**FORMALIZING THE SEMANTICS OF A DUAL-VALUED FUNCTIONAL LANGUAGE IN COQ**

APPROVED BY SUPERVISING COMMITTEE:

Jeffery von Ronne Ph.D., Chair

Prof Jianwei Niu, Ph.D.

Prof. Tongping Liu, Ph.D.

Accepted:

Dean, Graduate School

## DEDICATION

*I would like to dedicate this thesis to my wife, Michelle and my daughter, Kaitlyn. For their support and understanding throughout this process.*

# FORMALIZING THE SEMANTICS OF A DUAL-VALUED FUNCTIONAL LANGUAGE IN COQ

by

ERIC ALDERS, B.S.

THESIS
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2015

## ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. von Ronne for his support, late nights, and constant mentorship throughout this process. I would also like to thank my committee for their time and effort in getting through this process. Lastly, to my my family for their support and sacrifice throughout the last year and half.

December 2015

**FORMALIZING THE SEMANTICS OF A DUAL-VALUED FUNCTIONAL LANGUAGE IN COQ**

Eric Alders, M.S.
The University of Texas at San Antonio, 2015

Supervising Professor: Jeffery von Ronne Ph.D., Chair

We provide the formalization, in the Coq proof assistant, of a small dual-valued functional language. Particularly, we explore the semantic properties that are required for reasoning about dual-values within a functional language. We formalize four theorems, in Coq, showing their completeness and soundness, of the small-step semantic rules for our small dual-valued functional language. We also highlight differences between the Coq implementation of our proofs from the original work done in this area by Pottier and Simonet with their Core ML$^2$ language.

## TABLE OF CONTENTS

## LIST OF FIGURES

# CHAPTER 1: INTRODUCTION

Why can't my computer just protect my information? In an age where computers have advanced encryption, artificial intelligence and programming languages to work with, it is still very difficult to protect the information that they process. This is because there is no one way to achieve such protections, and systems that do can become very large and costly to maintain. Due in part, because it is very difficult to protect information from all angles.

## 1.1 Protecting Confidentiality & Integrity

Encryption and access control systems are some of today's more modern approaches to controlling the confidentiality and integrity of the information that resides on our computers. Access control systems allow users to customize how information is accessed and modified within their systems while encryption protects that information while it is at rest. However, today's computer hackers have grown sophisticated enough to circumvent such protections, allowing them to steal that information to do with as they please. To improve our security we must improve our defenses against such attacks.

Information flow theory provides one such way of protecting the confidentiality of data that is processed by a program. By tracking the flows of data through a program the system can determine if any of its "secret" data is leaked. This is done by tracking the flows of high information to low outputs. A programming language can use this idea to secure the data that it processes and certify itself mathematically correct, with regards to the information flow of data.

Programming languages track the flow of information by annotating the type system of the language with information labels that can determine the security level of the data. When the program is compiled these data flows are statically checked by the compiler to determine if any information is leaked. If so the compiler won't compile the program. The end result ensures that programs that do compile are secure.

The downside to designing such a language that tracks information flows is that the language

must prove, mathematically correct, the language's syntax, semantics and type system, showing that it does not leak secret data. In order to prove that the information flows of your language are secure, one must prove that the language adheres to non-interference. Non-interference is simply defined as, given different secret inputs to the system the non-secret outputs of that system should not differ. This can lead to many complicated math proofs that are difficult and time consuming to prove by hand.

## 1.2 Machine-Checked Proofs

In order to not do all our language proofs by hand we turn to using machine-checked proofs to help automate, and in many ways simplify and speed up the development of programming language development. There are many types of automated proof tools available. For our work, we will be using a tool named Coq for developing our language and formulating the mathematical proofs.

Machine-checked proofs can help in many ways. They are in themselves like programming languages. They allow us to programmatically define our language syntax, the operational semantic rules and type rules. We can then take those definitions and use them in a proof that can be programmed and reasoned about within the proof checker. With our programming language encoded in the proof checker we can ensure that all combinations of our language syntax and rules are evaluated during the proof process. This will prevent us from making mistakes that undoubtedly would occur through a manual process.

Automated proof tools also can help by supplying, encoded within them, several common proof tactics and techniques that can be applied during the working of the proof. These help to ensure correctness of the proof tactic and ensure completeness of the results. Customized tactics can also be developed in many of these systems, along with providing libraries of common mathematical properties and simple logic constructs that can be reused as needed.

These systems can also allow us to create unit tests of our language. Try out our syntax against our semantic rules to see if the rules, as defined, work the way we intended. This will allows us to not only prove our language but to also test it out and run some sample programs or type

checking validation logic. These tools provide a much needed programming language development environment for the design, testing and proving of the designed language.

## 1.3 Problem

The problem of securing the data on a computer is a tough issue to solve. Protecting information on a machine that has aging security controls that are not 100% reliable causes issues for defenders and system administrators. Instead of looking at new ways to improve these controls we approach the problem from a different angle. We look at providing a programming language with some information flow properties that can be proven, mathematically, secure. In order to provide a programming language that can be used to solve such a complex problem we turn to the use of Coq, a tool for doing machine-checked proofs.

In order to achieve, in part, of such a goal, we look at developing a small functional language with the ability to deal with dual-values. The idea of a dual-valued language was introduced in [42] as a technical device to reason about the non-interference of a corresponding single-valued language. We will show through the use of formalized machine-checked proofs that our dual-valued functional language is sound and complete to reason about dual-values at the same time.

> *The purpose of this thesis will be to develop a small dual-valued functional language and the small-step semantics to reason about such a language. We will then show using machine-checked proofs, in Coq, that our syntax and semantics developed is adequate to reason about dual-values. We provide four theorems that show the completeness and soundness of such a language and it's operational semantic rules.*

## 1.4 Methodology

In order to develop a small dual-valued functional language I will use Coq for designing the syntax, semantics and providing the formal proofs. The language will use a ML [30] based functional syntax and semantics based on a variant of Pottier's and Simonet's Core ML$^2$ work [42]. The design

methods used, within Coq, for such a language will draw heavily from Pierce et al. online book *Software Foundations* [41]. Pierce et al. lay out the fundamentals of using Coq as a formal theorem prover for programming language development and formalization. Other ideas and more advanced programming language development theory are taken from Pierce's book *Types and Programming Languages* [40].

# CHAPTER 2: BACKGROUND

## 2.1 Information Flow

### 2.1.1 Basics

Information flow is a concept described by Bishop [11] as the security policies that define the way information moves throughout a system. The goal of most typical information flow policies is to prevent the data within a system from flowing to unauthorized users of that system. Information may flow only to processes that preserve the confidentiality of the data.

A typical example of an information flow policy would be that of the Bell-LaPadula Model [7]. This model describes a military policy that controls the flow of information across four separate access levels, Unclassified, Confidential, Secret, Top Secret. In this model information is allowed to flow up to higher levels but not down. This means that a principal that is authorized at the Secret level can see information in the Unclassified, Confidential and Secret levels but is not authorized to see information at the Top Secret level. In order for information to flow downward in the model it would need to be declassified, a process that can authorize the flows of some data to a lower confidentiality level.

Information flows can be explicit meaning that a value assigned to $y$, such as $y = x$, is an explicit flow of information from $x$ to $y$. An implicit flow of information occurs when information flows from $x$ to $y$ without an explicit assignment of the form $y = f(x)$, where $f(x)$ is an arithmetic expression with the variable $x$ [11]. Implicit information flows normally occur during control flow statements within a program (e.g. if, while). Implicit flows use control flow dependencies to inadvertently transfer information from $x$ to $y$. In order to track the information flow within a system one must keep track of both explicit and implicit flows of information.

Compilers provide a way to certify the flows of information within a program through the use of secure-type systems. During the process of compilation and type checking a programs AST can be analyzed, through the process of static analysis, to determine if the flows of information in the

program adhere to the information flow security policy. Programs that do not pass these checks cause the compiler to reject the code and not compile the program, typically notifying the user with a user-friendly error message. Through static analysis processes, we can guarantee that the executable code produced by the compiler is considered "certified" in regards to the information flow security policy.

Information flow and security-type systems have been studied for the past four decades leading to several breakthroughs in theoretical reasoning models for information flows and static analysis techniques that can be applied to detect sensitive data flows. An overview of information flow research that has been done is written in the survey paper by Sabelfeld and Myers [44]. Their survey paper addresses the issues of end-to-end security policies within a system, centering on the issue of the confidentiality of that information.

### 2.1.2 Early Developments

The idea of information flow first appeared by Lampson in 1973 [33]. Although not referred to as "Information Flow" at the time, Lampson's paper discusses the confinement of information by an executing program as to only be exposed to the caller of that program. In this way security of that information can be maintained.

Later work done by Denning actually looked at applying these concepts more formally with mathematics [22]. Denning's lattice model for secure information flow provided a way to formally certify the information flows in a program. This allowed for the tracking of public and confidential inputs to corresponding outputs represented by a matrix model of information flows.

Denning furthered her work with information flow by creating static analysis techniques for language compilers [23]. Denning allowed for the compiler to certify that non-confidential data results were not dependent on confidential data inputs. The technique allowed the compiler to enforce specific security policies about programming language semantics during compile time.

Volpano et al. took the work performed by Denning and produced a concept for defining a secure programming language type system [50]. Their work showed that a type system built around

the concepts of information flow could be proved mathematically sound. This helped fill a much needed gap in Denning's original lattice work.

Denning's work in information flow for programming languages led to the concept known as non-interference. Non-interference was first defined by Goguen and Messeguer in 1982 [31]. Non-interference, basically is defined as confidential high inputs do not cause a variation on public outputs. The concept became an important security marker for information flow and certification of a program's information flows as secure. If a user could alter inputs to a program to reveal facts about a systems confidential data then a system did not meet confidentiality standards and is not certifiable by the compiler.

Work on non-interference done by Barthe et al. took the approach of mapping several general purpose logic frameworks to non-interference for sequential and non-deterministic programming languages [5]. Their work against Hoare logic framework is proven sound, however many of the other frameworks discussed are not. They do provide one contribution that is worth noting, that of being able to determine non-interference in a non-typed language.

Later work by Zdancewic on a counter position paper talked about the challenges with non-interference in practice [52]. Zdancewic looked at why non-interference and security-type systems were not utilized by many of the popular programming languages in use at the time. He concluded that pure non-interference was too strict of a security policy to be used in practice and recommended that security-type system designers allowed for some flexibility in the type system.

### 2.1.3 Declassification

Based on the concept that most programs written need to at some point release confidential data as part of their function, work was done to incorporate declassifications into information flow theory and security lattice models. The work done by Sabelfeld and Sands looks at how to define declassification into information flow models and presents a framework for the application and definition of declassification [45].

Further research done by Askarov and Myers provide an implementation of declassification

into a semantic framework that is proved sound [2]. They then apply that concept by formulating it to produce a secure-type system. Similar work is done by Chong and Myers except their framework focuses more on the security policies involved around the declassification of data and relating how those policies can be enforced by static analysis [15].

Zdancewic and Myers later worked on introducing the concept of robustness to declassification theory in their paper "Robust Declassification" [53]. The paper contributed the idea of "robustness" to declassification theory. If data could be declassified without an attacker able to effect the contents of that information then the information was considered to be "robust". Robust declassifications provided a way for researchers to formally model and type, in a way that could be correlated to the risk of attack.

In a continuation of robust declassification, Myers et al. combined previous works in declassification theory and took them to the next level. They applied robust declassification theory into the design of a more complete secure-type system [39]. Unlike previous type systems that they had developed, this one possessed ways to declassify confidential data streams against the principles of robustness. They also introduced the concept of upgrading data within the type system.

### 2.1.4 Language Paradigms (Java/OO, Functional, Javascript)

Myers's work eventually led him down the path of developing static analysis techniques for the Java Virtual Machine and a complete secure-type system based on information flow concepts. This work resulted in Java Flow or JFlow (later renamed by Myers to JIF) [37]. Up until this point, there had not been a fully developed secure-type system for an entire language or a way for the developer of that language to define those information flows. Myers's work marks the first for an object-oriented language such as Java.

The early work on information flow was based on imperative languages. In order to extend statically-typed functional programming languages, like ML, one needs techniques for handling parametric polymorphism and higher-order functions. Work done by Pottier and Simonet [42] set out to confront issues around information flow analysis in a functional language. Their work

created the first secure-type system for a functional language known as Flow Caml. The premise of the language was based on the functional language ML and the types used within the language. Most of the paper was focused on handling the information flows of values and expressions within the ML language and very little of the paper addressed higher-order functions.

The basis of both Myer's and Pottier's work center around the use of the language within a standalone trusted host. Information flow analysis and theory at this point had not ventured into code that may execute within other processes or are shared among processes through shared memory. Much of the analysis and research was focused on the flows of data within single host environments and not on the flow of code.

Further work continued with Java byte code due to its more structured type system over plain assembly language. Genaim and Spoto researched information flow applications over the Java byte code [29]. Their work led to identifying ways to build control flow graphs, binary functions, and binary decision diagrams that led to the first full functioning static analysis of information flows for the Java runtime.

Besides compilers and static analysis applications of information flow, researchers started to also look at how they could apply information flow theory during the execution of the system. Suh et al. worked on the issue of using dynamic information flow tracking to detect I/O that may be leaked to an untrusted output or injected through an unauthorized input to detect a possible attack [48]. Their approach was useful but did lead to some runtime overhead. Tse and Zdancewic also looked into the idea of tracking information flow over the security policies enforced during runtime through the security principal at execution [49]. Their work led to more expressive runtime security policies, Java stack inspection methods and the use of PKI cryptography.

Dynamic languages and determining how to track the information flows of code dynamically composed and executed at runtime was also studied. Askarov and Sabelfeld looked at the problem of providing a framework and on-the-fly static analysis for dynamic code evaluation and communication primitives, specifically Javascript [3]. Their work provided security policies for both termination sensitive and non-sensitive conditions. Other work in the dynamic language area has

also looked at providing fully defined type systems for Javascript. Bhargavan et al. provide such a type system and security policies for using information flow security to protect against attacks that focus on single sign-on systems and client side cryptography libraries [10]. Their work created a complete secure-type system for a language that they named Defensive Javascript.

Recent work by Gampe and von Ronne look at composing a secure-type system framework between information flows of a host and executed embedded language [28]. Their work defines a modeling approach that can be used to implement static flow analysis within a compiler. The approach breaks down two languages, usually found in practice such as Java and SQL and provides examples of how to track the information flow from the host language's type system to the embedded language and certify that they are secure.

### 2.1.5 Furthering JIF-based work

Myers and Liskov developed the distributed label model in JIF [38]. JIF's label model allowed for code annotated with labels to be static checked by the Java Virtual Machine in order to certify that the byte code that was getting ready to run was considered secure. The idea was to allow for code written in JIF such as a Java Applet to be downloaded by a browser and then check for inappropriate information flows by the runtime before it executed. This technique brought a different approach from the standard discretionary and mandatory access control checks used at the time. However, this approach centered mainly around data flows of the code particularly those flows on a single user system. Exceptions and how they affect confidential data flows was examined, but only in a limited fashion. Exceptions are checked by the JIF compiler for those defined by the programmer; however, the runtime library exceptions are not checked.

Chong and Myers expand on the JIF concept by taking a new look at erasure and declassification in the distributed environment [17]. Their work adds declassification to the type system of JIF while applying erasure checks through the runtime environment. Their work helps to understand how the elements of declassification and erasure from formal information theory can be used to support non-interference within the distributed environment. The result of their efforts created the

JIFE extension to JIF. Additional work of Chong and Myer's took on the concept of extending JIF with decentralized robustness [16]. This helped strengthened the JIF secure-type system by including robustness checks within the compiler.

The untrusted host of multiple systems was researched further in depth by Zdancewic et al. in their work on secure program partitioning [55]. The work proposed the idea that a compiler split the program by the information flows of confidential data. This approach allowed portions of the code to be partitioned so that confidential information flows were not exposed to any single host. Hosts in the distributed system are only aware of their portion of confidential data and confidential data that another host contained was never passed through multiple systems, or the information flow of the system, as a whole, would not be certified. The entire secure computations of the distributed system would be achieved without confidential data being shared. This new approach was built into Myer's JIF concept to create a new JIF/Split language.

JIF/Split was later improved upon by Chong et al. in work they did to create automatic partitioning for web applications [14]. The concept took the JIF/Split concept and defined a Web Intermediate Language to allow for the definition within the code of what logic should be run on the server vs the client. This expanded on the existing decentralized label model that was already included with JIF. Now it was possible to also define a location for code logic execution. Using this approach the compiler could then split the code using secure partitioning to make sure that confidential information flows were not leaked and certify that the program was information flow compliant. The language that came out of this endeavor was known as Swift.

Both the JIF/Split and Swift languages focused on the data information flows of the code. The authors of the systems relied heavily on Myer's and Liskov's work with the JIF programming language when tracking code flows. They make the assumption that since exceptions are covered by the JIF compiler that all information flow is considered certifiable and do not expand upon the original work of exceptions done in JIF.

### 2.1.6 Cryptographic Implementations

Other problem set areas within a distributed system have also been looked at using information flow theory. One such area is that of including cryptography primitives into the language's type system and using information flow to automatically encrypt and decrypt data based on the data flows between hosts and shared memory. In Fournet and Rezk's work they develop a secure-type system that uses cryptographic primitives with asymmetric encryption and PKI that is proved to be sound [27].

In later work, Fournet et al. [26] extend Fournet and Rezk's [27] original work by modeling their type system against a distributed environment/system where the adversary controls parts of the distributed system but not the whole thing. They extend the original work by allowing a more relaxed information flow without strict non-interference. They also extend the original cryptographic implementations from the original work which only uses asymmetric encryption for key exchange and instead use symmetric encryption for information exchange which is much more efficient. The compiler implemented, extends the Jif/Split code model to use cryptography for communications instead of assuming that communications happen over private trusted channels.The implementation focuses on using code slicing to split statements into different threads of information flow that are then analyzed to enforce cryptographic manipulation of the information shared amongst hosts. Shared information is replicated across hosts in the distributed environment. Information that is exchanged between hosts is automatically encrypted by the compiler based on the static analysis and information flow of the code. New cryptographic commands are added to the original for symmetric encrypt/decrypt and for generating/verifying message authentication codes (MACs).

## 2.2 Coq

### 2.2.1 What it Is

Coq is an interactive theorem prover that is developed and maintained by The French Institute for Research in Computer Science and Automation in France. Coq can be used to formulate logical expressions, mathematical theorems, and programming languages. Coq is derived from the calculus of constructions including some common theorem solving tactics. Information about Coq can be found at the url `https://coq.inria.fr/`. A popular book written about Coq and it's many uses is by Bertot et al. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions* [9]. There are several applications of the Coq proof assistant in recent research and projects. The CompCert compiler project [34], The Bedrock Coq library [13], and formalizing the Feit-Thompson theorem [25].

### 2.2.2 How it Works

Coq is based on the calculus of construction [18] a concept derived from the Curry-Howard Correspondence [19, 20]. The calculus of construction allows Coq to act as a typed programming language that can serve, as a foundation, for the creation of mathematical proofs. By extending Curry-Howard's simply typed lambda calculus, the calculus of construction allows for inference rules that can be used to prove both the operational semantic reductions and typing judgments of a programming language.

Coq being a functional programming language similar to ML and OCaml uses types, definitions, propositions, and axioms along with logical operations that can be used for defining programming language syntax, semantic rules, and typing rules. Then, mathematical theorems can be built around these definitions to allow for formally verifying their correctness. This allows for Coq to present formal language proofs that are machine-checked and verified.

### 2.2.3 Related Work

There are very few published writings about formal Coq verification of a functional language. The most notable example of any formal verification on ML would be that of Dubois [24]. In her paper she lays out the ML specification as defined by Milner [36]. She works through proving that the original definition of the ML language is sound by using Coq.

Other Coq implementations have focused on information flow at the byte-code level. One such worthy implementation is that of Kammuller [32]. In his paper he creates a simple byte-code language that is information flow aware in Coq. Kammuller laters formally shows that this byte-code variant formally conforms to non-interference and can be used to verify information flow at the byte-code level. Another byte-code implementation that shows formalization is the work done by Barthe et al. [6]. The focus of Barthe et al.'s work is to show that the non-interference of Java byte-code can be formally proven using Coq. Their approach focuses on the certification of the byte-code, showing formal non-interference using Coq, and similar to other earlier research in information flow for the Java language.

Another sizable research project that is being worked is by Azevedo et al. [4]. Their research focuses on creating a secure framework for an abstract computing machine. The work is focused more so on the architecture of the computing platform in Coq, similar to that of some of the previous byte-code type research done with Coq. One major difference in their work that sets them apart from earlier byte-code research is that they focus more on architecture of components in their implementations of Coq so that they may be reused by other researchers. They also supply proofs that can be used as a basis for other more advanced or specialized research work.

## 2.3 Core ML$^2$

The basis of this research will pull heavily from the work done by Pottier and Simonet on Core ML$^2$ [42]. The work done by Pottier and Simonet mark the basic foundation of showing how information flow theory can be applied to Core ML, a subset of ML, similar to that of Wright et

al. [51]. In Pottier and Simonet's work, Core ML$^2$, is defined as a technical device for reasoning about Core ML's non-interference. Core ML$^2$ is an extension of Core ML that provides the bracket syntax extensions and semantic rules for reasoning about dual-values. Pottier and Simonet provide the first formal evidence that shows that their language Core ML$^2$ is both sound and complete but also that the typing system preserves well-typed behavior. They then formally give a definition of non-interference and prove the non-interference of well-typed Core ML programs as a corollary of Core ML$^2$'s soundness. Our work is a formalism of these key ideas in Coq.

$v ::= x \mid () \mid k \mid \text{fix} f.\lambda x.e \mid m \mid (v, v) \mid \text{inj}_j\ v \mid \langle v, v \rangle \mid \text{void}$
$a ::= v \mid \text{raise}\ \epsilon\ v \mid \langle a, a \rangle$
$e ::= a \mid v\ v \mid \text{ref}\ v \mid v := v \mid !v \mid \text{proj}_j\ v \mid v\ \text{case}\ x \succ e\ e \mid \text{let}\ x = v\ \text{in}\ e \mid E[e] \mid \langle e, e \rangle$
$E ::= \text{bind}\ x = [\,]\ \text{in}\ e \mid [\,]\ \text{handle}\ \epsilon\ x \succ e \mid [\,]\ \text{handle}\ e\ \text{done} \mid [\,]\ \text{handle}\ e\ \text{raise} \mid [\,]\ \text{finally}\ e$

**Figure 2.1**: Core ML$^2$ syntax

The idea behind Core ML$^2$ is that non-interference can be reasoned about by using pairs of values as inputs. These value pairs can then be used by Core ML$^2$ to simulate two Core ML programs executing simultaneously. To do this, Core ML$^2$ extends the semantics of Core ML to include a bracket operator and void value. The purpose of the void value is to be used in memory references when no value is to be stored. The bracket operator is a bit more complex. The idea of the bracket is based on the syntax and semantics of a traditional pair, where there are two values stored and the language can access the first or second projection of the pair. However, the bracket is strictly used to represent dual-values within the language. Each side of the bracket represents a different value. The projection on that bracket of either first or second will provide the value that is located at either first or second. If the projection is not considered then the bracket itself is referenced as a whole or defined as the top level.

A configuration of the Core ML$^2$ language is defined as a triple of an expression, store, and projection value. These projection values are the set of values first, second, and top level. The projection values can be applied to expressions that are within the bracket and to stores of mem-

Basic reductions

$$(\text{fix } f.\lambda x.e)v \mathbin{/_i} \mu \to e[x \Leftarrow v][f \Leftarrow \text{fix } f.\lambda x.e] \mathbin{/_i} \mu \qquad (\beta)$$

$$\text{ref } v \mathbin{/_i} \mu \to m \mathbin{/_i} \mu \oplus [m \mapsto \text{new}_i\ v] \qquad \text{(ref)}$$

$$m := v \mathbin{/_i} \mu \to () \mathbin{/_i} \mu[m \mapsto \text{update}_i\ \mu(m)\ v] \qquad \text{(assign)}$$

$$!m \mathbin{/_i} \mu \to \text{read}_i\ \mu(m) \mathbin{/_i} \mu \qquad \text{(deref)}$$

$$\text{proj}_j\ (v_1, v_2) \mathbin{/_i} \mu \to v_j \mathbin{/_i} \mu \qquad \text{(proj)}$$

$$(\text{inj}_j\ v)\ \text{case } x \succ e_1\ e_2 \mathbin{/_i} \mu \to e_j[x \Leftarrow v] \mathbin{/_i} \mu \qquad \text{(case)}$$

$$\text{let } x = v \text{ in } e \mathbin{/_i} \mu \to e[x \Leftarrow v] \mathbin{/_i} \mu \qquad \text{(let)}$$

Sequencing

$$\text{bind } x = v \text{ in } e \mathbin{/_i} \mu \to e[x \Leftarrow v] \mathbin{/_i} \mu \qquad \text{(bind)}$$

$$\text{raise } \epsilon\ v \text{ handle } \epsilon\ x \succ e \mathbin{/_i} \mu \to e[x \Leftarrow v] \mathbin{/_i} \mu \qquad \text{(handle)}$$

$$\text{raise } \epsilon\ v \text{ handle } e \text{ done} \mathbin{/_i} \mu \to e \mathbin{/_i} \mu \qquad \text{(handle-done)}$$

$$\text{raise } \epsilon\ v \text{ handle } e \text{ raise} \mathbin{/_i} \mu \to e;\ \text{raise } \epsilon\ v \mathbin{/_i} \mu \qquad \text{(handle-raise)}$$

$$a \text{ finally } e \mathbin{/_i} \mu \to e;\ a \mathbin{/_i} \mu \qquad \text{(finally)}$$

$$E[a] \mathbin{/_i} \mu \to a \mathbin{/_i} \mu \qquad \text{(pop)}$$
$$if\ E\ handles\ neither\ \lfloor a \rfloor_1 nor\ \lfloor a \rfloor_2$$

Lifting

$$\langle v_1 \mid v_2 \rangle\ v \mathbin{/} \mu \to \langle v_1\ \lfloor v \rfloor_1 \mid v_2\ \lfloor v \rfloor_2 \rangle \mathbin{/} \mu \qquad \text{(lift-app)}$$

$$\langle v_1 \mid v_2 \rangle := v \mathbin{/} \mu \to \langle v_1 := \lfloor v \rfloor_1 \mid v_2 := \lfloor v \rfloor_2 \rangle \mathbin{/} \mu \qquad \text{(lift-assign)}$$

$$!\langle v_1 \mid v_2 \rangle \mathbin{/} \mu \to \langle !v_1 \mid !v_2 \rangle \mathbin{/} \mu \qquad \text{(lift-deref)}$$

$$\text{proj}_j\ \langle v_1 \mid v_2 \rangle \mathbin{/} \mu \to \langle \text{proj}_j v_1 \mid \text{proj}_j v_2 \rangle \mathbin{/} \mu \qquad \text{(lift-proj)}$$

$$\langle v_1 \mid v_2 \rangle\ \text{case } x \succ e_1\ e_2 \mathbin{/} \mu \to \langle v_1\ \text{case } x \succ \lfloor e_1 \rfloor_1\ \lfloor e_2 \rfloor_1 \mid v_2\ \text{case } x \succ \lfloor e_1 \rfloor_2\ \lfloor e_2 \rfloor_2 \rangle \mathbin{/} \mu \qquad \text{(lift-case)}$$

$$E[\langle a_1 \mid a_2 \rangle] \mathbin{/} \mu \to \langle \lfloor E \rfloor_1[a_1] \mid \lfloor E \rfloor_2[a_2] \rangle \mathbin{/} \mu \qquad \text{(lift-context)}$$
$$if\ none\ of\ the\ sequencing\ rules\ applies$$

Reduction under a context

$$\frac{e \mathbin{/_i} \mu \to e' \mathbin{/_i} \mu'}{E[e] \mathbin{/_i} \mu \to E[e'] \mathbin{/_i} \mu'} \qquad \text{(context)}$$

$$\frac{e \mathbin{/_i} \mu \to e' \mathbin{/_i} \mu' \qquad e_j = e'_j \qquad \{i, j\} = \{1, 2\}}{\langle e_1 \mid e_2 \rangle \mathbin{/} \mu \to \langle e'_1 \mid e'_2 \rangle \mathbin{/} \mu'} \qquad \text{(bracket)}$$

Auxiliary functions

$$\text{new}_\bullet\ v = v \qquad \text{update}_\bullet\ v\ v' = v' \qquad \text{read}_\bullet\ v = v$$
$$\text{new}_1\ v = \langle v \mid \text{void} \rangle \qquad \text{update}_1\ v\ v' = \langle v' \mid \lfloor v \rfloor_2 \rangle \qquad \text{read}_1\ v = \lfloor v \rfloor_1$$
$$\text{new}_2\ v = \langle \text{void} \mid v \rangle \qquad \text{update}_2\ v\ v' = \langle \lfloor v \rfloor_1 \mid v' \rangle \qquad \text{read}_2\ v = \lfloor v \rfloor_2$$

**Figure 2.2**: Core ML$^2$ semantics

ory references that may contain brackets. The configuration's projection value is what determines which side of the bracket will be accessed within the program and applied during program execution. When a projection is encountered by a memory store it will determine which side of a bracket to read, or if writing memory, will create a bracket with the appropriate value placed in the bracket, as indicated by the projection. A projection of a configuration is the projection of the expression and the projection of the store.

Brackets are introduced in the language by either explicitly being supplied by the programmer within the source code or they will be introduced during memory allocation operations at runtime. Brackets are not allowed to be nested within other brackets, as well as, voids are only allowed in brackets within memory. It is illegal to use a void anywhere else within the program. Voids should not be explicitly referenced during runtime by the program and represent an unbound memory reference. Reduction can occur outside brackets, within brackets and by lifting the bracket boundary.

During semantic operation of a Core ML expression, the language reduction of a bracket will always execute either the first or second projection (if not defined as top level). Other reduction steps have been added within Core ML$^2$ to facilitate reducing brackets and lifting brackets out of other expressions as to not prohibit progress of further execution. These reduction steps are introduced as lift operations that are applied at points to lift the bracket boundary while leaving the projections of the bracket unchanged. These lift operations can be applied during application reduction, assignment, memory dereferencing, pair projections and case evaluations.

Our work will use a much smaller language that we will call the small dual-valued functional language. The language will use the bracket concept from Core ML$^2$ to handle the reduction of dual-values. This will provide a template allowing us to setup our language syntax and semantics within Coq for formalization.

# CHAPTER 3: THE SMALL DUAL-VALUED FUNCTIONAL LANGUAGE

In this chapter we present our language the small dual-valued functional language. We will begin by introducing the syntax of the language and how we represented that syntax as inductive type structures within Coq. We will then present our operational small-step semantics of the language followed by followed by their inductive type structure representation in Coq.

## 3.1 Syntax

$$v ::= x \mid \text{true} \mid \text{false} \mid \lambda x.e \mid \langle v, v \rangle \mid \text{void}$$
$$e ::= v \mid e\ e \mid \text{if } e \text{ then } e \text{ else } e \mid \langle e, e \rangle$$

**Figure 3.1**: Small dual-valued functional language syntax

We begin design of our small dual-valued functional language with the syntax in BNF form located in Figure 3.1. Our language consists of the following values. The x represents variables. We follow that with our boolean values true and false. We decided to use boolean values instead of constant integers for this language for simplicity. We follow that by our lambda abstraction. Our last two value terms are the bracket and void terms from Pottier and Simonet's Core ML$^2$ [42]. The bracket will give us the dual-valued feature for our language that we desire. The void is included in our syntax for completeness but because we currently do not have memory reference features, they will not be be used.

```
Inductive Exp : Type :=
  | True : Exp
  | False : Exp
  | Var: id → Exp
  | App: Exp → Exp → Exp
  | Lambda : id → Exp → Exp
  | If: Exp → Exp → Exp → Exp
  | Void: Exp
  | Bracket: Exp → Exp → Exp.
```

**Figure 3.2**: Syntax in Coq

The expressions in our small dual-valued functional language syntax are represented by the application and our if than else. Brackets are also a legal expression within our language as long as one side of the bracket contains an expression term. A bracket that consists of two value terms is considered by definition a value. We choose these terms initially for simplicity, but also because both of these terms can use a lift rule that will allow us to prove a complete set of dual-valued semantics for the features of our language.

Figure 3.2 represents the same BNF grammar of our small dual-valued functional language in Coq. We use an inductive type called Exp for expression. As you can see the constructors of this type are the syntactic terms of our language. In Figure 3.3, we have the Coq representation of our value types in an inductive definition. This definition differs slightly from the one in our BNF because, instead of having a separate syntactic category for values, we define value as a property of expressions. The use of why we do this will become more apparent when we discuss the operational semantics of our language.

```
Inductive value : Exp → Prop :=
  | v_True : value True
  | v_False : value False
  | v_Func : ∀ x, ∀ e, value (Lambda x e)
  | v_Void : value Void
  | v_Bracket : ∀ v1 v2,
    value v1 →
    value v2 →
    value (Bracket v1 v2).
```

**Figure 3.3**: Value terms in Coq

### 3.1.1 Dual-Valued Features

The introduction of brackets into the language for dealing with dual-values caused us to create several other inductive definitions in our language for use within our proofs to reason about the language. In Figure 3.4, is our representation for the set of projection values in Coq. We use two different inductive types for this, one to reason about projections within the bracket and another to reason about the projection within a configuration value that is attached to the semantic operation.

This helps to constrain Coq's reasoning within our proofs to only valid projection values.

```
 Inductive proj_index: Type :=
  | fst: proj_index
  | snd: proj_index.

Inductive config_index: Type :=
  | top_lvl: config_index
  | cfg_index: proj_index → config_index.
```

**Figure 3.4**: Bracket & configuration projection types

When reasoning about brackets in our dual-valued language we need to be able to determine the projection of a given bracket given a valid projection. To do this we use a fixpoint function in Coq that matches on the expression and index and returns the proper value. We also need to reason about the contents of a bracket, or in this case, determine if the bracket is well-formed. This will allow us to enforce the rule within Coq that brackets cannot be nested. To do this we use two different inductive types, one to determine if the expression is not a bracket and the other, to enforce nesting recursively. The Coq inductive definitions are in Figure 3.5.

### 3.1.2 Stores

Currently, our simple dual-valued functional language does not contain any memory reference expressions, however, in order to add such functionality, in the future, we have added definitions for a memory store within our Coq inductive types. The store definition consists of a list of expressions shown in Figure 3.6. The store also includes a single fixpoint function that allows us to take the projection of a store given a valid projection value. This will allow us to reason about a store in our dual-valued language. The definition recursively traverses the store taking the bracket projection of each expression in the store. This definition is crucial when later we talk about configuration projections which are equivalent to the projection of the expression and the projection of the store.

```
Fixpoint bracket_proj (e:Exp) (i:proj_index) : Exp :=
  match e with
  | Bracket e1 e2 ⇒
    match i with
    | fst ⇒ e1
    | snd ⇒ e2
    end
  | True ⇒ e
  | False ⇒ e
  | Var x ⇒ e
  | App e1 e2 ⇒ App (bracket_proj e1 i) (bracket_proj e2 i)
  | Lambda x e1 ⇒ Lambda x (bracket_proj e1 i)
  | If e1 e2 e3 ⇒ If (bracket_proj e1 i) (bracket_proj e2 i) (bracket_proj e3 i)
  | Void ⇒ e
  end.

Inductive not_bracket : Exp → Prop :=
  | nb_true : not_bracket True
  | nb_false : not_bracket False
  | nb_var : ∀ x, not_bracket (Var x)
  | nb_app : ∀ e1 e2,
    not_bracket e1 →
    not_bracket e2 →
    not_bracket (App e1 e2)
  | nb_lambda : ∀ x e,
    not_bracket e →
    not_bracket (Lambda x e)
  | nb_if : ∀ e1 e2 e3,
    not_bracket e1 →
    not_bracket e2 →
    not_bracket e3 →
    not_bracket (If e1 e2 e3)
  | nb_void : not_bracket (Void).

Inductive well_formed : Exp → Prop :=
  | wf_true : well_formed True
  | wf_false : well_formed False
  | wf_var : ∀ x, well_formed (Var x)
  | wf_app : ∀ e1 e2,
    well_formed e1 →
    well_formed e2 →
    well_formed (App e1 e2)
  | wf_lambda : ∀ x e,
    well_formed e →
    well_formed (Lambda x e)
  | wf_if : ∀ e1 e2 e3,
    well_formed e1 →
    well_formed e2 →
    well_formed e3 →
    well_formed (If e1 e2 e3)
  | wf_bracket: ∀ e1 e2,
      not_bracket e1 → well_formed e1 →
      not_bracket e2 → well_formed e2 →
      well_formed (Bracket e1 e2).
```

**Figure 3.5**: Bracket functions & inductive types

## 3.2 Semantics

For the operational semantics of our small dual-valued functional language we use a small-step style of semantic. The listed semantic rules are found in Figure 3.7. For our basic reductions the

```
Definition store := list Exp.
Fixpoint store_proj (l:list Exp) (i:proj_index) : list Exp :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ (bracket_proj h i) :: (store_proj t i)
  end.
```

**Figure 3.6**: Store definition & functions

Basic reductions

$$\frac{e_1 \mathbin{/_i} \mu \to e_1' \mathbin{/_i} \mu'}{e_1\ e_2 \mathbin{/_i} \mu \to e_1'\ e_2 \mathbin{/_i} \mu'} \quad \text{(app1)}$$

$$\frac{e_2 \mathbin{/_i} \mu \to e_2' \mathbin{/_i} \mu'}{v\ e_2 \mathbin{/_i} \mu \to v\ e_2' \mathbin{/_i} \mu'} \quad \text{(app2)}$$

$$(\lambda.x.e)v \mathbin{/_i} \mu \to [v \mapsto x]e \mathbin{/_i} \mu \quad (\beta)$$

$$\frac{e_1 \mathbin{/_i} \mu \to e_1' \mathbin{/_i} \mu'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mathbin{/_i} \mu \to \text{if } e_1' \text{ then } e_2 \text{ else } e_3 \mathbin{/_i} \mu'} \quad \text{(if)}$$

$$\text{if true then } e_2 \text{ else } e_3 \mathbin{/_i} \mu \to e_2 \mathbin{/_i} \mu \quad \text{(if true)}$$

$$\text{if false then } e_2 \text{ else } e_3 \mathbin{/_i} \mu \to e_3 \mathbin{/_i} \mu \quad \text{(if false)}$$

$$\frac{e \mathbin{/_i} \mu \to e' \mathbin{/_i} \mu' \qquad e_j = e_j' \qquad \{i,j\} = \{1,2\}}{\langle e_1 \mid e_2 \rangle \mathbin{/} \mu \to \langle e_1' \mid e_2' \rangle \mathbin{/} \mu'} \quad \text{(bracket)}$$

Lifting

$$\langle v_1 \mid v_2 \rangle\ v \mathbin{/} \mu \to \langle v_1\ \lfloor v \rfloor_1 \mid v_2\ \lfloor v \rfloor_2 \rangle \mathbin{/} \mu \quad \text{(lift-app)}$$

$$\text{if } \langle v_1 \mid v_2 \rangle \text{ then } e_2 \text{ else } e_3 \mathbin{/} \mu \to \langle \text{if } v_1 \text{ then } \lfloor e_2 \rfloor_1 \text{ else } \lfloor e_3 \rfloor_1 \mid \text{if } v_2 \text{ then } \lfloor e_2 \rfloor_2 \text{ else } \lfloor e_3 \rfloor_2 \rangle / \mu \quad \text{(lift-if)}$$

**Figure 3.7**: Small dual-valued functional language semantic rules

rules are standard functional language semantics that are similar to Pierce's [40]. Our application rule for both the first and second expression. We choose to use a lambda abstraction here because our small language does not currently have any types. Our if rules for if expression reduction followed by our if true and if false rules that select the appropriate branch. The bracket rule is similar to Core ML$^2$ original bracket rule. The i subscript in our semantic rules represent the Core ML$^2$ idea of a configuration index. Rules without the subscript i are assumed to be taking place at

top level. The store is represented by the greek letter $\mu$ in our rules.

For dealing with our brackets inside of our small dual-valued functional language we have two unique lift rules. The lift rules allow us to lift the bracket out distributing the expression across the two sides of the bracket to prevent from halting reduction of the language. The first is the application lift rule, similar to the Core ML$^2$ original rule, which will take a value bracket expression and split it into two applications within the bracket followed by the projection of the second value term. The lift if rule allows us to lift the bracket out of the if reduction by taking the bracket value and split it across the two values, creating two separate if expressions on either side of the bracket. This will allow us to execute the appropriate if expression on each side of the bracket, taking the appropriate projection of the then and else expressions. You will notice that the lift rules only operate at the top level expression and never within a bracket itself.

In Figure 3.8 we show our encoded small-step reduction semantics for our language as encoded in Coq. The inductive type is represented by a tuple in Coq that is of the expression, a store and the configuration index. These relate directly to the Core ML$^2$ idea of a configuration. The majority of the rules represent the basic ML reduction semantics seen before with the small-step reduction semantics for if, application, and lambda abstraction which can by found in Pierce et al. [41]. The bracket rules differ slightly from their definitions in CoreML$^2$. Pottier and Simonet, define the original rule to execute non-deterministically. However, in our implementation we allow the bracket to reduce in a deterministic manner. This is solely to make the implementation into the Coq language easier to deal with.

```
Inductive step : Exp * store * config_index → Exp * store * config_index →
   Prop :=
  | ST_If : ∀ e1 e1' e2 e3 st st' i,
      e1 / st @ i ⇒ e1' / st' @ i →
      (If e1 e2 e3) / st @ i ⇒ (If e1' e2 e3) / st' @ i
  | ST_IfTrue : ∀ e1 e2 st i,
      If True e1 e2 / st @ i ⇒ e1 / st @ i
  | ST_IfFalse : ∀ e1 e2 st i,
      If False e1 e2 / st @ i ⇒ e2 / st @ i
  | ST_App1 : ∀ e1 e1' e2 st st' i,
      e1 / st @ i ⇒ e1' / st' @ i →
      (App e1 e2) / st @ i ⇒ (App e1' e2) / st' @ i
  | ST_AppLam : ∀ id e1 e2 st i,
       value e2 →
      (App (Lambda id e1) e2) / st @ i ⇒ [id := e2] e1 / st @ i
  | ST_App2 : ∀ v1 e2 e2' st st' i,
      value v1 →
      e2 / st @ i ⇒ e2' / st' @ i →
      (App v1 e2) / st @ i ⇒ (App v1 e2') / st' @ i
  | ST_Bracket : ∀ e1 e1' e2 st st',
      e1 / st @ (cfg_index fst) ⇒ e1' / st' @ (cfg_index fst) →
      Bracket e1 e2 / st @ top_lvl ⇒ Bracket e1' e2 / st' @ top_lvl
  | ST_Bracket2 : ∀ v1 e2 e2' st st',
      value v1 →
      e2 / st @ (cfg_index snd) ⇒ e2' / st' @ (cfg_index snd) →
      Bracket v1 e2 / st @ top_lvl ⇒ Bracket v1 e2' / st' @ top_lvl
  | ST_AppLift : ∀ v1 v2 v st,
      value v1 →
      value v2 →
      value v →
      (App (Bracket v1 v2) v) / st @ top_lvl ⇒ (Bracket (App v1 (bracket_proj
    v fst)) (App v2 (bracket_proj v snd))) / st @ top_lvl
  | ST_IfLift : ∀ v1_1 v1_2 e2 e3 st,
      value v1_1 →
      value v1_2 →
      (If (Bracket v1_1 v1_2) e2 e3) / st @ top_lvl
        ⇒ (Bracket (If v1_1 (bracket_proj e2 fst) (bracket_proj e3 fst))
                   (If v1_2 (bracket_proj e2 snd) (bracket_proj e3 snd)))
            / st @ top_lvl
   where " e1 '/' st1 '@' i1 '⇒' e2 '/' st2 '@' i2 " := (step (e1,st1,i1) (e2
    ,st2,i2)).
```

**Figure 3.8**: Small-step operational semantic rules in Coq

# CHAPTER 4: SEMANTIC THEOREMS

In order to show that our small dual-valued functional language is a valid construct to reason about dual-values, we need to show that the bracket reduction semantics can be reduced in a fashion similar to a language without dual-values. We must also show that if we can reduce both projections of a bracket expression through reduction that we can reduce the expression in whole. In order to do this we will prove our small dual-valued functional language semantics sound and complete relative to single-valued semantics using Coq. The proofs that we will use are based from the original proofs, Lemmas 1 - 4, in Pottier and Simonet [42]. However, our proofs will be built so that they work within the Coq environment proving our small dual-valued functional language complete and sound. The complete Coq source code is listed in the appendix for convenience.

## 4.1 Theorem 1: Configuration Projection Projects Store

Theorem one shows that if you take a step at the first or second projection then you can also take a step at the top level with the projection of the store at first or second. This proof for our small dual-valued functional language is trivial to solve because our language does not contain any memory reference cell operations. However, we provide this proof for completeness, and as a stepping stone for our second theorem which will use theorem one to prove the semantic rules around a bracket expression sound.

**Theorem 1.**

$$Let\ i \in \{1,\ 2\}.\ If\ e\ /_i\ \mu \rightarrow e'\ /_i\ \mu',\ then\ e\ /\ \lfloor \mu \rfloor_i \rightarrow e'\ /\ \lfloor \mu' \rfloor_i.$$

*Proof.* To prove, it is a trivial matter of performing induction over the semantic rules showing that both steps are equivalent. □

In Figure 4.1 we show theorem one as represented in Coq. The proof in Coq is the same.

```
Theorem One: ∀ i e e' st st',
  e / st @ (cfg_index i) ⇒ e' / st' @ (cfg_index i) →
  e / (store_proj st i) @ top_lvl ⇒ e' / (store_proj st' i) @ top_lvl.
```

**Figure 4.1**: Theorem 1

## 4.2 Theorem 2: Soundness

The second theorem for our small dual-valued functional language will show that our language is indeed sound for reasoning about dual-values. The purpose of this theorem will be to show that if we can take a step at the top level of our language then we can also take a step at the configuration projection of i. The configuration projection of i is equivalent to the bracket projection at i and the store projection at i. Our definition of theorem two is slightly different from Pottier and Simonet's. We found, while formalizing the theorem in Coq, the theorem doesn't cover the lift rules that are defined within the Core ML$^2$ semantics. This may have been an oversight of the original authors. In our definition of theorem two we have corrected this oversight by adding an or clause that covers the semantic transformations of the lift rules.

**Theorem 2.**

$$Let\ i \in \{1,\ 2\}.\ If\ e\ /\ \mu \rightarrow e'\ /\ \mu',\ then\ \lfloor e\ /\ \mu \rfloor_i \rightarrow \lfloor e'\ /\ \mu' \rfloor_i\ or\ (\lfloor e \rfloor_i = \lfloor e' \rfloor_i\ and\ \lfloor \mu \rfloor_i = \lfloor \mu' \rfloor_i).$$

*Proof.* Through induction on the semantic step, for each case we apply the corresponding semantic step rule. In the case of the bracket rules we apply the semantic rule to the corresponding projection while using theorem one to prove the other projection. For each lift rule the expressions and stores do not change, therefore they are equivalent. □

In Figure 4.2 we show the definition of theorem two as represented in Coq. The proof of theorem two is similar in Coq as is above. However, two lemmas were also needed in order to prove theorem two. We needed a lemma that showed that if we have a value term that it also implies the bracket projection of that value, since the bracket projection of a value is a value. The second

```
Theorem Two: ∀ i e e' st st',
  e / st @ top_lvl ⇒ e' / st' @ top_lvl →
  ((bracket_proj e i) / (store_proj st i) @ top_lvl ⇒
       (bracket_proj e' i) / (store_proj st' i) @ top_lvl)
    \/ ((bracket_proj e i) = (bracket_proj e' i)
     /\  (store_proj st i) = (store_proj st' i)).
```

**Figure 4.2**: Theorem 2

lemma needed to prove theorem two is showing that the bracket projection is distributive when substituting variables. This allowed us to show that variable substitution could still be processed correctly within a bracket, thus proving the lambda abstraction condition in theorem two. These lemma definitions are shown in Figure 4.3

```
Lemma bracket_proj_value: ∀ v i,
 value v →
 value (bracket_proj v i).

Lemma bracket_proj_subst_distributive: ∀ x e1 e2 i,
  bracket_proj (subst x e1 e2) i = (subst x (bracket_proj e1 i) (bracket_proj
    e2 i)).
```

**Figure 4.3**: Theorem 2 helper lemmas

## 4.3 Theorem 3: Stuckness Preserved by a Projection

Theorem three is the inverse of theorem two. It uses contradiction to prove that if a step is stuck that its projection is also stuck. Showing that our small dual-valued functional language is correct for theorem three serves as a check that getting stuck in our small dual-valued functional language corresponds to getting stuck in one of the corresponding pairs of single value programs. The Coq version of this theorem proved to be challenging. Coq represents false statements as implications making several of Coq's built in commands to do inversion ineffective for a contradiction proof of this size. Because of this, it took many trials of redesigning the theorem for successful completion within Coq.

**Theorem 3.**

$$If\ e\ /\ \mu\ is\ stuck,\ then\ \lfloor e\ /\ \mu \rfloor_i\ is\ stuck\ for\ some\ i \in \{1,\ 2\}.$$

*Proof.* By contradiction. Induction over the expresion e.

- If e is a value then e must be true, false, lambda abstraction, bracket value, or void. Void is not permitted as an expression and the other values are in their normal forms with reduction complete. Therefore, they are not stuck.
- If e is an application then e1 must be some sort of value, otherwise it would step. If e1 were a lambda abstraction or bracket then either the beta reduction or the lift rule would apply. If we have a bracket and take the projection of that bracket for i we would have another value term since the bracket itself is a value. The same reasoning would apply to e2 as a value term. Therefore we are stuck.
- The if follows similar reasoning. If the e1 is not a value it would step, as a value true or false would apply one of the If rules, a bracket would apply the lift rule. Since none of these rules apply, we are stuck.
- The bracket would be reducible by either bracket rule. But if none apply, then the bracket is a value. The projection at some i on that bracket would give us another non-bracketed value term showing that we are stuck.

□

```
Theorem Three: ∀ e,
   stuck e → (∃ i, stuck (bracket_proj e i)).
```

**Figure 4.4**: Theorem 3

The Coq definition for theorem three is listed in Figure 4.4. In order to reason about theorem three in Coq we needed to define what stuck meant. To do this we used the definitions and the

inductive types in Figure 4.5. The first two definitions are similar to definitions from Pierce et al. [41]. They define in Coq what a normal form is and what the stuck representation of a semantic should be. We use these definitions later for some helper lemmas to prove theorem three. The last three inductive definitions define syntactically, the different stuck scenarios that can take place within our language. By defining these inductive types syntactically, we can let Coq use the syntactic structure to do appropriate case analysis. Without dealing directly with the somewhat complicated definition of stuck (with its negation and existential quantification). This allowed us to take advantage of Coq's inversion command that was able to abstract the required information from the definitions that was needed to complete the proof.

```
Definition normal_form (e:Exp)(st:store) : Prop :=
  ~ ∃ e' st', e/st @top_lvl ⇒ e'/st' @top_lvl.

Definition stuck_semantic (e:Exp)(st:store) : Prop :=
  normal_form e st /\ ~ value e.

Inductive valueNotLambdaOrBracket: Exp → Prop :=
  | vnl_True : valueNotLambdaOrBracket True
  | vnl_False : valueNotLambdaOrBracket False
  | vnl_Void : valueNotLambdaOrBracket Void.

Inductive valueNotTrueFalseBracket: Exp → Prop :=
  | vntf_Void: valueNotTrueFalseBracket Void
  | vntf_Lambda: ∀ x e, valueNotTrueFalseBracket (Lambda x e).

Inductive stuck: Exp → Prop :=
  | Stk_Var: ∀ id, stuck (Var id)
  | Stk_App1: ∀ e1 e2, stuck e1 → stuck (App e1 e2)
  | Stk_App2: ∀ e1 e2, value e1 → stuck e2 → stuck (App e1 e2)
  | Stk_AppNotLam: ∀ e1 e2,
      valueNotLambdaOrBracket e1 → value e2 → stuck (App e1 e2)
  | Stk_Ifstk: ∀ e1 e2 e3, stuck e1 → stuck (If e1 e2 e3)
  | Stk_Ifval: ∀ e1 e2 e3,
      valueNotTrueFalseBracket e1 → stuck (If e1 e2 e3)
  | Stk_Bracket1: ∀ e1 e2, stuck e1 → stuck (Bracket e1 e2)
  | Stk_Bracket2: ∀ e1 e2, value e1 →
     stuck e2 → stuck (Bracket e1 e2).
```

**Figure 4.5**: Stuck definitions and inductive types

In Figure 4.6 we have several of our helper lemmas the first two were simple lemmas. Both lemmas allowed us to prove certain conditions to be false in Coq. The first value_not_step de-

fines that if we have a value condition then that value cannot take a step. The not_stuck_value lemma proves that for stuck expression that expression cannot be a value. These lemmas provided completion to several cases that where generated by Coq.

The last three lemmas establish the equivalence between the semantic definition of stuck (stuck _semantic) and syntactic definition of stuck (stuck). The first lemma stuck_is_stuck_semantic proves that if we have a syntactically stuck expression then that implies that the expression is stuck semantically. To prove this we had to specialize our induction hypotheses for the given configuration index values given to make sure that we could satisfy the exists i portion of the original theorem. Since a bracket expression is not completely stuck, it may still step through one of its projections while the other projection is stuck. Lemma stuck_semantic_is_stuck proves the inverse relationship between the syntactically stuck expression and the stuck semantic definition. This direction of the relationship required that we come up with a reasoning construct in Coq that would allow us to show a well-formed configuration. Since Coq wants to prove "for all" values of a given index, even though only one index value is valid at one time. The stuck_is_stuck lemma essential shows the if and only if relationship between the syntactically stuck expression and stuck semantic definition. This shows that our definitions about being stuck are complete and sound.

```
Lemma value_not_step: ∀ e1,
  value e1 → ∀ st1 e2 st2 i,
  e1 / st1 @ i ⇒ e2 /st2 @ i →
  Logic.False.

Lemma not_stuck_value: ∀ e, stuck e → ~ value e.

Lemma stuck_is_stuck_semantic: ∀ e st, stuck e → stuck_semantic e st.

Lemma stuck_semantic_is_stuck:
  ∀ e st, well_formed e → stuck_semantic e st → stuck e.

Lemma stuck_is_stuck:
  ∀ e st,
    well_formed e → (stuck_semantic e st ↔ stuck e).
```

**Figure 4.6**: Stuck helper lemmas

To complete the stuck semantic implies stuck, we need to come up with an inductive type to

reason syntactically about well-formed configurations. We also provided two lemmas that show that the reasoning about a well-formed configuration gives us a well-formed expression. The definitions for these are in Figure 4.7. The first is a inductive type definition that defines what a well-formed configuration is in Coq. The following two lemmas show the if and only if relationships between a well-formed configuration and a well-formed expression. We do this for both the projection index and the top level index, showing that all possible index values are covered.

```
Inductive well_formed_cfg: Exp → config_index → Prop :=
  | wfc_true : ∀ idx, well_formed_cfg True idx
  | wfc_false : ∀ idx, well_formed_cfg False idx
  | wfc_var : ∀ x idx , well_formed_cfg (Var x) idx
  | wfc_app : ∀ e1 e2 idx,
    well_formed_cfg e1 idx →
    well_formed_cfg e2 idx →
    well_formed_cfg (App e1 e2) idx
  | wfc_lambda : ∀ x e idx,
    well_formed_cfg e idx →
    well_formed_cfg (Lambda x e) idx
  | wfc_if : ∀ e1 e2 e3 idx,
    well_formed_cfg e1 idx →
    well_formed_cfg e2 idx →
    well_formed_cfg e3 idx →
    well_formed_cfg (If e1 e2 e3) idx
  | wfc_bracket_cfg: ∀ e1 e2,
        well_formed_cfg e1 fst →
        well_formed_cfg e2 snd →
        well_formed_cfg (Bracket e1 e2) top_lvl.

Lemma wf__wfc_pi: ∀ e pi, (not_bracket e /\ well_formed e) ↔ well_formed_cfg
   e (cfg_index pi).

Lemma wf__wfc_top: ∀ e, well_formed e ↔ well_formed_cfg e top_lvl.
```

**Figure 4.7**: Well-formed configuration definition and lemma definitions

## 4.4 Theorem 4: Completeness

Now that we have established that our small dual-valued functional language is sound for reasoning about dual-values we must now show that it is complete. To do this we will show that our language has supplied enough lift rules to reduce all possible expressions in our language. Again, we will prove by using theorem two and theorem three but the formalized Coq version of our theorem will

be much simpler.

**Theorem 4.**

$$Assume \lfloor e \mathbin{/} \mu \rfloor_i \rightarrow^* v_i \mathbin{/} \mu_i' \; for \; all \; i \in \{1, 2\}.$$

$$Then, \; there \; exists \; a \; configuration \; v \mathbin{/} \mu' \; such \; that \; e \mathbin{/} \mu \rightarrow^* v \mathbin{/} \mu'.$$

*Proof.* To start let us remark that the lift rules are the only rules that leave an expressions projection unchanged and that no reduction sequence can be made up entirely of lift rules. Let's assume we have an infinite reduction sequence. Then by theorem two we have an infinite reduction sequence for some projection index of i. But this is not possible because we know that the configuration projection of i when i is 1 and 2 can be reduced because this would be equivalent to our small dual-valued functional language without brackets. Let us also assume that our expression reduces to an irreducible configuration and it is stuck. Then we know that from theorem three only one of its projections is stuck and that theorem two reduces to a stuck configuration for some projection index. Therefore, we have a contradiction and can reduce to a successful configuration because some such projection index is successful. □

```
Theorem Four: ∀ e st,
  (∀ i,
    (∃ v st',
       value v →
      (bracket_proj e i) / (store_proj st i) @ top_lvl
          ⇒* v / st' @ top_lvl)) →
  (∃ v' st'',
    value v' →
    e / st @ top_lvl ⇒* v' / st'' @ top_lvl).
```

**Figure 4.8**: Theorem 4

The Coq definition for theorem four is given in Figure 4.8. We use a multi-step reduction definition that is borrowed from Pierce et al. [41]. This operator allows us to reason in Coq in a zero or more steps fashion. Because $v'$ is an existential variable in Coq we can easily assert the reduction semantic we wish to reduce too. Then it is a simple matter of showing in Coq that the

expression that we started with reduces in zero or more steps which can be done by showing that the before and after of the expression is reflexive.

# CHAPTER 5: CONCLUSION

Through our work we have been successful in showing how a small dual-valued functional language syntax and semantics can be developed and implemented with Coq. We have shown how to construct such a language in Coq and have provided formal machine-checked proofs to show that our language is correct and sound. Throughout proving the correctness of our small dual-valued functional language we discovered some oversights from the original proofs done on Core ML$^2$. Although, not big oversights, they do show the importance and the power of using a machine-checked prover such as Coq, to formally verify such work.

## 5.1 Future Work

### 5.1.1 Language Improvements

Although we have seen some success at formally proving and providing proofs for a small dual-valued functional language there are many more aspects to this language that need to be accomplished. The first will be to expand the semantics of our small dual-valued functional language to represent the full semantics of the original Core ML$^2$ that is used by Pottier and Simonet. This will allow us to expand and complete the initial proofs provided and even expand upon the functional language itself for future research into non-deterministic implementations and other distributive architectures.

We also need to look at adding a type system to our small dual-valued functional language and within this type system provide the needed information flow security information. This will allow us to provide a formalization of the type system proofs within Coq and prove non-interference for a functional language. This will allow us to then expand upon the type system for more advanced typing inference with information flow aspects.

### 5.1.2 Adapting to the Cloud

As organizations struggle with the need to expand their systems to cover increasing demand, they are also looking for ways to decrease costs. This has led to an increase in cloud based technologies and integration with other systems; creating multi-party distributed platforms. The benefit is it allows greater sharing of data, but the negative is, that now this data and potentially trade secrets in the form of code could be transferred to an untrusted host system that may be compromised or at risk of becoming compromised.

Distributed cloud systems are gaining popularity for many of today's computer software and infrastructure service providers. These systems can be found hosting enterprise level websites, processing millions of transactions per day. They provide clusters of systems that can work together to process large quantities of data, balancing the load across those systems, and allowing large workloads to be scaled across available resources. Hierarchies of machines can be built, that share data from one tier to the next, each working seamlessly to create an assembly line of information flow that can pass information between several systems, owned by different entities, located in different countries around the world. An attacker just needs to compromise one area in a distributed cloud system to tap into that information flow. That information may be insignificant to an organization or may be critical, maybe even life threating. In this case, it is important for that organization to be able to protect that information flow, control who sees that information and who may change that information [12]. Organizations today need a more automated way of protecting their critical information from attackers, especially when that information may have to be routed through an untrusted host(s) of a distributed cloud system.

Information flow research to date, has covered several aspects of partitioning by using compilers to analyze the location of code produced. This has led to several models that ensure proper flow between the server and client, originally targeted by the compiler, but lacks the ability to dynamically change these information flow relationships based on changes in the runtime environment(s) of those systems. Cryptographic approaches to information flow encrypt/decrypt data between

servers and clients using either third-party APIs or the hosts TPM for the cryptographic primitives. However, this approach assumes trust of these components which may also be vulnerable to attack and can suffer from the same problems as partitioning. Both of these approaches assume trust in the hosts that the code is running on and ignore the threat of an insider attack. Because of this both approaches are not flexible enough to prevent such an attack.

By improving on our language for the cloud we could prove that non-interference for information flows across distributed systems is possible. By doing this we could, potentially, create a new distributed functional language that could be used for passing data across systems in the cloud in a secure manner. This will allow us to protect the confidentiality of that data from attackers that would like gain access to it. Allowing us to use untrusted host systems in the cloud without fear of having our data stolen or compromised.

Research done for information flow within a distributed environment may provide clues to creating such a language. More recent work done for RPC in distributed systems using information flow methods can be found in [1, 35, 56]. Though it is also very possible that language that solves these problems may be found within research for advanced concurrency and actor based systems [8, 21,43,46,47,54]. These type of systems make up the heart of a distributed application environment. They may also hold the key to producing an information flow based language that can operate in such an environment.

# Appendix A: COQ CODE

## A.1 FML.v

```
Require Export SfLib.

Inductive proj_index: Type :=
  | fst: proj_index
  | snd: proj_index.

Inductive config_index: Type :=
  | top_lvl: config_index
  | cfg_index: proj_index → config_index.

Definition proj_index_config_index (i:proj_index) := (cfg_index i).
Coercion proj_index_config_index : proj_index >→ config_index.

(*Syntax*)
Inductive Exp : Type :=
  | True : Exp
  | False : Exp
  | Var: id → Exp
  | App: Exp → Exp → Exp
  | Lambda : id → Exp → Exp
  | If: Exp → Exp → Exp → Exp
  | Void: Exp
  | Bracket: Exp → Exp → Exp.

Tactic Notation "e_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "True" | Case_aux c "False"
  | Case_aux c "Var" | Case_aux c "App" | Case_aux c "Lambda" | Case_aux c "If
```

```
    "
  | Case_aux c "Void"  | Case_aux c "Bracket"].

(* Bracket Functions *)
Fixpoint bracket_proj (e:Exp) (i:proj_index) : Exp :=
  match e with
  | Bracket e1 e2 ⇒
    match i with
    | fst ⇒ e1
    | snd ⇒ e2
    end
  | True ⇒ e
  | False ⇒ e
  | Var x ⇒ e
  | App e1 e2 ⇒ App (bracket_proj e1 i) (bracket_proj e2 i)
  | Lambda x e1 ⇒ Lambda x (bracket_proj e1 i)
  | If e1 e2 e3 ⇒ If (bracket_proj e1 i) (bracket_proj e2 i) (bracket_proj e3
    i)
  | Void ⇒ e
  end.

Inductive not_bracket : Exp → Prop :=
  | nb_true : not_bracket True
  | nb_false : not_bracket False
  | nb_var : ∀ x, not_bracket (Var x)
  | nb_app : ∀ e1 e2,
    not_bracket e1 →
    not_bracket e2 →
    not_bracket (App e1 e2)
  | nb_lambda : ∀ x e,
    not_bracket e →
    not_bracket (Lambda x e)
```

```
  | nb_if : ∀ e1 e2 e3,
    not_bracket e1 →
    not_bracket e2 →
    not_bracket e3 →
    not_bracket (If e1 e2 e3)
  | nb_void : not_bracket (Void).

Hint Constructors not_bracket.

Inductive not_void : Exp → Prop :=
  | nv_true : not_void True
  | nv_false : not_void False
  | nv_var : ∀ x, not_void (Var x)
  | nv_app : ∀ e1 e2,
    not_void e1 →
    not_void e2 →
    not_void (App e1 e2)
  | nv_lambda : ∀ x e,
    not_void e →
    not_void (Lambda x e)
  | nv_if : ∀ e1 e2 e3,
    not_void e1 →
    not_void e2 →
    not_void e3 →
    not_void (If e1 e2 e3)
  | nv_bracket : ∀ e1 e2, not_void (Bracket e1 e2).

Hint Constructors not_void.

Inductive well_formed : Exp → Prop :=
  | wf_true : well_formed True
  | wf_false : well_formed False
```

```
  | wf_var : ∀ x, well_formed (Var x)
  | wf_app : ∀ e1 e2,
    well_formed e1 →
    well_formed e2 →
    well_formed (App e1 e2)
  | wf_lambda : ∀ x e,
    well_formed e →
    well_formed (Lambda x e)
  | wf_if : ∀ e1 e2 e3,
    well_formed e1 →
    well_formed e2 →
    well_formed e3 →
    well_formed (If e1 e2 e3)
  | wf_bracket: ∀ e1 e2, (* not_void e1 → not_void e2 → *)
        not_bracket e1 → well_formed e1 →
        not_bracket e2 → well_formed e2 →
        well_formed (Bracket e1 e2).

Hint Constructors well_formed.

Inductive value : Exp → Prop :=
  | v_True : value True
  | v_False : value False
  | v_Func : ∀ x, ∀ e, value (Lambda x e)
  | v_Void : value Void
  | v_Bracket : ∀ v1 v2,
    value v1 →
    value v2 →
    value (Bracket v1 v2).

Hint Constructors value.
```

```
Fixpoint subst (x:id) (s:Exp) (t:Exp) : Exp :=
  match t with
  | Var x' ⇒
      if beq_id x x' then s else t
  | App e1 e2 ⇒
    App (subst x s e1) (subst x s e2)
  | If e1 e2 e3 ⇒
    If (subst x s e1) (subst x s e2) (subst x s e3)
  | Lambda y e1 ⇒
    Lambda y (if beq_id x y then e1 else (subst x s e1))
  | True ⇒ True
  | False ⇒ False
  | Void ⇒ Void
  | Bracket e1 e2 ⇒
    Bracket (subst x (bracket_proj s fst) e1) (subst x (bracket_proj s snd) e2
   )
  end.

Notation "'[' x ':=' s ']' t" := (subst x s t) (at level 20).

(* Stores *)
Definition store := list Exp.
Fixpoint store_proj (l:list Exp) (i:proj_index) : list Exp :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ (bracket_proj h i) :: (store_proj t i)
  end.

Reserved Notation "e1 '/' st1 '@' i '⇒' e2 '/' st2 '@' i2" (at level 40, st1
   at level 39, e2 at level 39).

Inductive step : Exp * store * config_index → Exp * store * config_index →
```

```
 Prop :=
| ST_If : ∀ e1 e1' e2 e3 st st' i,
    e1 / st @ i ⇒ e1' / st' @ i →
    (If e1 e2 e3) / st @ i ⇒ (If e1' e2 e3) / st' @ i
| ST_IfTrue : ∀ e1 e2 st i,
    If True e1 e2 / st @ i ⇒ e1 / st @ i
| ST_IfFalse : ∀ e1 e2 st i,
    If False e1 e2 / st @ i ⇒ e2 / st @ i
| ST_App1 : ∀ e1 e1' e2 st st' i,
    e1 / st @ i ⇒ e1' / st' @ i →
    (App e1 e2) / st @ i ⇒ (App e1' e2) / st' @ i
| ST_AppLam : ∀ id e1 e2 st i,
     value e2 →
    (App (Lambda id e1) e2) / st @ i ⇒ [id := e2] e1 / st @ i
| ST_App2 : ∀ v1 e2 e2' st st' i,
    value v1 →
    e2 / st @ i ⇒ e2' / st' @ i →
    (App v1 e2) / st @ i ⇒ (App v1 e2') / st' @ i
| ST_Bracket : ∀ e1 e1' e2 st st',
    e1 / st @ (cfg_index fst) ⇒ e1' / st' @ (cfg_index fst) →
    Bracket e1 e2 / st @ top_lvl ⇒ Bracket e1' e2 / st' @ top_lvl
| ST_Bracket2 : ∀ v1 e2 e2' st st',
    value v1 →
    e2 / st @ (cfg_index snd) ⇒ e2' / st' @ (cfg_index snd) →
    Bracket v1 e2 / st @ top_lvl ⇒ Bracket v1 e2' / st' @ top_lvl
| ST_AppLift : ∀ v1 v2 v st,
    value v1 →
    value v2 →
    value v →
    (App (Bracket v1 v2) v) / st @ top_lvl ⇒ (Bracket (App v1 (bracket_proj
  v fst)) (App v2 (bracket_proj v snd))) / st @ top_lvl
| ST_IfLift : ∀ v1_1 v1_2 e2 e3 st,
```

```
      value v1_1 →
      value v1_2 →
      (If (Bracket v1_1 v1_2) e2 e3) / st @ top_lvl
        ⇒ (Bracket (If v1_1 (bracket_proj e2 fst) (bracket_proj e3 fst))
                   (If v1_2 (bracket_proj e2 snd) (bracket_proj e3 snd)))
            / st @ top_lvl
   where " e1 '/' st1 '@' i1 '⇒' e2 '/' st2 '@' i2 " := (step (e1,st1,i1) (e2
    ,st2,i2)).

Tactic Notation "step_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "ST_If" | Case_aux c "ST_IfTrue" | Case_aux c "ST_IfFalse"
  | Case_aux c "ST_App1" | Case_aux c "ST_AppLam"
  | Case_aux c "ST_App2" | Case_aux c "ST_Bracket" | Case_aux c "ST_Bracket2"
  | Case_aux c "ST_AppLift" | Case_aux c "ST_IfLift" ].

Hint Constructors step.

Notation multistep := (multi step).
Notation "e '/' st '@' i '⇒⋆' e' '/' st' '@' i'" := (multistep (e,st,i) (e',
    st',i')) (at level 40, st at level 39, e' at level 39).
```

## A.2 Theorem1.v

```
Require Export Coq.Program.Equality.
Require Import FML.

Theorem One: ∀ i e e' st st',
  e / st @ (cfg_index i) ⇒ e' / st' @ (cfg_index i) →
  e / (store_proj st i) @ top_lvl ⇒ e' / (store_proj st' i) @ top_lvl.
Proof with eauto.
intros.
(dependent induction H); intros; subst...
```

```
Qed.
```

## A.3 Theorem2.v

```
Require Import FML.
Require Import Theorem1.

Lemma bracket_proj_subst_distributive: ∀ x e1 e2 i,
  bracket_proj (subst x e1 e2) i = (subst x (bracket_proj e1 i) (bracket_proj
   e2 i)).
Proof with eauto.
intros.
e_cases (induction e2) Case; simpl...
Case "Var".
  remember (beq_id x i0) as f.
  destruct f; simpl...
Case "App".
  rewrite IHe2_1.
  rewrite IHe2_2...
Case "Lambda".
  remember (beq_id x i0) as f.
  destruct f; simpl...
  rewrite IHe2...
Case "If".
  rewrite IHe2_1; rewrite IHe2_2; rewrite IHe2_3...
Case "Bracket".
  destruct i; subst...
Qed.

Lemma bracket_proj_value: ∀ v i,
 value v →
 value (bracket_proj v i).
Proof with eauto.
```

```
intros.
e_cases (induction v) Case; simpl...
Case "App".
  inversion H.
Case "If".
  inversion H.
Case "Bracket".
  destruct i; subst...
  inversion H... inversion H...
Qed.

Lemma store_bproj_equiv: ∀ e e' st st' i1 i2,
   i1 <> i2 →
   e / st @ (cfg_index i1) ⇒ e' / st'@ (cfg_index i1) →
  (store_proj st i2) = (store_proj st' i2).
Proof with eauto.
intros.
(dependent induction H0); simpl...
Qed.

Theorem Two: ∀ i e e' st st',
  e / st @ top_lvl ⇒ e' / st' @ top_lvl →
  ((bracket_proj e i) / (store_proj st i) @ top_lvl ⇒ (bracket_proj e' i) / (
   store_proj st' i) @ top_lvl)
    \/ ((bracket_proj e i) = (bracket_proj e' i) /\  (store_proj st i) = (
   store_proj st' i)).
Proof with eauto.
intros.
step_cases (dependent induction H) Case; intros.
Case "ST_If".
  inversion IHstep.
  SCase "IHstep ⇒".
```

```
    left. apply ST_If. apply H0.
  SCase "IHstep =".
    right. inversion H0. simpl. rewrite ← H1. rewrite ← H2. eauto.
Case "ST_IfTrue".
  left. apply ST_IfTrue.
Case "ST_IfFalse".
  left. apply ST_IfFalse.
Case "ST_App1".
  inversion IHstep.
  SCase "IHstep ⇒".
    left. apply ST_App1. apply H0.
  SCase "IHstep =".
    right. inversion H0. simpl. rewrite ← H1. rewrite ← H2. eauto.
Case "ST_AppLam".
  left.
  rewrite bracket_proj_subst_distributive.
  apply ST_AppLam.
  apply bracket_proj_value...
Case "ST_App2".
  inversion IHstep.
  SCase "IHstep ⇒".
    left. apply ST_App2. apply bracket_proj_value. apply H. apply H1.
  SCase "IHstep =".
    right. inversion H1. simpl. rewrite ← H2. rewrite ← H3. eauto.
Case "ST_Bracket".
  destruct i.
  SCase "i = fst".
    left.
    apply One. apply H.
  SCase "i = snd".
    right. split. reflexivity.
    apply store_bproj_equiv with (e:=e1)(e':=e1')(i1:=fst).
```

```
    unfold not. intros. inversion H0.
    apply H.
Case "ST_Bracket2".
  destruct i.
  SCase "i = fst".
    right. split. reflexivity.
    apply store_bproj_equiv with (e:=e2)(e':=e2')(i1:=snd).
    unfold not. intros. inversion H1.
    apply H0.
  SCase "i = snd".
    left.
    apply One. apply H0.
Case "ST_AppLift".
  destruct i.
  SCase "i = fst".
    right. split. reflexivity. reflexivity.
  SCase "i = snd".
    right. split. reflexivity. reflexivity.
Case "ST_IfLift".
  right.
  destruct i; split; reflexivity.
Qed.
```

## A.4 Theorem3.v

```
Require Import FML.
Require Import Theorem2.
Require Import Coq.Program.Equality.

Definition normal_form (e:Exp)(st:store) : Prop :=
  ~ ∃ e' st', e/st @top_lvl ⇒ e'/st' @top_lvl.

Definition stuck_semantic (e:Exp)(st:store) : Prop :=
```

```
  normal_form e st /\ ~ value e.

Inductive valueNotLambdaOrBracket: Exp → Prop :=
  | vnl_True : valueNotLambdaOrBracket True
  | vnl_False : valueNotLambdaOrBracket False
  | vnl_Void : valueNotLambdaOrBracket Void.

Inductive valueNotTrueFalseBracket: Exp → Prop :=
  | vntf_Void: valueNotTrueFalseBracket Void
  | vntf_Lambda: ∀ x e, valueNotTrueFalseBracket (Lambda x e).

Inductive stuck: Exp → Prop :=
  | Stk_Var: ∀ id, stuck (Var id)
  | Stk_App1: ∀ e1 e2, stuck e1 → stuck (App e1 e2)
  | Stk_App2: ∀ e1 e2, value e1 → stuck e2 → stuck (App e1 e2)
  | Stk_AppNotLam: ∀ e1 e2, valueNotLambdaOrBracket e1 → value e2 → stuck (
   App e1 e2)
  | Stk_Ifstk: ∀ e1 e2 e3, stuck e1 → stuck (If e1 e2 e3)
  | Stk_Ifval: ∀ e1 e2 e3, valueNotTrueFalseBracket e1 → stuck (If e1 e2 e3)
  | Stk_Bracket1: ∀ e1 e2, stuck e1 → stuck (Bracket e1 e2)
  | Stk_Bracket2: ∀ e1 e2, value e1 → stuck e2 → stuck (Bracket e1 e2).

Hint Constructors stuck.


Tactic Notation "stuck_cases" tactic(first) ident(c) :=
  first;
  [ Case_aux c "Stk_Var"
  | Case_aux c "Stk_App1"
  | Case_aux c "Stk_App2"
  | Case_aux c "Stk_AppNotLam"
  | Case_aux c "Stk_If"
```

```
  | Case_aux c "Stk_Bracket1"
  | Case_aux c "Stk_Bracket2" ].

Lemma value_not_step: ∀ e1,
  value e1 → ∀ st1 e2 st2 i,
  e1 / st1 @ i ⇒ e2 /st2 @ i →
  Logic.False.
Proof with eauto.
intros.
generalize dependent e2.
generalize dependent i.
e_cases (induction e1) Case; intros; try solve by inversion.
Case "Bracket".
  inversion H0.
  inversion H.
  specialize (IHe1_1 H10 (cfg_index fst) e1'). apply IHe1_1. assumption.
  inversion H.
  specialize (IHe1_2 H12 (cfg_index snd) e2'). apply IHe1_2. assumption.
Qed.

Lemma not_stuck_value: ∀ e, stuck e → ~ value e.
Proof with eauto.
  unfold not.
  intros.
  e_cases (induction e) Case; try solve by inversion 2.
  Case "Bracket".
    inversion H.
    SCase "stuck e1".
      inversion H0.
      apply IHe1; assumption.
    SCase "Stuck e2".
      inversion H0; apply IHe2; assumption.
```

```
Qed.

Lemma stuck_is_stuck_semantic: ∀ e st, stuck e → stuck_semantic e st.
Proof with eauto.
  intros e st.
  unfold stuck_semantic; unfold normal_form; unfold not.
  split.
    assert (∀ i,  stuck e → (∃ (e' : Exp) (st' : store), e / st @ i ⇒ e' /
    st' @ i) → Logic.False).
    clear H.
    e_cases (induction e) Case; intros cfg_idx Hstk H; destruct H; destruct H;
    try solve by inversion.
    Case "App".
      specialize (IHe1 cfg_idx).
      specialize (IHe2 cfg_idx).
      inversion Hstk.
      SCase "ST_App1".
        assert (Hnv := H1); apply not_stuck_value in Hnv.
        inversion H; try (exfalso; apply Hnv; assumption);
          try (apply IHe1 in H1; [apply H1 | eauto]);
          try solve by inversion 2;
          try (destruct (e1); solve by inversion).
            SSSCase "e1 = Bracket v1 v2".
              exfalso; apply Hnv; subst; try constructor; assumption.
      SCase "ST_App2".
        assert (Hnv := H3); apply not_stuck_value in Hnv.
        assert (Hns := value_not_step). specialize (Hns e1).
        subst.
        inversion H; try (exfalso; apply Hnv; assumption);
          try (apply IHe2 in H3; [apply H3 | eauto]);
          try solve by inversion 2.
          e_cases (destruct e1) SSCase; try solve by inversion.
```

```
      exfalso; apply Hns with st e1' x0 cfg_idx; assumption.
  SCase "ST_AppLamNotLam".
    inversion H; e_cases (destruct e1) SSCase; try solve by inversion;
eauto;
    try (exfalso; apply value_not_step with e2 st e2' x0 cfg_idx;
assumption).
 Case "If".
  clear IHe2 IHe3.
  inversion Hstk.
  SCase "e1 stuck".
    assert (Hnv := H1); apply not_stuck_value in Hnv.
    inversion H; try (apply Hnv; subst; constructor); eauto.
  SCase "valueNotTrueFalseBracket e1".
    subst.
    e_cases (destruct e1) SSCase; subst; try solve by inversion 2.
 Case "Bracket".
  specialize (IHe1 fst).
  specialize (IHe2 snd).
  inversion Hstk.
  SCase "Stk_Bracket1".
    assert (Hnv := H1); apply not_stuck_value in Hnv.
    inversion H.
    SSCase "ST_Bracket1".
      apply IHe1. assumption. eauto.
    SSCase "ST_Bracket2".
      contradiction.
  SCase "Stk_Bracket2".
    assert (Hnv := H3); apply not_stuck_value in Hnv.
    subst.
    inversion H.
    SSCase "ST_Bracket1".
      apply value_not_step in H1.
```

```
            eauto.
            assumption.
        SSCase "ST_Bracket2".
          apply IHe2 in H3; try contradiction.
            ∃ e2'. ∃ x0.
            assumption.
    apply H0; assumption.
  Case "stuck → not value".
    intros contra.
    e_cases (induction e) SCase; try solve by inversion.
      inversion contra.
      inversion H.
      SSCase "stuck e1".
        apply IHe1; assumption.
      SSCase "stuck e2".
        apply IHe2; assumption.
Qed.

(* Need to add hypothesis that e is well-formed to get that Bracket's
   only reduce under top_lvl. This may mean the the assertion
   used for the inductive hypothesis also needs to be adjusted,
   since Bracket only ∃ at top_lvl. *)
Inductive well_formed_cfg: Exp → config_index → Prop :=
  | wfc_true : ∀ idx, well_formed_cfg True idx
  | wfc_false : ∀ idx, well_formed_cfg False idx
  | wfc_var : ∀ x idx , well_formed_cfg (Var x) idx
  | wfc_app : ∀ e1 e2 idx,
    well_formed_cfg e1 idx →
    well_formed_cfg e2 idx →
    well_formed_cfg (App e1 e2) idx
  | wfc_lambda : ∀ x e idx,
    well_formed_cfg e idx →
```

```
    well_formed_cfg (Lambda x e) idx
  | wfc_if : ∀ e1 e2 e3 idx,
    well_formed_cfg e1 idx →
    well_formed_cfg e2 idx →
    well_formed_cfg e3 idx →
    well_formed_cfg (If e1 e2 e3) idx
  | wfc_bracket_cfg: ∀ e1 e2, (* not_void e1 → not_void e2 → *)
        well_formed_cfg e1 fst →
        well_formed_cfg e2 snd →
        well_formed_cfg (Bracket e1 e2) top_lvl.

Lemma wf__wfc_pi: ∀ e pi, (not_bracket e /\ well_formed e) ↔ well_formed_cfg
   e (cfg_index pi).
  split; intros.
  Case "→".
   inversion H as [Hnb Hwf].
   clear H.
   induction Hwf;
     inversion Hnb; subst;
     try apply IHHwf in H0;
     try apply IHHwf1 in H1; try apply IHHwf2 in H2;
     try apply IHHwf1 in H2; try apply IHHwf2 in H3; try apply IHHwf3 in H4;
     try (constructor; repeat assumption; inversion Hnb; fail).
   Case "←".
     split.
     SCase "show not_bracket".
       dependent induction H; try (constructor; repeat assumption; fail).
     SCase "show well_formed".
       dependent induction H; try (constructor; repeat assumption; fail).
Qed.

Lemma wf__wfc_top: ∀ e, well_formed e ↔ well_formed_cfg e top_lvl.
```

```
Proof.
 split; intros.
 Case "→".
   induction H; try (constructor; repeat assumption; fail).
   SCase "Bracket".
     constructor; try (rewrite ← wf__wfc_pi; split; assumption; fail).
 Case "←".
   induction H; try (constructor; repeat assumption; fail).
   SCase "Bracket".
     assert(∀ e pi, well_formed_cfg e (cfg_index pi) → not_bracket e) as Hnb.
       intros.
       rewrite ← wf__wfc_pi in H1.
       inversion H1.
       assumption.
     assert(∀ e pi, well_formed_cfg e (cfg_index pi) → well_formed e) as Hwf.
       intros.
       rewrite ← wf__wfc_pi in H1.
       inversion H1.
       assumption.
     constructor; eauto.
Qed.

Lemma stuck_semantic_is_stuck:
  ∀ e st, well_formed e → stuck_semantic e st → stuck e.
Proof.
unfold stuck_semantic.
unfold normal_form.
unfold not.
intros e st.
assert (∀ idx,
          (well_formed_cfg e idx) →
          ((∃ (e' : Exp) (st' : store), e / st @ idx ⇒ e' / st' @ idx) →
```

```
   Logic.False)
           /\ (value e → Logic.False) →
          stuck e).
Case_aux AssertionContext "generalize for induction".

  e_cases (induction e) Case; intros idx Hwf; intros;
    inversion Hwf; subst;
    inversion H as [HnoStep HnotVal];
    try (constructor; assumption; fail);
    try (exfalso; apply HnotVal; constructor; assumption; fail).
  Case "App".
    clear HnotVal.
    assert (He1 := (excluded_middle (value e1))).
    assert (He2 := (excluded_middle (value e2))).
    destruct He1 as [Hve1 | Hnve1].
    SCase "val e1".
      destruct He2 as [Hve2 | Hnve2]; try (repeat constructor; fail).
      SSCase "al e2".
        e_cases (destruct e1) SSSCase;
          inversion Hve1;
          try (apply Stk_AppNotLam; [repeat constructor; assumption |
   assumption]).
        SSSCase "Lambda".
          subst.
          exfalso; apply HnoStep.
          ∃ ([i := e2] e1).
          ∃ st.
          constructor; assumption.
        SSSCase "Bracket".
          exfalso.
          apply HnoStep.
          inversion H2.
```

```
        ∃ (Bracket (App e1_1 (bracket_proj e2 fst)) (App e1_2 (bracket_proj
  e2 snd))).
        ∃ st.
        apply ST_AppLift; assumption.
    SSCase "not val e2".
        apply Stk_App2; try assumption.
        apply (IHe2 idx); try assumption.
        split; try assumption.
           intros. apply HnoStep.
           destruct H0 as [e2' [st']].
           ∃ (App e1 e2').
           ∃ (st').
           apply ST_App2; assumption.
  SCase "not val e1".
    apply Stk_App1.
    apply (IHe1 idx); try assumption.
    split; try assumption.
    intros.
    apply HnoStep.
    destruct H0 as [e1' [st']].
    ∃ (App e1' e2). ∃ st'.
    apply ST_App1; try assumption.
Case "If".
  clear IHe2 IHe3.
  assert (HisVal := (excluded_middle (value e1))).
  destruct HisVal as [He1_val | He1_nv].
  SCase "He1_val".
    e_cases (destruct e1) SSCase;
      try (apply Stk_Ifval; repeat constructor; fail);
      try (exfalso; eauto; fail);
      try (solve by inversion).
    SSCase "Bracket".
```

```
      inversion H3.
      inversion He1_val; subst.
      exfalso. eauto.
  SCase "He1_nv".
    apply Stk_Ifstk.
    apply (IHe1 idx); try assumption.
    split; try assumption.
    intros.
    apply HnoStep.
    destruct H0 as [e1' [st']].
    eauto.
Case "Bracket".
    assert (His_e1_val := (excluded_middle (value e1))).
    assert (His_e2_val := (excluded_middle (value e2))).
    destruct His_e1_val as [He1Val | He1nv].
    SCase "He1Val".
      destruct (His_e2_val) as [He2Val | He2nv].
      SSCase "He2Val". exfalso. apply HnotVal. auto.
      SSCase "He2nv".
        apply Stk_Bracket2; try assumption.
        apply (IHe2 snd); try assumption.
        split; try assumption.
        intros.
        apply HnoStep.
        destruct H0 as [e2' [st']].
        ∃ (Bracket e1 e2').
        ∃ st'.
        apply ST_Bracket2; try assumption.
    SCase "He1nv".
       apply Stk_Bracket1; try assumption.
       apply (IHe1 fst); try assumption.
       split; try assumption.
```

```
          intros.
          apply HnoStep.
          destruct H0 as [e1' [st']].
          ∃ (Bracket e1' e2).
          ∃ st'.
          apply ST_Bracket; assumption.

Case_aux AssertionContext "use generalization".
   rewrite wf__wfc_top.
   apply H.
Qed.

Lemma stuck_is_stuck:
  ∀ e st,
    well_formed e → (stuck_semantic e st ↔ stuck e).
Proof.
  intros.
  split.
  Case "→". apply stuck_semantic_is_stuck; assumption.
  Case "←". apply stuck_is_stuck_semantic.
Qed.

Theorem Three: ∀ e,
   stuck e → (∃ i, stuck (bracket_proj e i)).
Proof.
   e_cases (induction e) Case; intros; try solve by inversion; simpl;
     try (∃ fst; constructor; assumption; fail).
   Case "App".
     inversion H.
     SCase "Stk_App1". apply IHe1 in H1. destruct H1. ∃ x.
       constructor. assumption.
     SCase "Stk_App2". destruct IHe2.
```

```
        SSCase "stuck e2". assumption.
        SSCase "∃ i, stuck (bracket_proj e1 i)".
          ∃ x. apply Stk_App2. apply bracket_proj_value; assumption.
          assumption.
      SCase "Stk_AppNotLambda".
        ∃ fst.
        apply Stk_AppNotLam; [ eauto | apply bracket_proj_value; assumption ].
          e_cases (destruct e1) SSCase; try solve by inversion; try (
    constructor; fail).
    Case "If".
      inversion H.
      SCase "Stk_Ifstk".
        apply IHe1 in H1.
        destruct H1 as [i].
        ∃ i.
        constructor.
        assumption.
      SCase "Stk_Ifval".
        ∃ fst.
        inversion H1; try (simpl; apply Stk_Ifval; repeat constructor;
    assumption; fail).
    Case "Bracket".
      inversion H; [ ∃ fst | ∃ snd ]; assumption.
Qed.
```

## A.5 Theorem4.v

```
Require Import FML.
Require Import Theorem3.

Theorem Four: ∀ e st,
  (∀ i,
    (∃ v st',
```

```
       value v →
      (bracket_proj e i) / (store_proj st i) @ top_lvl ⇒* v / st' @ top_lvl))
    →
  (∃ v' st'',
    value v' →
    e / st @ top_lvl ⇒* v' / st'' @ top_lvl).
Proof with eauto.
intros.
e_cases (induction e) Case; intros.
Case "True".
  ∃ True. ∃ st.
  intros.
  eapply multi_refl.
Case "False".
  ∃ False. ∃ st.
  intros.
  eapply multi_refl.
Case "Var".
  ∃ (Var i). ∃ st.
  intros.
  eapply multi_refl.
Case "App".
  ∃ (App e1 e2). ∃ st.
  intros.
  eapply multi_refl.
Case "Lambda".
  ∃ (Lambda i e). ∃ st.
  intros.
  eapply multi_refl.
Case "If".
  ∃ (If e1 e2 e3). ∃ st.
  intros.
```

```
  eapply multi_refl.
Case "Void".
  ∃ Void. ∃ st.
  intros.
  eapply multi_refl.
Case "Bracket".
  ∃ (Bracket e1 e2). ∃ st.
  intros.
  eapply multi_refl.
Qed.
```

## BIBLIOGRAPHY


[1] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. Sharing mobile code securely with information flow control. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 191–205. IEEE, 2012.

[2] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *Programming Languages and Systems*, pages 64–84. Springer, 2010.

[3] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 43–59. IEEE, 2009.

[4] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. In *ACM SIGPLAN Notices*, volume 49, pages 165–178. ACM, 2014.

[5] Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[6] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *Programming Languages and Systems*, pages 125–140. Springer, 2007.

[7] D Bell. The bell-lapadula model. *Journal of computer security*, 4(2):3, 1996.

[8] Dave Berry, Robin Milner, and David N Turner. A semantics for ml concurrency primitives. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–129. ACM, 1992.

[9] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[10] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *Proceedings of USENIX Security*, 2013.

[11] Matt Bishop. *Introduction to computer security*. Addison-Wesley Professional, 2004.

[12] Sören Bleikertz, Anil Kurmus, Zoltán A Nagy, and Matthias Schunter. Secure cloud maintenance: protecting workloads against insider attacks. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 83–84. ACM, 2012.

[13] Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. *ACM SIGPLAN Notices*, 48(9):391–402, 2013.

[14] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 31–44. ACM, 2007.

[15] Stephen Chong and Andrew C Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209. ACM, 2004.

[16] Stephen Chong and Andrew C Myers. Decentralized robustness. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, pages 12–pp. IEEE, 2006.

[17] Stephen Chong and Andrew C Myers. End-to-end enforcement of erasure and declassification. In *Computer Security Foundations Symposium, 2008. CSF'08. IEEE 21st*, pages 98–111. IEEE, 2008.

[18] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and computation*, 76(2):95–120, 1988.

[19] Haskell B Curry, J Roger Hindley, and Jonathan Paul Seldin. To hb curry: essays on combinatory logic, lambda calculus, and formalism. 1980.

[20] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972.

[21] Fabien Dagnat, Marc Pantel, Matthias Colin, Patrick Sall, Mots-cls Acteurs, and Objets Concurrents. Typing concurrent objects and actors. In *In L'Objet - Methodes formelles pour les objets (L'OBJET*. Citeseer, 2000.

[22] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[23] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[24] Catherine Dubois. Proving ml type soundness within coq. In *Theorem Proving in Higher Order Logics*, pages 126–144. Springer, 2000.

[25] Walter Feit and John Thompson. Chapter i, from solvability of groups of odd order, pacific j. math, vol. 13, no. 3 (1963. *Pacific Journal of Mathematics*, 13(3):775–787, 1963.

[26] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 432–441. ACM, 2009.

[27] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *ACM SIGPLAN Notices*, volume 43, pages 323–335. ACM, 2008.

[28] Andreas Gampe and Jeffery von Ronne. A framework for composing security-typed languages. In *FCS 2013 Workshop on Foundations of Computer Security (Informal Proceedings)*, page 34, 2013.

[29] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Verification, Model Checking, and Abstract Interpretation*, pages 346–362. Springer, 2005.

[30] Stephen Gilmore. *Programming in Standard ML'97: A tutorial introduction*. University of Edinburgh, 1997.

[31] Joseph A Goguen and José Meseguer. Security policies and security models. security and privacy. In *IEEE Symposium on, 0*, volume 11, page 77, 1982.

[32] Florian Kammüller. Formalizing non-interference for a simple bytecode language in coq. *Formal Aspects of Computing*, 20(3):259–275, 2008.

[33] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[34] Xavier Leroy. The compcert c verified compiler, 2012.

[35] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 321–334. ACM, 2009.

[36] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[37] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[38] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[39] Andrew C Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 172–186. IEEE, 2004.

[40] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[41] Benjamin C Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjoberg, and Brent Yorgey. Software foundations. *Course notes, online at http://www. cis. upenn. edu/~ bcpierce/sf*, 2(3.1):3–2, 2010.

[42] François Pottier and Vincent Simonet. Information flow inference for ml. In *ACM SIGPLAN Notices*, volume 37, pages 319–330. ACM, 2002.

[43] John H Reppy. Higher-order concurrency. Technical report, Cornell University, 1992.

[44] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[45] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[46] Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 3–13. IEEE, 2003.

[47] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364. ACM, 1998.

[48] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 85–96. ACM, 2004.

[49] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):6, 2007.

[50] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2):167–187, 1996.

[51] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[52] Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, 2004.

[53] Steve Zdancewic and Andrew C Myers. Robust declassification. In *Computer Security Foundations Workshop, IEEE*, pages 15–15. IEEE Computer Society, 2001.

[54] Steve Zdancewic and Andrew C Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 29–43. IEEE, 2003.

[55] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328, 2002.

[56] Lantian Zheng, Stephen Chong, Andrew C Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 236–250. IEEE, 2003.

# VITA

Eric Alders is originally from Wichita, KS where he grew up. Eric attended Wichita State University from 98'-02' graduating with a Bachelors of Science Computer Science and minor in Mathematics. He attended Webster University in Kansas City, MO from 05'-08' receiving a Masters of Business Administration. Eric plans on completing his current degree, a Masters in Computer Science from UTSA in the fall of 15'.

Eric has over 15 years experience as a professional software engineer with government and private companies in health care, real estate, finance and security. Eric spent four years of his career in San Antonio, TX working with the Air Force on cyber security issues. Eric holds GSEC and CEH security certifications and enjoys research in security, programming languages and operating systems.

Eric is a husband and father that enjoys cooking and working out in his spare time. Eric currently resides in San Antonio, TX with his wife, Michelle, daughter Kaitlyn, and two dogs Maddie & Daisy. Eric continues to work for companies in the health care industry as a professional software engineer where he is the lead engineer for web development and security.