JavaScript `array`. The `ValueSet` stores objects which inherit from a `ValueMixin`[1] and must define a `compare` function. Objects are checked for equality using this function so that two objects may be considered equal based on selected attributes.

`ValueSet`s support the following operations, listed here along with their estimated worst-case time complexities:

s.size()	s.add(v)	s.delete(v)	s.has(v)
$O(1)$	$O(|s|)$	$O(|s|)$	$O(|s|)$

s.union(t)	s.intersect(t)	s.difference(t)
$O(|s| \cdot |t|)$	$O(|s| \cdot |t|)$	$O(|s| \cdot |t|)$

Using an alternate backing structure such as a hash table would provide constant-time implementations of `has`, `add` and `delete`, and thus also improve the run-time of operations which make use of these functions (`union`, `intersect` and `difference`). Given that the expected user input will only operate on small `ValueSet`s this is not a major concern, but a potential improvement that should be noted for future development.

5.3   Data-Flow Frameworks

Each data-flow framework defines the elements outlined in §2.3, namely:

- 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*.

- The *boundary* value specifying the initial value at the entry or exit to the CFG; and

- The *initial value*, $\top$, at each point in the graph.

The meet and transfer functions operate on `ValueSet`s. Frameworks must also specify the following additional information:

- A name and identifier;

- LaTeX representations of the meet and transfer functions; and

---

[1]A mixin class contains methods which may be inherited by another class, but is not necessarily the parent of that class.

- JavaScript functions to compute local information independent from the transfer function.

This allows properties of the framework to be displayed by the visual components in addition to the values calculated at each point.

5.4 Simulation Algorithm

Algorithm 5.1 shows a disassembled version of algorithm 2.3. The functions can be combined to enable a step-by-step evaluation or automatic playback.

Algorithm 5.1: Implementation of General Framework Algorithm

```
1   function reset ()
2       B = entry node of CFG
3       B.meet = framework.boundary
4       for each (node in the cfg)
5           node.meet = framework.top
6       step    = MEET
7       changed = false
8
9   function iterate ()
10      if (step is MEET)
11          B.meet, changed = framework.meet(b, this.cfg)
12          step = TRANSFER
13      else
14          B.transfer, changed = framework.transfer(b)
15          step = MEET
16          B    = next node
```

To demonstrate, listings and 5.1 and 5.2 show the implementation of some of the playback functions. In the fast forward function, the simulation is advanced until it is complete before triggering an update event which is propagated to other components of the system. In the automatic playback function, a function is called at set time intervals which advances the simulation by one iteration, then triggers the update. Controlling when update events are triggered events helps prevent slowdown from unnecessary re-rendering of components.

Listing 5.1: JavaScript Implementation of Fast Forward

```
1   this.fast_forward = function() {
2     while (!this.state.finished) {
3       this.iterate();
4     }
5     this.events.trigger('update');
6   }
```

Listing 5.2: JavaScript Implementation of Automatic Playback

```
1  this.play = function() {
2    this.state.paused = false;
3    var _this = this;
4    (function foo() {
5      if (!_this.state.paused && !_this.state.finished) {
6        // If the user hasn't pressed pause and we're not
              finished
7        _this.iterate();                        // Step forward
8        _this.events.trigger('update');    // Update components
9        setTimeout(foo, _this.play_speed); // Repeat at interval
10     }
11   })();
12 }
```

5.5   Understanding Programs

The simulation operates on programs written in ILOC, as described in *Engineering a Compiler*[2, appx. A]. ILOC is an assembly-like language designed for toy compilers as it is both simple to parse and human-readable. This makes it the perfect input language for the learning platform, with the added benefit that students will be able to step through examples from the textbook using the simulator.

The simulator supports the following ILOC features:

- The entire set of opcodes listed in *Engineering a Compiler*;

- Three types of literal: registers, labels, and integers;

- Simple programs with a single operation per instruction; and

- Distinction between three types of memory: registers, main memory, and the comparison register.

There is also limited support for multi-operation instructions, which may be used to emulate data-flow analysis on basic blocks. This functionality remains untested as priority was assigned to simulation on nodes as single instructions.

In order to understand ILOC programs, they must be to converted from text into a CFG. To do this the simulator uses the PEG.js[13] library which, given an input grammar, generates a parser written in JavaScript. This parser produces an abstract syntax tree (AST), which is then transformed into a CFG representing the program. The following sections describe this process in more detail.

5.5.0.1   Parsing Expression Grammar

A Parsing Expression Grammar (PEG) is very similar to a context-free grammar in that it describes a formal language using a set of rules for recognising strings

in that language. The description of ILOC in *Engineering a Compiler* describes a simple PEG for the language, which has been adapted for use in this system.

The rules of the grammar have been extended as shown in fig. 5.1. The reason for these changes is that ILOC allows the user to write to three types of storage: registers, main memory and comparison flags. Changing the grammar allows disambiguation in the abstract syntax tree between operations which write to or read from different types of storage. This is an important distinction to make because the built-in data-flows only consider register accesses when calculating value sets at each point (see §**??** for an explanation).

The full extended ILOC grammar is listed in appendix **??**.

$$
\begin{array}{rcl}
\textit{Operation} & \rightarrow & \textit{NormalOp} \\
 & | & \textit{ControlFlowOp} \\
 & | & \textit{MemoryLoadOp} \\
 & | & \textit{MemoryStoreOp} \\
\end{array}
$$

MemoryLoadOp	→	*LoadOpcode*	*OperandList*	⇒	*OperandList*
MemoryStoreOp	→	*StoreOpcode*	*OperandList*	⇒	*OperandList*
NormalOp	→	*NormalOpcode*	*OperandList*	⇒	*OperandList*
ControlFlowOp	→	*ControlFlowOpcode*	*OperandList*	⇒	*OperandList*

$$
\begin{array}{rcl}
\textit{LoadOpcode} & \rightarrow & \texttt{loadI} \\
 & | & \texttt{loadAO} \\
 & | & \dots \\[4pt]
\textit{StoreOpcode} & \rightarrow & \texttt{storeI} \\
 & | & \texttt{storeAO} \\
 & | & \dots \\[4pt]
\textit{NormalOpcode} & \rightarrow & \texttt{addI} \\
 & | & \texttt{rshift} \\
 & | & \dots \\[4pt]
\textit{ControlFlowOpcode} & \rightarrow & \texttt{jumpI} \\
 & | & \texttt{cbr\_GE} \\
 & | & \dots \\[4pt]
\textit{Operand} & \rightarrow & \textit{register} \\
 & | & \textit{num} \\
 & | & \textit{label} \\
 & | & \textit{cc} \\[4pt]
\textit{cc} & \rightarrow & \texttt{cc} \\
\end{array}
$$

Fig. 5.1: Extensions to ILOC Parsing Expression Grammar

5.5.1   Abstract Syntax Tree

A parse tree represents the exact structure of parsed data. In contrast, an AST represents the structure of data independent of its original representation; this may involve adding or removing nodes or re-structuring the tree entirely.

To construct an AST one would usually obtain a parse tree and then transform it into the desired structure. Parsers generated by the PEG.js library are capable of constructing the AST as they parse the input program. The developer may supply JavaScript code to be executed when a rule matches and this code is used to create an abstract representation of the data matched by that rule. This feature can also be used to annotate the tree as it is built.

Listing 5.3: Example of PEG.js Grammar Rules

```
1  Operand
2      = _ r:register _ { return r; }
3      / _ n:num _ { return n; }
4      / _ c:cc _ { return c; }
5      / _ l:label _ { return l; }
6
7
8  register
9      = _ "r" n:([0-9a-z_]i)+ _ {
10         return new ILOC.Operand({
11             type: ILOC.OPERAND_TYPES.register,
12             name: n.join("")
13         });
14     }
```

Listing 5.3 shows a snippet of the PEG.js grammar for ILOC. Groups of symbols may be assigned an identifier and referred to in the JavaScript code; on line 9 the identifier n is given to the register name so it can be set as the name of the returned Operand. The return value of the JavaScript code is passed to the above context; the identifier r on line 2 refers to the Operand returned by the register rule. More complex code may collect information and use it to annotate the AST; for example, the grammar used in this project annotates assignments to a given register with a unique identifier for use in data-flows such as reaching definitions.

Fig. 5.2 shows an example AST for the simple ILOC program in listing 5.4.

Listing 5.4: Simple ILOC Program

```
1  Start: addI     ra, 1   => rb
2         comp     ra, rb => cc
3         cbr_LE   cc      -> Start, Store
4  Store: storeAO ra       => rx, 32
```
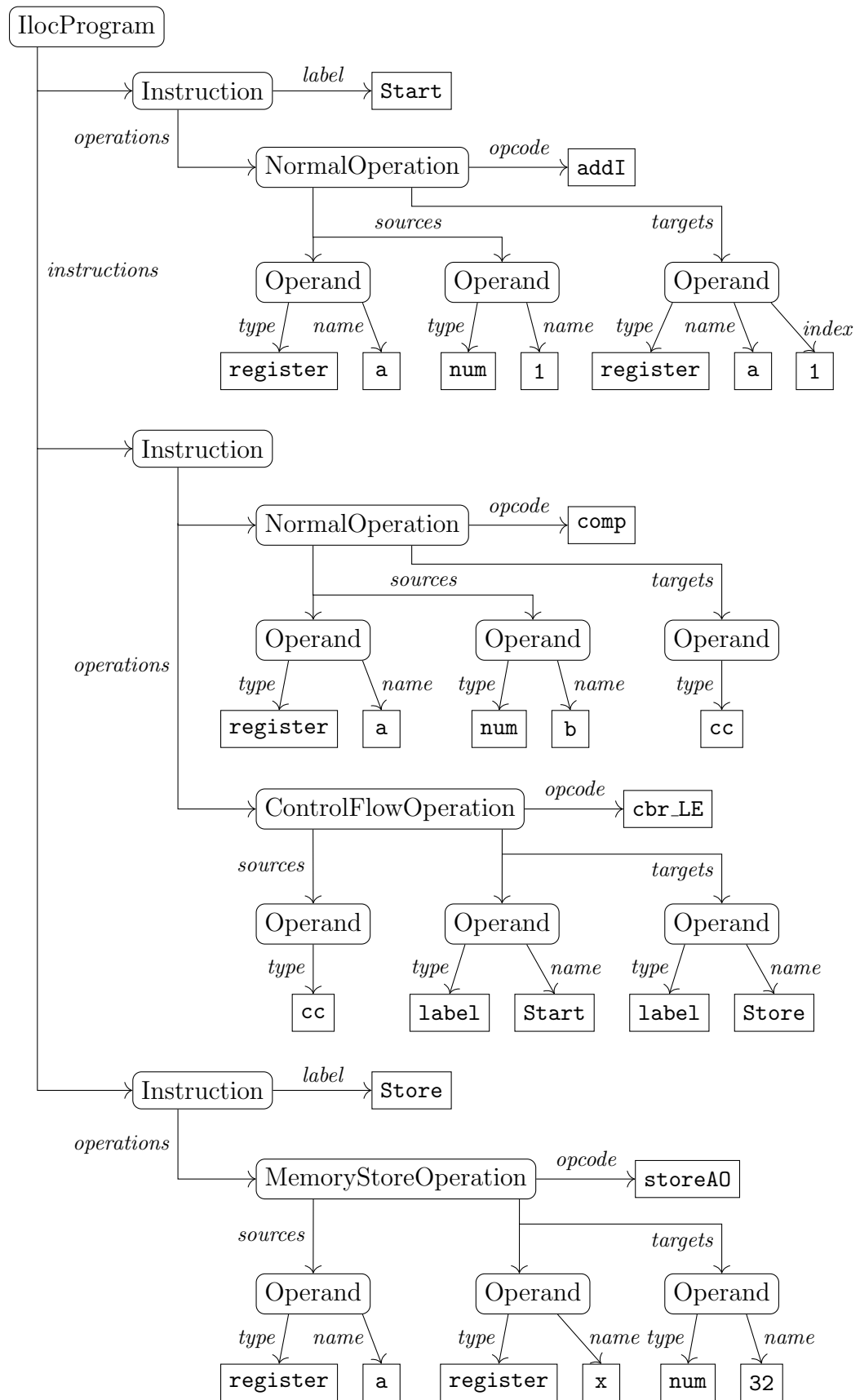
Fig. 5.2: The AST for the ILOC program in fig. 5.4.

5.5.2   Control-Flow Graph

The simulation constructs a CFG from the AST. Internally, the edges of the CFG are represented using an adjacency matrix. As the input is expected to be small the choice of internal representation has minimal effect on the efficiency of the simulation. If larger input was expected then a strong case could be made for switching to a different structure such as an adjacency list in order to reduce memory requirements; however, the CFG shares much of its underlying code with the Hasse diagrams (see §5.5.3) for which an adjacency matrix is better suited.

Algorithm 5.2 describes how the CFG is constructed from the AST. The algorithm first adds all of the instructions to the graph, identifying which labels refer to which instructions and storing this information in a map. Next, the algorithm populates the adjacency matrix. If the node is a ControlFlowOperation an edge to the target instructions (looked up using the label map), otherwise we add an edge to the next instruction in the sequence.

Algorithm 5.2: Constructing a CFG from an AST for an ILOC program.

```
1   labels = Map (Label → Instruction)
2
3   for each (instruction in the AST)
4       add instruction to the CFG as a node
5       if (instruction has label)
6           add (label → instruction) to labels
7
8   for each (node in the CFG)
9       if (node is a ControlFlowOperation)
10          for each (target in node)
11              add an edge from node to labels[target]
12      else
13          add an edge from node to next node
```

5.5.3   Lattices

The simulation uses the CFG to construct the meet semi-lattice for a given data-flow framework. The lattices in this simulation only provide support for bit-vector data-flows since these are the only type included with the system (see §**??**).

Due to this constraint, the process of generating the lattice is quite simple. The set of all combinations of values is a powerset, and our lattice must include each one. This is easy to see when we consider the meaning of the term bit-vector: the number of possible sets represented by n bits is 2^n.

First, the domain of values is identified. Then all possible sets of values must be generated, the code for which was taken from a snippet online[14]. Each value set becomes a node in the lattice. Next, the algorithm finds the meet of every pair of sets; an edge is added between those sets and the result of the meet operation. Finally, a transitive reduction is performed on the graph: if an edge exists from $x \rightarrow y$ and from $y \rightarrow z$, any edge from $x \rightarrow z$ may be removed as it is represented

by the path $x \to y \to z$. This constructs a Hasse diagram representing the meet semi-lattice, as described in algorithm 5.3.

Algorithm 5.3: Constructing a Hasse diagram for the meet semi-lattice of a CFG.

```
1   values = collect all values from the CFG
2   sets = generate all possible subsets of values
3
4   graph = Graph
5
6   for each (set in sets):
7       add set to graph as a node
8
9   // Find all the edges between the nodes
10  for i from 0 to length(sets) − 1
11      for j from i + 1 to length(sets)
12          temp = meet(sets[i], sets[j])
13          add edge to graph from sets[i] to temp
14          add edge to graph from sets[j] to temp
15
16  // Perform a transitive reduction of the graph
17  for i from 0 to length(sets)
18      for j from 0 to length(sets)
19          if (there exists an edge i → j)
20              for k from 0 to length(sets)
21                  if (there exists an edge j → k)
22                      remove the edge i → k
```

The simulator only generates a lattice for programs with a small set of values due to the exponential growth of the size of the lattice. The graph is backed by an adjacency matrix since this structure has constant-time operations for adding and removing edges, resulting in a huge performance increase over other structures such as an adjacency list.

Chapter 6

Front-End Implementation

This chapter discusses the implementation of the front-end elements which make up the user interface, including the visualisation and interactive teaching components.

6.1 Overview

The implementation of the user interface components uses a modular design. Each element is a `View`, a component which renders content to a HTML element referred to as that component's canvas. An inheritance model is used so that functionality is shared between `View`s, for example, each of the visual components inherits from `SimulatorView`, which handles registering callback functions with a simulator's event handler.

The front-end is implemented using JavaScript, HTML and CSS. A number of libraries provide additional functionality, including:

- d3.js and the extension dagre-d3 to draw directed graphs;

- jQuery and extensions such as tipsy for navigation and visual elements;

- MathJax for rendering of LaTeX math formulae;

- Handlebars.js for HTML templating; and

- Bootstrap v4 alpha for page layout and theming.

6.2 View Model

Each user interface component is implemented as its own class, or `View`. A `View` component renders HTML content inside another HTML element referred to as its canvas. Multiple `View`s may exist at one time, and a `View` may contain other views.

Beside from the individual view classes, there are multiple types of `View`; for example, all of the visual components inherit from the `SimulatorView` class. Each `SimulatorView` is required to implement callback functions for the `update` and `reset` events. These functions are called by the simulator's event handler when an event is triggered, which may cause the component to be re-rendered or the information contained within to be modified.

In addition to specifying the interface for various classes, the `View` model allows functionality to be shared between components. The `TutorialView` class implements navigation through and display an interactive tutorial, so that each one need only implement functions to control the content shown at each step.

This model is a very simple implementation of what is known as a single page app: a web application in which the page is only loaded once, after which JavaScript is used to update the page content and asynchronous calls request more data from the server. Some existing libraries are based around this concept; for example AngularJS[15] provides a similar framework for creating single page apps using re-usable components.

6.3   Visual Components

This section describes the implementation of the components which visualise the simulation, including the choice of any libraries and frameworks and the changes that were made from the original design.

6.3.1   Control-Flow Graphs

Control-flow graphs are rendered using the dagre-d3 extension for the graph visualisation library d3.js. An SVG component is created and the nodes and edges added to the dagre-d3 graph. The layout of the graph is handled entirely by the graphing library.

The dagre-d3 library was chosen due to the strength of the demonstrations on the project's website[16]. The available examples showed graphs very similar to the desired result, with only a small set of API calls required to draw a simple control-flow graph. Given more time, perhaps using a more fully-featured graphing library such as Cytoscape.js[17] or vis.js[18] would improve the aesthetics and performance of the component. However, dagre-d3 is more than adequate for a proof of concept.

To reduce clutter in the graph, the user has three options for displaying nodes: *current*, in which only the points relating to the current step are shown; *all*, in which every point is displayed; or *none*, in which only the nodes of the CFG are shown. Fig. 6.1 shows the CFG in the *current* setting.
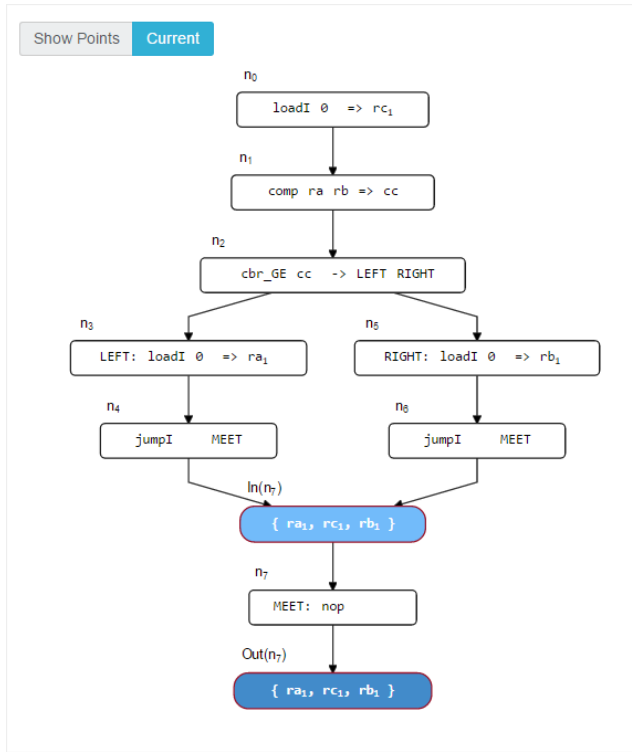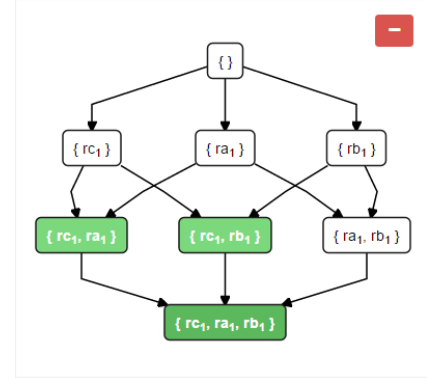
Fig. 6.1: Control-Flow Graph Visualisation

Fig. 6.2: Hasse Diagram Visualisation

A system was devised to allow dynamic insertion and removal of points from the CFG. In addition to automatically displaying nodes based on the settings described above, the component's API can be used to add or remove specific points. This allows the tutorial examples to display the exact information required rather than relying on the simulation to provide the correct results.

As the user steps through a simulation, nodes and points in the CFG are highlighted to visually relate them to actions occurring in other components. This is shown by fig. 6.1; the transfer function is reading information, shown in light blue, from $In(n_7)$ and modifying $Out(n_7)$, shown in dark blue.

6.3.2 Hasse Diagrams

The Hasse diagram component is implemented using much the same system as the control-flow graphs. An SVG component is created, the nodes and edges are added to a dagre-d3 graph and the result is rendered to the screen.

As the user steps through a simulation, nodes in the diagram are highlighted to visually relate them to actions occurring in other components – during the meet phase, the sets being considered are highlighted in light green and the resulting set in dark green.

6.3.3   Table of Results

The table of results, which displays the set of values at each point, is simply a HTML table containing the required information. Various layouts were considered for this table, with the final design shown in fig. 6.3.

Instruction	Local Information		Round	Global Information			
	defgen	defkill		1		2	
			Set	In	Out	In	Out
0	{ ra_1 }	{ ra_1 }		{}	{ ra_1 }	{}	{ ra_1 }
1	{ rb_1 }	{ rb_2, rb_1 }		{}	{ rb_1 }	{}	{ rb_1 }
2	{ rc_1 }	{ rc_2, rc_1 }		{}	{ rc_1 }	{}	{ rc_1 }
3	{ rc_2 }	{ rc_1, rc_2 }		{}	{ rc_2 }	{}	{ rc_2 }
4	{ rb_2 }	{ rb_1, rb_2 }		{}	{ rb_2 }	{ rc_2 }	{ rb_2 }
5	{ rd_1 }	{ rd_1 }		{}	{ rd_1 }	{ rb_2 }	{ rd_1, rb_2 }
6	{}	{}		{}	{}	{ rd_1 }	{ rd_1 }
7	{}	{}		{}	{}	{}	{}

Fig. 6.3: Current Table Layout

This design could be improved by changing the layout of the table when viewed on small screen sizes. In the current layout the entire table becomes scrollable if it is too large for its canvas. Perhaps changing the table so that only the rounds become scrollable (see fig. 6.4) would improve its readability; however, the amount of time required to implement this was deemed better spent on implementing other functionality as the cases in which it is required are uncommon.

Instruction	Local Information		Round	3		2		1	
	defgen	defkill	Set	In	Out	In	Out	In	Out
0	{ ra_1 }	{ ra_1 }		{}	{ ra_1 }	{}	{ ra_1 }	{}	{ ra_1 }
1	{ rb_1 }	{ rb_2, rb_1 }		{ ra_1 }	{ rb_1, ra_1 }	{ ra_1 }	{ rb_1, ra_1 }	{}	{ rb_1 }
2	{ rc_1 }	{ rc_2, rc_1 }		{ rb_1, ra_1 }	{ rc_1, rb_1, ra_1 }	{ rb_1 }	{ rc_1, rb_1 }	{}	{ rc_1 }
3	{ rc_2 }	{ rc_1, rc_2 }		{ rc_1, rb_1, rd_1, rb_2 }	{ rc_2, rb_1, rd_1, rb_2 }	{ rc_1, rd_1 }	{ rc_2, rd_1 }	{}	{ rc_2 }
4	{ rb_2 }	{ rb_1, rb_2 }		{ rc_2, rd_1 }	{ rb_2, rc_2, rd_1 }	{ rc_2 }	{ rb_2, rc_2 }	{}	{ rb_2 }
5	{ rd_1 }	{ rd_1 }		{ rb_2, rc_2 }	{ rd_1, rb_2, rc_2 }	{ rb_2 }	{ rd_1, rb_2 }	{}	{ rd_1 }
6	{}	{}		{ rd_1, rb_2 }	{ rd_1, rb_2 }	{ rd_1 }	{ rd_1 }	{}	{}
7	{}	{}		{ rd_1 }	{ rd_1 }	{}	{}	{}	{}

Fig. 6.4: Alternate Table Layout with Scrollable Rounds

6.3.4   Code Display

The code display handles displaying ILOC programs, with visual links to other components sharing the simulation. This is handled by a simple HTML table, as each ILOC instruction has a maximum of 7 fields: label, opcode, two sources, the → or ⇒ symbol, and two targets. Aligning each token in the instruction allows the code to be read much more easily than if the raw text were displayed on screen. Fig. 6.5 shows the code display component.

Fig. 6.5: Default Code Display
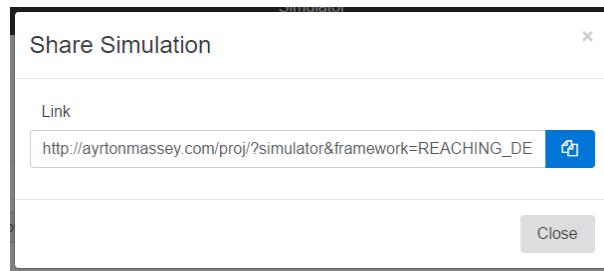

Fig. 6.6: Editing Code in the Simulator

Fig. 6.7: Share Code Dialog

In the simulator interface this view also handles editing or sharing code with others. Figures 6.6 & 6.7 show this in action. If the user enters invalid code an error message is displayed using line and column information obtained from PEGjs. Line numbers are displayed in the text area to help the user identify the source of the error; this functionality is provided by the jQuery Lined TextArea[19] plugin. The additional whitespace is added by the simulator to improve readability, but is not necessary.

6.4 Interactive Components

This section describes the implementation of the interactive components which allow the user to interact with the system, including the choice of any libraries and frameworks and the changes that were made from the original design.

6.4.1 Simulator Controls

The simulator controls simply activate the playback functions in the simulator through its API. The controls can be embedded in any other view, so whilst their main use is in the simulator interface it is also possible to include them in interactive tutorials or tests.

Fig. 6.8: Simulator
Controls

6.4.2   Questions

The QuestionView class was designed to be adapted to as many situations as possible. Each question displays a number of possible answer, and has a variety of configuration options:

- The order in which answers should be presented (or shuffled).

- Whether a question allows multiple answer choices.

- Which answers should be revealed upon selecting an answer, if any.

- Which answers should be disabled upon selecting an answer, if any.

The question and its answers are rendered using the MathJax library to display mathematical formulae. Submitting a question reveals the nature of each answer (correct or incorrect) and marks those selected by the user. Answers may be assigned flavour text which is displayed upon selection to provide the user with instant, detailed feedback. Two example questions are shown in figures 6.9 & 6.10
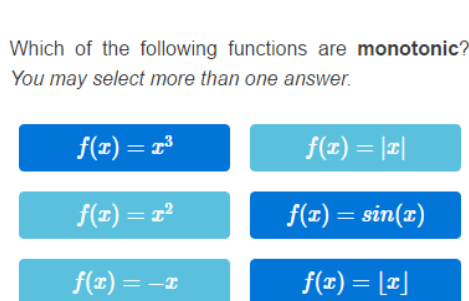

Which of the following functions are **monotonic**?
You may select more than one answer.

$f(x) = x^3$ $f(x) = |x|$

$f(x) = x^2$ $f(x) = sin(x)$

$f(x) = -x$ $f(x) = \lfloor x \rfloor$

The equation for $\text{In}(n)$ is:

$$\text{In}(n) = \bigcup_{p \in preds} \text{Out}(p)$$

What is the value of $\text{In}(n_0)$?

$\{\}$ ✓ $\{rx_1\}$

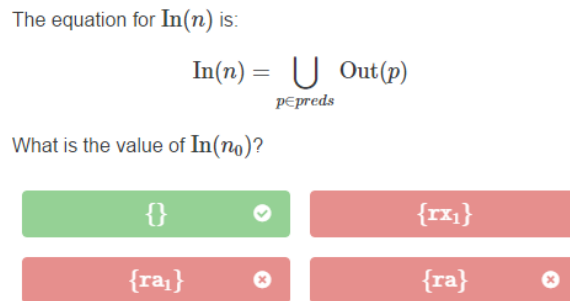$\{ra_1\}$ ✗ $\{ra\}$ ✗

Fig. 6.9: A multiple-choice
QuestionView.

Fig. 6.10: A QuestionView after it has
been submitted.

The highly configurable nature of the questions allows them to be re-used across the app. In TutorialViews the answers are revealed immediately, whereas in TestViews the test must be submitted before the user is given feedback. QuestionViews are also used in the evaluation to collect categorization information from the user.

Two callback functions can be passed to the QuestionView to control the application's behaviour upon selecting an answer. This function can perform such actions as allowing the user to proceed by enabling navigation buttons, or altering neighbouring components to provide visual feedback.

6.5 User Interface

This section discusses the implemented user interface and describes any changes made from the proposed design. These changes and the use of any libraries and frameworks are justified.

6.5.1 Menus

6.5.2 Simulation Interface

6.5.3 Tutorial Interface

6.5.4 Testing Interface

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.

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

[4] Coursera. About Coursera. `https://www.coursera.org/about/`, 2015. Accessed: 2015-09-20.

[5] HackerRank. Reverse a linked list: Challenges. `https://www.hackerrank.com/challenges/reverse-a-linked-list/`, 2015. Accessed: 2015-09-21.

[6] Khan Academy. Implementing binary search of an array. `https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/implementing-binary-search-of-an-array`, 2015. Accessed: 2015-09-21.

[7] B. Mustafa. Evaluating A System Simulator For Computer Architecture Teaching And Learning Support. *Innovation in Teaching and Learning in Information and Computer Sciences*, 9(1):100–104, 2010.

[8] R. M. Felder and L. K. Silverman. Learning and Teaching Styles In Engineering Education. *Engineering Education*, 78(7):674–681, 1988.

[9] B.J. Farr. *Individual Differences in Learning: Predicting One's More Effective Learning Modality*. PhD thesis, Catholic University of America, 1970.

[10] R. S. Dunn and K. J. Dunn. Learning Styles / Teaching Styles: Should They... Can They... Be Matched? *Educational Leadership*, 36:238 – 244, 1979.

[11] F. Coffield, D. Moseley, E. Hall, and K. Ecclestone. *Learning styles and pedagogy in post-16 learning*. The Learning and Skills Research Centre, 2004.

[12] M. D. Merrill. Instructional Strategies and Learning Styles: Which takes Precedence? *Trends and Issues in Instructional Technology*, 2000.

[13] David Majda. PEG.js – Parser Generator for JavaScript. `http://pegjs.org/`, 2016. Accessed: 2016-03-25.

[14] Troels Knak-Nielsen. Calculate All Combinations (permutations). `https://dzone.com/articles/calculate-all-combinations`, 2016. Accessed: 2016-03-25.

[15] Google Inc. AngularJS – Superheroic JavaScript MVW Framework. `https://angularjs.org/`, 2016. Accessed: 2016-03-26.

[16] Chris Pettitt. cpettitt/dagre-d3 Wiki. `https://github.com/cpettitt/dagre-d3/wiki#demos`, 2016. Accessed: 2016-03-26.

[17] Cytoscape Consortium. Cytoscape.js. `http://js.cytoscape.org/`, 2015. Accessed: 2016-03-26.

[18] Almende B.V. vis.js. `http://visjs.org/`, 2016. Accessed: 2016-03-26.

[19] Alan Williamson. jQuery Lined TextArea. `http://alan.blog-city.com/jquerylinedtextarea.htm`, 2016. Accessed: 2016-03-26.

[20] Wikipedia. Context-free grammar. `https://en.wikipedia.org/wiki/Context-free_grammar`, 2016. Accessed: 2016-03-25.

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, 43

bit-vector data-flow Data-flows in which values come as single items, and are either included or excluded in each set. Such data-flows may be represented by a vector of bits, each bit representing a value and a 0 or 1 indicating the presence or absence of that value in the set.. 7, 32

boundary The initial value at the starting point of an analysis, i.e. $\text{In}(ENTRY)$ or $\text{Out}(EXIT)$. 9, 26

context-free grammar A context-free grammar describes a formal language using rules of the form $A \rightarrow \alpha$, in which A is a single non-terminal symbol and α is a series of terminal or non-terminal symbols. The grammar is "context-free" in that its rules may be applied regardless of the context of the non-terminal A[20]. 28

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, 41

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, 6–10, 25, 30, 32, 41

data-flow analysis A technique for gathering information at various points in a control-flow graph. i, 1, 2, 5, 6, 8, 13, 17, 19–21, 25, 28

data-flow equations A system of equations which determine how data flows through a CFG. 8–10, 26

47

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, 26

domain The domain of values considered in a data-flow problem, e.g. definitions or expressions. 9, 11, 26

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]. 42, 43

fixed-point A computation in which the required process is repeated until the state stops changing. 8, 10, 25

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, 8

Likert scale A method of gauging opinion on a topic, consisting of a collection of Likert items. Each Likert item contains a statement followed by an odd number of responses, containing an equal number of positive and negative responses balanced such that the difference between responses is uniform. An example Likert item is the oft-used Strongly Agree / Strongly Disagree scale, which may be weighted from from 1-5. The Likert scale is the sum of weights of responses to Likert items. 15, 17

liveness analysis A variable is *live* if its current value will be used later in the program's execution. 1, 43

local information Information specific to a given node in the CFG, e.g. in reaching definitions DefGen is the set of definitions which a given node generates.. 6

meet An equation (or set of equations) which determines how data flows *between* nodes in a CFG. 6, 10, 26

meet operator An operator which defines how the meet of two sets is obtained, such as $\cup$, $\cap$ or another operator entirely. 9, 11, 26

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, 11

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. 6–9, 30, 43

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, 26, 27

tuple-valued data-flow Data-flows in which the values come in tuples; for example, in constant propagation each variable is paired with a value indicating whether it holds a constant.. 7

A.2   Acronyms

API application programming interface. 22, 25

AST abstract syntax tree. 25, 28, 30–32

CFG control-flow graph. 6–10, 25, 26, 28, 32, 41, 42

COPT Compiler Optimisations. 2, 19, 21

ILOC Intermediate Language for Optimising Compilers. 3, 25, 28–32

PEG Parsing Expression Grammar. 28, 29

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.