

Compiler Support for Learning Program Invariants

**A Thesis Presented in Fulfillment
of the Requirements for the Degree of
Master of Research**

Michael Lay

B.IT, Macquarie University, 2018



School of Computing
Faculty of Science & Engineering
Macquarie University, NSW 2109, Australia

Submitted February 2023

©Michael Lay 2023

Declaration

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

This research has received ethics approval (Project ID: 11915) from the Science & Engineering Ethics Subcommittee and meets the requirements set out in the National Statement on Ethical Conduct in Human Research. All steps have been taken to ensure that the research adheres to these guidelines.

Signed:

Date:

Dedication

To finishing this Masters.

Acknowledgements

My supervisors Matthew Roberts and Matt Bower for all the advice and support they have given, without which this thesis would not have been completed. My partner for supporting me in writing this thesis.

Abstract

Formal methods are a skill in high demand for programmers in safety-critical industries. However, the teaching and learning of formal methods is not well understood. We perform a controlled trial of two possible approaches to teaching one fundamental skill from formal methods, and analyse the differences in effectiveness. The fundamental skill is the understanding and creation of program invariants. We compare the traditional method of teaching invariants, which we determined by extensive survey, to a compiler-supported technique. We find no conclusive evidence of a difference in learning speed or depth, however, we do find indications that compiler support is valuable for learners. We provide an in-depth analysis of our data and all material required to build on it. The results indicate a need for larger trials.

Contents

Declaration	iii
Dedication	v
Acknowledgements	vii
Abstract	ix
List of Figures	xvi
List of Tables	xvii
1 Introduction	1
2 Literature Review	5
2.1 Invariant-Based Programming	5
2.2 Tools supporting invariants and compiler feedback	6
2.3 Teaching Invariants	8
2.4 Required Knowledge	9
2.5 Experimental Design	12
2.6 Financial Incentives	15
3 Methods	17
4 Participant Recruitment	21
4.1 Recruitment Sources	22
4.2 Calculating participants required	24

5	Workshop	27
5.1	Minimum requirements	28
5.2	Pre/Post-Learning Test Design	28
5.3	Dafny	31
5.4	Workshop content	32
5.5	Teaching technique	34
6	Participant Analysis	35
7	Results	39
7.1	Workshop Results	39
7.2	Post Questionnaire	42
7.2.1	Likert-Scale Questions	44
7.2.2	Short-Response Prompts	47
8	Discussion	53
8.0.1	Post Questionnaire	54
8.0.2	Short-Response Prompts	54
8.0.3	Confounding Factors	55
9	Conclusion	59
9.1	Future work	61
A	Data Tables	63
B	Participant Consent Form	67
C	Posters	73
D	Learning Tests	75
D.1	Pre-Learning Test	75

D.2	Post-Learning Test	82
E	Workshop Questions	89
E.1	Traditional group	89
E.2	Compiler group	92
F	Ethics Approval Letter	97

List of Figures

2.1	An example Why3 proof using the built in GUI	7
2.2	An example OpenJML annotated program using eclipse	7
2.3	An example Dafny program using the VSCode IDE	8
4.1	Participant prize selection choices	22
4.2	Sources of participant recruitment	23
4.3	Participant registrations by poster version	24
5.1	Example of a pre-learning test question. This question fits into the “Selecting a correct loop invariant” category.	30
5.2	An example of a Dafny function that reverses a given array	32
6.1	Number of registered and attended participants for each workshop session. Weekday sessions in blue, weekend sessions in orange. . . .	36
6.2	Number of participants in each skill level	37
6.3	Participants’ self rating of their confidence in writing correct programs	38
6.4	Confidence in writing correct programs for each skill level	38
7.1	Average difference in participant scores across topics by experimental group	40
7.2	Number of participants in each skill level by experimental group . . .	43
7.3	Responses to "I am confident writing correct programs", asked before and after the workshop	44
7.4	Responses to "I am confident writing correct programs" split by ex- perimental group	45

7.5	Graphed responses to the Post Questionnaire engagement questions by experimental group	47
7.6	Graphed responses to the Post Questionnaire understanding questions by group	48

List of Tables

2.1	Summary of several papers on teaching invariants and their evidence for effectiveness	9
2.2	Formal methods courses across Australia and their pre/co-requisites .	11
7.1	Average difference in participant scores, standard deviation and Hedges G by topics and experimental group	40
7.2	Variance results by topics and experimental group	42
7.3	Average difference between experimental groups and variance results grouped by skill level	43
7.4	Responses to the Post Questionnaire engagement questions by experimental group	46
7.5	Responses to the Post Questionnaire understanding questions by group	46
7.6	Number of quotes per theme identified	49
A.1	Participants' learning test averages grouped by total and skill levels .	64
A.2	Participant responses to Likert scale questions	65

Introduction

Formal verification is a powerful method that provides a strong guarantee of correctness in the programs we write. Although there are many benefits to the technique, widespread adoption has been lacking outside of safety critical projects such as space flight or air travel. Several reasons can be attributed to its slow adoption such as having a high barrier to entry [1], less usable tools, and greater overhead. Different types of formal methods are taught at a tertiary level. Some examples include pre and post conditions, invariants, theorem provers and model checking. Each of these require varied amounts of requisite knowledge before being taught, with theorem proving and model checking often requiring substantial background knowledge.

Surveying the area of computer science education in formal verification reveals a lack of rigorous methodology employed and quantifiable data that can be compared across studies. The current landscape predominately involves studies that analyse qualitative student feedback from course evaluations or analysing trends in course results across multiple offerings. This is unsurprising due to factors such as students unfavourable views towards formal verification, often viewing it as mathematically focused [2], and the restrictive rules in place for human research on tertiary course students in Australia.

In addition to this, tool support for formal verification is growing more sophisticated

and user friendly. Where formally verifying software was once a specialist position requiring years of experience, the tools might now be usable by a larger set of programmers and IT professionals. Research into student use of these new tools is ongoing but often lacks the rigorous methodology to definitively prove any broad claims.

The hypothesis for this study is that adding compiler assisted support for formal verification can improve student success when learning to generate and validate *invariants* for the types of programs studied. Specifically, we focus on invariants which have been shown to be a problem area in student understanding [3]. A programming invariant is a set of assertions which must remain true during the execution of a program for the given program to remain valid. This ensures that the program meets its predefined constraints and goals. Providing compiler support for generating invariants will be approached using a compiler tool that provide automatic feedback about the validity of invariants specified. Our research will utilise an accepted experimental design that can produce quantifiable data on student learning of the topic. With this data we will perform statistical analysis to determine the effects of compiler support on student learning of invariants. For this research “Learning” of invariants is defined as the ability to generate and validate invariants. This research could inform how invariants and formal verification is taught, by supporting the use of modern tools for improving student learning.

Specifically we aim to address the following research questions:

- What effects does compiler assisted learning using Dafny have on student learning of invariants compared to traditional methods?
- Does compiler assisted learning using Dafny promote further engagement compared with traditional methods?

The main contributions of this thesis are:

- Performing a controlled trial to evaluate the effectiveness of two approaches to teaching invariants, i.e. a fundamental skill in formal methods
- Surveying the current landscape for formal methods education at the tertiary level in Australia
- Developing an assessment instrument to gauge student learning of invariants
- Providing the required material to build on the research

The thesis is organised into the following chapters:

- **Chapter 1** - Presents an introduction to the problem, motivation for the research, the research questions addressed and the contributions made.
- **Chapter 2** - Provides a literature review of the current research into invariant-based programming, the current tools, the landscape for formal methods education in Australia, an overview of experimental design and, financial incentives for motivating participants.
- **Chapter 3** - Discusses the methods used in the research including the assessment instrument and motivations behind the chosen experimental design.
- **Chapter 4** - Presents an overview of the participant recruitment strategy, sources of recruitment, and an analysis of the number of participants needed for the study.
- **Chapter 5** - Provides motivations and details about the research workshop conducted including the minimum requirements to attend, an overview of the assessment instrument, the tools used, and the workshop content.

- **Chapter 6** - Presents an analysis of the participants who registered and attended the workshop. Details on how participants are divided and analysed as subgroups are also provided.
- **Chapter 7** - Presents the results from the learning assessment instrument and qualitative feedback given by participants.
- **Chapter 8** - Discussion of results.
- **Chapter 9** - Conclusion, summary and future work.

Literature Review

2.1 Invariant-Based Programming

Invariant-Based Programming is an approach where the programmer formulates the specifications of the program and generating invariants for a program before writing the program code. This method aims to produce software that is correct by construction. [Back \[4\]](#) uses a nested invariant diagram starting with figures illustrating the data structures involved in a selection sort. The pre and post conditions of the selection sort are considered to determine the initial and final program states. The program flow is then represented by the nested invariant diagrams. The nested invariant diagram is appended with additional blocks as intermediate situations are encountered with their respective invariants. A number of studies have investigated the effectiveness of the technique, several of which we summarise below.

[Mannila \[3\]](#) analyses the errors that novice programmers make in developing invariant based programs, specifically presenting the general types of errors students make when expressing invariants. They conclude that invariant-based programming is a suitable approach to teaching programming from an early stage of computer science study without being "too advanced". [Kabbani et al. \[5\]](#) take a similar approach and use a self built web IDE to demonstrate the iterative process of creating and refining loop invariants. They find that using an iterative approach for developing and debugging a system from a specification, students can develop

provably correct software. In addition to generating a correct program through the use of invariants, [Back](#) [6] proposes using a nested invariant diagram. An invariant diagram describes invariants as sets and program code as transitions between sets. [Back](#) argues the systematic use of these figures makes it straightforward to formulate the invariants needed for a program. [Back](#) [7] extends the research done in [6] by exploring student difficulties when constructing invariant based programs and the benefits of the proposed figures as a tool for identifying situation constraints.

2.2 Tools supporting invariants and compiler feedback

Several tools exist supporting invariant specification and compiler feedback. Some of these tools include Why3, OpenJML and Dafny.

Why3 is a platform used for deductive program verification, providing a rich language for specifying requirements and programming. Why3 has been used in several papers such as work by [Blazy](#) [8] as a tool to initiate students into formal methods. Why3 relies on external theorem provers to provide both an automated, and when not possible, interactive method for verifying conditions. An automated program extraction tool also provides correct by construction OCaml programs. The language is supported through a web IDE and a VSCode extension.

OpenJML is a tool for the verification of Java programs, allowing users to check the correctness of specifications annotated using the Java Modelling Language. OpenJML has been used in a number of courses teaching formal specification such as work by [Cataño and Rueda](#) [9] and [Poll](#) [10] where JML is used as the tool to teach a formal methods course, and work by [Divasón and Romero](#) [11] where Krakatoa which supports OpenJML is used to complement theoretical lessons. OpenJML works similarly to Why3 using deductive reasoning to verify a specification. The

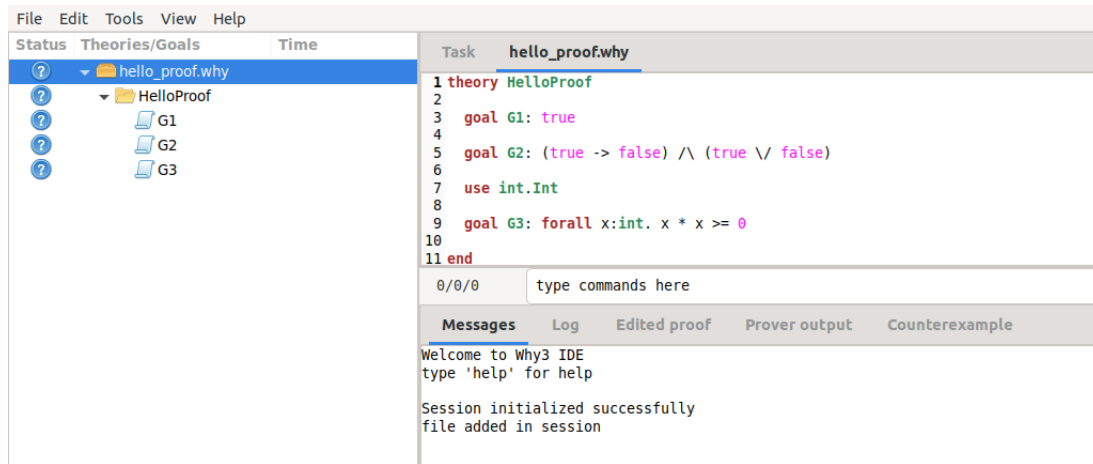


Figure 2.1: An example Why3 proof using the built in GUI

primary differentiating factor of OpenJML is its focus on the Java language and ease of use for practitioners and students. OpenJML is supported in Krakatoa [12] which provides a front-end. An Eclipse plugin is available but is outdated and does not currently work with the latest version of Eclipse.

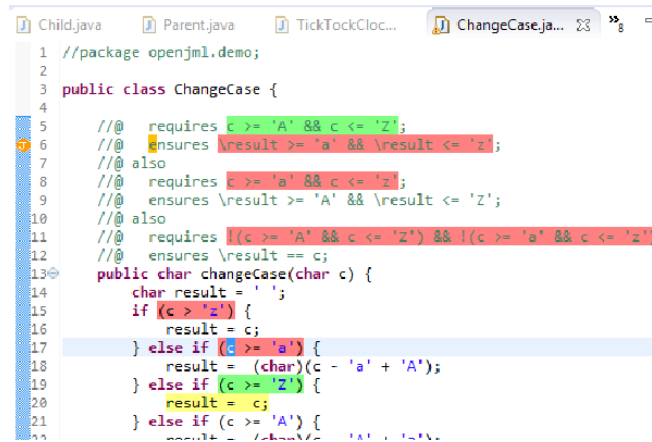


Figure 2.2: An example OpenJML annotated program using eclipse

Dafny [13] is a programming language that is “verification ready”. Its main feature is its verifier, which reveals errors as a program is written, providing counterexamples both inline within the code and in the output console. Dafny also offers code generation, being able to compile Dafny code into a variety of languages such as C#,

Java and JavaScript. Dafny has been used in a number of papers related to formal methods education such as work by Ettinger [14] where Dafny is used to teach formal specification using “small steps of refinement” wherein a stepwise approach for turning specifications into code is explored. Dafny is supported by both a web IDE and a VSCode extension.

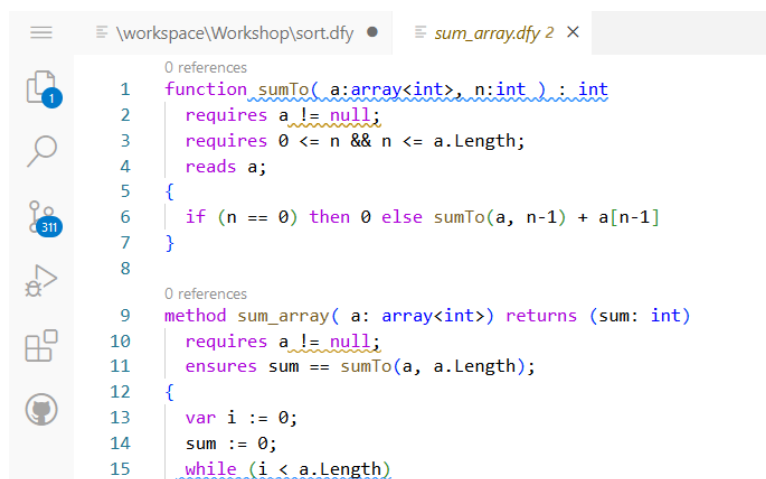


Figure 2.3: An example Dafny program using the VSCode IDE

All three tools utilise a variety of backend theorem provers which allow verification of programs. Some of these backend theorem provers include z3 [15], and CVC4 [16]. Both OpenJML and Dafny use z3 by default with Why3 using CVC4. These two provers provide the capability for the tools to generate counterexamples from falsifiable assertions, providing a powerful source of information for learners.

2.3 Teaching Invariants

A number of methods are used to teach programming invariants ranging from traditional methods, using worked board examples to computer aided tools used to both help learn and generate invariants. Surveying the literature, we see that these are the two primary categories that invariant education techniques fall into. We summarise several papers in Table 2.1 listing their approach and evidence of

its effectiveness. Where there are multiple sources of evidence for an approaches claim of effectiveness, we have listed the most quantitative.

Paper	Tool assistance	Evidence	Sample
Teaching loop invariants to beginners by example [17]	No	None	-
Invariant based programming: basic approach and teaching experiences [7]	No	Student feedback	Unknown
Teaching Programming to Liberal Arts Students: Using Loop Invariants [18]	No	None	-
Inculcating invariants in introductory courses [19]	Yes	Assignment results	15
Tool-supported Invariant-based programming [20]	Yes	Assignment results	7

Table 2.1: Summary of several papers on teaching invariants and their evidence for effectiveness

There is a lack of strong evidence that supports each approaches effectiveness, with few offering any supporting quantitative data. Due to this lack of quantitative data, it is difficult to compare the effectiveness of each approach across different teaching techniques. In addition, the small sample of each work also leads to more skeptical results.

2.4 Required Knowledge

Learning formal verification is often approached through several avenues such as tertiary education, online courses or self learning. Each of these paths require some previous knowledge often varying from institution to institution. In this section, we provide an overview and analysis of the current knowledge required at various Australian institutions before studying formal verification.

To produce a consistent analysis of required knowledge, we categorise the various pre-requisites and co-requisites into groups based on the ACM Computer Science Curricula 2013 [21]. The ACM Curricula groups content into well defined subject areas such as placing “Basic Analysis”, “Algorithmic Strategies”, “Fundamental Data Structures and Algorithms” and “Advanced Computational Complexity” into the “Algorithms and Complexity” group. As each pre-requisite and co-requisite course contains topics from a large number of categories, we categorise using the overall groupings as opposed to finely categorising all the required knowledge. This means

that instead of specifying “Object Oriented Programming”, “Syntax Analysis” or “Functional Programming” for each unit, we group all of these under the larger group of “Programming Languages”.

Institution	Formal Verification Course		Level	Required Knowledge			
	Unit Code	Unit Name		Unit Code	Unit Name	Type	Mapping
UNSW	COMP3153 [22]	Algorithmic Verification	Undergraduate	COMP1927	Computing 2	Prerequisite	ADT, PL
UNSW	COMP6721 [23]	(In-)Formal Methods: The Lost Art	Undergraduate	COMP2521	Data Structures and Algorithms	Prerequisite	DS, PL, SDF, AL
MQ	COMP4000 [24]	Formal Methods	Undergraduate	MATH1081	Discrete Mathematics	Prerequisite	AD/TL
MQ	COMP7010 [25]	Advanced Topics in Theory and Practice of Software	Postgraduate	COMP2521	Data Structures and Algorithms	Prerequisite	DMTH
ANU	COMP2600 [26]	Formal Methods in Software Engineering	Undergraduate	COMP3000	Programming Languages	Prerequisite	ADT, PL
				COMP3010	Algorithm Theory and Design	Prerequisite	ALG
					Maturity		DS, SDF, AL
				COMP1110	Introduction to Software Systems	Prerequisite	FP
				COMP1510	Introduction to Software Engineering	Prerequisite	DMTH
				MATH1005	Discrete Mathematical Models	Prerequisite	DS
				MATH1014	Mathematics and Applications 2	Prerequisite	DS
					24cp @ 300 level		
					12 cp @ 900 level		
UOW	CSCI410 [27]	Software Requirements, Specifications and Formal Methods	Undergraduate	SENG2130	Systems Analysis and Design	Prerequisite	SE
UOW	CSC910 [28]	Software Requirements, Specifications and Formal Methods	Postgraduate	SENG6350	Systems Analysis and Design	Prerequisite	SE
UON	SENG3320 [29]	Software Verification and Validation	Undergraduate	SOFT2201	Software Construction and Design 1	Prerequisite	SE, PL
USYD	SOFT3202 [31]	Software Verification and Validation	Undergraduate	MATH1061	Discrete Mathematics	Prerequisite	DMTH
UQ	CSSE4603 [32]	Models of Software Systems	Undergraduate	MATH1061	Discrete Mathematics	Prerequisite	DS
UQ	CSSE7032 [33]	Models of Software Systems	Postgraduate	MATH1061	Discrete Mathematics	Prerequisite	DS
UQ	CSSE7640 [34]	Formal Modelling and Verification	Postgraduate	MATH7861	Discrete Mathematics	Prerequisite	DS
JCU	CP3110 [35]	Fundamentals of Software Engineering	Undergraduate	CP2004	Object Oriented Programming with Java	Prerequisite	PL
JCU	CP5610 [36]	Fundamentals of Software Engineering	Postgraduate	CP2004	Object Oriented Programming with Java	Prerequisite	PL
DU	SIT218 [37]	Secure Coding	Undergraduate	SIT102	Introduction to Programming	Prerequisite	FP
MU	CSC3050 [38]	Formal Methods II	Undergraduate	SIT192	Discrete Mathematics	Prerequisite	DMTH
MU	FIT5171 [39]	System validation and verification, quality and standards	Postgraduate	CSC2030	Formal Methods I	Prerequisite	ADT, PL
				CSC2040	Algorithms and data structures	Prerequisite	DS, AL, PL
				FIT9132	Introduction to databases	Prerequisite	SDF, AL, DS
				FIT9131	Programming foundations in Java	Prerequisite	IM
				MAT1830	Discrete Mathematics	Prerequisite	FP, PL
				FIT2004	Algorithms and Data Structures	Corequisite	SDF, PL
						Corequisite	DMTH
						Corequisite	ADT
						Corequisite	DS, AL

Mapping

ADT = Algorithms and Data Structures

DBS = Database Systems

DMTH = Discrete Mathematics

FP = Foundations of Programming

PL = Programming Languages

SE = Software Engineering

ACM Mapping

AL = Algorithms and Complexity

DS = Discrete Structures

IM = Information Management

PL = Programming Languages

SDF = Software Development Fundamentals

SE = Software Engineering

Table 2.2: Formal methods courses across Australia and their pre/co-requirements

Across each institution, the formal methods courses required a variety of knowledge. A large number of courses required a background in programming languages, with several requiring a CS2 level of programming ability. Here we define CS1 as an introductory level of programming knowledge with mastery of basic skills, such as loops and conditionals, while a CS2 level implies further understanding of data structures and other concepts. Often, a CS2 level of programming ability is required before taking on an advanced data structures unit, meaning that some level of programming aptitude is a precursor for a formal verification course. Discrete mathematics was also a recurring pre-requisite for several formal verification courses. Discrete mathematics has overlaps with a data structures course in addition to teaching various logics such as predicate and propositional logic. The necessity for some background in maths is unsurprising as formal methods employ rigorous mathematical techniques. There are several outliers, such as Monash University which only require CS1 as a prerequisite but requires CS2 as a co-requisite alongside a course on discrete mathematics. Several courses did not have specific courses as pre-requisites, but required a number of units to have been completed or to be admitted to a specific program. We have listed these as 'Maturity' as by the time most students reach these units, the completion of a data structures or discrete mathematics course is almost guaranteed.

Overall, from Table 2.2, it can be seen that a background in programming at a CS2 level or higher was required, in addition to having taken a course in discrete mathematics.

2.5 Experimental Design

Choosing an appropriate experimental design is pivotal for any study. The classic experimental design [40] involves the testing of a dependant or independent variable through the use of pre-testing, post-testing and using a control and experimental

group. Several additions to this accepted methodology are used such as randomising participants where the inclusion and exclusion criteria for participants group selection is 'random' [41] or randomized encouragement designs where participants are invited to one treatment group but may choose to receive a treatment or not [42]. Savović et al. [43] finds that specific study design characteristics of randomised controlled trials may lead to exaggeration of intervention effect estimates. Of these characteristics, subjectively assessed outcome measures held the greatest risks. Within these study designs additional methodological features can also be added to influence the validity of a trial such as ensuring appropriate sample sizes and allocation concealment.

These experiment designs have been used in the computer science education field in work such as Papastergiou [44] where the learning effectiveness and motivational appeal of computer games for learning memory concepts was used. In this experiment students were randomly assigned into an experimental group where games were used and a control group where no games were used with a computer memory knowledge test used as a pre and post test. Burnette et al. [45] investigated whether a growth mindset intervention could promote performance and interest in computer science. In this study students were randomly assigned to a growth mindset group or matched control with the intervention administered over four modules across a semester. The pre-test and post-test used established mindset measures where "intelligence" was replaced with the term 'computer science' to measure the intervention effect. Bockmon et al. [46] looks into the impacts of spatial skills on introductory programming abilities. In the first year of the study professors collected information on the cohort without the intervention. A pre test and post test was used where students completed the same assessment instrument. In the second year the spatial skills intervention was applied and measured using the same pre and post test assessment instrument. All three studies are examples

of the classic experimental design (examples 1 and 2) and an interventional study without concurrency control (example 3) [41].

Mixed methods research is a research approach that is seeing popular use. The goal of mixed methods research is to collect both quantitative and qualitative data within the same study to draw both their strengths and uncover relationships that exist between the intricate layers of multifaceted research questions [47]. Several mixed methods approaches exist. Almeida [48] synthesizes and describes several mixed methods approaches, which are placed into four major groups. Sequential design is described as the most popular mixed method technique characterised by two steps, data collection and analysis of quantitative data, followed by data collection and analysis of qualitative data or vice versa. Concurrent design establishes a main methodology (quantitative or qualitative) which the study follows before proceeding with collecting and analysing the other. Multiphase design involves using a chain of quantitative, qualitative and further mixed method studies and building each new study on what was learned previously. Finally multilevel design assumes a multi-layer complex problem where the number of steps taken is equal to the number of problem layers. Data collection and analysis is performed for each layer before interpreting the overall results. Almeida [48] states that sequential design is the most popular mixed methods techniques due to the greater ease of adoption but suffers from an increase in development time of the study.

Mixed methods research has been used in the computer science education area in work such as Zahedi et al. [49] where the influence of gamification and its effects on genders are explored. Quantitative data in this study was collected to determine if gamification had an effect on student performance while qualitative data was collected to answer questions about the effect of gamification on identity development and self-efficacy of women.

2.6 Financial Incentives

Using the correct incentives, often financial, for motivating participants to engage with research is an important aspect for the success of a research project. Research conducted has confirmed that monetary incentives for participation is an effective strategy [50] [51]. Although providing monetary compensation for research participation is effective, there are issues around the integrity of the collected data and its quality. [Litman et al. \[51\]](#) finds that the quality of data collected from Amazon's Mechanical Turk had a direct relationship with the compensation rates offered.

Other alternatives to a set monetary reward such as a lottery system are also in use. A lottery system offers participants a chance at winning a generally higher value reward in exchange for not having a guaranteed monetary reward. The ethical merit of lotteries has been heavily scrutinised. Issues include that lotteries undermine the "equal treatment of all participants because they prevent equal distribution of rewards" [52], they exploit those who are not competent enough to calculate their chances at winning, and that they expose participants to gambling. The effectiveness of a lottery system has seen mixed results. [Ulrich et al. \[53\]](#) randomises the type of reward that participants received and found that when offering no reward the response rate was 42.2%, using a lottery gave a 44.7% response rate and an unconditional \$5 cash incentive increased the response rate to 64.2%. [Halpern et al. \[54\]](#) performs three trials comparing the effectiveness of different probability and different value payouts compared to a monetary reward. [Halpern et al.](#) finds that a lottery based incentive did not improve the response rate compared to offering no incentive at all while an unconditional fixed reward produced a larger response compared to an actuarially equivalent lottery. [Porter and Whitcomb \[55\]](#) performs a controlled trial to test the effects of lottery incentives. [Porter and Whitcomb](#) find that "more is not better... increasing the size of the prize

did not result in a linear increase in response rates" bringing about the issue that a prize not valuable enough will not affect response rates at all but a prize too valuable will also not affect response rates. Given the mixed reported effectiveness of using a lottery reward, other factors also exist for choosing this incentive. [Zangeneh et al. \[52\]](#) samples 50 graduate students performing research on human participants and finds that the availability of funding is the main determinant for the use of a lottery reward.

Methods

We use a modified randomised controlled trial design, similar in approach to the randomised controlled trial design discussed in section 2.5. This design type involves splitting participants between two randomised groups, a control and experimental group. Each group undergoes a controlled trial where all reasonable variables are held constant with the exception of the independent variable being tested. The 'modified' represents the experiment not being truly randomised in the classical sense of a randomised trial as explained further below.

The control group for the experiment will be taken through a invariant learning workshop using the 'traditional' method wherein participants are taught about invariants using lecture slides and are given workshop questions with feedback provided by the instructor. The experimental/intervention group will undergo the same workshop with the addition of compiler provided feedback given throughout the workshop. The control and experimental groups will henceforth be referred to as the 'traditional' and 'compiler' groups. Due to the intended way that participants will register for individual workshop sessions, we cannot assign each individual participant to the traditional or compiler group on an individual basis. Instead groups of participants will be assigned to the traditional or compiler group based on the workshop they have signed up for. As such participant group designation will not be absolutely random due to this designation system. The reason for participant

designation in this manner is due to a lack of personnel to operate both a traditional group and compiler group for each intended session time. We believe that assigning participants to the traditional or compiler group by chosen workshop session will be sufficient to fulfill the random criteria as the research team has no influence on the workshop sessions that participants select.

A pre-learning and post-learning assessment instrument was designed to assess participants' baseline knowledge before the workshop and the learning that occurred over the session. A range of questions from Boolean logic to invariants will be asked with a range of difficulties tested, helping to differentiate the various levels of student understanding. This assessment instrument will form the core of the quantitative data collected in the experiment. The differences in average improvement from the pre and post-learning tests of the traditional and compiler groups will be compared. A t-test will be used to determine if a statistically significant result is present with various measures of variance used to determine the specific t-test used. Metrics on effect size such as *hedges g* will also be computed.

In addition to the quantitative data collected from the assessment instrument, we will also collect qualitative data to help answer research question 2. We use an approach similar to the sequential mixed methods approach where additional qualitative data is collected after the research workshop. The primary source of the qualitative data will be from the post questionnaire sent after the workshop to collect participants' thoughts and experiences on the workshop session. Observations taken during the workshop will also be used to explain and enrich any trends seen in the resulting data.

As with any study requiring participants, it is possible that we will have issues with a lack of participants. To address this, several measures were taken to make the research workshop more attractive. First, we accept participants with a minimum understanding of CS1 concepts (variables, loops, functions, arrays) to ensure that

we have a larger population of participants. The research workshop will also be advertised through Australian and New Zealand universities and other online platforms. Finally, we opt to use a lottery system of rewards to attract participants. A lottery system is used in favour of a set monetary amount for each participant due to budget constraints.

This research has received ethics approval (Project ID: 11915) from the Science & Engineering Ethics Subcommittee and meets the requirements set out in the National Statement on Ethical Conduct in Human Research. All steps have been taken to ensure that the research adheres to these guidelines. The full ethics approval letter can be viewed in Appendix F.

Participant Recruitment

Attracting enough participants to sign up and attend the workshops was pivotal to the success of the research workshops. To facilitate this, materials used in advertising the workshop were produced with consultation from marketing experts. From these discussions it was clear that two points needed to be satisfied, 1) incentivising participants to click on a link or scan a QR code and 2) providing an easy to complete consent form that sets participants up with a workshop time.

To motivate participants to sign up, two options were considered. The first was using the standard monetary reward. Using this option with a budget of \$1500 AUD and an estimate of 62 participants needed, we would allocate \$24 AUD to each participant. We did not opt for this option as it was unlikely that \$24 AUD was substantial enough to motivate participants to attend a multiple hour workshop. Instead we opted for a second option which involved offering 3 high value prizes that participants could enter into a draw for. The three prizes offered were a PlayStation 5 Digital Edition, \$400 AUD Gift Card and a Nintendo Switch. When completing the participant consent form, each participant could choose to be entered into the draw for one of the prizes or none at all. The distribution of participant prize choices can be seen in Figure 4.1.

Although we are unable to compare the effectiveness of both approaches, from the 103 participants that registered, we are satisfied that the second option was

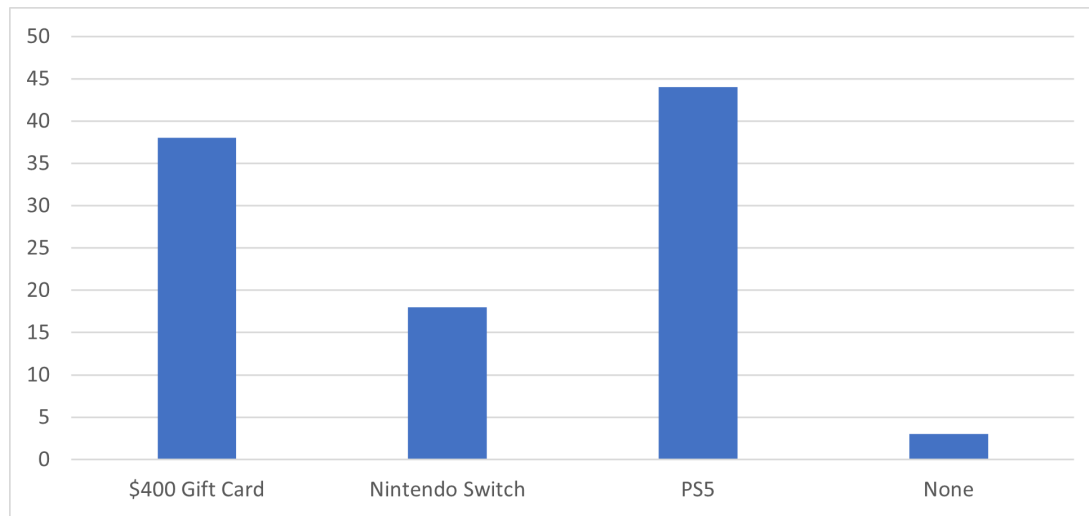


Figure 4.1: Participant prize selection choices

effective.

All forms of advertisement contained a direct link to the participant consent form. The participant consent form was made up of two pages; the first page contained summarised information about the research such as eligibility criteria, workshop learning outcomes and format. The second page collected participant details, consent and their chosen workshop time (see appendix B for the full participant consent form). By streamlining the process into a short 2 page form, we aimed to reduce the hurdles needed for participants to register.

4.1 Recruitment Sources

Participants were recruited from a wide range of sources including direct university channels (e.g. unit announcement posts), university clubs and online social media 4.2. Overall 103 participants signed up for the research workshop. The recruitment channel of participants was tracked using the embedded data feature of the Qualtrics survey platform. This feature allowed the links provided in the advertisements to have a version attached. Of these recruitment channels, direct university channels

drew in the most number of participants. Direct university channels included announcement posts, emails from faculty staff and direct referrals to the workshop by staff.

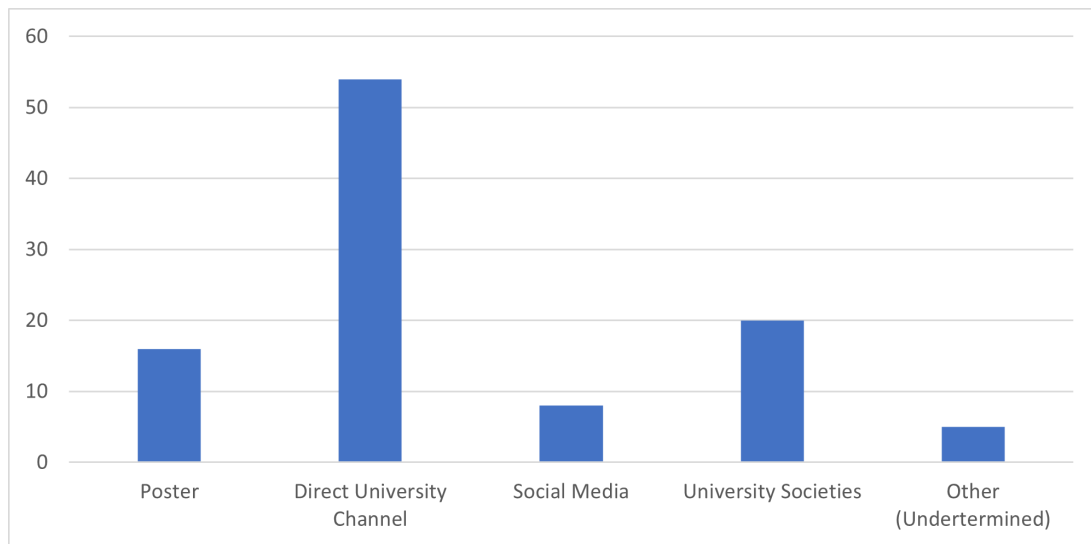


Figure 4.2: Sources of participant recruitment

The other categories made up roughly half of the total participant sign-ups. Advertisements from the university societies were made through the announcement channels of their official discord servers, a popular communication program used by computer science students. The poster category consisted of physical posters plastered around the computing/IT buildings of different universities. These posters were accompanied by a QR code and a catchy headline that directed interested participants to the participant consent form. 4 different posters were used (see Appendix C) each focusing on marketing a different aspect of the workshop:

- Poster 1 focused on the prize rewards that participants were entered for.
- Poster 2 provided information summarised information about the workshop.
- Poster 3 prominently displayed the workshop title.
- Poster 4 used a click bait headline targeting program correctness.

All four posters featured a prominent QR code that directed viewers to the participant consent form. Posters 1, 3 and 4 used colourful eye catching backgrounds while Poster 2 used a white background due to the abundant information on display. Poster 1 proved to be the only effective poster, recruiting 15 participants while the other posters were much less effective.

Social media recruitment involved advertisement posts on several sites including Red and Facebook. The 'Other' recruitment source collates any registrations where the unique version number used to track registrations in the URL was removed by the participant.



Figure 4.3: Participant registrations by poster version

4.2 Calculating participants required

To determine if a statistically significant result exists for a given intervention, a minimum number of participants is required based on the effect size of the intervention. To guide the study, we calculate the minimum sample size required using work done by [Chow et al. \[56\]](#). The formula given requires several parameters such as the expected effect size of the intervention, the average means of the two groups, and

the standard deviations of both groups. Given these parameters, the formula gives us a minimum required sample size to determine if a statistically significant result exists. We provide estimated values for the given formula and their justifications below.

$$\left(1 + \frac{1}{\kappa}\right) * \left(\sigma * \frac{z_{1-\alpha/2} + z_{1-\beta}}{\mu_A - \mu_B}\right)^2 \quad (4.1)$$

We use the following motivations and listed values:

- The effect size of similar studies in the area: [Scherer et al. \[57\]](#) provides a survey of the effect sizes of different types of studies in the computer science education area. Specifically, the study provides an overall effect size for instructional approaches of several categories. Providing compiler support falls under the ‘feedback’ and ‘problem-solving’ categories which have a weighted average hedges g of 0.5.
- The mean of the traditional group: this will be assumed to be 50.
- The standard deviation: assuming a normal distribution of test results we use a standard deviation of 12.5.
- A sampling ratio of 1 as both groups will have an equal or near equal number of participants.
- Although we cannot ensure a sampling ratio of 1, we attempted to evenly split the groups as much as possible. It is otherwise unlikely to estimate another value for the sampling ratio.
- The mean of the compiler group: using the effect size chosen above and the estimated standard deviation, the compiler group will use a mean score of 56.25.

- We used a standard α value of 0.05 and β value of 0.2.

Using the above values as inputs for the formula, we require a sample size of $62.79104 \approx 63$ to show if a statistically significant result exists when using compiler support through Dafny for learning invariants as compared to the traditional approach of learning invariants.

Workshop

The research workshop consisted of a pre-learning test, workshop content and a post-learning test. These three activities were fit inside a 3 hour time window divided as follows, not including breaks:

- Introductions - 10 minutes
- Pre-learning test - 20 minutes
- Workshop activities - 1 hour 30 minutes
 - Discussion on program correctness
 - Pre and post conditions
 - Hoare Logic
 - Examples for invariants on non array based problems
 - Self guided questions on non array based problems
 - Examples for invariants on array based problems
 - Self guided questions on array based problems
- Post-learning test - 20 minutes

All workshop materials used can be found here ¹

¹<https://github.com/MicAu/Workshop>

5.1 Minimum requirements

In order to register and attend the workshop we required a baseline amount of knowledge in order to understand what was being taught. When deciding on this baseline we did not want to make the requirements too high or we would risk having too little of a population to draw participants from. In line with this we accepted any participants who had completed at least a CS1 (introductory programming) level of knowledge. The motivation for this was 1) having a larger population to draw participants from and 2) being able to analyse whether compiler support had a greater effect on more advanced programmers when compared to novice programmers. To determine whether a participant met these minimum requirements, we asked about participants skills in the participant consent form.

5.2 Pre/Post-Learning Test Design

The motivation of running both a pre-learning test and a post-learning test was to measure the actual learning that was done during the duration of the workshop. By having a baseline benchmark for the knowledge that each participant had before each workshop, we could take their final results and subtract the two scores to find the difference in learning.

The questions asked on the learning tests attempted to gauge student understanding of several areas that we believed were important to understanding invariants. The learning tests asked about the following topic areas:

- Boolean logic
 1. Validity of a combined boolean expression, using non mixed boolean operators
 2. Validity of a combined boolean expression using a mix of boolean opera-

tions

- Pre/post conditions
 1. Choosing a pre condition that would allow the given function to produce the correct result
 2. Selecting the correct post condition for a given function with regards to a given pre condition
- Hoare logic
 1. Selecting the valid hoare triple
 2. Choosing the correct final assertion of a hoare triple
 3. Choosing the correct first assertion of a hoare triple
- Invariants
 1. Determining the incorrect invariant
 2. Selecting a correct loop invariant
 3. Selecting a correct loop invariant to ensure the post condition
 4. Selecting a correct array based loop invariant
 5. Selecting a correct complicated array based loop invariant

Choosing the specific areas to test and gauge student understanding was a difficult task as a complete validated concept inventory for learning invariants and formal verification has not yet been studied. As such we opted to use common concepts from concept inventories studied in the computer science area which relate to correctness [58]. In particular, boolean logic, mathematical specification, modular reasoning and correctness proofs were checked in the learning tests. Boolean logic reasoning was tested directly in the first two questions. Mathematical specification

was tested in the pre and post condition questions where mathematical inequalities and ranges are used as well as in the invariant questions where inequalities and existential quantifiers are tested. Modular reasoning is tested by focusing on the correctness of specific functions. Finally, correctness proofs are checked in the invariant questions where participants must choose the correct invariant that would complete the proof.

Q9. Select the most correct loop invariant

```
def sumOfFirstN(n : int):  
    i = 0  
    total = 0  
    while i <= n:  
        # Invariant here  
        total += i  
    # Post Condition: total is the sum of 0 ... n
```

☐ $0 \leq i \leq n$, total = sum of 0 ... i

☐ $0 \leq i \leq n + 1$, total = sum of 0 ... i + 1

☐ $0 \leq i \leq n$, total = sum of 0 ... i + 1

☐ $0 \leq i \leq \text{total}$, total = sum of 0 ... i

Figure 5.1: Example of a pre-learning test question. This question fits into the “Selecting a correct loop invariant” category.

Both the traditional and compiler group received the same pre and post-learning test. To ensure fairness between the two groups, the test used the Python language as a basis for the programs specified. Python was chosen as the syntax would be familiar for programmers even from different backgrounds without experience in python specifically. As a language that is similar to pseudo-code, work by [Tew and Guzdial \[59\]](#) suggests that pseudo-code can be used to achieve language independence when measuring student learning. The usage of a pre and post-learning test to measure learning also works well with participants of different background skill levels,

including those who may have already had experience with learning invariants. As only the difference in results between the pre and post-learning tests are calculated, a participant who had more knowledge before the workshop will not result in a higher “learning” measured by the learning tests. This is because it is the relative difference between the pre and post-learning test results that are measured.

Validating that both the pre-learning and post-learning test measured the same outcomes at a similar difficulty was also needed. To ensure that the two versions were of similar content, all questions in the post-learning test were based on the same content as presented in the itemised list above. Ensuring a similar difficulty was a more difficult task. To attempt this we created the post-learning test using variations of the first tests questions. As an example, question 10 of the pre-learning test asks participants to find the invariant for a given loop that increments its control variable, while the post-learning test asks the same question but with a loop that uses a decreasing control variable and modified post condition.

5.3 Dafny

As discussed in section 2.2, several usable tools exist for teaching formal verification. We decided to use Dafny for several reasons — the language is still receiving frequent updates meaning that it should be more usable on modern systems, there is tool support in popular IDEs such as VSCode so that participants can develop code in a familiar environment, and the language syntax is similar enough to other modern languages such that the transition to understanding or using the language should be easier even without direct instruction. Other tools such as JML featured some of these advantages as well but lacked the full suite of usability features.

Dafny as a language provides a familiar syntax for specifying the core essentials such as variables, loops and functions. In addition to this Dafny also allows users to

specify pre conditions, post conditions, invariants and assertions. Figure 5.2 shows an example of a Dafny program that defines a pre and post condition indicated using the 'requires' and 'ensures' keyword respectively at the start of the method. Invariants can be defined at the beginning of loops and are specified using boolean expressions, universal quantifiers and implications.

```

1  method rev(a : array<int>)
2      requires a.Length > 0;
3      modifies a;
4      ensures forall k :: 0 <= k < a.Length ==> a[k] == old(a[(a.Length - 1) - k]);
5  {
6      var i := 0;
7      while (i < a.Length - 1 - i)
8      ...
9          invariant 0 <= i <= a.Length/2;
10         invariant forall k :: 0 <= k < i || a.Length - 1 - i < k <= a.Length - 1
11         ...
12         ==> a[k] == old(a[a.Length - 1 - k]);
13         invariant forall k :: i <= k <= a.Length - 1 - i ==> a[k] == old(a[k]);
14     {
15         a[i], a[a.Length - 1 - i] := a[a.Length - 1 - i], a[i];
16         i := i + 1;
17     }
18 }

```

Figure 5.2: An example of a Dafny function that reverses a given array

Setting up participants with the correct tooling was the next step to be considered. To make the process as easy as possible we used the online service Gitpod. Gitpod takes a GitHub repository with a special 'gitpod.yml' file which sets up a VSCode environment in the participants browser. We provided each participant with a Gitpod link that setup a complete development environment with all the Dafny dependencies and tooling pre installed. The environment also provides participants with the slides used, workshop questions and pre generated Dafny files for each of the examples and workshop questions.

5.4 Workshop content

The workshop section focused on teaching participants about program correctness and invariants. First, participants discussed program correctness and its motivations

accompanied by a real life example. The formal verification approach was then compared against what participants may have seen before, namely unit testing. An introduction on pre/post conditions and Hoare logic was then given with several examples demonstrating their use. To transition to loop invariants, a program example featuring a loop is given with participants asked to apply hoare logic. The feasibility of the hoare logic approach when loops are concerned is discussed before moving onto loop invariants for helping to prove the given program. Up to this point, the experience of both the traditional group and compiler supported group is mostly the same with the exception of the compiler group setting up their development environment.

As participants completed the invariant examples the traditional groups resources featured only the slides and written notes while the compiler group had examples explained using the slides, written notes and Dafny. The compiler groups explanations focused on the hints given by dafny such as 'cannot prove termination, try supplying a decreases clause for the loop'. The first example asked participants to prove the correctness of a program to calculate the power of a given number, presented as a fully worked example. The second example calculated the triangle number of a given value n . Participants were expected to use the same technique of taking the post condition and using the loop counter to form the loop invariant. As a similar example, participants were given less guidance in completing this task. After working through these first two examples, participants were given four workshop practical questions to work on by themselves. These questions focused on the basics of invariants such as an invariant needing to be true when the loop terminated. The traditional group received feedback and support by communicating with the instructor while the compiler group had access to the instructor in addition to the feedback given by the Dafny compiler. After completing the four questions, two further invariant examples were explained featuring arrays. The third example

was a sum array question with the invariant involving a helper function similar to the first example. The fourth and final example asked participants to find the invariant to an array max question. The fourth example differed from the others as after developing an invariant using a similar approach to the previous examples, we introduce universal and existential quantifiers (for all, there exists) to help formalise the 'word based' invariant that was produced into an invariant that Dafny could help verify. After these two examples, participants were once again tasked with two workshop practical questions with help provided in the same manner to the first set of practical questions. The two questions focused on arrays and the existential quantifiers introduced in the previous examples.

5.5 Teaching technique

As discussed in 2.3 several methods exist for teaching invariants. The method we are using, as described above, is a cross between the traditional method but augmented with compiler feedback at various stages of the teaching process. For example, the "Examples for invariants on non array based problems" uses a traditional PowerPoint slide style lesson, guiding students through the construction of invariants for several non array based problems. This teaching method would fit into any traditional course on invariants, however, for the compiler group, this workshop will additionally be augmented with tool support by walking participants through the invariant construction and proof using Dafny. This additional tool will be demonstrated to the compiler group after being given the same explanation for each example as given to the traditional group. As such the use of tool support can be seen as an "extra" as opposed to a whole lesson change itself.

Participant Analysis

Of the 103 participants who signed up, 47 participants attended the workshop from which 42 completed both the required tasks (the pre and post-learning test). 5 participants attended the workshop but only completed one of the required tasks and are excluded from the results analysis. 24 female and 79 male participants registered with 11 female and 31 male participants attending and completing the workshop. Participants were recruited primarily from universities across both Australia and New Zealand.

Participants could sign up for one of the 9 workshops that were made available over a 3 week period. The workshop dates were chosen to coincide partially with a mid semester break at many of the Australian Universities. The workshop sessions were also offered at different times and days including both the weekday and weekend. The number of participants registered for each session varied as seen in Figure 6.1. No major difference in participant attendance was observed from offering sessions on weekends as compared to weekdays. A large amount of admin work was required with session scheduling. Although participants chose their initial session without any further work needed by the research team, a large amount of time was taken up by rescheduling individual participants. If a participant did not make their chosen workshop session, we first sent an email asking if they would like to reschedule which would then require several further emails. This

was exacerbated by many participants needing to reschedule multiple times. In an attempt to retain participants and combat a large amount of rescheduling we sent out a reminder about the workshop one week and one day prior to their chosen workshop session.

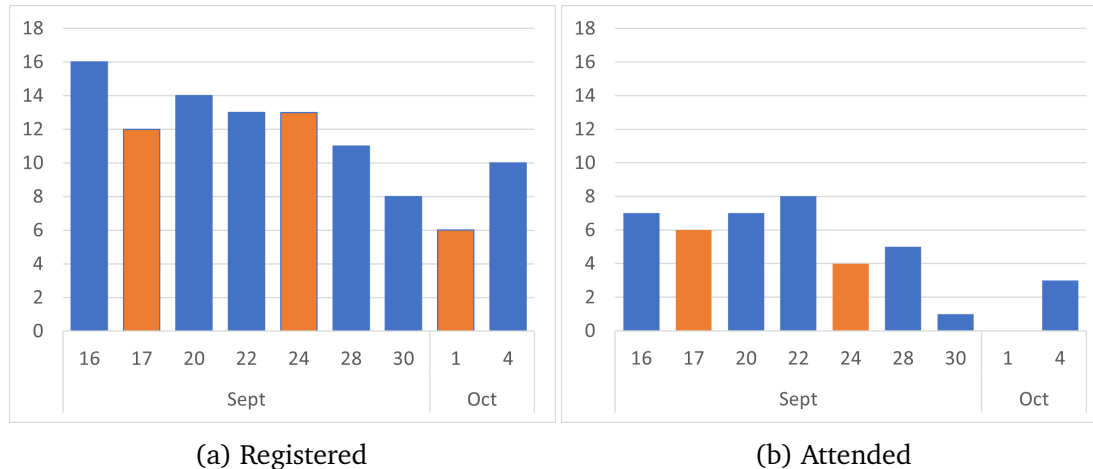


Figure 6.1: Number of registered and attended participants for each workshop session. Weekday sessions in blue, weekend sessions in orange.

Based on participants answers to the 'select your completed/undergoing skills' questions, we grouped participants into three levels. Level 1 represented novice programmers who had only completed an introductory programming unit involving learning variables, loops, conditionals, functions and arrays. Level 2 included participants who had learnt about more involved data structures such as lists, objects and problem solving techniques like recursion. Level 3 included any further skills such as compiler theory, algorithm correctness and advanced structures. Figure 6.2 shows the makeup of participants by assigned skill level. The makeup of skill level between participants who signed up compared against the makeup of participants who attended and completed all tasks reveals a similar ratio. The workshop attracted more participants from those who had studied programming extensively compared to programmers who were considered novices. Several reasons can be attributed to this, it may be less likely for novice programmers to partake in activities that are

considered extra curricular or the workshop may have appeared intimidating as the words used to advertise the workshop may be unfamiliar. Experienced programmers may also be more comfortable learning new content or are familiar with the concept of invariants.

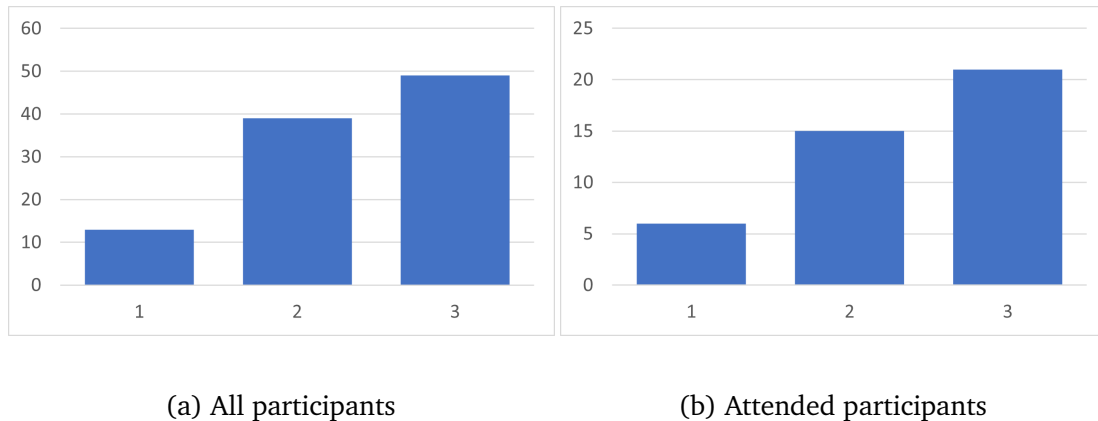


Figure 6.2: Number of participants in each skill level

Participants were asked about their confidence in writing correct programs when registering. The definition of what 'correct' means was not elaborated on as well as the size of programs being considered. This was an area that should have been further clarified in the participant consent form. Of the 103 participants the majority (45% all, 43% attended) responded with 'agree' indicating that most participants were confident in writing correct programs.

Breaking down the confidence ratings by level, we see a similar ratio is maintained.

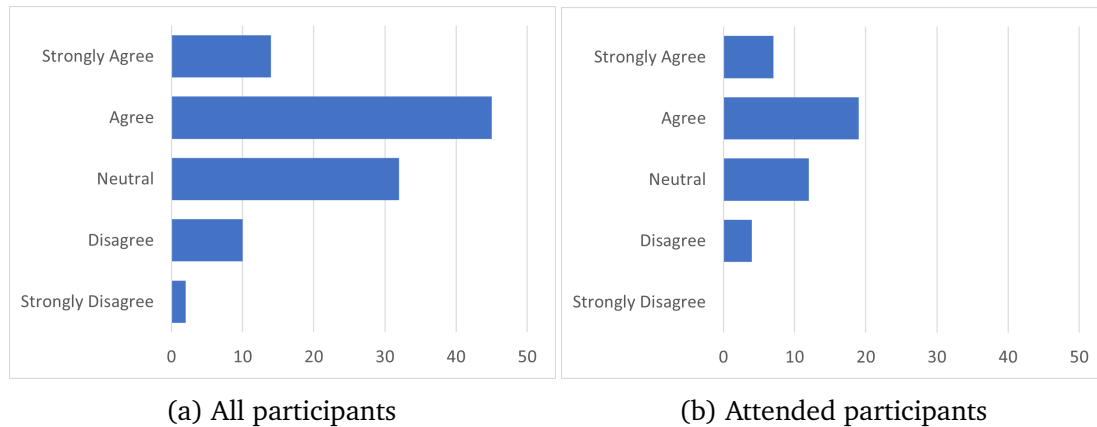


Figure 6.3: Participants' self rating of their confidence in writing correct programs

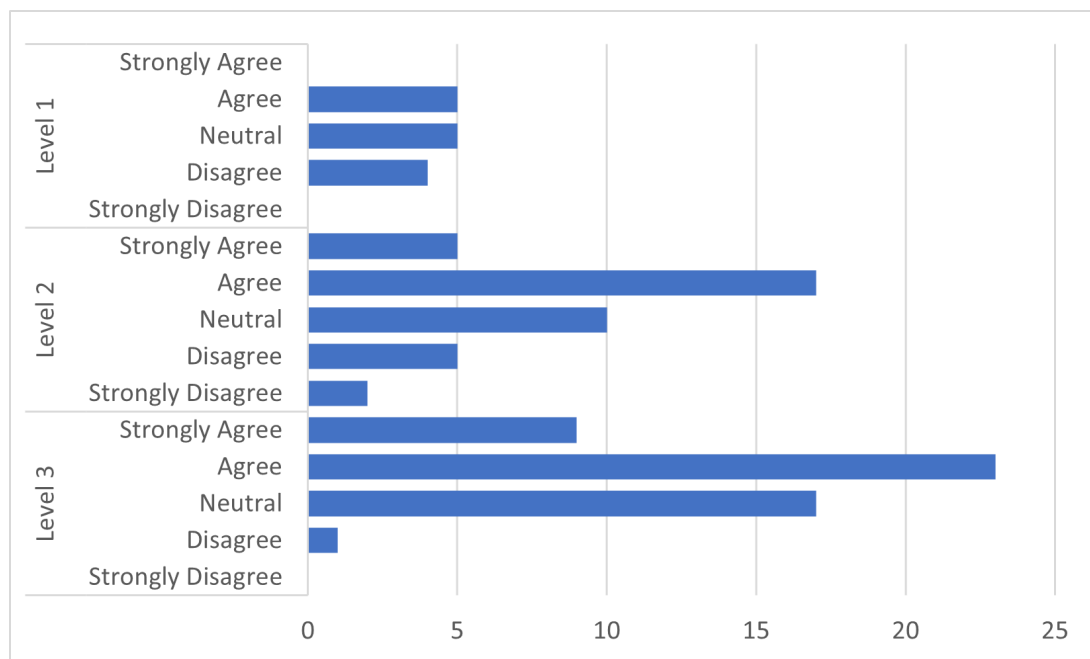


Figure 6.4: Confidence in writing correct programs for each skill level

Results

7.1 Workshop Results

We take each participants answers to the pre-learning test and post-learning test and calculate the difference between each test. As an example if a participant scored 2/4 (50%) in the boolean logic section of Test A and scored 4/4 (100%) in the Boolean logic section of test b Test B we say that this is a 50% improvement over their result in Test A.

Figure 7.1 shows the differences between the traditional and compiler group broken down by question categories and total. In total there were 42 participants with 19 in the traditional group and 23 in the compiler group. Across topic areas, the results between the traditional and compiler group varied. The compiler group performed worse in the post tests for boolean logic and pre/post conditions while performing better on Hoare logic and invariants. When all topic groups were added together, the compiler group out performed the traditional group. On the topic of invariants, the traditional group performed worse in the post test as compared to the pre test.

Several methods exist to measure the effect size of a given intervention. The effect size is a quantitative measure of the relationship strength between two variables. We opt to use Hedges g to measure effect size as the two groups have a varying

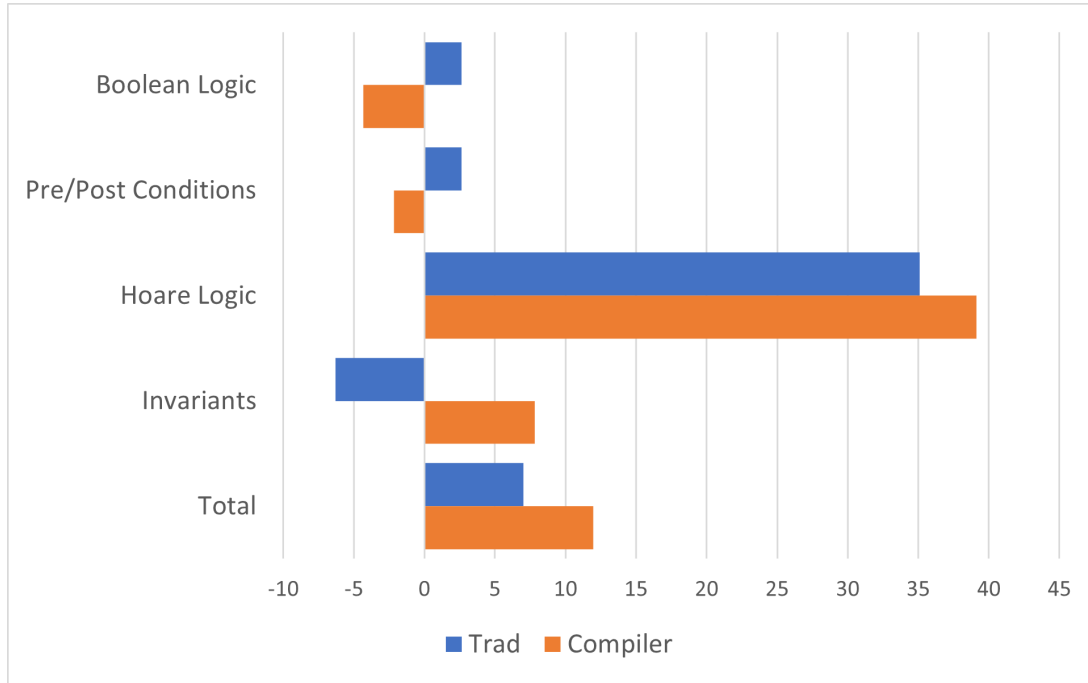


Figure 7.1: Average difference in participant scores across topics by experimental group

sample size which Cohens d does not take into account. The value of hedges g is interpreted in the same way as Cohens d , a value around 0.2 indicates a small effect, a value around 0.5 a medium effect and a value around 0.8 indicates a large effect. As shown in Table 7.1, the effect of compiler intervention in pre/post conditions and Hoare logic is below a small effect. A medium effect is seen for the other topic areas with the greatest effect seen in the invariants topic.

	Difference AVG			Std Dev		
	Trad	Compiler	Difference	Trad	Compiler	Hedges G
Boolean Logic	0.026	-0.043	-0.070	0.262	0.144	0.339
Pre/Post	0.026	-0.022	-0.048	0.424	0.412	0.115
Hoare Logic	0.351	0.391	0.040	0.392	0.343	0.110
Invariants	-0.063	0.078	0.141	0.353	0.350	0.402
Total	0.070	0.120	0.049	0.183	0.196	0.260

Table 7.1: Average difference in participant scores, standard deviation and Hedges G by topics and experimental group

We use a t-test to determine if a statistically significant difference exists between the

averages of the two groups. Several types of t-tests may be used based on whether an equal or unequal variance exists between the results of each group. Variance in this context refers to the spread of data points relative to the mean. A small variance would imply that the data points are focused closely to the mean, while a large variance implies data points spread further apart. An equal variance between the two groups means that the data points from both groups are spread similarly relative to the mean while an unequal variance implies the opposite. To determine which t-test to use we calculate the f distribution between the two results. The f distribution allows us to determine whether the variance between the means of two populations significantly differ from each other.

Table 7.2 shows that the f distribution across all topics with the exception of boolean logic sits above the 0.05 p value threshold. Given that boolean logic is an outlier, we can assume the variance between both groups are equal. Using an equal variance t-test we assume the null hypothesis H_0 to be that compiler supported learning does not produce a discernible difference as compared to the traditional approach i.e. the average results of both groups should be similar.

We calculate the p-value to determine the chance of a type 1 error (rejecting a correct null hypothesis). Using the standard acceptance value of 0.05, we find that across all topic areas, the chance of a type 1 error is above the 0.05 threshold. This result supports the null hypothesis i.e. there exists a possibility that even if the results show a difference between the compiler and traditional approach, the null hypothesis is actually still true.

Each of the t variance values also sit within the 95% region of acceptance, which supports acceptance of the null hypothesis. The acceptance region is a metric that partitions outcomes into two subsets based on a threshold value. The two subsets represent either an acceptance region where the null hypothesis is accepted, or the rejection region where the alternative hypothesis is accepted.

	Variance			F Distribution	P-value	T-Test
	Trad	Compiler	Variance Ratio			
Boolean Logic	0.065	0.020	3.280	0.005	0.280	1.094
Pre/Post	0.170	0.163	1.048	0.453	0.713	0.371
Hoare Logic	0.146	0.113	1.296	0.279	0.724	-0.356
Invariants	0.118	0.117	1.007	0.488	0.202	-1.298
Total	0.032	0.037	1.161	0.365	0.404	-0.844

Table 7.2: Variance results by topics and experimental group

These calculations are repeated with each of the groups divided into their respective skill levels. It is pertinent to note that by dividing the groups into further subgroups, there are fewer participants within each group, so any trends shown should be analysed skeptically at best. Figure 7.2 shows the distribution of participants into each skill level by group. Of note is that there is only 2 participants in the compiler group for level 1 while level 3 has a large imbalance of participants between the traditional and compiler groups. As a result of low numbers in the level 1 group, several table entries in Table 7.3 are missing due to insufficient data. Participants in the skill 1 and 2 levels showed a strong improvement in results of $\approx 10 - 20\%$ in various topic areas. Participants in the group 3 skill level showed a decrease in results of $\approx 5\%$ on all topics. A subset of data is shown in Table 7.3, the complete table can be found in Table A.1.

7.2 Post Questionnaire

Participants were emailed a post questionnaire after the completion of their workshop session. The aim of the post questionnaire was to gather further data about participant experiences and confidence in the topics taught during the workshop. The first set of questions used a Likert scale to measure participants attitudes to the given questions. The Likert questions can be grouped into two categories, the first about participants confidence in the material and second about their engagement with the workshop. The last set of questions provided short-response prompts that

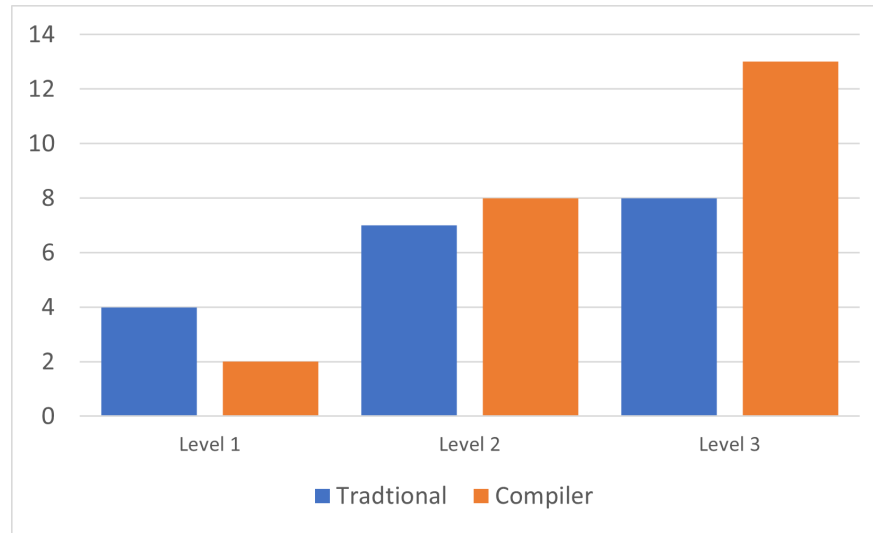


Figure 7.2: Number of participants in each skill level by experimental group

	Level 1	Difference	Hedges G	Variance Ratio	F Distribution	P-value	T-Test
Boolean Logic		0.00	-	-	-	-	-
Pre/Post Conditions		-0.13	0.45	1.33	0.55	0.63	0.52
Hoare Logic		0.25	0.39	3.19	0.39	0.68	-0.45
Invariants		0.30	1.03	2.25	0.45	0.30	-1.19
Total		0.17	1.26	-	-	0.22	-1.46
Level 2							
Boolean Logic		-0.13	0.50	3.73	0.05	0.35	0.97
Pre/Post Conditions		0.01	0.02	1.14	0.43	0.97	-0.04
Hoare Logic		0.13	0.37	1.04	0.47	0.48	-0.72
Invariants		0.34	0.82	1.20	0.40	0.14	-1.58
Total		0.15	0.78	1.31	0.36	0.15	-1.52
Level 3							
Boolean Logic		-0.04	0.20	3.52	0.03	0.67	0.44
Pre/Post Conditions		-0.15	0.34	1.07	0.44	0.46	0.75
Hoare Logic		-0.04	0.13	1.51	0.25	0.77	0.30
Invariants		-0.01	0.02	1.32	0.32	0.97	0.04
Total		-0.04	0.22	1.46	0.27	0.63	0.49

Table 7.3: Average difference between experimental groups and variance results grouped by skill level

allowed participants to express their experiences in more detail.

15 participants responded to the post questionnaire with 7 from the traditional group and 8 from the compiler group. The Likert scale question data showed a positive trend in metrics related to engagement for participants from both groups, backed by participant responses in the short-response section. Questions related to

understanding showed some mixed results between the traditional and compiler groups.

7.2.1 Likert-Scale Questions

The first question asked gauged participants confidence in writing correct programs. This question was also asked in the participant consent form when registering for the workshop sessions. As such we are able to give a comparison between the results from before the workshop and after the workshop as seen in Figure 7.3. Overall, participants tended towards selecting 'Agree' more when the same question was asked after the workshop session. Analysing the choices of individual participants, out of 15 participants, only two participants made a less confident choice with one choosing 'Agree' before and 'Neutral' after and another participant choosing 'Strongly Agree' before and 'Agree' after. 4 participants chose a more confident answer while 9 participants made the same choice.

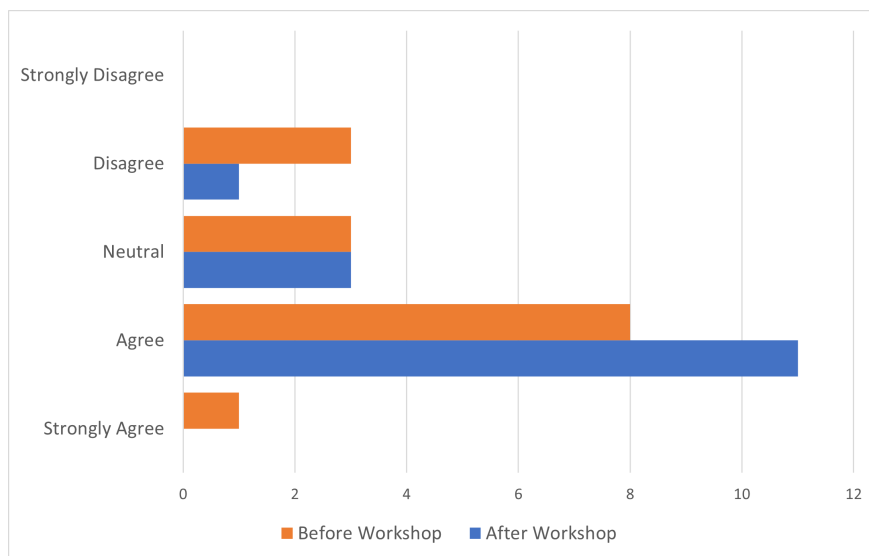


Figure 7.3: Responses to "I am confident writing correct programs", asked before and after the workshop

Splitting the results by traditional and compiler groups (Figure 7.4) we see that the same trends are observed as when the data is combined.

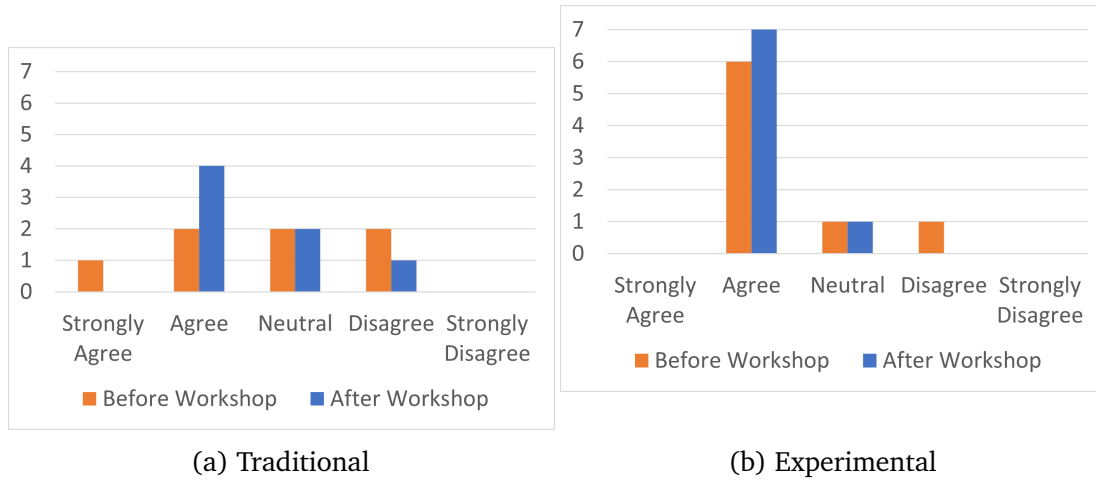


Figure 7.4: Responses to "I am confident writing correct programs" split by experimental group

Questions about workshop engagement were also asked as outlined in Table 7.4. The first question asked if the methods used in the workshop helped participants learn about invariants. Comparing the results between the traditional and compiler groups, no discernible difference from adding compiler support can be observed. The second question asked whether the way that invariants were taught made the learning process more enjoyable, which saw stronger agreement from the compiler supported group. Similarly, when asked whether the way invariants were taught contributed to the participants overall engagement, the compiler group saw strong agreement as compared to the traditional group. The last question asked whether participants would be interested in learning about other forms of software verification saw the largest effect from the addition of compiler support. Participants from the compiler group were overwhelmingly interested in pursuing further study into other forms of software verification. From this, it can be interpreted that participants from the Compiler group saw a small increase in general engagement metrics as compared to the traditional group.

Questions on understanding were asked next. The first two questions focused on understanding of invariants while the last two focused on understanding as a

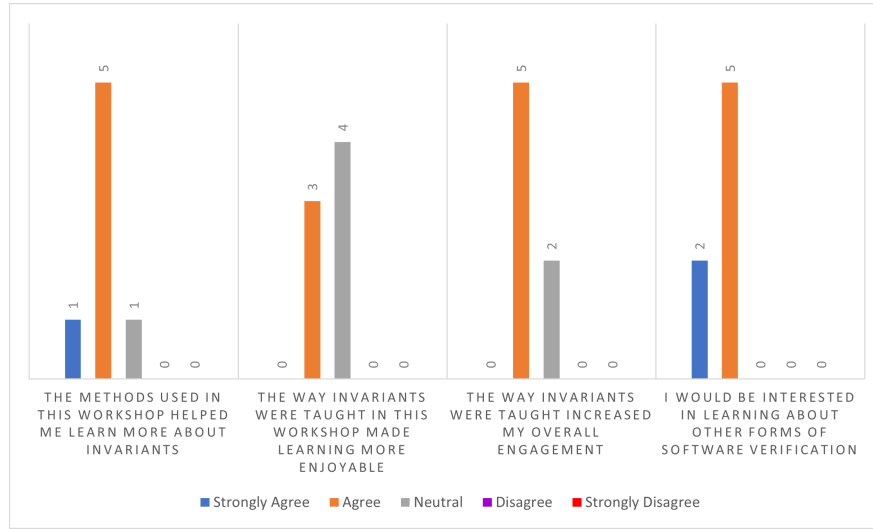
Traditional	S Agree	Agree	Neutral	Disagree	S Disagree
The methods used in this workshop helped me learn more about invariants	1	5	1	0	0
The way invariants were taught in this workshop made learning more enjoyable	0	3	4	0	0
The way invariants were taught increased my overall engagement	0	5	2	0	0
I would be interested in learning about other forms of software verification	2	5	0	0	0
Compiler	S Agree	Agree	Neutral	Disagree	S Disagree
The methods used in this workshop helped me learn more about invariants	1	7	0	0	0
The way invariants were taught in this workshop made learning more enjoyable	3	3	2	0	0
The way invariants were taught increased my overall engagement	2	4	2	0	0
I would be interested in learning about other forms of software verification	5	3	0	0	0

Table 7.4: Responses to the Post Questionnaire engagement questions by experimental group

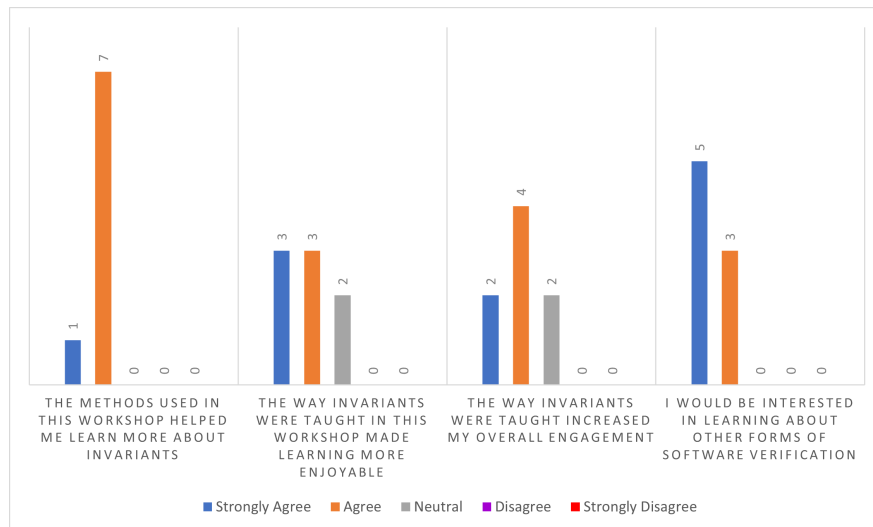
programmer. The first question asked if participants felt that they understood the concept of invariants. Responses from the traditional group centered around neutral while participants in the compiler group responded more strongly, centering around agreeing. A similar result is seen for the next question which asked if participants were confident in formulating invariants. Participants from the traditional group tended more towards disagreeing while those from the compiler group centered more around a neutral response. The following two understanding questions about programming showed mixed results. The traditional group when asked whether learning about invariants has made reasoning about programs easier reported stronger agreement as compared to the compiler group. When asked about whether learning invariants had made participants better programmers, the compiler group had a higher agree response. Overall, participants from the compiler group reported a small increase in agreement as compared to the traditional group.

Traditional	S Agree	Agree	Neutral	Disagree	S Disagree
I feel that I understand the general concept of invariants	0	3	1	3	0
I feel confident coming up with invariants	0	2	1	4	0
Learning about invariants has made reasoning about my programs easier	0	6	1	0	0
Learning about invariants has made me a better programmer	0	3	4	0	0
Compiler	S Agree	Agree	Neutral	Disagree	S Disagree
I feel that I understand the general concept of invariants	2	6	0	0	0
I feel confident coming up with invariants	0	2	5	1	0
Learning about invariants has made reasoning about my programs easier	0	4	4	0	0
Learning about invariants has made me a better programmer	1	5	1	1	0

Table 7.5: Responses to the Post Questionnaire understanding questions by group



(a) Traditional

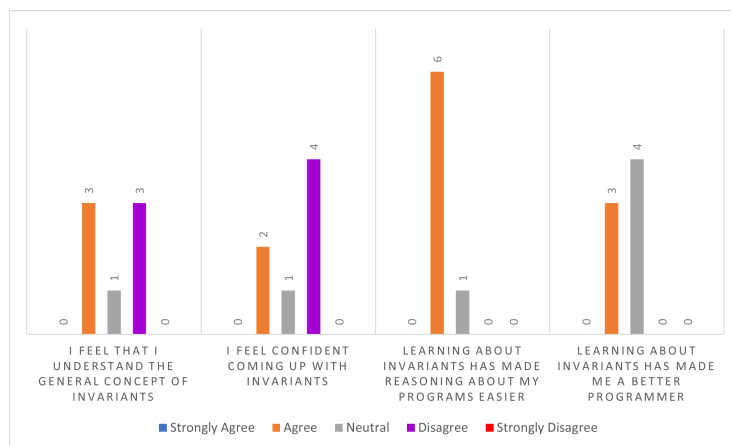


(b) Compiler

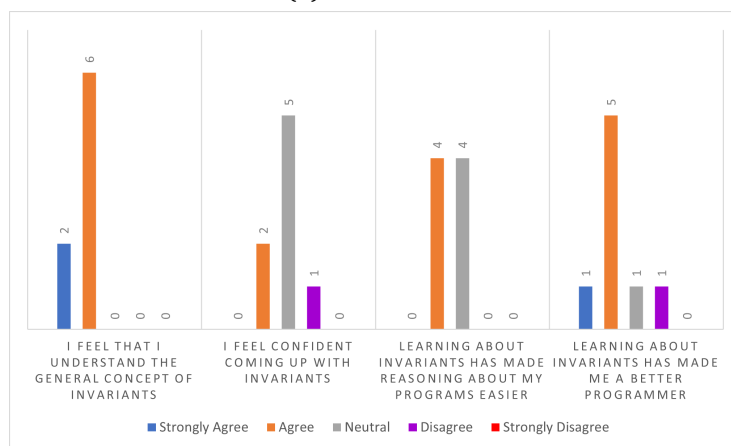
Figure 7.5: Graphed responses to the Post Questionnaire engagement questions by experimental group

7.2.2 Short-Response Prompts

Several short-response prompts were asked to understand participants insights and pain points in relation to invariants, with a final general catch all question asked. Out of the 15 participants who responded to the questions using the Likert scale, only 11 participants provided answers to the short-response questions. Analysis on



(a) Traditional



(b) Compiler

Figure 7.6: Graphed responses to the Post Questionnaire understanding questions by group

the short responses was performed with several themes emerging. These themes are presented in Table 7.6 and analysed below. The analysed quotes are presented verbatim.

The first prompt asked 'Describe an insight you had while learning about invariants'. The responses largely focused around insights that invariants gave participants about program correctness. Several responses centered around how invariants helped participants break down problems.

Theme	Responses
Insights	2
Confidence	4
Testing	3
Engagement	3
Timing	5
General issues	4
Logic issues	1
Syntax issues	2
Other	1

Table 7.6: Number of quotes per theme identified

- One insight was on the structure of complex code and how it can almost always be broken down into simple and testable components. Learning about invariants helped break down coding problems into smaller logical steps which in itself helped with the correctness of my code.
- Thinking ahead and physically writing out what the expected results would be really helped me become a better programmer

Other responses focused on the confidence that invariants provided participants in the correctness of their programs.

- Invariants allow you to be confident that your algorithm will run without errors from it's structure
- By learning about invariants I got a deeper understanding about how my code can work. I still am not 100% confident with them but it has helped.
- Invariants must always hold true for the program to be valid. If I master it, It would allow me to test results in algorithms.

Several miscellaneous responses were also provided with common themes centered around the importance of testing.

- While learning about invariants within this class, I kept thinking about my

previous experience and how I would have been able to use it in that situation.

Furthermore, how can I change my programming style to use invariants.

- Realised they make debugging easier
- The importance of testing your code.

The next prompt asked participants to "Describe some trouble that you had coming up with invariants". The responses to this prompt fit into roughly 4 categories, namely, time based issues, difficulty with logic, difficulty with syntax and general difficulty.

Timing issues included both comments on not having sufficient time in the workshop to fully understand the concept of invariants and insufficient time to develop invariants for the given problems.

- Forming the simplest invariants for a given problem in a short time.
- Since they were so new wrapping my head around them was a bit difficult. Need more time to practice to fully understand them
- The coding style I have mostly adopted is to test and find edge cases through trial and error and through the debugging process which is a more reactive approach. Invariants on the other hand were a more proactive and it felt more necessary to fully understand a problem or code segment to be able to come up with the invariants which can be hard without having gotten hands on with the writing and testing of the code

Issues with the underlying logic used to specify invariants was also present. These comments centered around being uncomfortable with specifying logical statements and the required precision of the statements. These issues have also been observed in the work by [Mannila](#) [3] who notes imprecise logical notation as a common error type when students develop invariants.

-
- I'm not very good at coming up with logical statements so it was hard for me to devise a fitting invariant.
 - I had some trouble knowing how specific i needed to be.

One comment discussed difficulty with the syntax of Dafny.

- I found the more complicated invariants difficult because I struggled with the syntax. I was not familiar with all of the operators.

Lastly, some miscellaneous responses were provided that expressed general frustration and difficulty with developing invariants.

- I struggled to find the invariants or to know where to start when dealing with invariants.
- I wasn't sure what rules applied to invariants and making them felt like guessing and checking
- I had trouble coming up with invariants for the harder functions and trying to figure out the longer functions.
- Struggled to understand it as I've never learnt it before.

The final prompt "List any additional comments" allowed participants to provide any additional feedback that they had. The common themes from the feedback centered around engagement/interest and workshop timing.

Feedback centered around engagement and interest was overall positive. It is important to note however that bias would exist with this feedback as it is more likely that participants who enjoyed or were interested in the content would give feedback in this post questionnaire.

- The content was engaging and interesting!

- I think there is more to learn about using invariants, but the workshop provided a good foundation to start from.
- Invariants were a very inspiring tool to learn about when thinking of efficiency and correctness when looking at testing. It is definitely a more challenging approach than other testing methods, but it felt like a skill that if mastered would make testing much more effortless and fool proof in the future

Several statements about the timing of the workshop were also given. All participants who gave feedback about timing were intermediate programmers in the level 2 skill group.

- Workshop was well organised and was paced well. I felt like i was given sufficient time to absorb the concepts being taught before moving onto the next thing. Thank you for teaching us about invariants!
- Was cool, but i prefer to learn a concept over the span of a few days

A final positive note for the instructor was provided :)

- I think you did great at teaching what is probably a challenging topic.

Discussion

Although no statistically significant results were seen between the traditional and compiler groups, the results still give some hope that a relationship exists between compiler support and improved learning of invariants. With participants learning test results viewed both as a whole and in separate skill levels, the results of the compiler group tended towards an improvement in average score difference. An outlier to this result existed within the level 3 skill group comprised of more experienced programmers who performed comparatively worse when compared to the traditional group. It is uncertain what caused the drop in results in the level 3 skill group. One possible explanation is that as more experienced programmers, the extra information provided by compiler support was overwhelming whereas participants with less extensive experience may have not attempted to use or understand all the features of the language. Overall, to draw a more conclusive result a larger sample size is required.

An anomaly existed in the boolean logic results of the learning tests. It is uncertain why the compiler group performed worse in the post test on boolean logic as compared to the pre test as the questions administered were similar in both tests. Furthermore, between the two groups, this topic was taught in the same manner and both groups were given the same pre and post test. It is possible that due to the small sample this anomaly was caused by coincidence, however, the addition of compiler support may also be a factor albeit unlikely.

8.0.1 Post Questionnaire

The engagement questions showed that the compiler group had stronger agreement in overall enjoyment metrics. This indicates that compiler support may add to the enjoyment of learning the topics covered. Observations from the workshop sessions also support the trends shown in the data. Participants from the compiler session asked more questions and were more engaged with the content, while those from the traditional group asked fewer targeted questions and did not inquire past the material presented.

Data from the understanding questions indicate some interesting results. From the data of both groups, it is clear that learning about invariants, regardless of the method used, produced more confident programmers. Although participants from the compiler group agreed that invariants had made them better programmers, they were less confident that it made reasoning about their programs easier as compared to the traditional group. One possible reason for this is that by using Dafny, the compiler group may have linked concepts about invariants more closely to the Dafny language specifically as opposed to a general concept that could be applied to any other language. This would have been emphasised by the unique syntax and constructs specific to Dafny. By not using a language with special syntax, the traditional group would not have associated invariants with a specific language and thus be more receptive to invariants as a general program reasoning tool.

The results from the "confidence in writing correct programs" question gives some confidence that learning about invariants has a small but visible effect on participants confidence.

8.0.2 Short-Response Prompts

The responses to the post questionnaire and specifically the short-response prompts will exhibit response bias as only motivated participants would attend the workshop,

and only the even more engaged participants would respond to the post questionnaire. From the themes identified it is clear that timing related issues were a primary source of problems for participants. Given the resources and time, it would have been preferable to run the workshop over multiple sessions to mitigate issues with cognitive load. The response about difficulties with syntax highlights several key issues. In the response the participant outlines that they "found the more complicated invariants difficult" because they "struggled with" and were "not familiar with all of the operators". If the participant was given more time to familiarise themselves with the Dafny language this may not have been an issue. If participants were given more time to familiarise themselves with the Dafny language, perhaps compiler support would have a much larger effect on student learning. Given the short time frame of the workshops however and participants being only given a quick overview of Dafny, having only one comment focused on syntax is still promising for the general adoption of Dafny. The participants comments about syntax issues could also be interpreted to mean issues with general logic. The "more complicated" invariant questions that they had troubles with focused on questions that utilised universal and existential quantifiers (there exists, for all, etc.) which were specified using Dafny specific syntax. Perhaps the combination of all of these factors overwhelmed the participants and a more structured course or higher entry requirements would alleviate this.

8.0.3 Confounding Factors

While administering the workshops, it became apparent that there were several variables that could have been controlled more tightly. The first of these was simply the balance of participants between the traditional and compiler group. With how participants signed up for their designated workshops, each workshop had to be designated as a traditional or compiler group at the start of the workshop. Future workshops would then be designated as traditional or compiler to balance this. With

enough budget, a solution to this issue would be to split each workshop group into both a traditional and compiler group by hiring additional staff. This would resolve any issues with unbalanced group as each workshop would be divided evenly.

Participant fatigue after a 3 hour workshop (inclusive of all tasks) was also an apparent issue. The goal of the workshop format was to entice participants by keeping the research session as short as possible and not requiring participants to attend on multiple days. This however also had a negative effect on the post-learning test as by the time each participant reached the end, it was clear that fatigue had set in. This issue was apparent with several participants choosing to leave before the final learning test due to fatigue. Although we tried to ease the effects of fatigue by providing small breaks in between each section, this was clearly not enough to overcome these issues.

Dedicating only 2 hours to learning the material before being tested may have not been sufficient. Due to the workshop schedule the amount of time spent on each topic before moving onto a new topic meant that participants may have not had sufficient time to let a concept sink in before a new idea was introduced. Normally one would expect to learn a concept and be given several days to understand, however, the workshop design meant that participants did not have this down time. This issue would have been exacerbated for the final more involved invariant questions where the subtleties of invariant correctness was being tested.

Although the experiment attempts to simulate student learning using the traditional "pen and paper" method, one advantage that participants in the traditional group enjoyed was more access to the instructor. In the traditional setting, an instructor would not be as free to answer all questions asked by students due to the number of students assigned per class. In the research workshop, each workshop session was composed of few participants meaning that each participant had quicker and more available access to the instructor. This is a factor that directly combats the ad-

vantage that compiler support provides which is instantaneous feedback regardless of instructor availability.

Conclusion

Formal verification is a powerful yet underused method of guaranteeing the correctness of the programs we write. Unfortunately, widespread adoption has been lacking due to the high barrier to entry [1], less usable tools as compared to mainstream programming software and the required knowledge. Computer science education research into formal verification has suffered from a lack of rigorous methodology and quantifiable data that can be compared. In this thesis, we performed a controlled trial to compare two approaches to teaching programming invariants, the traditional method and a compiler-supported technique using a modern formal verification language Dafny. The controlled trial was administered in the form of a 3 hour workshop, composed of the following activities (excluding short breaks):

- Introductions - 10 minutes
- Pre-learning test - 20 minutes
- Workshop activities - 1 hour 30 minutes
 - Discussion on program correctness
 - Pre and post conditions
 - Hoare Logic
 - Examples for invariants on non array based problems

- Self guided questions on non array based problems
 - Examples for invariants on array based problems
 - Self guided questions on array based problems
- Post-learning test - 20 minutes

Issues with participant attendance and an insufficient sample size was apparent in the data. No conclusive evidence is found that supports a difference in student learning speed or depth. Indications from the results, point towards compiler support being a valuable resource for learners. Participant responses to the post questionnaire and workshop observations support that compiler aided learning promotes further engagement as compared to the traditional method.

9.1 Future work

Given the self contained nature of each workshop session, future work for this research includes running more workshop sessions and adding to the already collected data. With all the workshop materials, slides and worksheets available it would be simple for a new instructor to run further sessions in a consistent manner. By adding further data and gathering a large enough sample size we could conclusively determine the effect of compiler support on the learning of invariants.

A different selection criteria for participants could also produce interesting results. Due to limitations in both time and available participants, this research used a low bar for entry, requiring only introductory programming knowledge. This low bar meant that the material was designed in such a way that it would be understood by a larger audience. Perhaps restricting the recruitment criteria further by requiring more programming experience and having suitable workshop material for that level would produce different results. Another selection criteria could be to select participants by abstract reasoning aptitude by using several short filter questions on sign up. The motivation for this selection criteria is that abstract reasoning is a general skill that is understood to be important for learning and understanding. By reducing the number of participants that just “won’t get it” regardless of how the material is taught (traditional, compiler support, or other means), we can focus on participants where the intervention may possibly have an effect.

Modifying the structure of the workshop into a multipart workshop that is run over several weeks or as part of a course would also be a meaningful path forward. The lack of time available expressed by participants was clearly a strong confounding variable for the results gathered. Several smaller modifications to the workshop such as better analysis of participant backgrounds e.g. assessing their prior knowledge of invariants and adding post workshop or course interviews would also be useful. The

addition of interviews as opposed to a 'static' post questionnaire would be useful for gathering information about specific participant pain points when generating invariants. Through these interviews we could also aim to build a concept inventory to catalog the fundamental skills needed for learning and generating invariants.

Several questions used across the pre and post-learning tests, workshop questions and PowerPoint slide examples were inconsistent in syntax. This would be one further area for improvement in future work.

Data Tables



Total	Difference AVG		Std Dev		Hedges G	Variance		F Distribution	P-value	T-Test
	Trad	Compiler	Trad	Compiler		Trad	Compiler			
Boolean Logic	0.03	-0.04	0.26	0.14	0.34	0.07	0.02	0.00	0.28	1.09
Pre/Post Conditions	0.03	-0.02	0.42	0.41	0.12	0.17	0.16	0.45	0.71	0.37
Hoare Logic	0.35	0.39	0.39	0.34	0.11	0.15	0.11	0.28	0.72	-0.36
Invariants	-0.06	0.08	0.35	0.35	0.40	0.12	0.12	0.49	0.20	-1.30
Total	0.07	0.12	0.18	0.20	0.26	0.03	0.04	0.37	0.40	-0.84
Level 1										
Boolean Logic	0.00	0.00	0.00	0.00	-	0.00	0.00	-	-	-
Pre/Post Conditions	-0.13	-0.25	0.25	0.35	0.45	0.05	0.06	0.55	0.63	0.52
Hoare Logic	0.42	0.67	0.69	0.47	0.39	0.35	0.11	0.39	0.68	-0.45
Invariants	0.00	0.30	0.23	0.42	1.03	0.04	0.09	0.45	0.30	-1.19
Total	0.08	0.25	0.15	0.00	1.26	0.02	0.00	-	0.22	-1.46
Level 2										
Boolean Logic	0.07	-0.06	0.35	0.18	0.50	0.10	0.03	0.05	0.35	0.97
Pre/Post Conditions	-0.07	-0.06	0.45	0.42	0.02	0.17	0.15	0.43	0.97	-0.04
Hoare Logic	0.29	0.42	0.36	0.35	0.37	0.11	0.10	0.47	0.48	-0.72
Invariants	-0.14	0.20	0.44	0.40	0.82	0.17	0.14	0.40	0.14	-1.58
Total	0.01	0.17	0.21	0.18	0.78	0.04	0.03	0.36	0.15	-1.52
Level 3										
Boolean Logic	0.00	-0.04	0.27	0.14	0.20	0.06	0.02	0.03	0.67	0.44
Pre/Post Conditions	0.19	0.04	0.46	0.43	0.34	0.18	0.17	0.44	0.46	0.75
Hoare Logic	0.38	0.33	0.28	0.33	0.13	0.07	0.10	0.25	0.77	0.30
Invariants	-0.03	-0.03	0.35	0.29	0.02	0.10	0.08	0.32	0.97	0.04
Total	0.11	0.07	0.18	0.21	0.22	0.03	0.04	0.27	0.63	0.49

Table A.1: Participants' learning test averages grouped by total and skill levels

Question	Group	S Agree	Agree	Neutral	Disagree	S Disagree
Before workshop - I am confident writing correct programs	Traditional	0	4	2	1	0
	Compiler	0	7	1	0	0
	Traditional	0	2	2	2	0
	Compiler	0	6	1	1	0
Engagement						
The methods used in this workshop helped me learn more about invariants	Traditional	0	5	1	0	0
	Compiler	1	7	0	0	0
The way invariants were taught in this workshop made learning more enjoyable	Traditional	0	3	4	0	0
	Compiler	3	3	2	0	0
The way invariants were taught increased my overall engagement	Traditional	0	5	2	0	0
	Compiler	2	4	2	0	0
I would be interested in learning about other forms of software verification	Traditional	0	5	0	0	0
	Compiler	5	3	0	0	0
Understanding						
I feel that I understand the general concept of invariants	Traditional	0	3	1	3	0
	Compiler	2	6	0	0	0
I feel confident coming up with invariants	Traditional	0	2	1	4	0
	Compiler	0	2	5	1	0
Learning about invariants has made reasoning about my programs easier	Traditional	0	6	1	0	0
	Compiler	0	4	4	0	0
Learning about invariants has made me a better programmer	Traditional	0	3	4	0	0
	Compiler	1	5	1	1	0

Table A.2: Participant responses to Likert scale questions

Participant Consent Form



Background Information

Come learn about invariants and be entered for a chance to win 3 great prizes including a PS5, Nintendo switch and a \$400 Gift Card.

Can I join?
Anyone who has taken an introductory computer science unit (variables, loops, etc) is eligible to attend.

Why should I come along?
Invariants are a useful tool that can be applied in any program to ensure that it is working correctly. Unfortunately, too few students know about them, and even less know how to use them properly!

What are invariants?
Invariants are a set of assertions that describe what must be true during the execution of a program.


What's involved?
A 3-hour online workshop at your selected time. The workshop will include a pre/post-test, short lecture, practical activities and an optional interview as part of the research study.



How will the prize winners be chosen?
Prize winners will be chosen based on a random draw after all workshops have been completed. Each participant will be entered into the draw for one of the prizes chosen during signup.

Will this affect my marks in any units of study?
No, taking part in this study will not contribute to any additional marks in your units of study.

If you have any additional questions that are unanswered, please email michael.lay@students.mq.edu.au

If you are interested in signing up, please continue to the next page.





Participant Consent Form

You are invited to participate the study of 'Compiler support for learning Invariants'. This research will investigate the effects of providing compiler support to student outcomes when learning invariants.

The study is conducted by Michael Lay to fulfill the requirements of the Master of Research degree. Supervision is provided by Dr Matthew Roberts, School of Computing, Macquarie University and Professor Matt Bower, School of Education, Macquarie University.

This study will consist of the following activities:

- Attending a 3-hour workshop at a selected time of your choice, online via Zoom. The workshop will consist of various activities related to learning invariants including a short lecture and practical activity component.
- Selected participants will be invited to a 1-hour 1:1 interview at an agreed time. The interview will be recorded for further analysis. Transcripts may be used as part of the research project, but voice recordings will not be released. Interviewees will be given the opportunity to review their answers both during and after the interview.

This study is considered low-risk research with the only foreseeable risk being discomfort.

All information and personal details used as part of this research will be treated as strictly confidential, except as required by law. All data will be de-identified before publication. Only members of the research team (Michael Lay, Matthew Roberts and Matt Bower) will have access to the data.

You may withdraw from the study at any point without consequence by sending an email to michael.lay@students.mq.edu.au.

Taking part in this study will not contribute to any additional marks in your units of study.

On completion of the study you may request a copy of the final thesis by emailing michael.lay@students.mq.edu.au.

I agree to the following

	Yes	No
I have read the information above	<input type="radio"/>	<input type="radio"/>
I am willing to participate in this research	<input type="radio"/>	<input type="radio"/>
I understand that I can withdraw from participation in this research at any time without consequence	<input type="radio"/>	<input type="radio"/>

Select all the options where you have completed some or all the listed skills

Introductory Programming (Variables, Conditions, Loops, Functions, Arrays)

Fundamental Programming (Lists, Recursion, Classes/Objects)

Algorithms and Structures (Invariants, Algorithm Correctness, Trees/Graphs)

Compiler Theory

Algorithm Theory (Reduction, Asymptotic Notations)

Select the appropriate option

	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
I am confident writing correct programs	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

First Name**Last Name****Email Address**

I am

Male

Female

I would like to attend the following session

September 16th (Friday) - 1pm

September 17th (Saturday) - 2pm

September 20th (Tuesday) - 1pm

September 22nd (Thursday) - 1pm

September 24th (Saturday) - 2pm

September 28th (Wednesday) - 1pm

September 30th (Friday) - 1pm

October 1st (Saturday) - 2pm

October 4th (Tuesday) - 1pm

The prize i would like to be entered for is the

PS5

Nintendo Switch

\$400 Gift Card

None, I'm just helping!

Signature

×

SIGN HERE

clear



Posters

Poster 1



Poster 2

Win a PS5, Nintendo Switch or \$400 Gift Card by participating in research

INVARIANT LEARNING WORKSHOP

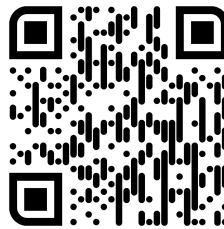
A useful skill in the Finance and Crypto industry!

Why should you come along? Invariants are a useful tool that can be applied in any program to ensure that it is working correctly. Unfortunately, too few students know about them, and even less know how to use them properly!

What are invariants? Invariants are a set of assertions that describe what must be true during the execution of a program.

What's involved? A 3-hour online workshop including a pre/post-test, optional interview as part of the research study and a post workshop questionnaire.

Can I join? Anyone who has taken an introductory computer science unit (variables, loops, etc) is eligible to attend.

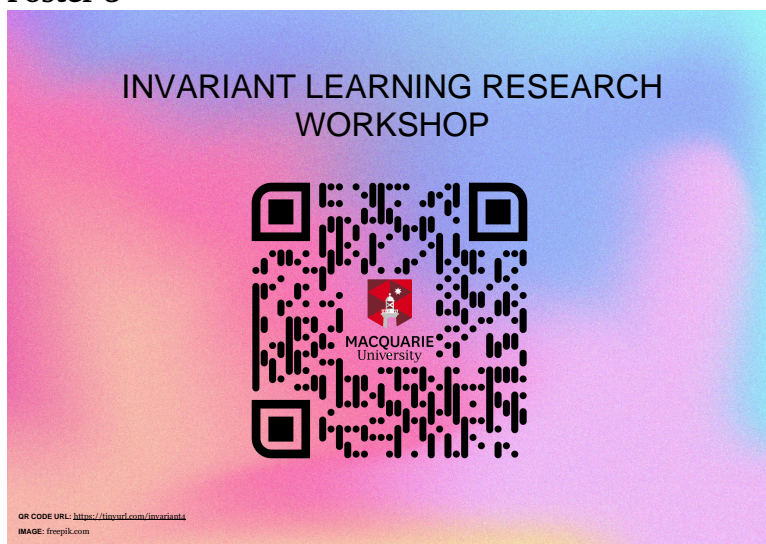
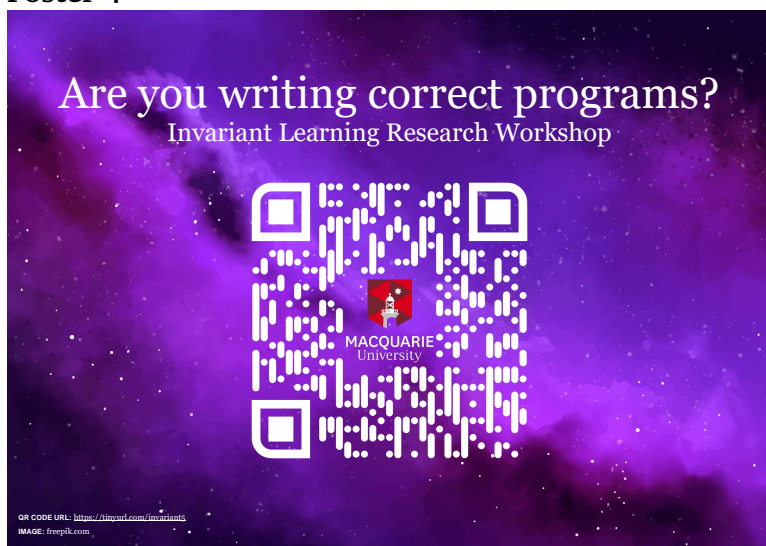


Online via Zoom

Sign up by scanning the QR code above!



FIND OUT MORE AND SIGNUP AT:
tinyurl.com/invariant3



Poster 3**Poster 4**

Learning Tests

D.1 Pre-Learning Test



Q1. Given the following values, select the equation that is true
x = 5
y = 3

☐ $x == y \ \&\& \ x + 2 == y$

☐ $x != y \ \&\& \ x + 2 == y$

☐ $x != y \ \&\& \ x - 2 == y$

☐ $x == y \ \&\& \ x - 2 == y$

Q2. Given the following values, select the equation that is true

x = 4

y = 1

true || (true && false)

true && x != y && 1 > y

false || (true && x > y)

false || (x > y && false)

Q3. Select the statement that must be true such that the function returns the correct result. The function should return the first even number in a passed integer array, and -1 if no even numbers exist.

```
int findFirstEven(int[] arr) {  
    for(int i = 0; i < arr.length; i++)  
        if(arr[i] % 2 == 0)  
            return arr[i];  
    return -1;  
}
```

arr contains one or more elements

arr must only contain integers

arr is not null/none

arr must contain an even number

Q4. Select the statement that should be at the end of the following function:

```
def squareRoot(x : int):  
    # pre condition: x is a positive integer  
  
    # computation to find the square root of x, held in a variable named 'y'  
  
    # post condition: ???  
    return y
```

`y * y == x`

`x * x == y`

`y * x == x * y`

`x is an integer`

Q5. The following truth triples are comprised of a pre-condition, series of statements and a post condition. Select the truth triple that is valid:

`{a = b} a = a + 2 {b = a + 2}`

`{a = b} a = b + 1, b = a + 1 {a > b}`

`{a < b} b = a + 1, a = b + 1 {a > b}`

`{true} while (x < a): x = x + 1 {x == a}`

Q6. Given the following truth triple, select the expression that must be true at the end
 $\{a = b\} \ a = a + 1 \ \{???\}$

$\{a = b + 1\}$

$\{b = b + 1\}$

$\{a = b\}$

$\{b = a + 1\}$

Q7. Given the following truth triple, select the expression that would come first
 $\{???\} \ a = a + 1 \ \{a \geq b\}$

$a \leq b$

$a < b$

$a == b$

$a \geq b - 1$

Q8. The following program has an incorrect invariant. Why is the invariant incorrect?

```
def sumOfArray(arr : List[int]):
    sum = 0
    i = 0
    while i < len(arr):
        # invariant 0 <= i <= arr.length
        # invariant forall k :: 1 <= k < i ==> arr[k] + arr[k - 1] == sum
        sum = sum + arr[i]
        i = i + 1
    return sum
```

i should start at 1 as multiplying by a 0 will cause the result to become 0

It is possible for i to reach the array length and cause an error

k is only checking the current and previous array entry

arr could be null/none

Q9. Select the most correct loop invariant

```
def sumOfFirstN(n : int):
    i = 0
    total = 0
    while i <= n:
        # Invariant here
        total += i
    # Post Condition: total is the sum of 0 ... n
```

$0 \leq i \leq n$, total = sum of 0 ... i

$0 \leq i \leq n + 1$, total = sum of 0 ... i + 1

$0 \leq i \leq n$, total = sum of 0 ... i + 1

$0 \leq i \leq \text{total}$, total = sum of 0 ... i

Q10. Given the following loop, select the invariant that would ensure the post condition after the loop holds

```
def foo(n : int):  
    i = 0  
    while i < n:  
        # Invariant here  
        i = i + 1  
  
    # Post Condition {i == n}
```

`0 <= i`

`0 < i`

`0 <= i <= n`

`0 < i < n`

Q11. Given the following loop, select the correct loop invariant.

```
# Return the index where 'value' is found, -1 if value does not exist in the array  
def findIndex(value : int, arr : List[int]):  
    i = 0  
    while i < len(arr):  
        # Invariant here  
        if arr[i] == value:  
            return i  
        i = i + 1  
    return -1
```

`0 <= i <= arr.Length && forall k :: 0 <= k < i ==> arr[k] != value`

`0 <= i < arr.Length && forall k :: 0 <= k < i ==> arr[k] != value`

`0 <= i <= arr.Length && forall k :: 0 < k <= i ==> arr[k] != value`

Q12. Given the following loop, select the correct invariant

Return the index of the maximum value from an array

```
def findMax(arr : List[int]):  
    # Pre Condition: arr.length != 0  
    i = 0  
    maxIdx = 0  
    while i < len(arr):  
        # Invariant here  
        if arr[i] > arr[maxIdx]:  
            maxIdx = i  
        i = i + 1  
    return maxIdx
```



$0 \leq i \leq \text{arr.Length} \ \&\& \ 0 \leq \text{maxIdx} < \text{arr.Length} \ \&\& \ 0 \leq \text{maxIdx} \leq i \ \&\& \ \text{forall } k :: 0 \leq k < i \implies \text{arr}[\text{maxIdx}] \geq \text{arr}[k]$

$0 \leq i \leq \text{arr.Length} \ \&\& \ 0 \leq \text{maxIdx} \leq \text{arr.Length} \ \&\& \ 0 \leq \text{maxIdx} \leq i \ \&\& \ \text{forall } k :: 0 \leq k < i \implies \text{arr}[\text{maxIdx}] \geq \text{arr}[k]$

$0 \leq i \leq \text{arr.Length} \ \&\& \ \text{forall } k :: 0 \leq k < \text{maxIdx} < i \implies \text{arr}[\text{maxIdx}] \geq \text{arr}[k]$

$0 \leq i \leq \text{arr.Length} \ \&\& \ \text{forall } k :: 0 \leq k < i \leq \text{maxIdx} \implies \text{arr}[\text{maxIdx}] \geq \text{arr}[k]$

D.2 Post-Learning Test



Q1. Given the following values, select the equation that is true
a = 5
b = 2

`b == a && a - 3 == b`

`b != a && a + 3 == b`

`b != a && a - 3 == b`

`b == a && b + 3 == a + 3`

Q2. Given the following values, select the equation that is true

k = 3

l = 8

(true && true) && false

false || (k + 2 > l && true)

false || (l < k && true)

false || (true && k < l)

Q3. Select the statement that must be true such that the function returns the correct result. The function should return the highest value in the passed integer array, and None if the array contains no values.

```
def getHighestValue(arr : List[int]):  
    if arr == None:  
        return None  
    highest = arr[0]  
    i = 0  
    while i < len(arr):  
        if arr[i] > highest:  
            highest = arr[i]  
    return highest
```

arr contains one or more elements

arr must only contain integers

arr is not null/none

arr must contain a large number

Q4. Select the statement that should be at the end of the following function:

```
def runFuncNTimes(N : int, func : Callable):  
    # pre condition: N is a positive integer  
  
    # computation to run the passed function N number of times.  
    # the number of times run is stored in 'i'  
  
    # post condition: ???
```

$N > i$

$i * x == x * i$

$N == i$

N is an integer

Q5. The following truth triples are comprised of a pre-condition, series of statements and a post condition. Select the truth triple that is valid:

$\{a = b\} b = 3 + a \{a = b + 3\}$

$\{a \neq b\} a = a + 1 \{a = b\}$

$\{a \neq b\} a = b + 1, b = a + 1 \{a = b\}$

$\{a > b\} a = b + 1, b = b + 2 \{b > a\}$

Q6. Given the following truth triple, select the expression that must be true at the end
 $\{a = b\} \ b = b - 1 \ \{???\}$

$\{a = b\}$

$\{b = a + 1\}$

$\{b = a - 1\}$

$\{b + 2 = a + 3\}$

Q7. Given the following truth triple, select the expression that would come first
 $\{???\} \ b = a + 1 \ \{a = b\}$

$\{a = b\}$

$\{a \leq b\}$

$\{a \geq b\}$

None of the above

Q8. The following program has an incorrect invariant. Why is the invariant incorrect?

```
def sumOfArray(arr : List[int]):  
    sum = 0  
    i = 0  
    while i < len(arr):  
        # invariant 0 <= i < arr.Length  
        # invariant sum == sum of numbers between 0 to i in arr  
        sum = sum + arr[i]  
        i = i + 1  
    return sum
```

The first invariant is not true when the loop terminates

The second invariant does not include the last number as part of its running total

i should start at 1 as summing from 0 will not include a number

arr could be null/none

Q9. Select the most correct loop invariant

```
# Returns the first number that can divide n, None if no numbers can divide n  
def firstFactor(n : int):  
    i = 2  
    while i < n:  
        # Invariant here  
        if n % i == 0:  
            return i  
        i += 1  
    return None
```

2 ... i - 1 does not include a factor of n

2 ... i does not include a factor of n

2 ... i + 1 does not include a factor of n

1 ... i does not include a factor of n

Q10. Given the following loop, select the invariant that would ensure the post condition after the loop holds

```
def foo(n : int):  
    i = n  
    while i >= 0:  
        # Invariant here  
        i = i - 1  
    # Post Condition {i <= 0}
```

☐ $0 \leq i \leq n$

☐ $0 \leq i$

☐ $-1 \leq i \leq n$

☐ $-1 \leq i$

Q11. Given the following loop, select the correct loop invariant.

Return the last index of where 0 is in the array, -1 if no 0's exist

```
def findLastZero(arr : List[int]):  
    i = len(arr) - 1  
    while i >= 0:  
        # Invariant here  
        if arr[i] == 0:  
            return i  
        i = i - 1  
    return -1
```

☐ forall $k :: i < k < \text{arr.length} \ \&\& \ k \geq 0 \implies \text{arr}[k] \neq 0$

☐ forall $k :: i \leq k < \text{arr.length} \ \&\& \ k > 0 \implies \text{arr}[k] \neq 0$

☐ forall $k :: i \leq k < \text{arr.length} \ \&\& \ k \geq 0 \implies \text{arr}[k] == 0$

☐ forall $k :: i \leq k < \text{arr.length} \ \&\& \ k \geq 0 \implies \text{arr}[k] \neq 0$

Q12. Given the following loop, select the correct invariant

```
def findMin(arr : List[int]):  
    # Pre Condition: arr.length > 0  
    i = len(arr) - 1  
    smallestIndex = i  
    while i >= 0:  
        # Invariant here  
        if arr[smallestIndex] > arr[i]:  
            smallestIndex = i  
        i = i - 1  
    return smallestIndex
```

$0 \leq i \leq \text{arr.length} \ \&\& \ 0 \leq \text{smallestIndex} < \text{arr.length} \ \&\& \ i \leq \text{smallestIndex} < \text{arr.length} \ \&\& \ \text{forall } k :: i \leq k < \text{arr.length} \implies \text{arr}[\text{smallestIndex}] < \text{arr}[k]$

$0 \leq i \leq \text{arr.length} \ \&\& \ 0 \leq \text{smallestIndex} < \text{arr.length} \ \&\& \ i \leq \text{smallestIndex} < \text{arr.length} \ \&\& \ \text{forall } k :: i \leq k < \text{arr.length} \implies \text{arr}[\text{smallestIndex}] \leq \text{arr}[k]$

$0 \leq i \leq \text{arr.length} \ \&\& \ 0 \leq \text{smallestIndex} < \text{arr.length} \ \&\& \ i \leq \text{smallestIndex} < \text{arr.length} \ \&\& \ \text{forall } k :: i \leq k \leq \text{arr.length} \implies \text{arr}[\text{smallestIndex}] \leq \text{arr}[k]$

$0 \leq i \leq \text{arr.length} \ \&\& \ 0 \leq \text{smallestIndex} < \text{arr.length} \ \&\& \ \text{forall } k :: i \leq k \leq \text{arr.length} \implies \text{arr}[\text{smallestIndex}] \leq \text{arr}[k]$

Workshop Questions

Several of these questions have been taken and repurposed from [Nipkow \[60\]](#).

E.1 Traditional group

Question 1

The below code attempts to loop from 0 up to n using the variable i

```
i = 0
while i < n:
    invariant 0 <= i
    i = i + 1
assert: i == n
```

The invariant provided is not sufficient to prove loop correctness.
Why is it not sufficient? Give one example where the invariant would not hold.

Question 2

Using the same program from Question 1, the invariant has been modified but is still not sufficient. Why is the invariant specified not sufficient? Can you suggest a fix?

```
i = 0
while i < n:
    invariant 0 <= i < n
    i = i + 1
assert: i == n
```

Question 3

Using the solution for Question 2, if the loop condition was changed from $i < n$ to $i \neq n$, would the assertion after the loop

still verify? Why or why not?

Question 4

The below program is an extension from the solution of Question 2/3.

```
def fib(n: int):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

def computeFib(n : int):
    if n == 0:
        return 0
    i = 1
    a = 0
    b = 1
    while i < n:
        invariant 0 < i <= n
        # Missing invariant
        # Missing Invariant
        a, b = b, a + b
        i = i + 1
```

This program attempts to compute the fibonacci number of n into the variable b by keeping track of the previous number in a .

$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$

Two invariants are missing which relate to the postcondition and variables a and b . Add these two invariants.

HINT: Each invariant will relate to the individual variables a and b , and may use the $\text{fib}(n)$ function defined above.

Question 5

The following code attempts to reverse a list of integers. The achieve this, the program swaps the first value with the last value and continues in this fashion until it reaches the middle e.g.

```
[1, 2, 3, 4, 5, 6] - i = 0, swaps 1 and 6
[6, 5, 3, 4, 2, 1] - i = 1, swaps 2 and 5
[6, 5, 4, 3, 2, 1] - i = 2, swaps 3 and 4
```

```

def rev(a : List[int])
  i = 0
  while i < a.length - 1 - i:
    invariant 0 <= i <= a.Length/2;
    //Invariant here
    //Invariant here

    a[i], a[a.Length - 1 - i] = a[a.Length - 1 - i], a[i];
    i = i + 1
  post condition: forall k :: 0 <= k < a.Length ==> a[k] == old(a
    [(a.Length - 1) - k]); //old represents accessing the old/
    original array before it was modified

```

Two invariants are missing related to the values that have been swapped and the values that have yet to be swapped. Find the appropriate invariants to prove the loop is correct.

Question 6

The following code returns an array of size n containing the values 0 up to and excluding n
 e.g. `arrayUpToN(3)` returns `[0, 1, 2]`

```

def arrayUpToN(n: int):
  i = 0;
  a = new int[n];
  while i < n
    invariant 0 <= i <= n
    invariant forall k :: 0 <= k < i ==> a[k] >= 0
    invariant forall k :: 0 <= k < i ==> a[k] == k
    invariant forall j, k :: 0 <= j <= k < i ==> a[j] <= a[k]

    a[i] = i;
    i = i + 1;

  post conditions: 0 <= i <= n
                  forall j :: 0 < j < n ==> a[j] >= 0
                  forall j, k : int :: 0 <= j <= k < n ==> a[j] <=
                    a[k]

```

4 invariants are provided in the loop.

Give a short explanation of what each of the three invariants attempt to show.

- 1.
- 2.
- 3.
- 4.

Which invariants are not needed?

E.2 Compiler group

Question 1

The below code attempts to loop from 0 up to n using the variable i

```
var i: int := 0;
while i < n
  invariant 0 <= i
{
  i := i + 1;
}
assert i == n;
```

The invariant provided is not sufficient to prove loop correctness.

Why is it not sufficient? Give one example where the invariant would not hold. (Test out the program in a dafny file!)

Question 2

Using the same program from Question 1, the invariant has been modified but is still not sufficient. Why is the invariant specified not sufficient? Can you suggest a fix?

```
var i := 0;
while i < n
  invariant 0 <= i < n
{
  i := i + 1;
}
assert i == n;
```

Question 3

Using the solution for Question 2, if the loop condition was changed from $i < n$ to $i \neq n$, would the assertion after the loop still verify? Why or why not?

Question 4

The below program is an extension from the solution of Question 2/3.

```
function fib(n: nat): nat
{
  if n == 0 then 0 else
  if n == 1 then 1 else
  fib(n - 1) + fib(n - 2)
}

method ComputeFib(n: nat) returns (b: nat)
  ensures b == fib(n);
{
  if (n == 0) { return 0; }
  var i := 1;
  var a := 0;
  b := 1;
  while (i < n)
    invariant 0 < i <= n;
    //Missing invariant
    //Missing Invariant
  {
    a, b := b, a + b;
    i := i + 1;
  }
}
```

This program attempts to compute the fibonacci number of n into the variable b by keeping track of the previous number in a .

$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$

Two invariants are missing which relate to the postcondition and variables a and b . Add these two invariants.

HINT: Each invariant will relate to the individual variables a and b , and may use the $\text{fib}(n)$ function defined above.

Question 5

The following code attempts to reverse a list of integers. The achieve this, the program swaps the first value with the last value and continues in this fashion until it reaches the middle e.g.

[1, 2, 3, 4, 5, 6] - $i = 0$, swaps 1 and 6

[6, 5, 3, 4, 2, 1] - $i = 1$, swaps 2 and 5

[6, 5, 4, 3, 2, 1] - i = 2, swaps 3 and 4

```
method rev(a : array<int>)
  requires a != null;
  modifies a;
  ensures forall k :: 0 <= k < a.Length ==> a[k] == old(a[(a.
    Length - 1) - k]);
{
  var i := 0;
  while (i < a.Length - 1 - i)
    invariant 0 <= i <= a.Length/2;
    //Invariant here
    //Invariant here
    {
      a[i], a[a.Length - 1 - i] := a[a.Length - 1 - i], a[i];
      i := i + 1;
    }
}
```

Two invariants are missing related to the values that have been swapped and the values that have yet to be swapped. Find the appropriate invariants to prove the loop is correct.

Question 6

The following code returns an array of size n containing the values 0 up to and excluding n
e.g. arrayUpToN(3) returns [0, 1, 2]

```
method arrayUpToN(n: int) returns (a: array<int>)
  requires n >= 0
  ensures a.Length == n
  ensures forall j :: 0 < j < n ==> a[j] >= 0
  ensures forall j, k : int :: 0 <= j <= k < n ==> a[j] <= a[k]
{
  var i := 0;
  a := new int[n];
  while i < n
    invariant 0 <= i <= n
    invariant forall k :: 0 <= k < i ==> a[k] >= 0
    invariant forall k :: 0 <= k < i ==> a[k] == k
    invariant forall j, k :: 0 <= j <= k < i ==> a[j] <= a[k]
    {
      a[i] := i;
      i := i + 1;
    }
}
```

```
    }  
}
```

4 invariants are provided in the loop.

Give a short explanation of what each of the three invariants attempt to show.

- 1.
- 2.
- 3.
- 4.

Which invariants are not needed?

Ethics Approval Letter

Science & Engineering Subcommittee
Macquarie University, North Ryde
NSW 2109, Australia



01/08/2022

Dear Dr Roberts,

Reference No: 520221191540189
Project ID: 11915
Title: Compiler Support for Learning Invariants

Thank you for submitting the above application for ethical review. The Science & Engineering Subcommittee has considered your application.

I am pleased to advise that ethical approval has been granted for this project to be conducted by Michael Lay, and other personnel: Professor Matthew Bower.

This research meets the requirements set out in the National Statement on Ethical Conduct in Human Research 2007, (updated July 2018).

Standard Conditions of Approval:

1. Continuing compliance with the requirements of the National Statement, available from the following website:
<https://nhmrc.gov.au/about-us/publications/national-statement-ethical-conduct-human-research-2007-updated-2018>.
2. This approval is valid for five (5) years, subject to the submission of annual reports. Please submit your reports on the anniversary of the approval for this protocol. You will be sent an automatic reminder email one week from the due date to remind you of your reporting responsibilities.
3. All adverse events, including unforeseen events, which might affect the continued ethical acceptability of the project, must be reported to the subcommittee within 72 hours.
4. All proposed changes to the project and associated documents must be submitted to the subcommittee for review and approval before implementation. Changes can be made via the [Human Research Ethics Management System](#).

The HREC Terms of Reference and Standard Operating Procedures are available from the Research Services website:
<https://www.mq.edu.au/research/ethics-integrity-and-policies/ethics/human-ethics>.

It is the responsibility of the Chief Investigator to retain a copy of all documentation related to this project and to forward a copy of this approval letter to all personnel listed on the project.

Should you have any queries regarding your project, please contact the [Faculty Ethics Officer](#).

The Science & Engineering Subcommittee wishes you every success in your research.

Yours sincerely,

Dr Peter Busch
Chair, Science & Engineering Subcommittee

Bibliography

- [1] M. Khazeev, M. Mazzara, D. De Carvalho, and H. Aslam, “Towards A Broader Acceptance Of Formal Verification Tools: The Role Of Education,” *arXiv:1906.01430 [cs]*, Jun. 2019, arXiv: 1906.01430. [Online]. Available: <http://arxiv.org/abs/1906.01430>
- [2] “Motivating Study of Formal Methods in the Classroom | SpringerLink.” [Online]. Available: https://link-springer-com.simsrad.net.ocs.mq.edu.au/chapter/10.1007/978-3-540-30472-2_3
- [3] L. Mannila, “Invariant Based Programming in Education - An Analysis of Student Difficulties,” *Informatics in Education*, vol. 9, no. 1, pp. 115–132, Apr. 2010. [Online]. Available: <https://infedu.vu.lt/doi/10.15388/infedu.2010.07>
- [4] R.-J. Back, “Teaching the Construction of Correct Programs Using Invariant Based Programming,” in *Petri Nets and Other Models of Concurrency - ICATPN 2006*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, S. Donatelli, and P. S. Thiagarajan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4024, pp. 1–18, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/11767589_1
- [5] N. M. Kabbani, D. Welch, C. Priester, S. Schaub, B. Durkee, Y.-S. Sun, and M. Sitaraman, “Formal Reasoning Using an Iterative Approach with an Integrated Web IDE,” *Electronic Proceedings in Theoretical Computer Science*,

- vol. 187, pp. 56–71, Aug. 2015, arXiv: 1508.03896. [Online]. Available: <http://arxiv.org/abs/1508.03896>
- [6] R.-J. Back, “Invariant Based Programming Revisited,” Jul. 2008.
- [7] —, “Invariant based programming: basic approach and teaching experiences,” *Formal Aspects of Computing*, vol. 21, no. 3, pp. 227–244, May 2009. [Online]. Available: <https://doi.org/10.1007/s00165-008-0070-y>
- [8] S. Blazy, “Teaching Deductive Verification in Why3 to Undergraduate Students,” in *Formal Methods Teaching*, ser. Lecture Notes in Computer Science, B. Dongol, L. Petre, and G. Smith, Eds. Cham: Springer International Publishing, 2019, pp. 52–66.
- [9] N. Cataño and C. Rueda, “Teaching Formal Methods for the Unconquered Territory,” vol. 5846, Nov. 2009, pp. 2–19.
- [10] E. Poll, “Teaching Program Specification and Verification Using JML and ESC/Java2,” in *TFM*, 2009.
- [11] J. Divasón and A. Romero, “Using Krakatoa for Teaching Formal Verification of Java Programs,” in *Formal Methods Teaching*, ser. Lecture Notes in Computer Science, B. Dongol, L. Petre, and G. Smith, Eds. Cham: Springer International Publishing, 2019, pp. 37–51.
- [12] “Krakatoa and Jessie: verification tools for Java and C programs.” [Online]. Available: <http://krakatoa.lri.fr/>
- [13] “Community,” Jul. 2022, original-date: 2016-04-16T20:05:38Z. [Online]. Available: <https://github.com/dafny-lang/dafny>
- [14] R. Ettinger, “Lessons of Formal Program Design in Dafny,” in *Formal Methods Teaching*, ser. Lecture Notes in Computer Science, J. F. Ferreira, A. Mendes,

-
- and C. Menghi, Eds. Cham: Springer International Publishing, 2021, pp. 84–100.
- [15] “Z3,” Jul. 2022, original-date: 2015-03-26T18:16:07Z. [Online]. Available: <https://github.com/Z3Prover/z3>
- [16] “About CVC4.” [Online]. Available: <https://cvc4.github.io/>
- [17] W. C. Tam, “Teaching loop invariants to beginners by examples,” in *Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, ser. SIGCSE ’92. New York, NY, USA: Association for Computing Machinery, Mar. 1992, pp. 92–96. [Online]. Available: <https://doi.org/10.1145/134510.134530>
- [18] D. Arnow, “Teaching programming to liberal arts students.”
- [19] D. Evans and M. Peck, “Inculcating invariants in introductory courses,” in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE ’06. New York, NY, USA: Association for Computing Machinery, May 2006, pp. 673–678. [Online]. Available: <https://doi.org/10.1145/1134285.1134388>
- [20] J. Eriksson, “Tool-Supported Invariant-Based Programming.”
- [21] ACM Computing Curricula Task Force, Ed., *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, Inc, Jan. 2013. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2534860>
- [22] “Handbook - Algorithmic Verification.” [Online]. Available: <https://www.handbook.unsw.edu.au/undergraduate/courses/2019/COMP3153>
- [23] “Handbook - comp6721 (in-)formal methods: The lost art.” [Online].

- Available: <https://www.handbook.unsw.edu.au/undergraduate/courses/2023/COMP6721>
- [24] M. University, “Formal Methods,” publisher: Macquarie University. [Online]. Available: <https://coursehandbook.mq.edu.au/2020/units/COMP4000/>
- [25] —, “Advanced Topics in Theory and Practice of Software,” publisher: Macquarie University. [Online]. Available: <https://coursehandbook.mq.edu.au/2020/units/COMP7010/>
- [26] “Formal Methods in Software Engineering - ANU.” [Online]. Available: <https://programsandcourses.anu.edu.au/2016/course/comp2600>
- [27] “UOW Course Handbook 2021: Subject CSCI410.” [Online]. Available: <https://courses.uow.edu.au/subjects/2021/CSCI410>
- [28] “Subject Descriptions - Subject Information.” [Online]. Available: https://solss.uow.edu.au/apir/public_subjectview.subject_info_view?p_subject_id=179447
- [29] “Software Verification and Validation (SENG3320),” Jul. 2013, last Modified: 2017-10-16 23:44:15. [Online]. Available: <https://www.newcastle.edu.au/course/details>
- [30] “Software Verification and Validation (SENG6320) / Course / The University of Newcastle, Australia.” [Online]. Available: <https://www.newcastle.edu.au/course/SENG6320>
- [31] “SOFT3202: Software Construction and Design 2.” [Online]. Available: <https://www.sydney.edu.au/units/SOFT3202>
- [32] “Models of Software Systems - my.UQ - The University of Queensland, Australia.” [Online]. Available: https://my.uq.edu.au/programs-courses/course.html?course_code=CSSE4603

-
- [33] “Models of Software Systems - my.UQ - The University of Queensland, Australia.” [Online]. Available: https://my.uq.edu.au/programs-courses/course.html?course_code=CSSE7032
- [34] “Formal Modelling and Verification - my.UQ - The University of Queensland, Australia.” [Online]. Available: https://my.uq.edu.au/programs-courses/course.html?course_code=CSSE7640
- [35] “CP3110 Fundamentals of Software Engineering - Subject Search - JCU Australia.” [Online]. Available: <https://secure.jcu.edu.au/app/studyfinder/index.cfm?subject=CP3110&year=2009&transform=subjectwebview.xslt>
- [36] “CP5610 Fundamentals of Software Engineering - Subject Search - JCU Australia.” [Online]. Available: <https://secure.jcu.edu.au/app/studyfinder/?subject=CP5610&year=2008&transform=subjectwebview.xslt>
- [37] D. University, “Unit.” [Online]. Available: <https://www.deakin.edu.au/courses/unit>
- [38] “CSC3050: Formal methods II.” [Online]. Available: <https://www3.monash.edu/pubs/98handbooks/it/CSC3050.html>
- [39] M. University, “System validation and verification, quality and standards,” publisher: Monash University. [Online]. Available: <https://handbook.monash.edu/2020/units/FIT5171>
- [40] P. J. Brink, *Advanced Design in Nursing Research*. SAGE, 1998, google-Books-ID: D5E5DQAAQBAJ.
- [41] R. Aggarwal and P. Ranganathan, “Study designs: Part 4 – Interventional studies,” *Perspectives in Clinical Research*, vol. 10, no. 3, pp. 137–139, 2019. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6647894/>

- [42] S. G. West, N. Duan, W. Pequegnat, P. Gaist, D. C. Des Jarlais, D. Holtgrave, J. Szapocznik, M. Fishbein, B. Rapkin, M. Clatts, and P. D. Mullen, “Alternatives to the Randomized Controlled Trial,” *American Journal of Public Health*, vol. 98, no. 8, pp. 1359–1366, Aug. 2008, publisher: American Public Health Association. [Online]. Available: <https://ajph.aphapublications.org/doi/full/10.2105/AJPH.2007.124446>
- [43] J. Savović, H. Jones, D. Altman, R. Harris, P. Jůni, J. Pildal, B. Als-Nielsen, E. Balk, C. Gluud, L. Gluud, J. Ioannidis, K. Schulz, R. Beynon, N. Welton, L. Wood, D. Moher, J. Deeks, and J. Sterne, “Influence of reported study design characteristics on intervention effect estimates from randomised controlled trials: combined analysis of meta-epidemiological studies.” *Health Technology Assessment*, vol. 16, no. 35, Sep. 2012. [Online]. Available: <https://www.journalslibrary.nihr.ac.uk/hta/hta16350/>
- [44] M. Papastergiou, “Digital Game-Based Learning in high school Computer Science education: Impact on educational effectiveness and student motivation,” *Computers & Education*, vol. 52, no. 1, pp. 1–12, Jan. 2009. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0360131508000845>
- [45] J. L. Burnette, C. L. Hoyt, V. M. Russell, B. Lawson, C. S. Dweck, and E. Finkel, “A Growth Mind-Set Intervention Improves Interest but Not Academic Performance in the Field of Computer Science,” *Social Psychological and Personality Science*, vol. 11, no. 1, pp. 107–116, Jan. 2020, publisher: SAGE Publications Inc. [Online]. Available: <https://doi.org/10.1177/1948550619841631>
- [46] R. Bockmon, S. Cooper, W. Koperski, J. Gratch, S. Sorby, and M. Dorodchi, “A CS1 Spatial Skills Intervention and the Impact on Introductory Programming

-
- Abilities,” in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Portland OR USA: ACM, Feb. 2020, pp. 766–772. [Online]. Available: <https://dl.acm.org/doi/10.1145/3328778.3366829>
- [47] A. Shorten and J. Smith, “Mixed methods research: expanding the evidence base,” *Evidence-Based Nursing*, vol. 20, no. 3, pp. 74–75, Jul. 2017, publisher: Royal College of Nursing Section: Research made simple. [Online]. Available: <https://ebn.bmj.com/content/20/3/74>
- [48] F. Almeida, “STRATEGIES TO PERFORM A MIXED METHODS STUDY,” *European Journal of Education Studies*, no. 0, Aug. 2018, number: 0. [Online]. Available: <https://www.oapub.org/edu/index.php/ejes/article/view/1902>
- [49] L. Zahedi, J. Batten, M. Ross, G. Potvin, S. Damas, P. Clarke, and D. Davis, “Gamification in education: a mixed-methods study of gender on computer science students’ academic performance and identity development,” *Journal of Computing in Higher Education*, vol. 33, no. 2, pp. 441–474, Aug. 2021. [Online]. Available: <https://link.springer.com/10.1007/s12528-021-09271-5>
- [50] J. P. Bentley and P. G. Thacker, “The influence of risk and monetary payment on the research participation decision making process,” *Journal of Medical Ethics*, vol. 30, no. 3, pp. 293–298, Jun. 2004, publisher: Institute of Medical Ethics Section: Research ethics. [Online]. Available: <https://jme.bmj.com/content/30/3/293>
- [51] L. Litman, J. Robinson, and C. Rosenzweig, “The relationship between motivation, monetary compensation, and data quality among US- and India-based workers on Mechanical Turk,” *Behavior Research Methods*, vol. 47, no. 2, pp. 519–528, Jun. 2015. [Online]. Available: <https://doi.org/10.3758/s13428-014-0483-x>
- [52] M. Zangeneh, R. Barmaki, H. Gibson-Wood, M.-J. Levitan, R. Romeo,

- and J. Bottoms, "Research Compensation and Lottery: An Online Empirical Pilot Study," *International Journal of Mental Health and Addiction*, vol. 6, no. 4, pp. 517–521, Oct. 2008. [Online]. Available: <https://doi.org/10.1007/s11469-008-9177-x>
- [53] C. M. Ulrich, M. Danis, D. Koziol, E. Garrett-Mayer, R. Hubbard, and C. Grady, "Does It Pay to Pay?: A Randomized Trial of Prepaid Financial Incentives and Lottery Incentives in Surveys of Nonphysician Healthcare Professionals," *Nursing Research*, vol. 54, no. 3, pp. 178–183, Jun. 2005. [Online]. Available: https://journals.lww.com/nursingresearchonline/Abstract/2005/05000/Does_It_Pay_to_Pay___A_Randomized_Trial_of_Prepaid.5.aspx
- [54] S. D. Halpern, R. Kohn, A. Dornbrand-Lo, T. Metkus, D. A. Asch, and K. G. Volpp, "Lottery-Based versus Fixed Incentives to Increase Clinicians' Response to Surveys," *Health Services Research*, vol. 46, no. 5, pp. 1663–1674, Oct. 2011. [Online]. Available: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3207198/>
- [55] S. R. Porter and M. E. Whitcomb, "The Impact of Lottery Incentives on Student Survey Response Rates," p. 19.
- [56] S.-C. Chow, J. Shao, and H. Wang, *Sample size calculations in clinical research*, 2nd ed., ser. Chapman & Hall/CRC biostatistics series. Boca Raton, Fla. ;: Chapman & Hall/CRC, 2008, no. 20.
- [57] R. Scherer, F. Siddiq, and B. Sánchez Viveros, "A meta-analysis of teaching and learning computer programming: Effective instructional approaches and conditions," *Computers in Human Behavior*, vol. 109, p. 106349, Aug. 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0747563220301023>

-
- [58] J. Krone, J. E. Hollingsworth, M. Sitaraman, and J. O. Hallstrom, “A Reasoning Concept Inventory for Computer Science,” p. 7.
- [59] A. E. Tew and M. Guzdial, “The FCS1: a language independent assessment of CS1 knowledge,” in *Proceedings of the 42nd ACM technical symposium on Computer science education - SIGCSE '11*. Dallas, TX, USA: ACM Press, 2011, p. 111. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1953163.1953200>
- [60] T. Nipkow, “Getting started with Dafny: A guide,” *Software Safety and Security: Tools for Analysis and Verification*, vol. 33, p. 152, 2012, publisher: IOS Press.