

THE HOLY GRAIL OF GRADUAL SECURITY

Tianyu Chen

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science,
Indiana University
May 2025

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Jeremy G. Siek, Ph.D.

Amr Sabry, Ph.D.

Chung-chieh Shan, Ph.D.

Sam Tobin-Hochstadt, Ph.D.

Date of Defense: 05/02/2025

Copyright © 2025

Tianyu Chen

To my parents and my grandmother.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Prof. Jeremy Siek for his outstanding guidance and unwavering support. Prof. Siek is my role model for being a researcher. He encourages me to think in a deep and rigorous way while presenting the results in a lucid and accessible way. Prof. Siek is also my role model for being an educator. He shows me how to present the course materials in an incremental and easily accessible way that is relatable to the experiences of the students. I owe all the achievements during my graduate studies to Prof. Siek.

I would like to thank the professors in my Ph.D. advisory committee (in no particular order): Prof. Amr Sabry, Prof. Chung-chieh Shan, and Prof. Sam Tobin-Hochstadt for providing feedback and suggestions to my research and this dissertation. In addition, I would like to thank Prof. Dan Friedman: like many students at Indiana University, I was introduced to the world of programming languages by Prof. Friedman’s “little books” and the lectures of C311/B521 “Programming Language Principles.”

I would like to thank the exceptional community of programming language researchers (“PL Wonks”) at Indiana University. I am especially grateful to the following professors, students, and postdoctoral researchers (in no particular order): Prof. Carlo Angiuli, Prof. Ryan Newton, Dr. David Christiansen, Joshua Crofts, Caner Derici, Chenchao Ding, Fred Fu, Aria Givens, Ethan Hawk, Artem Iurchenko, Sanad Kadu, Caleb Schultz Kisby, Darshal Shetty, Zixiu Su, Jifeng Wu, Yafei Yang, Andre Kuhlenschmidt, Ryan Scott, Tulip Amalie, Kartik Sabharwal, Annie Pompa, Vikraman Choudhury, Sam Bowman, Chaitanya Koparkar, Rajan Walia, Sarah Spall, Weixi Ma, Matthew Heimerdinger, Chao-Hong Chen, Michael Vollmer (now professor at the University of Kent), Victoria Vollmer, Kuang-Chen Lu, Joshua Larkin, Deyaaeldeen Almahallawi, Aaron Hsu, Paulette Koronkevich, Andrew Kent, and Jason Hemann (now professor at Seton Hall University). *Thank you!*

I thank Zihan Chen, Che Jingyin, Katharine Khamhaengwong, Jacob Striebel, Xiaorui

Pan, and Kan Yuan at Indiana University for friendship.

I am grateful to all the students whom I taught. In particular, I would like to thank Calvin Josenhans, Gautam Hari, Sparsh Nair, Oleksandra Tkachuk, Zeshawn Zahid, Jacob Herbert, Dylan Jacoby, Solomon Zinn Krulewitch, and Yuntian Zeng. I also thank Dr. Akesha Horton for her lectures on pedagogy.

I thank Dr. Vilhelm Sjöberg for hosting me as a summer intern on software verification at CertiK. I thank Prof. Kristopher Micinski, Prof. Sergey Bratus, and Prof. Gang Tan for advice and discussion.

I would like to thank the operating system research group at Columbia University led by Prof. Junfeng Yang, who provided a valuable summer research opportunity to me when I was an undergrad. I appreciate the friendship and help from (in no particular order) Dr. Heming Cui (now professor at the University of Hong Kong), Rui Gu, Yang Tang (now professor at New York University), Gang Hu, Xinhao Yuan, Lingmei Weng, and Chang Lou (now professor at the University of Virginia). I thank Prof. Yu-Ping Wang for lecturing me on symbolic execution while I was doing undergraduate independent study at Tsinghua. Also, I express special gratitude to the friends and roommates from my undergraduate days at Tsinghua: Hongyin Luo, Yuan Yang, Hao Wang, Hongyi Wen (now professor at New York University Shanghai), Xintian Li, Hanxuan Yu, and Sêkai Zhou.

I would like to thank Hattie Fu for her love, support, and company during the final (and one of the hardest) stage of my dissertation writing.

I thank my grandmother, who taught physics and electrical engineering before retirement, for introducing me to mathematics and applied sciences. I thank my aunts for buying me the first computer in my life: a used graphic workstation with an Intel 80486 processor. Lastly, and most importantly, I would like to say thank you to my parents. I could not have completed this quest without your support and understanding.

(This dissertation is based upon work supported by the National Science Foundation under Grant No. 1763922 “Performant Sound Gradual Typing.”)

Tianyu Chen

THE HOLY GRAIL OF GRADUAL SECURITY

Ensuring the security and privacy of personal data typically involves tracking and checking the flow of information, which can be performed either statically using a type system or dynamically using runtime monitoring. The dynamic approach of information-flow control (IFC) requires less effort from the programmer while the static approach provides stronger guarantees and less runtime overhead. Languages with gradual IFC combine static and dynamic techniques to prevent security leaks, so the programmer is free to choose when it is appropriate to increase the precision of type annotations and put in the effort to pass the static checks, versus when it is appropriate to reduce the precision of type annotations, thereby deferring the enforcement to runtime. Gradual programming languages should satisfy the gradual guarantee: programs that only differ in the precision of their type annotations should behave the same modulo cast errors. Unfortunately, Toro et al. [2018] identify a tension between the gradual guarantee and information security. They conjecture that it is not possible to enforce noninterference and satisfy the gradual guarantee.

In my PhD dissertation, I harmoniously combine static and dynamic enforcement of IFC in one programming language, λ_{IFC}^* , which satisfies both noninterference and the gradual guarantee at the same time without making any sacrifices. λ_{IFC}^* (1) enforces information flow security, (2) satisfies the gradual guarantee, (3) supports type-based reasoning, and (4) requires no extra static analysis prior to program execution. The key to the design of λ_{IFC}^* is to exclude the unknown label from runtime security labels. On the technical side, the semantics of λ_{IFC}^* is

the first gradual information-flow control language to be specified using coercion calculi (à la Henglein). Casts in λ_{IFC}^* are represented by security coercions, which enforce the flow of information while satisfying the gradual guarantee.

I mechanize the proofs of type safety and the gradual guarantee for λ_{IFC}^* in the Agda proof assistant. I prove noninterference for λ_{IFC}^* by simulating λ_{IFC}^* with its dynamic extreme.

In summary, my thesis is that it is possible to design a gradual IFC programming language that satisfies noninterference and the gradual guarantee while supporting type-based reasoning, by excluding the unknown label from run-time security labels and using security coercions to represent casts.

Jeremy G. Siek, Ph.D.

Amr Sabry, Ph.D.

Chung-chieh Shan, Ph.D.

Sam Tobin-Hochstadt, Ph.D.

TABLE OF CONTENTS

Acknowledgements	v
Abstract	vii
List of Tables	xiii
List of Figures	xiv
Chapter 1: Introduction	1
1.1 Security Is Important, and Programming Languages Can Help!	1
1.2 Information-Flow Control via Static, Dynamic, and Hybrid Mechanisms	4
1.2.1 Static IFC in Programming Languages	4
1.2.2 Dynamic IFC in Programming Languages	7
1.2.3 Dynamic IFC Augmented With Static Pre-processing	8
1.2.4 Programmer-Controlled Hybrid IFC	10
1.3 The Tension Between Gradual Typing and Information-Flow Control	10
1.4 Key Enabling Insight	12
1.5 Thesis Statement	13
1.6 Technical Contributions and Outline	14
1.7 Data Availability	17

Chapter 2: Gradual Information-Flow Control (IFC) in λ_{IFC}^*	18
2.1 The Gradual IFC Language λ_{IFC}^*	19
2.1.1 Security Labels and Types	19
2.1.2 Syntax of λ_{IFC}^*	20
2.1.3 Type System of λ_{IFC}^*	21
2.1.4 Semantics of λ_{IFC}^*	26
2.1.5 λ_{IFC}^* Programs	27
2.2 λ_{IFC}^* in Action	27
2.2.1 The Gradual Transition Between Static and Dynamic IFC in λ_{IFC}^*	28
2.2.2 Implicit Flow, NSU Checks, Unknown Security, and the Gradual Guarantee	33
2.2.3 Type-Based Reasoning in λ_{IFC}^*	38
2.3 The Static and Dynamic Extremes of λ_{IFC}^*	40
2.3.1 The Static Extreme of λ_{IFC}^* : SSL_{Ref}	40
2.3.2 The Dynamic Extreme of λ_{IFC}^* : $\lambda_{\text{IFC}}^{\text{DYN}}$	40
2.3.3 Formal Definition of $\lambda_{\text{IFC}}^{\text{DYN}}$	41
2.3.4 Comparing $\lambda_{\text{IFC}}^{\text{DYN}}$ to λ^{info} and LIO	43
Chapter 3: The Definition of the Cast Calculus λ_{IFC}^c	46
3.1 Why Coercions?	46
3.2 A Coercion Calculus for Security Labels	47
3.2.1 Syntax, Typing, and Semantics of the Coercion Calculus for Security Labels	47
3.2.2 Monitoring Explicit and Implicit Flows	49

3.2.3	Security Label Expressions	51
3.3	A Coercion Calculus on Values	54
3.4	The Cast Calculus λ_{IFC}^c : An Intermediate Language For Gradual IFC	56
3.4.1	Syntax of λ_{IFC}^c	56
3.4.2	Type System of λ_{IFC}^c	56
3.4.3	Operational Semantics for λ_{IFC}^c	59
Chapter 4:	Compiling From λ_{IFC}^* to λ_{IFC}^c	65
4.1	Coerce Functions	65
4.2	Compilation	66
Chapter 5:	Type Safety of λ_{IFC}^*	70
5.1	Type Safety of λ_{IFC}^c by Progress and Preservation	70
5.2	Compilation From λ_{IFC}^* to λ_{IFC}^c Preserves Types	71
5.3	Type Safety of λ_{IFC}^*	71
Chapter 6:	Gradual Guarantee of λ_{IFC}^*	74
6.1	Simulation Between More and Less Precise Coercion Sequences	74
6.2	Simulation Between λ_{IFC}^c Terms of Different Precision	78
6.3	The Gradual Guarantee of λ_{IFC}^*	84
Chapter 7:	Noninterference of λ_{IFC}^*	88
7.1	The Normalization of Coercions Checks Information Flow	89
7.2	Noninterference of $\lambda_{\text{IFC}}^{\text{DYN}}$	90
7.3	Simulation Between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$	92

7.4	Noninterference of λ_{IFC}^*	111
Chapter 8: Future Work and Conclusion		113
8.1	Future Work	113
8.1.1	RQ1: How to mechanize the noninterference proof for λ_{IFC}^* ?	114
8.1.2	RQ2: Is it possible to make λ_{IFC}^* space efficient?	115
8.1.3	RQ3: Is λ_{IFC}^* expressive?	115
8.1.4	RQ4: Is it possible to implement λ_{IFC}^* with high performance?	117
8.2	Conclusion	119
References		120
Curriculum Vitae		

LIST OF TABLES

1.1	Proposed sources of tension between security and the gradual guarantee . . .	11
-----	--	----

LIST OF FIGURES

1.1	The user input grammar for a hypothetical application	2
1.2	The parse tree generated from the example user input. All terminals are represented as labeled values: the red ones, such as the digits of SSN, are of high-security, while the green ones, such as the keys of the record and first name / last name, are of low-security.	3
2.1	Security labels and types	20
2.2	Syntax of λ_{IFC}^* (highlighted security labels ℓ default to low if omitted)	20
2.3	Auxiliary operators for security labels and types: join w.r.t precision ($-\sqcup-$), consistent join ($-\tilde{\vee}-$ for labels and $-\tilde{\vee}-$ for types), and consistent meet ($-\tilde{\wedge}-$ for labels and $-\tilde{\wedge}-$ for types). Stamping for types	22
2.4	Consistent subtyping for labels and types	23
2.5	Precision of security labels and types	24
2.6	Typing rules of λ_{IFC}^* . Side conditions about the heap policy are highlighted	25
2.7	Evaluation of λ_{IFC}^*	26
2.8	Precision rules of λ_{IFC}^*	36
2.9	The syntax and the reduction semantics of $\lambda_{\text{IFC}}^{\text{DYN}}$. The checks that enforce heap policy in reference creation and assignment are highlighted	42
3.1	Syntax, typing, normal forms, and semantics of security coercions and coercion sequences	48
3.2	Composing and stamping coercions	50

3.3	Syntax, typing, normal forms, and semantics of label expressions	52
3.4	Stamping and security level operators for security label expressions	53
3.5	Syntax and semantics of coercions on values	54
3.6	Composition of coercions on values	55
3.7	Syntax of the cast calculus λ_{IFC}^c	56
3.8	Typing rules of the cast calculus λ_{IFC}^c . The side conditions that enforce the heap policy statically during memory write operations are highlighted	57
3.9	Operational semantics of λ_{IFC}^c (Part I).	60
3.10	Operational semantics of λ_{IFC}^c (Part II). NSU checks that are represented using label expressions are highlighted	61
3.11	Stamping on values of λ_{IFC}^c	62
3.12	Multi-step reduction of λ_{IFC}^c	64
4.1	Coerce functions of security labels and types	66
4.2	Compilation from λ_{IFC}^* to λ_{IFC}^c	67
5.1	Well-typed evaluation result	72
6.1	Precision relation of coercions on security labels	75
6.2	Precision relation of coercions on values	78
6.3	Precision relation of label expressions	79
6.4	Precision rules of λ_{IFC}^c (Part I)	80
6.5	Precision rules of λ_{IFC}^c (Part II)	81
6.6	Precision rules of λ_{IFC}^c (Part III)	82
6.7	Precision rules of λ_{IFC}^c (Part IV)	83

6.8	Precision rules of λ_{IFC}^*	85
7.1	Big-step operational semantics (successful cases) of $\lambda_{\text{IFC}}^{\text{DYN}}$	91
7.2	Simulation relation between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$ (Part I)	93
7.3	Simulation relation between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$ (Part II)	94
7.4	Erasure from λ_{IFC}^c to $\lambda_{\text{IFC}}^{\text{DYN}}$	110

CHAPTER 1

INTRODUCTION

1.1 Security Is Important, and Programming Languages Can Help!

Increasingly relying on computing devices and the Internet in their daily lives, people are more and more concerned about the confidentiality of their personal data and the integrity of their online assets. People fear that sensitive personal information, such as social security numbers, medical records, bank account balances, etc., may be revealed to malicious third parties. People also worry that their digital photo albums, signatures on online legal documents, spreadsheets in cloud storage, etc., may be manipulated by potential attackers.

Indeed, the fears are justified by recent news events. In 2018, the Cambridge Analytica scandal hit the world headlines, where the data collected from 87 million social media users was misused without their consent [1, 2, 3, 4]. In the healthcare sector, from 2005 to 2019, 249 million people were affected by data breaches that caused exposure of sensitive medical data [5]. Researchers have found ways to tamper with the analytics APIs [6] and damage the integrity of metadata, such as the numbers of likes, follows, and views, of major social media platforms [7]. To deal with the security and privacy challenges of the increasingly digitalized world, the European Union introduced the General Data Protection Regulation (GDPR) to reform and regulate the collection and processing of personal data. However, studies show that business entities experience challenges in complying with GDPR or auditing for compliance [8], particularly small-to-medium size enterprises [9, 10, 11].

From a technical perspective, ensuring the security and privacy of personal data typically involves tracking and checking the flow of information. To ensure confidentiality, data must not flow to inappropriate destinations, so that sensitive personal information is

$$\begin{aligned}
\langle RECORD \rangle &::= \{ \text{FirstName}=\langle ID \rangle; \\
&\quad \text{LastName}=\langle ID \rangle; \\
&\quad \text{SSN}=\langle SSN \rangle \} \\
\langle ID \rangle &::= w, w \in \{A, \dots, Z, a, \dots, z\}^+ \\
\langle SSN \rangle &::= \langle D \rangle \langle D \rangle \langle D \rangle - \langle D \rangle \langle D \rangle - \langle D \rangle \langle D \rangle \langle D \rangle \langle D \rangle \\
\langle D \rangle &::= d, d \in \{0, \dots, 9\}
\end{aligned}$$

Figure 1.1: The user input grammar for a hypothetical application

not revealed. Dually, to ensure integrity, data must not flow from inappropriate sources so that valuable digital assets are not corrupted [12, 13]. In practice, such enforcement of the flow of information is often difficult to implement. Take confidentially for example, software applications accept user input where selected fields are sensitive, whose confidentiality is required during their entire life cycle, including both parsing and data processing. To rule out information leaks, neither the sensitive fields nor any data that depends on those fields can be revealed to a low-privilege observer. Consider a web application that receives three fields from its user: (1) first name, (2) last name, and (3) social security number, the grammar of which is defined in Figure 1.1, where terminals are divided into low-security and high-security. The digits d for social security number, being confidential to users of the web application, are of high-security, so they are marked **red**. Other terminals, such as the keys of the record and the strings w for first name/last name, being safe to disclose, are of low-security, marked in **green**. Consider the following user input:

$$\{ \text{FirstName}=\text{Mad}; \text{LastName}=\text{Hatter}; \text{SSN}=012-34-5678 \}$$

It is tedious for the developer of this imaginary web application to track the security level of data and then check for information leaks. Software developers tend to focus more on functionality in order to meet the tight software release schedule and budget, thus software security comes as an afterthought [14, 15, 16]. Retrofitting security-related code often requires extensive modification to an existing code-base and relies on the programmers' skills and experience to decide when and where such code should be placed. Furthermore,

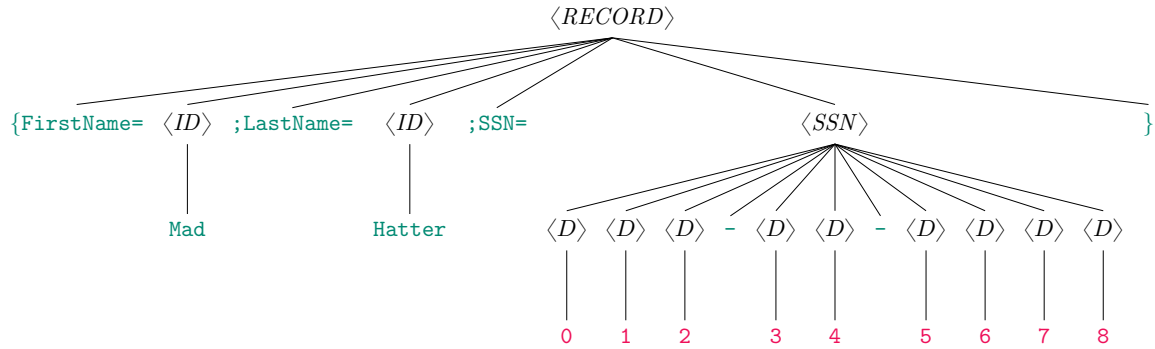


Figure 1.2: The parse tree generated from the example user input. All terminals are represented as labeled values: the red ones, such as the digits of SSN, are of high-security, while the green ones, such as the keys of the record and first name / last name, are of low-security.

this modification is error-prone: a single missing check could undermine the security of the entire application.

Alternatively, the author of the web application could implement a parser for the grammar in Figure 1.1 with a programming language that enforces information-flow security. Each terminal in the grammar would be labeled with a security level, and the programming language, instead of ad-hoc checks implemented by the programmer of the web application, would guarantee that high-security information is only present in those parts of the output parse tree that are marked as high-security. For example, according to the grammar in Figure 1.1, the example user input string is parsed into the parse tree in Figure 1.2, where the terminal nodes that represent digits of the social security number are of high-security, while the terminals that compose the rest of the input string are of low-security. The confidentiality of SSN is guaranteed during data processing by the programming language itself. When the web application interacts with the outside world, such as making a foreign function interface (FFI) call or storing into a database, the conceptual language should encrypt whatever values labeled as high security before they are passed into a foreign routine. The programming language-based approach of information-flow security [12] alleviates the security burden of software development, because it forms an abstraction over the flows of information and decouples security from the functionality of a software application.

1.2 Information-Flow Control via Static, Dynamic, and Hybrid Mechanisms

Information-flow control (IFC) ensures that information transfers within a program adhere to a security policy, for example, by preventing high-security data from flowing to a low-security channel. This adherence can be enforced statically using a type system [17, 18, 19], or dynamically using runtime monitoring [20, 21, 22, 23, 24, 25], or using static analysis to pre-compute information that facilitates runtime monitoring [26, 27, 28, 29, 30, 31]. The static and dynamic approaches each have complementary strengths and weaknesses; the static approach provides stronger guarantees and less runtime overhead while the dynamic approach requires less effort from the programmer. The main theorem of an IFC system is *noninterference* [32]. Informally, noninterference states that, if the secretive user input varies, the public output of the program must stay the same.

In Section 1.2.1 and Section 1.2.2, we review the literature about static and dynamic enforcement of IFC. We then review dynamic IFC enforcement with static pre-processing in Section 1.2.3. Finally in Section 1.2.4, we briefly review the hybrid technique of Buiras, Vytiniotis, and Russo [33], which offers the programmer control over the regions where IFC is enforced statically versus dynamically in a single program.

1.2.1 Static IFC in Programming Languages

In the 1970s, the interest in enforcing confidentiality and regulating the flow of information in a computer program arose with applications in a military or government setting [34, 35]. Denning [36] builds an information flow model using a lattice of security labels in 1976. Denning and Denning [37] describe a static analysis for information flow and prove that a certified program will not transmit confidential input to non-confidential output in 1977. They distinguish between two types of information flows: explicit flow and implicit flow. Consider the assignment

$$x := y + z$$

There are explicit flows from y to x and from z to x , because both y and z affect the value that x is assigned to. The certification checks whether the join (least upper bound) of the security of y and z is less than or equal to the security of x . Implicit flows, on the other hand, arise from the branching structure of a program. Consider the program

```
if x then y := y + 1 else ()
```

An observer is able to learn whether x is true or false, by inspecting whether y increments. To control the implicit flow, the certification checks whether the security of x is less than or equal to that of y .

Volpano, Irvine, and Smith [17] further develop the static enforcement and propose a typed-based approach to IFC by defining a type system for an imperative programming language, then prove its security with a type soundness proof. The type system approach benefits IFC enforcement because it is compositional: the security of the entire program is determined from the security of its constituent components; secure components form a larger secure system as long as their type signatures agree [12]. By writing a program that type checks, the software developer constructs a proof that the program is indeed secure.

We explain how a type system defends against illegal information flow using examples. Consider two IO functions: `private-input` and `publish`: the former returns a high-security boolean that represents sensitive user input information; the latter takes a low-security boolean and publishes it into a publicly visible channel.

Explicit flow. Consider the following program with an illegal explicit flow from private input to public output:

```
let input = private-input () in publish (¬ input)
```

The program is rejected by the type checker because `(¬ input)` is typed at high security (`Boolhigh`) but `publish` expects its argument to be of low security (`Boollow`). High-security information must not flow into a low-security sink (`high` $\not\leq$ `low`). As a result, the type

system prevents information from leaking through explicit flow. On the other hand, the program

```
let input = private-input () in publish (¬ true)
```

is accepted by the type checker, because $(\neg \text{true})$ is typed at Bool_{low} , which can flow into `publish`. Indeed, the output remains constant regardless of user input, so no information leaks through the output.

Implicit flow. Guarding against illegal implicit flows is more involved: the type system estimates the security of an if-conditional by joining the security of its branches with that of the branch condition (often referred to as stamping [38]). Consider the following program with an illegal information flow from private input to public output:

```
let input = private-input () in  
  publish (if input then false else true)
```

The input influences the output through the branching structure of the program: if `input` is true, the then-branch is taken and the output is false; if `input` is false, the else-branch is taken and the output is true. The branch condition is typed at $\text{Bool}_{\text{high}}$ and the branches are typed at Bool_{low} , so the entire `if` has type $\text{Bool}_{\text{low} \vee \text{high}} = \text{Bool}_{\text{high}}$ because of stamping. Again, `publish` expects Bool_{low} , so the program is rejected. Consequently, the type system guards against the information leak through implicit flow.

Heintze and Riecke [39] and Zdancewic [38] further develop the type system approach to IFC by adding support for higher-order functions. Researchers also build IFC type systems for bytecode intermediate languages [40], for object-oriented languages [41], and for reactive programming languages [42]. Although the aforementioned languages are mostly theoretical, efforts have also been made to integrate information flow control into off-the-shelf programming languages such as Jif for Java [19] and Flow Caml for OCaml [43, 44].

1.2.2 Dynamic IFC in Programming Languages

IFC can also be enforced dynamically using runtime monitoring, an idea presented by Fenton [45] in 1974. In dynamic IFC, values typically have security labels associated with them and the runtime monitor tracks those labels during program execution. In the illegal explicit flow example

```
let input = private-input () in publish (¬ input)
```

the value of `input` is tagged with `high` and so is its negation. When `publish` is called, the function checks whether the label on the argument is `low`. The check fails, so the monitor reports a runtime error. In the constant example

```
let input = private-input () in publish (¬ true)
```

`true` is tagged with `low` and so is its negation; the monitor succeeds and publishes `false`.

In the illegal implicit flow example

```
let input = private-input () in
  publish (if input then false else true)
```

the label associated with the evaluation result of the if-conditional is the join between that of the branch taken and that of the branch condition; in this case, $\text{low} \vee \text{high} = \text{high}$. The `publish` function again checks the label against `low`, $\text{high} \not\leq \text{low}$, so it results in a runtime error.

Li and Zdancewic [46, 47] study dynamic IFC for purely functional languages. They add dynamic checking of information flow control to Haskell by utilizing arrows and type-classes.

Austin and Flanagan [21] consider IFC for JavaScript: a language with mutable references. They propose a dynamic approach called *no-sensitive-upgrade* (NSU) checking, which guards against illegal implicit flows through the heap. During execution, the language runtime keeps track of a special security label called the “program counter”. The

program counter starts from low and is updated to high when the program branches on a high-security value. NSU terminates the execution whenever the program attempts to modify a low-security memory location under a high-security program counter. Austin and Flanagan [48] study a sound yet more flexible enforcement strategy called *permissive-upgrade*. Compared to NSU, permissive-upgrade allows more programs to run to completion. Austin and Flanagan [49] propose *faceted values*, another approach to dynamically handle implicit flows by simulating multi-execution in a single process. Compared with no-sensitive-upgrade and permissive-upgrade, this faceted execution approach avoids conservative monitor failures [24].

Stefan et al. [23, 50, 51] design a Haskell library called LIO, which is implemented as a domain-specific language (DSL) embedded in Haskell. Even though Haskell itself is purely functional, it is worth noting that the LIO DSL does support mutable references (LIORef). Inspired by IFC operating systems [52, 53, 54, 55], LIO makes two unique design choices: (1) coarse-grained labeling and (2) a floating current label. LIO is “coarse-grained” in that not all values are labeled by default; a programmer need to use the `toLabeled` primitive to explicitly protect a value. Consequently, a programmer may choose to label a value when it is necessary to impose flow control policies and omit the labels for security-insensitive parts of the program. In LIO, the “current label” serves as a security upper bound of all values. Under the hood, LIO keeps track of the current label as a monad. During program execution, the LIO runtime raises the current label to “float” above the security of all data read by the current computation. Similar to NSU, LIO performs dynamic checking on heap write operations and disallows writing to memory locations whose security is below the current label.

1.2.3 Dynamic IFC Augmented With Static Pre-processing

Researchers have also explored facilitating runtime IFC monitors with static analyses prior to program execution to achieves goals that are hard to accomplish by using dynamic IFC

alone. It is worth noting that, in contrast to user-controlled hybrid IFC and gradual IFC (which we are going to discuss in Section 1.2.4 and Section 1.3 respectively), papers in this category do not aim at offering the programmer the *control* over statically versus dynamically enforced regions in a single program.

Le Guernic and Jensen [26] explore a hybrid IFC monitor for sequential programs that combines runtime IFC with a static analysis that gathers information about the non-executed branches. This hybrid IFC accepts programs that are conservatively rejected by fully static IFC. They further show that it is possible to alter the behavior of executions that may be unsafe by resetting output values using the information from their hybrid analysis to reclaim confidentiality. Le Guernic [27] extends this hybrid IFC approach to concurrent programs and prove noninterference for any execution. Shroff, Smith, and Thober [29] build a runtime monitor augmented with a pre-computed fixed point of dependencies. Similar to Le Guernic and Jensen [26], they observe that this hybrid technique is less conservative than a fully-static system. They also show that their hybrid monitor is able to support user-defined policy, so that different policies can be applied to the same program.

Chandra and Franz [28] implement a hybrid monitor for the Java virtual machine. Their implementation add IFC annotations to a executable file using a static analysis. The runtime monitor then uses those annotations to update the labels of variables in alternative execution paths to enforce IFC, while maintaining backward-compatibility with existing Java class files.

Russo and Sabelfeld [30] study dynamic IFC in a flow-sensitive setting. They utilize a static analysis that detects variables in untaken branches whose security must be upgraded at runtime to prevent illegal implicit flows through the heap. Moore and Chong [31] improve the efficiency of the hybrid monitoring of Russo and Sabelfeld [30] by selectively tracking variables and incorporating memory abstractions.

1.2.4 Programmer-Controlled Hybrid IFC

Similar to gradual IFC but different from dynamic IFC with static pre-processing, Hybrid LIO (HLIO) [33] supports the programmer’s choice of static or dynamic IFC in different regions of a single program. By default, the checking is static, but a programmer can insert a `defer` clause to say that the security constraints should be checked at runtime. There are two major differences between HLIO and *gradual* IFC programming languages (we are going to discuss gradual IFC in the next section). In HLIO, the developer has to embed explicit `defer` into the program, while in a gradual language, the switch between static and dynamic is directed by types, with no `defer` or explicit casts needed. Moreover, there is no theorem about adding and removing `defer` in HLIO, while in a gradual language, the gradual guarantee theorem relates the runtime behavior of programs that differ only in the precision of their type annotations.

1.3 The Tension Between Gradual Typing and Information-Flow Control

Taking inspiration from gradual typing [56, 57], researchers have explored new ways to give programmers control over which parts of the program are secured statically versus dynamically, directed by *type annotations*. In general, gradually typed languages support the seamless transition between static and dynamic enforcement through the *precision* of type annotations. Gradual security is useful because the programmer is free to choose when it is appropriate to increase the precision of the type annotations and put in the effort to pass the static checks, versus when it is appropriate to reduce the precision of type annotations, thereby deferring the enforcement to runtime.

The main challenge in the design of gradually-typed languages is controlling the flow of values (and information) between the static and dynamic regions of code, which is accomplished using runtime casts. Typically source programs are compiled to an intermediate language (called a *cast calculus*) that includes explicit syntax for runtime casts. Disney and

Table 1.1: Proposed sources of tension between security and the gradual guarantee

Language	Security (noninterference)	Gradual Guarantee	Type-guided classification	NSU checking	Runtime security labels
GSL_{Ref}	✓ Yes	✗ No	✓ Yes	✓ Yes	{low, high, *}
GLIO	✓ Yes	✓ Yes	✗ No	✓ Yes	{low, high}
λ_{SEC}^*	✓ Yes	✓ Yes	✗ No	✓ Yes	{low, high}
WHILE ^G	✓ Yes	✓ Yes	✓ Yes	✗ No	{low, high, *}
λ_{IFC}^* (this dissertation)	✓ Yes	✓ Yes	✓ Yes	✓ Yes	{low, high}

Flanagan [58] design a cast calculus with IFC for a pure lambda calculus and prove noninterference. Fennell and Thiemann [59] design a cast calculus named ML-GS with mutable references using the no-sensitive-upgrade (NSU) runtime checks of Austin and Flanagan [21]. Fennell and Thiemann [60] design a cast calculus for an imperative, object-oriented language.

The main property of gradually typed languages is the *gradual guarantee* [61], which states that removing type annotations should not change the runtime behavior. Adding type annotations should also result in the same behavior except that it may introduce more trapped errors because those new type annotations may contain mistakes. Since the formulation of the gradual guarantee as a criterion for gradually-typed languages [61], researchers have explored the feasibility of satisfying both the gradual guarantee and noninterference. Toro, Garcia, and Tanter [62] identify a tension between the gradual guarantee and security enforcement; they analyze the semantics of runtime casts through the lens of Abstracting Gradual Typing [63] and propose a type-driven semantics for gradual security. However, Toro, Garcia, and Tanter [62] discover counterexamples to the gradual guarantee in the GSL_{Ref} language. Therefore, they conjecture that it is not possible to enforce noninterference and satisfy the gradual guarantee at the same time.

Azevedo de Amorim, Fredrikson, and Jia [64] conjecture one possible source of the tension: the *type-guided classification* performed in GSL_{Ref} [62]. They propose a new gradually typed language, GLIO, which sacrifices type-guided classification. They prove

that GLIO satisfies both noninterference and the gradual guarantee using a denotational semantics. Our previous design λ_{SEC}^* [65] follows GLIO and also sacrifices type-guided classification. Bichhawat, McCall, and Jia [66] conjecture that *NSU checking* could be another possible source of the tension. As an alternative, they propose a hybrid approach that leverages static analysis ahead of program execution to determine the write effects in untaken branches. They study a simple imperative language with first-order stores (called WHILE^G) and prove both noninterference and the gradual guarantee for it.

An ideal gradual IFC language (1) should enforce security by satisfying noninterference, (2) should satisfy the gradual guarantee, (3) should provide type-based reasoning through vigilance and type-guided classification, (4) and should not require additional static analyses prior to program execution. Contrary to the prior work, I will show in this dissertation that one does not have to give up on any of the four requirements to resolve the tension between noninterference and the gradual guarantee. Instead, the real source of the tension is that GSL_{Ref} allows \star as a *runtime* security label. By walking back this usual design choice, I present a gradual IFC programming language λ_{IFC}^* and prove that λ_{IFC}^* satisfies both noninterference and the gradual guarantee without any sacrifices.

1.4 Key Enabling Insight

In GSL_{Ref} , one can write a literal such as true_\star in a program. At runtime, the literal becomes a value of unknown security level. I observe that allowing \star as a *runtime* security label is the main reason that GSL_{Ref} violates the gradual guarantee. That design choice of GSL_{Ref} was unusual when compared to other gradually-typed languages, because the unknown type \star is traditionally used in gradual languages to represent the lack of static information, not the lack of dynamic information. The design is also unusual when compared to dynamic systems for IFC, as those systems do not use an unknown security level [20, 21, 22, 23, 24].

Based on this observation, I propose a new gradual, IFC language λ_{IFC}^* , which (1) en-

forces information flow security, (2) satisfies the gradual guarantee, (3) enjoys type-based reasoning through free theorems, and (4) utilizes NSU checking to enforce implicit flows through the heap with no static analysis required. In λ_{IFC}^* , runtime security labels do not include \star , only `low` and `high` (or any lattice of security labels). On the other hand, to support gradual typing, the security labels in a type annotation may include \star . Surprisingly, I discover that removing \star from the runtime labels is sufficient to reclaim the gradual guarantee, without sacrificing type-guided classification as in GLIO or NSU checking as in WHILE^G. This finding is the primary contribution of this dissertation. In λ_{IFC}^* , the security level of a literal defaults to `low`, similar to systems like Jif [67] and GLIO, but different from GSL_{Ref} and WHILE^G.

One might think that allowing \star as a label on literals and therefore on values is necessary so that programmers can run legacy code (without any security annotations) in a gradual language, by making \star the default label for literals. However, prior information-flow languages use `low` security as the default security label for literals [67] and for good reasons. The security of a literal is something that only the programmer can know. That is, the identification of high-security data in a program must be considered as an input to an information flow system, and not something that can or should be inferred. When migrating legacy code into a system that supports secure information flow, a necessary part of the process for the programmer is to identify whether there is any high-security information in the legacy code. The choice of `low` as the default label is because most literals (if not all) in real programs are low security. In fact, it is bad practice to embed high-security literals, such as passwords, in program text.

1.5 Thesis Statement

A gradual IFC programming language is able to satisfy the gradual guarantee while supporting type-based reasoning, as long as there is no \star among runtime security labels, and the language uses security coercions as the representation for casts:

My thesis:

It is possible to design a gradual IFC programming language that satisfies both noninterference and the gradual guarantee while supporting type-based reasoning, by excluding the unknown label \star from runtime security labels and using security coercions to represent casts.

1.6 Technical Contributions and Outline

The semantics of $\lambda_{\text{IFC}}^{\star}$ is given by translation to a new security cast calculus λ_{IFC}^c . I first define a syntax, type system, and operational semantics for λ_{IFC}^c . I then compile $\lambda_{\text{IFC}}^{\star}$ into λ_{IFC}^c in a type-preserving way.

In λ_{IFC}^c , *security coercions* serve as the runtime security monitor, in which I adapt ideas from the Coercion Calculus [68, 69] to IFC. A coercion on security labels can be an *injection* from a security label to \star or a *projection* from \star to a security label. Security labels on literals in $\lambda_{\text{IFC}}^{\star}$ become the sources of injections in λ_{IFC}^c , while runtime NSU checks are treated as a special kind of projection where the target is the security level of the memory location to modify. Information flow policies are enforced when security coercions are reduced to their normal forms. *Composing* coercions models *explicit flows*, while the action of *stamping* a coercion in normal form models *implicit flows*.

Compared to prior work on gradual IFC languages, the λ_{IFC}^c cast calculus supports an additional feature called *blame tracking* [70]. Blame tracking is important because it enables modular runtime error messages. As an example of their practicality, they play an important role in production-quality languages such as Typed Racket [71, 72].

Compared to prior work on gradual IFC based on abstracting gradual typing (AGT) [62], my use of a cast calculus makes it clear where in the program there exists runtime overhead from dynamic checking (i.e., the casts). This is important for the programmer to know

because one may wish to avoid runtime overhead in hot regions of a programs. In the translation from λ_{IFC}^* to λ_{IFC}^c , casts are only inserted where there is insufficient information during compilation to decide whether or not a security policy is enforced. In particular, casts are not inserted in statically typed regions. In this way, information flows that are statically ensured to be safe will never be checked again at runtime. On the contrary, the AGT mechanism for dynamic checking (called evidence) is attached to most nodes in the syntax tree.

In summary, my dissertation makes the following technical contributions:

1. I identify the main cause of the tension between information flow security and the gradual guarantee in GSL_{Ref} : the inclusion of \star in the runtime security labels. I discussed the key insights in §1.4 and will show example programs in §2.2.2.
2. I define λ_{IFC}^* in §2.1. It is the first gradual IFC language with type-based reasoning (§2.2.3) that satisfies both noninterference and the gradual guarantee.
3. I define two coercion calculi that serve as the runtime IFC monitor: a coercion calculus for security labels (§3.2) and a coercion calculus for secure values (§3.3).
4. I define a cast calculus λ_{IFC}^c with IFC (§3.4) that defines the dynamic semantics of λ_{IFC}^* .
5. The Agda formalization of λ_{IFC}^* and its cast calculus λ_{IFC}^c (§1.7).
6. I prove type safety for λ_{IFC}^* (Theorem 9).
7. I prove the gradual guarantee for λ_{IFC}^* (Theorem 21).
8. I prove noninterference for λ_{IFC}^* (Theorem 45), by simulating its cast calculus λ_{IFC}^c with its dynamic extreme $\lambda_{\text{IFC}}^{\text{DYN}}$.

The rest of this dissertation is organized as follows:

- In Chapter 2, I introduce the gradual IFC language λ_{IFC}^* . I first define security labels and types (§2.1.1). I then present the syntax (§2.1.2), type system (§2.1.3), and semantics of λ_{IFC}^* (§2.1.4). After that, I demonstrate how λ_{IFC}^* works using examples (§2.2). Finally, I define the static and the dynamic extremes of λ_{IFC}^* (§2.3). The dynamic extreme, called $\lambda_{\text{IFC}}^{\text{DYN}}$, is the more interesting one, so I compare $\lambda_{\text{IFC}}^{\text{DYN}}$ to other dynamic IFC languages in the literature, such as λ^{info} and LIO (§2.3.4).
- In Chapter 3, I present the design of a new IFC cast calculus λ_{IFC}^c . In λ_{IFC}^c , we use security coercions as our cast representation. I first explain the benefits of using coercions to model security checks (§3.1). I then introduce two coercion calculi: one for security labels (§3.2) and the other for the values of λ_{IFC}^c (§3.3). Finally, I define the syntax, type system, and semantics of λ_{IFC}^c (§3.4).
- In Chapter 4, I define a type-preserving compilation from λ_{IFC}^* to its cast calculus λ_{IFC}^c , so that the semantics of λ_{IFC}^* can be given by λ_{IFC}^c . I first present the coerce functions that produce coercions between security labels and types (§4.1). I then define the compilation function from λ_{IFC}^* to λ_{IFC}^c (§4.2).
- In Chapter 5, I prove type safety for λ_{IFC}^* , so untrapped errors (or “undefined behaviors”) never occur in λ_{IFC}^* . The proof follows a three-step approach: first, I prove type safety for the cast calculus λ_{IFC}^c by progress and preservation (§5.1). Second, I prove that compiling from λ_{IFC}^* to λ_{IFC}^c preserves types (§5.2). Finally, I prove type safety for λ_{IFC}^* , which says that the evaluation result of a λ_{IFC}^* program is always well-typed (§5.3).
- In Chapter 6, I prove the gradual guarantee for λ_{IFC}^* . The proof follows a three-step approach: first, I prove a simulation lemma between more and less precise security coercion sequences (§6.1). Second, I prove another simulation lemma, between λ_{IFC}^c

terms of different precision (§6.2). Finally, I prove the gradual guarantee of λ_{IFC}^* by using the simulation lemma of λ_{IFC}^c (§6.3).

- In Chapter 7, I prove noninterference for λ_{IFC}^* by simulating its cast calculus with the dynamic extreme. I first show that security coercions can check information flow, because coercion sequences strongly normalize and the normalization is deterministic (§7.1). I then present the proof of noninterference, which again follows a three-step approach. In the first step, I prove noninterference for the dynamic extreme, $\lambda_{\text{IFC}}^{\text{DYN}}$ (§7.2). This proof for $\lambda_{\text{IFC}}^{\text{DYN}}$ follows the standard erasure technique. In the second step, I prove a simulation lemma between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$, which says that a λ_{IFC}^c term always produces a value that is as secure as the one produced by its related $\lambda_{\text{IFC}}^{\text{DYN}}$ term (§7.3). The noninterference property of λ_{IFC}^c follows directly from the simulation lemma and the noninterference of $\lambda_{\text{IFC}}^{\text{DYN}}$. Finally, in the third step, I prove noninterference for λ_{IFC}^* as a corollary of the noninterference result of λ_{IFC}^c (§7.4).
- In Chapter 8, I discuss future research directions about λ_{IFC}^* as well as gradual IFC in general (§8.1). After that, I summarize the entire dissertation (§8.2).

1.7 Data Availability

The accompanying Agda development, which contains the mechanization of λ_{IFC}^* and its cast calculus λ_{IFC}^c , is available online in the following software repository:

<https://github.com/Gradual-Typing/LambdaIFCStar>

CHAPTER 2

GRADUAL INFORMATION-FLOW CONTROL (IFC) IN λ_{IFC}^*

In this chapter, I present λ_{IFC}^* , which is the first language that satisfies both noninterference and the gradual guarantee while supporting type-guided classification. Previously, Toro, Garcia, and Tanter [62] designed GSL_{Ref} , a language that satisfies noninterference and supports type-guided classification but sacrifices the gradual guarantee. Azevedo de Amorim, Fredrikson, and Jia [64] designed GLIO, a language that satisfies noninterference and the gradual guarantee but sacrifices type-guided classification. In addition, Azevedo de Amorim, Fredrikson, and Jia [64] identify the exclusion of type-guided classification as the reason that GLIO satisfies the gradual guarantee and GSL_{Ref} does not.

λ_{IFC}^* satisfies noninterference and the gradual guarantee while supporting type-guided classification because it (1) excludes \star from runtime labels and (2) uses security coercions to represent casts. Before I designed λ_{IFC}^* , I followed GLIO and created λ_{SEC}^* [65]. Like GLIO and unlike λ_{IFC}^* , the language λ_{SEC}^* does not use coercions as the cast representation and does not support type-guided classification. During the development of λ_{SEC}^* , I realized that NSU checking could be modeled using a sequence of casts, which could be expressed as coercions. The source of the first coercion and the target of the last coercion must be specific (**low** or **high**) no matter how long the sequence is. The key to designing a gradual IFC language is to identify the source of an injection and the target of a projection. Injection happens when a value is cast from some specific security label to \star . A constant must be associated with a specific security label so that when the constant is injected, its label becomes the source of the injection. Implicit flow causes the program counter (PC) to be injected when branching on an injected branch condition. Projection, on the other hand, happens when an injected value is cast to a specific security label. NSU checking is a projection: the PC is cast to the security level of the memory location to write to. If successful,

the projection justifies the heap policy that the memory location is at least as secure as PC. As a result, each memory location should also be associated with a specific security label. By excluding \star from runtime labels and using coercions as the cast representation, I came up with a new IFC language λ_{IFC}^* [73]. Like GLIO and unlike GSL_{Ref} , λ_{IFC}^* satisfies the gradual guarantee. Like GSL_{Ref} and unlike GLIO, λ_{IFC}^* supports type-guided classification, so λ_{IFC}^* enjoys the same type-based reasoning capabilities through free theorems as GSL_{Ref} .

The rest of this chapter is organized as follows. I first define the gradual IFC surface language λ_{IFC}^* , by presenting its syntax, type system, and semantics in §2.1. After that, I put λ_{IFC}^* into action in §2.2. I present example programs to demonstrate how λ_{IFC}^* enables a gradual and smooth transition between static and dynamic IFC, while supporting type-based reasoning and satisfying the gradual guarantee at the same time. Finally, in §2.3, I conclude the chapter by discussing the static and the dynamic extremes of λ_{IFC}^* . The static extreme, SSL_{Ref} , is a static IFC language. The dynamic extreme, $\lambda_{\text{IFC}}^{\text{DYN}}$ is a dynamic IFC language. λ_{IFC}^* enables a continuum of IFC enforcement between the static and dynamic extremes that it embeds.

2.1 The Gradual IFC Language λ_{IFC}^*

This section is organized as follows. I first define security labels and types in Section 2.1.1. I then present the syntax of λ_{IFC}^* in Section 2.1.2 and the type system for λ_{IFC}^* in Section 2.1.3. I discuss the semantics of λ_{IFC}^* by defining `eval` in Section 2.1.4. Finally, I define whole programs of λ_{IFC}^* in Section 2.1.5.

2.1.1 Security Labels and Types

I define security labels in Figure 2.1. For simplicity, I will use a two-point security lattice $\langle \{\text{high}, \text{low}\}, \leq, \vee, \wedge \rangle$, where `high` is for private, sensitive data and `low` is for public, disclosable data. Of course, any lattice of security labels could be used in place of `low` and `high`. The ordering is standard: `low` \leq `high` and `high` $\not\leq$ `low`. So information is

specific security labels	ℓ	\in	$\{\text{low}, \text{high}\}$
security labels	g	$::=$	$\star \mid \ell$
base types	ι	$::=$	$\text{Unit} \mid \text{Bool}$
raw types	T, S	$::=$	$\iota \mid A \xrightarrow{g} B \mid \text{Ref } (T_g)$
types	A, B	$::=$	T_g

Figure 2.1: Security labels and types

blame labels	p, q	
terms	L, M, N	$::= x \mid (\$ k)_\ell \mid (\lambda^g x : A. N)_\ell \mid (L M)^p$ $\mid (\text{if } L \text{ then } M \text{ else } N)^p \mid \text{let } x = M \text{ in } N$ $\mid (\text{ref } \ell M)^p \mid !^p M \mid (L := M)^p \mid (M : A)^p$

Figure 2.2: Syntax of λ_{IFC}^* (highlighted security labels ℓ default to `low` if omitted)

allowed to flow from public sources to private sinks but not the other way around. I refer to $\{\text{high}, \text{low}\}$ as *specific security labels*.

I also define security types in Figure 2.1. Types in λ_{IFC}^* have security labels associated with them, for example, $\text{Bool}_{\text{high}}$ is the type for booleans with high security, Unit_{low} is the type for the unit value with low security, and Bool_\star is the type of a boolean whose security level is unknown at compile time. I refer to $\{\text{high}, \text{low}, \star\}$ as *security labels*. A function type $(A \xrightarrow{g_2} B)_{g_1}$ carries an additional security label g_2 , which is the type of the program counter (PC) to evaluate the body of the function.

2.1.2 Syntax of λ_{IFC}^*

The syntax of the gradual language λ_{IFC}^* is shown in Figure 2.2. For readers familiar with GSL_{Ref} , the syntax of λ_{IFC}^* is similar to that of GSL_{Ref} . The main syntactic difference is that in λ_{IFC}^* , the security labels of literals and newly created memory cells (highlighted in Figure 2.2) default to a specific label such as `low`, while in GSL_{Ref} they default to a runtime unknown security level \star . I am going to show in Section 2.2.2 that the design choice of defaulting to a specific security label helps us resolve the tension between noninterference and the gradual guarantee in λ_{IFC}^* .

To enable information-flow control, λ_{IFC}^* allows the programmer to annotate constants,

mutable references, and λ -abstractions with a specific security label. λ_{IFC}^* ensures that if a value is annotated with `high`, it will not flow into a sink that is `low` security. As I mentioned, if the programmer does not annotate a value with a label, λ_{IFC}^* defaults the value's label to `low`, so `true` is shorthand for `truelow`. λ_{IFC}^* supports higher-order functions, mutable references, and explicit type annotations. One thing to note is that compared with literals, λ -abstractions in λ_{IFC}^* carry an additional security label annotation g , which is the type of the PC label expression used to evaluate the body of the λ . Same as the security labels in type annotations, g defaults to \star if omitted.

Some terms in λ_{IFC}^* are annotated with an identifier called a blame label (p). When compiled to the intermediate representation, those terms generate runtime checking (casts) that may fail. In case a check fails, it raises a cast error, called *blame*, that contains its blame label. In this way, the programmer knows which cast is causing the problem and which part of the program generates that cast.

2.1.3 Type System of λ_{IFC}^*

I first define operators and relations on security labels and security types. I then define the type system of λ_{IFC}^* using those operators and relations.

Operators and Relations on Security Labels and Types

Figure 2.3 presents auxiliary operators on security labels and types. These operators include join w.r.t precision, consistent join, consistent meet, and the stamping operation on types. The operators are standard by following those of GSL_{Ref} and GLIO . Join w.r.t precision returns the least upper bound of the precision of two labels or two types, for example

$$(\text{Bool}_{\text{low}} \xrightarrow{\star} \text{Bool}_{\text{high}})_{\star} \sqcup (\text{Bool}_{\text{low}} \xrightarrow{\text{low}} \text{Bool}_{\text{high}})_{\star} = (\text{Bool}_{\text{low}} \xrightarrow{\text{low}} \text{Bool}_{\text{high}})_{\star}$$

$$\begin{array}{l}
\ell \sqcup \ell = \ell \quad \boxed{g \sqcup g} \\
\star \sqcup g = g \\
g \sqcup \star = g \\
\iota \sqcup \iota = \iota \quad \boxed{T \sqcup T} \\
(\mathbf{Ref} A) \sqcup (\mathbf{Ref} B) = \mathbf{Ref} (A \sqcup B) \\
(A \xrightarrow{g_1} B) \sqcup (C \xrightarrow{g_2} D) = (A \sqcup C) \xrightarrow{g_1 \sqcup g_2} (B \sqcup D) \\
S_{g_1} \sqcup T_{g_2} = (S \sqcup T)_{g_1 \sqcup g_2} \quad \boxed{A \sqcup A} \\
\\
\ell_1 \tilde{\vee} \ell_2 = \ell_1 \vee \ell_2 \quad \boxed{g \tilde{\vee} g} \\
- \tilde{\vee} \star = \star \\
\star \tilde{\vee} - = \star \\
\iota \tilde{\vee} \iota = \iota \quad \boxed{T \tilde{\vee} T} \\
(\mathbf{Ref} A) \tilde{\vee} (\mathbf{Ref} B) = \mathbf{Ref} (A \sqcup B) \\
(A \xrightarrow{g_1} B) \tilde{\vee} (C \xrightarrow{g_2} D) = (A \tilde{\wedge} C) \xrightarrow{g_1 \tilde{\wedge} g_2} (B \tilde{\vee} D) \\
S_{g_1} \tilde{\vee} T_{g_2} = (S \tilde{\vee} T)_{g_1 \tilde{\vee} g_2} \quad \boxed{A \tilde{\vee} A} \\
\\
\ell_1 \tilde{\wedge} \ell_2 = \ell_1 \wedge \ell_2 \quad \boxed{g \tilde{\wedge} g} \\
- \tilde{\wedge} \star = \star \\
\star \tilde{\wedge} - = \star \\
\iota \tilde{\wedge} \iota = \iota \quad \boxed{T \tilde{\wedge} T} \\
(\mathbf{Ref} A) \tilde{\wedge} (\mathbf{Ref} B) = \mathbf{Ref} (A \sqcup B) \\
(A \xrightarrow{g_1} B) \tilde{\wedge} (C \xrightarrow{g_2} D) = (A \tilde{\vee} C) \xrightarrow{g_1 \tilde{\vee} g_2} (B \tilde{\wedge} D) \\
S_{g_1} \tilde{\wedge} T_{g_2} = (S \tilde{\wedge} T)_{g_1 \tilde{\wedge} g_2} \quad \boxed{A \tilde{\wedge} A}
\end{array}$$

$$\mathit{stamp} (T_{g_1}) g_2 = T_{g_1 \tilde{\vee} g_2}$$

Figure 2.3: Auxiliary operators for security labels and types: join w.r.t precision ($-\sqcup-$), consistent join ($-\tilde{\vee}-$ for labels and $-\tilde{\vee}-$ for types), and consistent meet ($-\tilde{\wedge}-$ for labels and $-\tilde{\wedge}-$ for types). Stamping for types

$$\boxed{g_1 \lesssim g_2}$$

$$\lesssim^* \frac{}{g \lesssim^*} \quad \star \lesssim \frac{}{\star \lesssim g} \quad \lesssim^{-l} \frac{l_1 \leq l_2}{l_1 \lesssim l_2}$$

$$\boxed{S \lesssim T}$$

$$\lesssim^{-\iota} \frac{}{\iota \lesssim \iota} \quad \lesssim^{-ref} \frac{A \lesssim B \quad B \lesssim A}{\text{Ref } A \lesssim \text{Ref } B} \quad \lesssim^{-fun} \frac{g_2 \lesssim g_1 \quad C \lesssim A \quad B \lesssim D}{A \xrightarrow{g_1} B \lesssim C \xrightarrow{g_2} D}$$

$$\boxed{A \lesssim B}$$

$$\lesssim^{-\tau} \frac{g_1 \lesssim g_2 \quad S \lesssim T}{S_{g_1} \lesssim T_{g_2}}$$

Figure 2.4: Consistent subtyping for labels and types

Consistent join of security labels resorts to security lattice join if both labels are statically known; otherwise the operator returns \star if at least one label is \star . Consistent join of types is recursive on the structure of the types, with the PC labels and the domain types in function types being contravariant and the referenced types being invariant. The consistent meet operators of security labels and types are analogous to consistent join. The stamping operator is a shorthand that computes the consistent join of the top-level label of the type with another label, while keeping the rest of the type unchanged.

Figure 2.4 presents the definitions of consistent subtyping for security labels and types, which will be used in the type system of λ_{IFC}^* . Consistent subtyping is the composition of consistency and subtyping. The following are equivalent: (1) $A \lesssim B$, (2) $A \sim C <: B$ for some C , and (3) $A <: D \sim B$ for some D . The consistent subtyping relations in λ_{IFC}^* are standard, similar to those in GSL_{Ref} and GLIO , with the PC and domain type in a function type being contravariant (rule \lesssim^{-fun}) and the referenced type being invariant (rule \lesssim^{-ref}).

I define a precision ordering \sqsubseteq on security labels and security types. For security labels, the statically unknown label \star is the most imprecise, so $\star \sqsubseteq g$ for any label g and $\ell \sqsubseteq \ell$ for any specific security label ℓ . The precision ordering extends to types in a natural way, so for example, $\text{Bool}_{\star} \sqsubseteq \text{Bool}_{\text{low}}$ and $(\text{Bool}_{\text{low}} \xrightarrow{\star} \text{Bool}_{\text{high}})_{\star} \sqsubseteq$. Figure 2.5 gives the

$$\begin{array}{c}
\boxed{g_1 \sqsubseteq g_2} \\
\\
\star \sqsubseteq \frac{}{\star \sqsubseteq g} \qquad \ell \sqsubseteq \ell \frac{}{\ell \sqsubseteq \ell} \\
\\
\boxed{S \sqsubseteq T} \\
\\
\sqsubseteq\text{-}\iota \frac{}{\iota \sqsubseteq \iota} \quad \sqsubseteq\text{-}\text{ref} \frac{A \sqsubseteq B}{\text{Ref } A \sqsubseteq \text{Ref } B} \quad \sqsubseteq\text{-}\text{fun} \frac{g_1 \sqsubseteq g_2 \quad A \sqsubseteq C \quad B \sqsubseteq D}{A \xrightarrow{g_1} B \sqsubseteq C \xrightarrow{g_2} D} \\
\\
\boxed{A \sqsubseteq B} \\
\\
\sqsubseteq\text{-}\tau \frac{g_1 \sqsubseteq g_2 \quad S \sqsubseteq T}{S_{g_1} \sqsubseteq T_{g_2}}
\end{array}$$

Figure 2.5: Precision of security labels and types

definition of precision on types.

*Typing Rules for λ_{IFC}^**

The typing rules for λ_{IFC}^* are shown in Figure 2.6. They are directly adapted from those of GSL_{Ref} , by changing the security labels on values to disallow the \star label. The type system of GSL_{Ref} is derived from its static extreme SSL_{Ref} by replacing labels and types as well as their operators and predicates with the gradual variants, while SSL_{Ref} is in turn an adaptation of prior security-typed languages such as Fennell and Thiemann [59], Heintze and Riecke [39], and Zdancewic [38].

For example, in SSL_{Ref} the typing rule of application looks like:

$$\vdash\text{-}\text{app-SSLRef} \frac{\Gamma; pc \vdash L : (A \xrightarrow{pc'} B)_\ell \quad \Gamma; pc \vdash M : A' \quad A' <: A \quad pc \leq pc' \quad \ell \leq pc'}{\Gamma; pc \vdash (L M) : \text{stamp } B \ell}$$

where $A' <: A$ is the usual type subsumption of function argument. The side conditions $\ell \leq pc'$ and $pc \leq pc'$ restricts the PC label on the function type so that no information is leaked through side effects. The type of the application has label that is the join of the label on B and ℓ ($\text{stamp } B \ell$). In λ_{IFC}^* , the typing judgment takes the form $\Gamma; g \vdash M : A$,

$$\boxed{\Gamma; g \vdash M : A}$$

$$\begin{array}{c}
\vdash var \frac{\Gamma \ni x : A}{\Gamma; g \vdash x : A} \quad \vdash const \frac{k : \iota}{\Gamma; g \vdash (\$ k)_\ell : \iota_\ell} \\
\vdash lam \frac{(\Gamma, x:A); g_2 \vdash N : B}{\Gamma; g_1 \vdash (\lambda^{g_2} x:A. N)_\ell : (A \xrightarrow{g_2} B)_\ell} \\
\vdash app \frac{\Gamma; g \vdash L : (A \xrightarrow{g_2} B)_{g_1} \quad \Gamma; g \vdash M : A' \quad A' \lesssim A \quad g \lesssim g_2 \quad g_1 \lesssim g_2}{\Gamma; g \vdash (L M)^P : stamp B g_1} \\
\vdash let \frac{\Gamma; g \vdash M : A \quad (\Gamma, x:A); g \vdash N : B}{\Gamma; g \vdash \text{let } x = M \text{ in } N : B} \\
\vdash if \frac{\Gamma; g_2 \vdash L : \text{Bool}_{g_1} \quad \Gamma; g_2 \tilde{\simeq} g_1 \vdash M : A \quad \Gamma; g_2 \tilde{\simeq} g_1 \vdash N : B \quad A \tilde{\simeq} B = C}{\Gamma; g_2 \vdash (\text{if } L \text{ then } M \text{ else } N)^P : stamp C g_1} \\
\vdash ref \frac{\Gamma; g_2 \vdash M : T_{g_1} \quad T_{g_1} \lesssim T_\ell \quad g_2 \lesssim \ell}{\Gamma; g_2 \vdash (\text{ref } \ell M)^P : (\text{Ref } T_\ell)_{\text{low}}} \quad \vdash deref \frac{\Gamma; g_2 \vdash M : (\text{Ref } A)_{g_1}}{\Gamma; g_2 \vdash !^P M : stamp A g_1} \\
\vdash assign \frac{\Gamma; g_2 \vdash L : (\text{Ref } T_{\hat{g}})_{g_1} \quad \Gamma; g_2 \vdash M : A \quad A \lesssim T_{\hat{g}} \quad g_2 \lesssim \hat{g} \quad g_1 \lesssim \hat{g}}{\Gamma; g_2 \vdash (L := M)^P : \text{Unit}_{\text{low}}} \quad \vdash ann \frac{\Gamma; g \vdash M : A' \quad A' \lesssim A}{\Gamma; g \vdash (M : A)^P : A}
\end{array}$$

Figure 2.6: Typing rules of λ_{IFC}^* . Side conditions about the heap policy are highlighted

evaluation result $r ::= k \mid \text{fun} \mid \text{addr} \mid \text{diverge} \mid \text{stuck}$

$$\boxed{obs(V) = r}$$

$$\begin{aligned} obs(\$ k) &= k \\ obs(\$ k \langle c \rangle) &= k \\ obs(\lambda x. N) &= \text{fun} \\ obs((\lambda x. N) \langle c \rangle) &= \text{fun} \\ obs(\text{addr } n) &= \text{addr} \\ obs((\text{addr } n) \langle c \rangle) &= \text{addr} \end{aligned}$$

$$\boxed{eval(M, b) = r}$$

$$\begin{aligned} eval(M, b) &= obs(V) && \text{if } (\mathcal{C} M)[x := \$ b] \mid \emptyset \Downarrow V \mid \mu \\ eval(M, b) &= \text{diverge} && \text{if } (\mathcal{C} M)[x := \$ b] \mid \emptyset \Downarrow \text{blame } p \mid \mu \\ &&& \text{or } (\mathcal{C} M)[x := \$ b] \mid \emptyset \Uparrow \\ eval(M, b) &= \text{stuck} && \text{otherwise} \end{aligned}$$

Figure 2.7: Evaluation of λ_{IFC}^*

where the static PC label g and the type A become gradual (may be or contain \star). Like GSL_{Ref} , I replace label partial order with label consistent subtyping, type subtyping with type consistent subtyping, and label join with label consistent join and get rule \vdash_{app} .

The only major difference from the type system of GSL_{Ref} is that because of the concrete label restriction on the syntax of constants and λ -abstractions, these terms must have concrete labels at the top level of their respective types (rule \vdash_{const} and \vdash_{lam}). Similarly, the type of the value in a newly allocated cell (rule \vdash_{ref}) has a concrete top-level label: $(\text{Ref } T_{\ell})_{\text{low}}$. The reference itself has a `low` label because it is newly created and cannot leak information.

2.1.4 Semantics of λ_{IFC}^*

I define the evaluation function ($eval$ in Figure 2.7) for λ_{IFC}^* by (1) compiling from λ_{IFC}^* to λ_{IFC}^c , (2) modeling input using substitution, and (3) running the compiled λ_{IFC}^c term using

the operational semantics of λ_{IFC}^c . I will define the compilation function in Chapter 4. The compilation function is of the form $\mathcal{C} M = M'$, which takes a well-typed λ_{IFC}^* term M and returns a cast-calculus (λ_{IFC}^c) term M' . I will define the $M \mid \mu \Downarrow N \mid \mu'$ operator, which evaluates a λ_{IFC}^c term M to a value or a blame, as well as the $M \mid \mu \Uparrow$ operator, which means the λ_{IFC}^c term M diverges, in Chapter 3. The evaluation function is of the form $\text{eval}(M, b) = r$. It takes a well-typed λ_{IFC}^* term $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M : A$ and a boolean constant $b \in \{\text{true}, \text{false}\}$ as input. The evaluation of M either produces the observation of a value $\text{obs}(V)$, or diverges, or gets stuck. We treat runtime errors as a special case of diverge. This is because in the context of IFC, one may not want blame information to be observable in a production system as it could reveal information. Instead, we force the program to diverge whenever blame is detected and send a private error message to the software developer for debugging purposes.

2.1.5 λ_{IFC}^* Programs

A λ_{IFC}^* program is defined as a λ_{IFC}^* term that takes a high-security boolean input and returns a low-security boolean output:

Definition 1 (Whole programs of λ_{IFC}^*). *A λ_{IFC}^* program is a λ_{IFC}^* term that is well-typed at $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M : \text{Bool}_{\text{low}}$.*

2.2 λ_{IFC}^* in Action

This section is organized as follows. In Section 2.2.1, I review the basics of gradual IFC using λ_{IFC}^* programs. I show that λ_{IFC}^* enables a gradual transition between static and dynamic IFC. In Section 2.2.2, I review the counterexamples of Toro, Garcia, and Tanter [62] and demonstrate that the tension between security and the gradual guarantee can be solved by removing \star from the runtime security labels. Finally, in Section 2.2.3, I show that λ_{IFC}^* enables the same type-based reasoning capabilities through free theorems as GSL_{Ref} , because λ_{IFC}^* is vigilant and performs type-guided classification.

2.2.1 The Gradual Transition Between Static and Dynamic IFC in λ_{IFC}^*

In this section I review the basic concepts of gradual information flow control using λ_{IFC}^* , establishing the intuition that λ_{IFC}^* enables a smooth, gradual transition between static and dynamic IFC. I start with fully static λ_{IFC}^* programs and show that λ_{IFC}^* can behave like a static security-typed language, guarding against both illegal explicit and implicit flows at compile time. I then replace some security label annotations in types with \star , so that the programs become partially typed and the typing information alone is insufficient to enforce IFC. I show that security coercions, serving as the runtime security monitor of λ_{IFC}^* , are able to capture both explicit flow and implicit flow violations at runtime, thereby preventing information leakage and enforcing security.

I model I/O with two functions, `user-input` and `publish`: the former returns a high-security boolean that represents sensitive input information; the latter takes a low-security boolean and publishes it into a publicly visible channel.

Gradual IFC includes static IFC

For statically typed programs, λ_{IFC}^* behaves just like a statically typed IFC language. Consider the following well-behaved λ_{IFC}^* program that takes in a high-security user input, passes it to the function `fconst` that ignores the input and returns `false`, which is then published.

```
1 let fconst = λ b : Boolhigh. false in
2 let input  = user-input () in
3 let result = fconst input in
4   publish result
```

The program type-checks and runs without error, with no need for runtime checks to enforce security. Indeed, a malicious party cannot infer anything about the high-security input because (1) the return value of `fconst` is always the same value `false` (2) the value `false` is of low security, so the explicit flow into `publish` is allowed.

If we replace `fconst` with the identity function `fid` with parameter type `Boollow`, the program becomes ill-typed as is usual for a statically typed IFC language: the type system disallows the explicit flow from the high-security input to `fid`, which expects a low-security boolean value.

```

1 let fid      = λ b : Boollow . b in
2 let input   = user-input () in
3 let result  = fid input in      // static error
4   publish result

```

Sometimes the observable behaviors of a program can depend on its branching structure. If some of the branch conditions have a data dependency on high-security input, a malicious party might be able to infer it from the observable behaviors, giving rise to illegal *implicit flows* [36], which must be ruled out to guarantee security.

Consider the following program in which the function `flip` contains a conditional expression, whose condition is dependent on a high-security user input. Its two branches return different low-security booleans, creating a potential implicit flow from high to low.

```

1 let flip : Boolhigh -> Boollow =
2   λ b : Boolhigh . if b then false else true in
3 let input   = user-input () in
4 let result  = flip input in
5   publish result

```

Perhaps the programmer mistakenly annotated the return type of `flip` thinking that it must return `Boollow`, because both branches contain low-security values. As is typical of statically typed IFC languages, the type system of λ_{IFC}^* rejects this program, thereby preventing an information leak through an implicit flow. To see why, note that the branch condition is of high security, so the type of the `if` expression as a whole is `Boolhigh`. In particular, the type checker computes the security level of an `if` to be the join of its branches (both `low`) and the condition (`high`), yielding `low \vee high = high`. The `flip` function is ex-

pected to return `Boollow` according to its type annotation, but returns `Boolhigh` because of the conditional, `high` $\not\leq$ `low`, so the program is ill-typed.

To summarize, λ_{IFC}^* behaves just like a static security-typed language in the above examples. When everything is statically typed, the type system of λ_{IFC}^* guards against illegal information flows, whether explicit or implicit.

Gradual IFC enables a mixture of static and dynamic IFC

I have shown that security labels (`low` and `high`) can appear in type annotations in a program, such as `Boollow` and `Boolhigh`. λ_{IFC}^* also provides the *unknown security label*, written `*`, for use in type annotations. I will explain how the unknown security label works in the following discussion.

Let us return to the `fconst` example, except this time the type of parameter `b` is `Bool*`.

```
1 let fconst = λ b : Bool*. false in
2 let input  = user-input () in
3 let result = fconst input in
4   publish result
```

The type system of λ_{IFC}^* accepts this program because, in the function call `fconst input`, it allows an implicit conversion from the type of `input`, which is `Boolhigh`, to the parameter type `Bool*`. This program runs to completion and publishes `false`.

Now suppose I again replace `fconst` with `fid`, but keep the parameter type of `Bool*`.

```
1 let fid     = λ b : Bool*. b in
2 let input  = user-input () in
3 let result = fid input in
4   publish result
```

The type system of λ_{IFC}^* still accepts this program. The type of `result` changes to `Bool*` but in the call `publish result`, the type system allows an implicit conversion from `Bool*` to `Boollow`. The security leak in this program is not caught statically; instead it is caught

dynamically.

The dynamic semantics of λ_{IFC}^* is defined by compilation into the λ_{IFC}^c calculus by inserting casts. In λ_{IFC}^c , explicit casts are represented as *security coercions* that monitor the flow of information. I use the standard syntax for coercions [68] but with adaptations to handle IFC. A coercion whose target is \star (and source is not \star) is an *injection*, and is indicated by an exclamation mark. A coercion whose source is \star (and target is not \star) is a *projection*, and is indicated by a question mark. The projections perform runtime checks that may fail.

The translation from λ_{IFC}^* to λ_{IFC}^c inserts a security coercion wherever an implicit cast occurred in the type checking of the λ_{IFC}^* term. Here is the result of cast insertion on the above program:

```
1 let fid      = λ b. b in
2 let input    = user-input () in
3 let result  = fid (input <high!>) in
4   publish (result <low?p>)
```

The coercion on Line 3 (`high!`) is an injection, casting from `high` to \star . The coercion on line 4 (`low?p`) is a projection, casting from \star to `low`. At runtime, a projection checks whether the incoming value has a security level that is less than or equal to the target security level. Now suppose we run the above example with input `true`. The injection on line 3 will create an injected value `truehigh <high!>`. This value is passed to and returned from `fid`, and then projected to `low`. Because `high` is greater than `low`, the projection fails.

Each projection is annotated with an identifier called a blame label (p). In case a projection fails, it raises a cast error, called *blame*, that contains its blame label. In this way, the programmer knows which cast is causing the problem. This feature is often referred to as *blame tracking* [70, 74]. Blame tracking is especially useful during the software development process, but in the context of IFC, one may not want blame to be observable in a production system as it could reveal information. This can be handled by causing the

program to diverge whenever blame is detected, possibly sending a private error message to the software developer.

Next let us return to the `flip` example to see how gradual IFC prevents illegal implicit flows. Suppose that I change the parameter type of the λ from `Boolhigh` to `Bool*`. The return type remains `Boollow`, to conform with the signature of `publish`. The definition of `flip` (line 1 and line 2) thus becomes:

```
1 let flip : Bool* -> Boollow =
2   λ b : Bool*. if b then false else true in
```

This change makes the program well-typed in λ_{IFC}^* . The IFC enforcement of the implicit flow is deferred until runtime because the branch condition now has type `Bool*`, with an unknown security level.

Next, let us consider the runtime behavior of this program. The result of cast insertion on the λ_{IFC}^* program is the following λ_{IFC}^c term:

```
1 let flip    = λ b. ((if* b then (false <low!> )
2               else (true <low!> )) <low?p> ) in
3 let input  = user-input () in
4 let result = flip (input <high!> ) in
5   publish result
```

where the `if` is changed to `if*` because the condition expression has static security level `*`. The type checking rule for `if*` requires the two branches to have security level `*` and the security level of the `if*` as a whole is also `*`. If we run the program with `true` or `false` as input, the λ_{IFC}^c term reduces to `blame p` with either input, thus capturing the illegal implicit flow and preventing the leakage of the private user input. The following reduction shows the highlights of running this program with input `true` (`false` is analogous):

```

→* let result = (λ b. ((if* b then (false <low!>)
                        else (true <low!>)) <low?p>)) (true <high!>) in ...

→* let result = prot low ((if* (true <high!>) then (false <low!>)
                              else (true <low!>)) <low?p>) in ...

→* let result = prot low ((prot high (false <low!>)) <low?p>) in ...

→* let result = prot low (false <↑;high!> <low?p>) in ...

→* blame p

```

First, the λ is bound to `flip` and the input `true` is bound to the user input value. We then inject the input, producing the value `(true <high!>)`. We call `flip` and the `if*` branches on this injected boolean, evaluating the “then” branch to the result `(false <low!>)` and then, to protect against implicit flows, the `if*` upgrades the result to `high` to match the runtime security level of the condition `(true <high!>)`, producing `(false <↑; high!>)`. The subtype coercion \uparrow sends the security level of `false` from `low` to `high`. The last step in the body of `flip` is to apply the coercion `low?p` to the value `(false <↑; high!>)`, which errors because `high` is greater than `low`.

2.2.2 Implicit Flow, NSU Checks, Unknown Security, and the Gradual Guarantee

The tension between information-flow security and the gradual guarantee arises from an interaction between implicit flows and the use of no-sensitive-upgrade checks to guard writes to mutable references. In brief, when \star is allowed as a runtime security label, some NSU checks have to conservatively trigger an error to preserve noninterference, even though no error would occur if the label was instead `high`. But the gradual guarantee says that if a program with a precise annotation runs without error, it should also run without error when that annotation is changed to \star .

In preparation to discuss this scenario in more detail, I first review the no-sensitive-upgrade technique [21] that protects against illegal implicit flows through writes to mutable references. I then show how allowing \star as a runtime security label leads to a situation where a language designer is forced to choose between noninterference and the gradual guarantee. Finally, I show how this problem is resolved in the λ_{IFC}^* language by walking back the choice of allowing \star as a runtime security label.

The main idea of no-sensitive-upgrades is to prevent data leaks through the mutable references by terminating execution whenever the program attempts to modify a low-security heap cell in a high-security execution context. In λ_{IFC}^* , NSU checks happen at runtime when type information is insufficient to statically decide whether a heap-modifying operation is secure or not. Consider the following well-typed program in λ_{IFC}^* :

```

1 let input : Bool $\star$  = user-input () in
2 let a      = ref low true in
3 let _      = if input then a := false else a := true in
4   publish (! a)

```

The assignments to a in the two branches try to write different low-security booleans into the cell at the address in a , depending on a branch condition whose security level is statically unknown. However, at runtime the `user-input` function returns a high-security boolean, so the branch condition is actually high security, and if the writes were successful, the program would leak information via an implicit flow. Fortunately, if we run this program, it reduces to `blame` regardless of the input. The way NSU checking works in λ_{IFC}^* is that a security level is associated with the current program counter. At the point of every write that requires an NSU, the system projects the program counter’s security level to the level of the memory location, making sure that the later is at least as high as the former. In the above example, the NSU check fails because the program counter’s security is `high` but the write is to a `low` security location.

In GSL_{Ref} [62], the dynamic enforcement of IFC through the heap is also based on

NSU checks. Consider the following pair of programs adapted from Section 6.3 of their paper. The program on the left is derived from the program on the right by replacing some of the `high` annotations with the unknown label `*`. Both variants of the program type check but the more precise variant runs to completion while the less precise variant triggers an error, thus violating the gradual guarantee. Let us examine their runtime behavior in further detail.

Less precise, more dynamic:

```

1 let x = user-input () in
2 let y = ref Bool* true* in
3 if x then (y := falsehigh) else ()

```

More precise, more static:

```

1 let x = user-input () in
2 let y = ref Boolhigh truehigh in
3 if x then (y := falsehigh) else ()

```

The program on the `right` runs without error in GSL_{Ref} because, at the assignment on line 3, variable `y` references a memory cell of `high` security and the PC's security level is also `high`, so the assignment is allowed.

In contrast, when the program on the `left` is run with input `truehigh`, the assignment is conservatively rejected by the NSU check. This is because GSL_{Ref} considers `*` as corresponding to the interval `[low, high]`, and the lower bound of this interval is not greater than or equal to the `high` PC label. So we see that this more precise program (`right`) runs successfully while the less precise one (`left`) errors in GSL_{Ref} .

In λ_{IFC}^* , the `*` security label can be used in type annotations, as one would expect of a gradually-typed language, but `*` is not allowed as a runtime security label and therefore also not allowed as a label on program literals and other introduction forms. I present the formal definition of the precision relation in Figure 6.8. The precision relation takes the form $\vdash M \sqsubseteq N$, where N is the same λ_{IFC}^* program as M , except that all type annotations in N are at least as precise as those in M . In rule $\sqsubseteq\text{-const}$ and rule $\sqsubseteq\text{-lam}$, the ℓ on both sides is the same, because it denotes the security level of the constant or the function. In rule $\sqsubseteq\text{-ref}$, both sides are annotated with the same ℓ , because it is for the security level of the newly created memory location.

$$\boxed{\vdash M \sqsubseteq N}$$

$$\begin{array}{c}
\sqsubseteq\text{-const} \frac{}{\vdash (\$ k)_\ell \sqsubseteq (\$ k)_\ell} \quad \sqsubseteq\text{-var} \frac{}{\vdash x \sqsubseteq x} \\
\sqsubseteq\text{-lam} \frac{g_1 \sqsubseteq g_2 \quad A_1 \sqsubseteq A_2 \quad \vdash N_1 \sqsubseteq N_2}{\vdash (\lambda^{g_1} x : A_1 . N_1)_\ell \sqsubseteq (\lambda^{g_2} x : A_2 . N_2)_\ell} \quad \sqsubseteq\text{-app} \frac{\vdash L_1 \sqsubseteq L_2 \quad \vdash M_1 \sqsubseteq M_2}{\vdash (L_1 M_1)^p \sqsubseteq (L_2 M_2)^p} \\
\sqsubseteq\text{-if} \frac{\vdash L_1 \sqsubseteq L_2 \quad \vdash M_1 \sqsubseteq M_2 \quad \vdash N_1 \sqsubseteq N_2}{\vdash (\text{if } L_1 \text{ then } M_1 \text{ else } N_1)^p \sqsubseteq (\text{if } L_2 \text{ then } M_2 \text{ else } N_2)^p} \\
\sqsubseteq\text{-ann} \frac{\vdash M_1 \sqsubseteq M_2 \quad A \sqsubseteq B}{\vdash (M_1 : A)^p \sqsubseteq (M_2 : B)^p} \\
\sqsubseteq\text{-let} \frac{\vdash M_1 \sqsubseteq M_2 \quad \vdash N_1 \sqsubseteq N_2}{\vdash \text{let } x = M_1 \text{ in } N_1 \sqsubseteq \text{let } x = M_2 \text{ in } N_2} \\
\sqsubseteq\text{-ref} \frac{\vdash M_1 \sqsubseteq M_2}{\vdash (\text{ref } \ell M_1)^p \sqsubseteq (\text{ref } \ell M_2)^p} \quad \sqsubseteq\text{-deref} \frac{\vdash M_1 \sqsubseteq M_2}{\vdash !^p M_1 \sqsubseteq !^p M_2} \\
\sqsubseteq\text{-assign} \frac{\vdash L_1 \sqsubseteq L_2 \quad \vdash M_1 \sqsubseteq M_2}{\vdash (L_1 := M_1)^p \sqsubseteq (L_2 := M_2)^p}
\end{array}$$

Figure 2.8: Precision rules of λ_{IFC}^*

The following adapts the above examples from GSL_{Ref} to λ_{IFC}^* . The fully static variant (down) is nearly identical to its GSL_{Ref} counterpart. To obtain the less-precise program (up), we change the type annotation on variable y to model the similar loss of precision in the GSL_{Ref} counterpart. We do not change the labels on the `ref` or `true` to \star because that is not allowed in λ_{IFC}^* as we just mentioned.

Less precise, more dynamic:

```

1 let x = user-input () in
2 let y : (Ref Bool $\star$ ) $\star$  = ref high truehigh in
3   if x then (y := falsehigh) else ()

```

More precise, more static:

```

1 let x = user-input () in
2 let y : (Ref Boolhigh)high = ref high truehigh in
3   if x then (y := falsehigh) else ()

```

Branching on high-security input and assigning to a high-security memory location should

be allowed. Indeed, both variants reduce to the unit value regardless of the input, thereby not violating the gradual guarantee. The fully annotated version (`down`) evaluates to unit without any overhead from runtime checking.

In the less-precise program (`up`), $(\text{Ref Bool}_*)^*$ replaces $(\text{Ref Bool}_{\text{high}})_{\text{high}}$. This change in type annotation means that both the security level of the memory and the security of the reference itself are statically unknown and should be checked at runtime. When executing the program, at the assignment on line 3, an NSU check happens and the assignment to high-security memory under high PC is allowed. As a result, the less-precise program also evaluates successfully to unit.

One might worry that the less precise program has a heavy annotation burden. However, as we mentioned, the default security label is `low` so the programmer does not have to label constants in λ_{IFC}^* . So we can remove the labels on constants to obtain the following program, which also reduces successfully to unit:

```

1 let x = user-input () in
2 let y : (Ref Bool*)* = ref high true in
3   if x then (y := false) else ()

```

The low-security `true` is classified as high security during reference creation because the security level of the cell is `high` (line 2). Similarly, during assignment the `false` is also classified as high security because the security level of the cell (line 3). Assigning to a high-security memory cell is allowed under a high PC by the NSU check, so the program evaluates successfully to unit.

One might think that requiring the programmer to annotate the reference creation with `high` (line 2) is still a burden and that GSL_{Ref} is better in this regard. However, while GSL_{Ref} allows the unannotated version of this program to compile, it errors during program execution. It is better to require the programmer to annotate references during program development than to have the programs compile but then fail during program execution, perhaps only detected after the program is deployed.

2.2.3 Type-Based Reasoning in λ_{IFC}^*

Type-based reasoning in gradual IFC languages arises from two language design choices: vigilance and type-guided classification. Vigilance gives us type-based reasoning for explicit flows, while type-guided classification provides type-based reasoning about implicit flows. In this section, I am going to show that λ_{IFC}^* is both vigilant and performs type-guided classification, so λ_{IFC}^* enables type-based reasoning in the sense of Toro, Garcia, and Tanter [62].

λ_{IFC}^ is Vigilant*

A language with casts is *vigilant* if it checks whether all the casts that are applied to the same value are consistent with each other, and triggers an error if they are not.

Toro, Garcia, and Tanter [62] present the following example to demonstrate how vigilance is needed for type-based reasoning and free theorems in the sense of Wadler [75]:

```
1 let mix : Intlow -> Inthigh -> Intlow =  
2   λ pub priv . if pub < (priv : Int* : Intlow) then 1 else 2 in  
3 mix 1low 5low
```

The example involves casts from `low` (line 3, the label annotation on `5low`) to `high` (line 1, the type annotation `Inthigh` in the signature of `mix`) and then back to `low` via the unknown security level `*` (line 2, the nested type annotations `Int*` and `Intlow`). The type signature of `mix` should guarantee the free theorem that either (1) the result of `mix`, which is low security, never depends on the high-security `priv` argument or (2) `mix` produces a runtime error. In this case, the output of `mix` does depend on `priv` via an implicit flow, so the free theorem says that an error should be triggered at runtime. Let us focus on the three casts applied to `5low`, where the first cast sends the security level from `low` to `high` because of the type annotation on line 1, the second cast is an injection due to the first type annotation

on line 2, and the third cast projects to `low` due to the second type annotation on line 2:

$$5_{\text{low}} \langle \text{low} \Rightarrow \text{high} \rangle \langle \text{high} \Rightarrow \star \rangle \langle \star \Rightarrow \text{low} \rangle$$

In λ_{IFC}^* , these casts produce the sequence of coercions: $5_{\text{low}} \langle \uparrow; \text{high}!; \text{low}^?^p \rangle$, which trigger an error when `high` collides with `low`, blaming label p .

Similarly, the interval refinement mechanism of GSL_{Ref} detects the conflict between the intermediate cast to `high` and the final cast to `low`. Surprisingly, in GLIO and in systems prior to GSL_{Ref} [58, 59], the program runs successfully and produces 1_{low} because they are forgetful [76] regarding the intermediate cast of 5_{low} to `high` and only check that 5_{low} can be cast to `low`. Those systems still satisfy noninterference, because the labels *on values* track their security similar to fully dynamic IFC. It is just that the security labels on *type annotations* will not trigger errors at runtime, even if they are inconsistent.

λ_{IFC}^* Performs Typed-Guided Classification

A gradual IFC language employs *typed-guided classification* if the security-level of a value can be changed when the value flows through a cast.

The following example from Section 2 of Toro, Garcia, and Tanter [62] demonstrates how type-guided classification interacts with implicit flow and type-based reasoning:

```

1 let mix : Intlow -> Int* -> Intlow =
2   λ pub priv. if pub < priv then 1 else 2 in
3 let smix : Intlow -> Inthigh -> Intlow =
4   λ pub priv. mix pub priv in
5 smix 1low 5low

```

Type-based reasoning tells us that the `smix` function should either fail or return a value that does not depend on its high-security parameter `priv`. However, `smix` calls `mix` and there is an implicit flow from `priv` to the result value, so this program should fail.

In λ_{IFC}^* the program produces an error as type-based reasoning suggests. Security coercions in λ_{IFC}^c classify values, so when 5_{low} is passed into `smix` and then `mix`, it is wrapped in a coercion: $5_{\text{low}} \langle \uparrow; \text{high}! \rangle$ and is classified as high-security. Consequently, the `if` reduces to its then-branch protected with `high`. This implicit flow affects the result value of the then-branch, by (1) inserting a subtype coercion before the injection and (2) promoting the source of the injection to `high` to preserve types. So the result of the `if` is $1_{\text{low}} \langle \uparrow; \text{high}! \rangle$. The injection, whose source is `high`, collides with a projection to `low` (to match the `Intlow` return type of `mix`), causing the program to error as expected, blaming the projection.

2.3 The Static and Dynamic Extremes of λ_{IFC}^*

Gradual IFC embeds both static and dynamic IFC. In this section, we discuss the static and dynamic extremes of λ_{IFC}^* . We compare them to existing static and dynamic IFC languages in the literature.

2.3.1 The Static Extreme of λ_{IFC}^* : SSL_{Ref}

Same as GSL_{Ref} , the static extreme of λ_{IFC}^* is SSL_{Ref} . In SSL_{Ref} , all security labels in all the types are statically known; the type system alone enforces IFC. The type system of SSL_{Ref} is standard and similar to other static IFC languages in the literature, such as λ_{SEC} of Zdancewic [38] and SLam of Heintze and Riecke [39].

2.3.2 The Dynamic Extreme of λ_{IFC}^* : $\lambda_{\text{IFC}}^{\text{DYN}}$

In the dynamic extreme of GSL_{Ref} , all security labels in all the types are unknown (\star). The dynamic extreme of GSL_{Ref} is a dynamic IFC programming language named $\lambda_{\text{IFC}}^{\text{DYN}}$. In the dynamic extreme, every memory location is associated with a fixed security level. $\lambda_{\text{IFC}}^{\text{DYN}}$ is particularly useful for the noninterference proof in Chapter 7. The proof is by simulation between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$. In the rest of this section, I present the formal definition of $\lambda_{\text{IFC}}^{\text{DYN}}$ in

§2.3.3 and compare $\lambda_{\text{IFC}}^{\text{DYN}}$ to dynamic IFC languages in the literature in §2.3.4.

2.3.3 Formal Definition of $\lambda_{\text{IFC}}^{\text{DYN}}$

I define the syntax and semantics of $\lambda_{\text{IFC}}^{\text{DYN}}$ in Figure 2.9. As is standard for security-typed languages [39, 59, 62], the protection term (`prot`) ensures that the computed value and the side effects of its sub-term must be at least as secure as the security level of the protection term. In rule *prot-ctx*, the sub-term of `prot` is reduced under the join of the current PC and the security of `prot` ($pc \vee \ell$). In rule *prot-val*, the sub-term has been reduced to a value V , so `prot` stamps the label ℓ onto the value, producing a new value whose security is at least as high as ℓ . The if-conditional reduces to `prot` (rule *β -if-true* and rule *β -if-false*), capturing the implicit flows from the branch condition to the resulting value as well as through the heap.

When creating a reference, we need to perform a runtime NSU check $pc \leq \ell$ so that the memory location to write to is at least as secure as the current PC (rule *ref?-ok*); if the NSU check fails, the program errors (rule *ref?-fail*). In this way, programs such as

```
publish (! (if (input ()) (ref low false) (ref low true)))
```

will error at runtime due to NSU, guarding against illegal implicit information flows through the heap. When assigning to a reference, we perform the NSU check $pc \vee \ell \leq \hat{\ell}$ (rule *assign?-ok*); if the NSU check fails, the program errors (rule *assign?-fail*). The NSU check on assignment is equivalent to $pc \leq \hat{\ell}$ and $\ell \leq \hat{\ell}$. The first part ($pc \leq \hat{\ell}$) is similar to the check on reference creation; it ensures that the security level ($\hat{\ell}$) of the memory location to write to is at least as high as the current PC (pc). The second part ($\ell \leq \hat{\ell}$) ensures that the memory location is at least as secure as the address itself. The second part is necessary because choosing which address to assign to may leak information. Consider the example:

terms	L, M, N	$::=$	$x \mid (\$ k)_\ell \mid (\text{addr } n_{\hat{\ell}})_\ell \mid (\lambda x. N)_\ell$ $\mid L M \mid \text{if } L M N$ $\mid \text{ref}^? \ell M \mid ! M \mid L :=? M \mid \text{prot } \ell M \mid \text{error}$
values	V, W	$::=$	$(\$ k)_\ell \mid (\text{addr } n_{\hat{\ell}})_\ell \mid (\lambda x. N)_\ell$
frames	F	$::=$	$\square M \mid V \square \mid \text{if } \square M N \mid \text{ref}^? \ell \square$ $\mid ! \square \mid \square :=? M \mid V :=? \square$

$$\boxed{M \mid \mu \mid pc \longrightarrow N \mid \mu'}$$

$$\begin{array}{c}
\xi \frac{M \mid \mu \mid pc \longrightarrow M' \mid \mu'}{\text{plug } M F \mid \mu \mid pc \longrightarrow \text{plug } M' F \mid \mu'} \quad \xi\text{-err} \frac{}{\text{plug error } F \mid \mu \mid pc \longrightarrow \text{error} \mid \mu} \\
\text{prot-val} \frac{}{\text{prot } \ell V \mid \mu \mid pc \longrightarrow V \vee \ell \mid \mu} \quad \text{prot-ctx} \frac{M \mid \mu \mid pc \vee \ell \longrightarrow M' \mid \mu'}{\text{prot } \ell M \mid \mu \mid pc \longrightarrow \text{prot } \ell M' \mid \mu'} \\
\text{prot-err} \frac{}{\text{prot } \ell \text{error} \mid \mu \mid pc \longrightarrow \text{error} \mid \mu} \\
\beta \frac{}{(\lambda x. N)_\ell V \mid \mu \mid pc \longrightarrow \text{prot } \ell (N[x := V]) \mid \mu} \\
\beta\text{-if-true} \frac{}{\text{if } (\$ \text{true})_\ell M N \mid \mu \mid pc \longrightarrow \text{prot } \ell M \mid \mu} \\
\beta\text{-if-false} \frac{}{\text{if } (\$ \text{false})_\ell M N \mid \mu \mid pc \longrightarrow \text{prot } \ell N \mid \mu} \\
\text{ref}^?\text{-ok} \frac{\text{pc} \leq \ell \quad n \text{ FreshIn } \mu(\ell)}{\text{ref}^? \ell V \mid \mu \mid pc \longrightarrow (\text{addr } n_\ell)_{\text{low}} \mid (\mu, \ell \mapsto n \mapsto (V \vee \ell))} \\
\text{ref}^?\text{-fail} \frac{\text{pc} \not\leq \ell}{\text{ref}^? \ell V \mid \mu \mid pc \longrightarrow \text{error} \mid \mu} \\
\text{deref} \frac{\mu(\hat{\ell}, n) = V}{!(\text{addr } n_{\hat{\ell}})_\ell \mid \mu \mid pc \longrightarrow \text{prot } \ell V \mid \mu} \\
\text{assign}^?\text{-ok} \frac{\text{pc} \vee \ell \leq \hat{\ell}}{(\text{addr } n_{\hat{\ell}})_\ell :=? V \mid \mu \mid pc \longrightarrow (\$ \text{unit})_{\text{low}} \mid [\hat{\ell} \mapsto n \mapsto (V \vee \hat{\ell})] \mu} \\
\text{assign}^?\text{-fail} \frac{\text{pc} \vee \ell \not\leq \hat{\ell}}{(\text{addr } a)_\ell :=? V \mid \mu \mid pc \longrightarrow \text{error} \mid \mu}
\end{array}$$

Figure 2.9: The syntax and the reduction semantics of $\lambda_{\text{IFC}}^{\text{DYN}}$. The checks that enforce heap policy in reference creation and assignment are highlighted

```

1 let a1 = ref low true in
2 let a2 = ref low false in
3   (if (input ()) a1 a2) := false
4   publish (! a1)

```

On line 3, we assign to low memory and PC is low, so the first part of the check succeeds. To prevent leaking information through the heap, we must also check the security of the address against the security of the memory location. In the example above, the address is high because the branch condition is of high, and we are assigning to low memory, so the NSU check fails. The program errors, thereby preventing information leak.

2.3.4 Comparing $\lambda_{\text{IFC}}^{\text{DYN}}$ to λ^{info} and LIO

We are going to compare $\lambda_{\text{IFC}}^{\text{DYN}}$ with other dynamic IFC languages in the literature, specifically λ^{info} [21] and LIO [51].

Comparing $\lambda_{\text{IFC}}^{\text{DYN}}$ to λ^{info}

$\lambda_{\text{IFC}}^{\text{DYN}}$ is different from λ^{info} , because the security of a memory location is fixed during program execution in the former but can change in the latter. In $\lambda_{\text{IFC}}^{\text{DYN}}$, the security level of a memory location is decided by the programmer during reference creation and remains unchanged throughout program execution. In λ^{info} , on the other hand, the security level of a memory location is decided by the current PC during reference creation and can increase monotonically as the program runs.

I choose to specify the security of the memory location in the syntax of reference creation so that λ_{IFC}^* satisfies the gradual guarantee. Imagine a gradual IFC language that supports \star as an annotation on reference creation, in which the term `(ref \star true)` would create a new memory location whose security is decided at run time by the current PC and store the value `true` in it (analogous to the semantics of λ^{info}). Consider the following program:

(More precise, more static)

```
1 let i = user-input () in
2 let a = ref high true in
3 if (i : Bool*) then
4   a := false
5 else ()
6 ! a
```

This program runs successfully. On line 3, the branch condition is injected, so NSU checking happens during assignment. On line 4, PC is promoted to high by the branch condition. The program assigns to a high memory location, $\text{high} \leq \text{high}$, so the NSU check is successful. On line 6, the program dereferences address a, resulting in a high boolean that is the negation of the input.

If we change the annotation on the reference creation (line 2) and make the security of the memory location decided by PC at runtime:

(Less precise, more dynamic)

```
1 let i = user-input () in
2 let a = ref * true in
3 if (i : Bool*) then
4   a := false
5 else ()
6 ! a
```

The **less precise** program errors due to failed NSU. PC initially starts at low, so on line 2, a memory location of low security is created. The memory cell stores true. The branch condition is injected, so NSU checking happens on line 4 to ensure that the security of the memory location is at least as high as that of PC. The program attempts to write to low memory under high PC, so the NSU check fails and signals a runtime error. The more precise program runs to completion but the less precise one errors, which violates the gradual guarantee. This counterexample of the alternative design shows that to satisfy

the gradual guarantee, the security of a memory location must be specified in the syntax of reference creation, as opposed to being decided by PC.

Comparing $\lambda_{\text{IFC}}^{\text{DYN}}$ to LIO

$\lambda_{\text{IFC}}^{\text{DYN}}$ is similar to LIO in that the security of a memory location stays the same during program execution. However, compared with $\lambda_{\text{IFC}}^{\text{DYN}}$, LIO supports an additional feature: first-class labels. When creating a reference in LIO, the security of the new memory location may come from a security-label value that is produced at runtime. In comparison, in λ_{IFC}^* the programmer has to specify this security level during development. Implementing first-class labels in λ_{IFC}^* is left for future work.

CHAPTER 3

THE DEFINITION OF THE CAST CALCULUS λ_{IFC}^c

In this chapter, I first describe the motivations of using coercion as the cast representation in Section 3.1. I then present a coercion calculus for security labels in Section 3.2. I show that coercion on security labels can model both explicit and implicit information flows. After that, I define a second coercion calculus whose purpose is to cast a program value from one type to another type in Section 3.3. I use this second coercion calculus to represent casts between security types in the intermediate language λ_{IFC}^c . Finally, with both coercion calculi in hand, I present the full definition of λ_{IFC}^c in Section 3.4.

3.1 Why Coercions?

As we have seen in Section 2.2, gradual information flows can be modeled as casts. For example, the cast sequence `high` \Rightarrow `*` \Rightarrow `low` should be statically accepted but dynamically rejected, while the sequence `low` \Rightarrow `*` \Rightarrow `high` should be statically and dynamically accepted, promoting the security of data to high. Such sequences of casts can be arbitrarily long (for example, `low` \Rightarrow `high` \Rightarrow `*` \Rightarrow `*` \Rightarrow `low`), which motivates us to represent the casts on security labels as *coercions*. In λ_{IFC}^* , the source security label of a coercion sequence comes from literals, while the sink is whatever security level that the observer has: for example, the `publish` function of Section 2.2 is of `low`. Coercions can be easily sequenced and composed. Checking information flow at runtime is accomplished by reducing coercion sequences to their normal forms.

There are two noteworthy benefits of using coercions to represent IFC. First, coercions can be used to represent NSU checking while satisfying the gradual guarantee. In brief, whenever a memory location is written to, the current PC is coerced to the security level of that location. We are going to formally introduce label expressions as our representation

for PC in Section 3.2.3 and discuss NSU in detail in Section 3.4. Second, the coercion representation benefits mechanization because it enables modular reasoning. The main simulation lemma (Lemma 18) depends on the simulation results of coercion sequences and label expressions, which are stated as separate lemmas and reasoned independently in our Agda code.

3.2 A Coercion Calculus for Security Labels

In this section, I describe a coercion calculus for security labels. This coercion calculus is an important stepping stone to our representation of casts between security types. I first define the syntax, the type system, and the semantics of the coercion calculus for security labels (Section 3.2.1). I then demonstrate that we can use coercion composition to model explicit flows and use coercion stamping to model implicit flows (Section 3.2.2). Finally, I define how these coercions act on security labels by defining a language of label expressions whose meaning is defined by a reduction relation (Section 3.2.3). Label expressions are used to model security checks that enforce the heap policy in λ_{IFC}^c .

3.2.1 Syntax, Typing, and Semantics of the Coercion Calculus for Security Labels

The syntax and typing for security coercions and coercion sequences is defined in Figure 3.1. A security label is either `low`, `high`, or statically unknown (`*`). There are five security coercions: identity (`id(g)`), subtype (`↑`), injection (`ℓ!`), and projection (`ℓ?p`), and blame (`⊥p`). Projection, which corresponds to the notion of a runtime check, is the only one responsible for blame, so it carries a blame label `p`. A coercion sequence \bar{c} starts with either success `id(g)` or failure (`⊥p g1 g2`). Each coercion has a source and target type $g_1 \Rightarrow g_2$. The `id(g)` casts the label `g` to itself; `↑` promotes security from `low` to `high`; injection casts to `*` from a specific label `ℓ` and projection does the opposite. Appending a single coercion to a coercion sequence makes the target security label that of the single coercion.

Information flow is enforced in the reduction semantics of security coercions, shown

specific security labels $\ell \in \{\text{low}, \text{high}\}$
 security labels $g ::= \star \mid \ell$
 blame labels p, q
 security coercions $c, d ::= \text{id}(g) \mid \uparrow \mid \ell! \mid \ell?^p \mid \perp^p$
 coercion sequences $\bar{c}, \bar{d} ::= \text{id}(g) \mid \perp^p g_1 g_2 \mid \bar{c}; c$

$\boxed{\vdash c : g_1 \Rightarrow g_2}$

$$\frac{}{\vdash \text{id}(g) : g \Rightarrow g} \quad \frac{}{\vdash \uparrow : \text{low} \Rightarrow \text{high}} \quad \frac{}{\vdash \ell! : \ell \Rightarrow \star}$$

$$\frac{}{\vdash \ell?^p : \star \Rightarrow \ell} \quad \frac{}{\vdash \perp^p : \text{high} \Rightarrow \text{low}}$$

$\boxed{\vdash \bar{c} : g_1 \Rightarrow g_2}$

$$\frac{}{\vdash \text{id}(g) : g \Rightarrow g} \quad \frac{\vdash \bar{c} : g_1 \Rightarrow g_2 \quad \vdash c : g_2 \Rightarrow g_3}{\vdash \bar{c}; c : g_1 \Rightarrow g_3} \quad \frac{}{\vdash \perp^p g_1 g_2 : g_1 \Rightarrow g_2}$$

$\boxed{\text{NF } \bar{c}}$

$$\frac{}{\text{NF id}(g)} \quad \frac{}{\text{NF id}(\star); \ell?^p} \quad \frac{\text{NF } \bar{c}}{\text{NF } \bar{c}; \ell!} \quad \frac{\text{NF } \bar{c}}{\text{NF } \bar{c}; \uparrow}$$

$\boxed{c; c \longrightarrow c}$

$$?-id \frac{}{\ell!; \ell?^p \longrightarrow \text{id}(\ell)} \quad ?-\uparrow \frac{}{\text{low}!; \text{high}?^p \longrightarrow \uparrow} \quad ?-\perp \frac{}{\text{high}!; \text{low}?^p \longrightarrow \perp^p}$$

$\boxed{\bar{c} \longrightarrow \bar{d}}$

$$id \frac{\text{NF } \bar{c}}{\bar{c}; \text{id}(g) \longrightarrow \bar{c}} \quad \perp \frac{\text{NF } \bar{c} \quad \vdash \bar{c} : g_1 \Rightarrow g_2}{\bar{c}; \perp^p \longrightarrow \perp^p g_1 \text{ low}} \quad \xi-\perp \frac{\vdash c : g_2 \Rightarrow g_3}{\perp^p g_1 g_2; c \longrightarrow \perp^p g_1 g_3}$$

$$\xi_L \frac{\bar{c} \longrightarrow \bar{d}}{\bar{c}; c \longrightarrow \bar{d}; c} \quad \xi_R \frac{\text{NF } \bar{c} \quad c; d \longrightarrow c'}{\bar{c}; c; d \longrightarrow \bar{c}; c'}$$

Figure 3.1: Syntax, typing, normal forms, and semantics of security coercions and coercion sequences

in Figure 3.1. Injection followed by projection to the same label collapses to the identity ($?-id$). Flowing from `low` to `high` is allowed, so an injection from `low` followed by a projection to `high` collapses into the \uparrow coercion ($?-\uparrow$). An information flow from `high` to `low` is prohibited, so an injection to `high` followed by a projection to `low` triggers an error that blames the projection ($?-\perp$). The predicate **NF** that specifies the normal forms of coercion sequences. The reduction rules for coercion sequences are also defined in Figure 3.1. Appending $\text{id}(g)$ onto a coercion sequence reduces to the that sequence (id). The failure coercions annihilate the other coercions in the sequence (\perp and $\xi-\perp$). We choose the evaluation order in a coercion sequence to be from left to right (ξ_L and ξ_R), because that corresponds to the direction of information flow from source to sink: in the example above, `low` \Rightarrow `high` \Rightarrow \star \Rightarrow \star \Rightarrow `low`, we validate that `low` can flow to `high` before we check and reject the flow from `high` through \star to `low`.

3.2.2 Monitoring Explicit and Implicit Flows

We can model explicit flow using security coercions. We define coercion composition in Figure 3.2. We can compose two coercion sequences $(\bar{c} \circ \bar{d})$, where $\bar{c} : g_1 \Rightarrow g_2$ and $\bar{d} : g_2 \Rightarrow g_3$, to form a flow from g_1 to g_3 .

We can also model implicit flows using security coercions. The stamping operation captures the intuition of an implicit flow from the security level ℓ' to a coercion sequence \bar{c} . We define the stamping operation in Figure 3.2 as two functions, $\text{stamp}(\bar{c}, \ell)$ and $\text{stamp}!(\bar{c}, \ell)$. Both function require \bar{c} to be in normal form and that its source label is not \star . The $\text{stamp}!$ operator promotes the security of the coercion \bar{c} to be at least ℓ and then injects the coercion if necessary, while stamp only promotes the security but does not inject. These stamping operations satisfy the gradual guarantee, because when stamping on a more precise coercion sequence and a less precise coercion sequence, stamping preserves the precision relation between them (Lemma 15). The stamping operations of coercion sequences are used in the stamping operations of (1) label expressions, which are our representation of

$$\boxed{\bar{c} \circ \bar{c} = \bar{c}}$$

$$\begin{aligned} \bar{c} \circ \perp^p g_2 g_3 &= \perp^p g_1 g_3 \quad \text{where } \vdash \bar{c} : g_1 \Rightarrow g_2 \\ \bar{c} \circ \mathbf{id}(g) &= \bar{c}; \mathbf{id}(g) \\ \bar{c}_1 \circ (\bar{c}_2; c) &= (\bar{c}_1 \circ \bar{c}_2); c \end{aligned}$$

$$\boxed{\text{stamp } \bar{c} \ell = \bar{c}}$$

$$\begin{aligned} \text{stamp } \bar{c} \text{ low} &= \bar{c} \\ \text{stamp } \mathbf{id}(\text{low}) \text{ high} &= \mathbf{id}(\text{low}); \uparrow \\ \text{stamp } \mathbf{id}(\text{high}) \text{ high} &= \mathbf{id}(\text{high}) \\ \text{stamp } (\mathbf{id}(\text{low}); \text{low}!) \text{ high} &= \mathbf{id}(\text{low}); \uparrow; \text{high}! \\ \text{stamp } (\mathbf{id}(\text{high}); \text{high}!) \text{ high} &= \mathbf{id}(\text{high}); \text{high}! \\ \text{stamp } (\mathbf{id}(\text{low}); \uparrow; \text{high}!) \text{ high} &= \mathbf{id}(\text{low}); \uparrow; \text{high}! \\ \text{stamp } (\mathbf{id}(\text{low}); \uparrow) \text{ high} &= \mathbf{id}(\text{low}); \uparrow \end{aligned}$$

$$\boxed{\text{stamp}! \bar{c} \ell = \bar{c}}$$

$$\begin{aligned} \text{stamp}! \bar{c} \text{ low} &= \begin{cases} \bar{c} & \text{if } \vdash \bar{c} : \ell \Rightarrow \star \\ \bar{c}; \ell_2! & \text{if } \vdash \bar{c} : \ell_1 \Rightarrow \ell_2 \end{cases} \\ \text{stamp}! \mathbf{id}(\text{low}) \text{ high} &= \mathbf{id}(\text{low}); \uparrow; \text{high}! \\ \text{stamp}! \mathbf{id}(\text{high}) \text{ high} &= \mathbf{id}(\text{high}); \text{high}! \\ \text{stamp}! (\mathbf{id}(\text{low}); \text{low}!) \text{ high} &= \mathbf{id}(\text{low}); \uparrow; \text{high}! \\ \text{stamp}! (\mathbf{id}(\text{high}); \text{high}!) \text{ high} &= \mathbf{id}(\text{high}); \text{high}! \\ \text{stamp}! (\mathbf{id}(\text{low}); \uparrow; \text{high}!) \text{ high} &= \mathbf{id}(\text{low}); \uparrow; \text{high}! \\ \text{stamp}! (\mathbf{id}(\text{low}); \uparrow) \text{ high} &= \mathbf{id}(\text{low}); \uparrow; \text{high}! \end{aligned}$$

Figure 3.2: Composing and stamping coercions

PC and (2) values in the cast calculus. Those three types of stamping together formalize the notion of implicit flow in λ_{IFC}^* .

Coercions capture the notion of gradual security checks. We define of the security level of a coercion sequence with the $|-|$ operator:

Definition 2 (Security level of a coercion). *Given a coercion sequence \bar{c} in normal form with type $\vdash \bar{c} : \ell \Rightarrow g$, then its security is given by the following $|-|$ operator:*

$$|\text{id}(\ell)| = \ell \quad |\text{id}(\ell); \ell!| = \ell \quad |\text{id}(\text{low}); \uparrow; \text{high}!| = \text{high} \quad |\text{id}(\text{low}); \uparrow| = \text{high}$$

The security level operator can be generalized to a larger lattice. The subtype coercion \uparrow would be parameterized by two labels: $\vdash (\uparrow_{\ell_1}^{\ell_2}) : \ell_1 \Rightarrow \ell_2$ where $\ell_1 < \ell_2$. The security would be the greater one: $|\uparrow_{\ell_1}^{\ell_2}| = \ell_2$. Here the lattice only consists of **low** and **high**, so $|\uparrow| = |\uparrow_{\text{low}}^{\text{high}}| = \text{high}$. In Chapter 7, we will formally reason about the security levels of security coercions with respect to composition and stamping.

3.2.3 Security Label Expressions

In this section we introduce security label expressions, which we use to model the security level of the PC. Security label expressions are crucial for implementing NSU checking in a way that satisfies the gradual guarantee.

A label expression is either (1) a specific security label, (2) blame (to signify an error), or (3) a coercion applied to a label expression (Figure 3.3). A label expression is in normal form (**NF**) if it is either (1) a specific security label or (2) an irreducible coercion applied to a specific security label. (A coercion is irreducible if it is a non-identity coercion in normal form). PC ranges over label expressions in normal form. The reduction relation for label expressions steps a label expression towards its normal form. The idea is that given a label expression of the form $e \langle \bar{d} \rangle$, we first reduce e to normal form and then apply the coercion \bar{d} . For example, if e reduces to a label wrapped in coercion $\ell \langle \bar{c} \rangle$, then the $lcomp$ rule says to reduce by composing the two coercions, producing $\ell \langle \bar{c}; \bar{d} \rangle$. Furthermore, in a

label expressions $e ::= \ell \mid \text{blame } p \mid e \langle \bar{c} \rangle$

$\vdash e \Leftarrow g$

$$\begin{array}{c} \vdash l \frac{}{\vdash l \Leftarrow l} \quad \vdash lcast \frac{\vdash e \Leftarrow g_1 \quad \vdash \bar{c} : g_1 \Rightarrow g_2}{\vdash e \langle \bar{c} \rangle \Leftarrow g_2} \\ \vdash lblame \frac{}{\vdash \text{blame } p \Leftarrow g} \end{array}$$

Irreducible \bar{c}

$$\frac{\mathbf{NF} \bar{c} \quad \vdash \bar{c} : g_1 \Rightarrow g_2 \quad g_1 \neq g_2}{\mathbf{Irreducible} \bar{c}}$$

NF e

$$\frac{}{\mathbf{NF} \ell} \quad \frac{\mathbf{Irreducible} \bar{c}}{\mathbf{NF} (\ell \langle \bar{c} \rangle)}$$

$e_1 \longrightarrow e_2$

$$\begin{array}{c} \xi-l \frac{e_1 \longrightarrow e_2}{e_1 \langle \bar{c} \rangle \longrightarrow e_2 \langle \bar{c} \rangle} \quad \xi-lblame \frac{}{\text{blame } p \langle \bar{c} \rangle \longrightarrow \text{blame } p} \\ \beta-id \frac{}{\ell \langle \text{id}(\ell) \rangle \longrightarrow \ell} \quad lcast \frac{\bar{c} \longrightarrow^+ \bar{d} \quad \mathbf{NF} \bar{d}}{\ell \langle \bar{c} \rangle \longrightarrow \ell \langle \bar{d} \rangle} \\ lblame \frac{\bar{c} \longrightarrow^* \perp^p \ell g}{\ell \langle \bar{c} \rangle \longrightarrow \text{blame } p} \quad lcomp \frac{\mathbf{Irreducible} \bar{c}}{\ell \langle \bar{c} \rangle \langle \bar{d} \rangle \longrightarrow \ell \langle \bar{c} \bar{d} \rangle} \end{array}$$

Figure 3.3: Syntax, typing, normal forms, and semantics of label expressions

$$\boxed{\text{stamp } e \ell = e}$$

$$\begin{aligned} \text{stamp } \ell \text{ low} &= \ell \\ \text{stamp } \text{low } \text{high} &= \text{low} \langle \text{id}(\text{low}); \uparrow \rangle \\ \text{stamp } \text{high } \text{high} &= \text{high} \\ \text{stamp } (\ell \langle \bar{c} \rangle) \ell' &= \ell \langle \text{stamp } \bar{c} \ell' \rangle \end{aligned}$$

$$\boxed{\text{stamp! } e \ell = e}$$

$$\begin{aligned} \text{stamp! } \ell \ell' &= \ell \langle \text{stamp! } \text{id}(\ell) \ell' \rangle \\ \text{stamp! } (\ell \langle \bar{c} \rangle) \ell' &= \ell \langle \text{stamp! } \bar{c} \ell' \rangle \end{aligned}$$

$$\boxed{|e| = \ell}$$

$$\begin{aligned} |\ell| &= \ell \\ |\ell \langle \bar{c} \rangle| &= |\bar{c}| \end{aligned}$$

Figure 3.4: Stamping and security level operators for security label expressions

label expression of the form $e \langle \bar{d} \rangle$, the coercion \bar{d} may also need to be reduced, which is accomplished by the *lcast* rule that refers to the reduction relation for coercion sequences (Figure 3.1). If the coercion reduces to an identity, then the coercion application goes away (*β -id*), whereas if the coercion reduces to a failure, then the label expression reduces to blame (*lblame*).

The stamping and security level operators for label expressions are defined in Figure 3.4. They both require their input to be in normal form, which can be either (1) a specific security label ℓ , or (2) a label wrapped with an irreducible coercion sequence $\ell \langle \bar{c} \rangle$. For (1), stamping **low** with **high** results in **low** $\langle \uparrow \rangle$, otherwise the label expression remains unchanged; for (2), we directly stamp the coercion sequence using *stamp* for coercion sequences defined in Figure 3.2. The definition of *stamp!* is analogous, except that it turns to the *stamp!* operator of coercion sequences. The security level operator $| - |$ is defined such that (1) a specific security label indicates the security level for itself and (2) the security of the coercion sequence $|\bar{c}|$ records the security level for $\ell \langle \bar{c} \rangle$.

In Section 3.4, I am going to describe how label expressions are used to implement

base types	ι	$::=$	Unit Bool
raw types	T, S	$::=$	ι $A \xrightarrow{g} B$ Ref (T_g)
types	A, B	$::=$	T_g
raw coercions	c_r, d_r	$::=$	$\mathbf{id}(\iota)$ $\mathbf{Ref} \ c \ d$ $(\bar{d}, c \rightarrow d)$
coercions	\mathbf{c}, \mathbf{d}	$::=$	c_r, \bar{c}

$V \langle \mathbf{c} \rangle \longrightarrow M$

$cast$	$\frac{\bar{c} \longrightarrow^+ \bar{d} \quad \mathbf{NF} \ \bar{d}}{V_r \langle c_r, \bar{c} \rangle \longrightarrow V_r \langle c_r, \bar{d} \rangle}$	$cast-blame$	$\frac{\bar{c} \longrightarrow^* \perp^p \ g_1 \ g_2}{V_r \langle c_r, \bar{c} \rangle \longrightarrow \mathbf{blame} \ p}$
$cast-id$	$\frac{}{V_r \langle \mathbf{id}(\iota), \mathbf{id}(g) \rangle \longrightarrow V_r}$	$cast-comp$	$\frac{\mathbf{Irreducible} \ c}{V_r \langle \mathbf{c} \rangle \langle \mathbf{d} \rangle \longrightarrow V_r \langle \mathbf{c} \ ; \ \mathbf{d} \rangle}$

Figure 3.5: Syntax and semantics of coercions on values

NSU checks, which enforce the heap policy for write operations.

3.3 A Coercion Calculus on Values

In this section, we define a second coercion calculus whose purpose is to cast a program value from one type to another type. We use this coercion calculus as the representation of casts in the intermediate language λ_{IFC}^c . These coercions on values make use of the coercions on security labels that we defined in Section 3.2, because the types in λ_{IFC}^c are annotated with security labels, as is usual for a static and gradually-typed IFC languages.

We begin with the definition of types in Figure 3.5, which is standard for gradual security type systems: each type has a security label ascription on it, which is either a specific label ℓ or \star . Function types have one extra label gc , which is a static approximation of the security level of the PC while executing the function body. In a reference type $(\text{Ref } T_{\hat{g}})_g$, the label \hat{g} of the referenced type also doubles as the security level of the memory cell.

The syntax and semantics for coercions on values is defined in Figure 3.5. Each coercion \mathbf{c} consists of a raw coercion c_r that casts the type of the value and the label coercion \bar{c} that casts the security label of the value. There are three kinds of raw coercions: identity coercions $\mathbf{id}(g)$, coercions between reference types $(\mathbf{Ref} \ c \ d)$, and coercions between function types $(\bar{d}, c \rightarrow d)$. In the coercion on functions, the \bar{d} casts the PC of the function.

$$\boxed{c \circ c = c}$$

$$\begin{aligned} & (\text{id}(\iota), \bar{c}) \circ (\text{id}(\iota), \bar{d}) = (\text{id}(\iota), \bar{c} \circ \bar{d}) \\ & (\mathbf{Ref} \ c_1 \ c_2, \bar{c}) \circ (\mathbf{Ref} \ d_1 \ d_2, \bar{d}) = (\mathbf{Ref} \ (d_1 \circ c_1) \ (c_2 \circ d_2), \bar{c} \circ \bar{d}) \\ & (\bar{c}_1, c_1 \rightarrow c_2, \bar{c}_2) \circ (\bar{d}_1, d_1 \rightarrow d_2, \bar{d}_2) = (\bar{d}_1 \circ \bar{c}_1, (d_1 \circ c_1) \rightarrow (c_2 \circ d_2), \bar{c}_2 \circ \bar{d}_2) \end{aligned}$$

$$\boxed{\text{Irreducible } c}$$

$$\frac{\text{NF } \bar{c} \quad \ell \neq g}{\text{Irreducible } (\text{id}(\iota), \bar{c})} \vdash \bar{c} : \ell \Rightarrow g$$

$$\frac{\text{NF } \bar{c}}{\text{Irreducible } (\mathbf{Ref} \ c \ d, \bar{c})} \quad \frac{\text{NF } \bar{c}}{\text{Irreducible } (\bar{d}, c \rightarrow d, \bar{c})}$$

Figure 3.6: Composition of coercions on values

The syntax for values is not defined until the next section, so here we remark that V ranges over values, which can either be a raw value V_r (constant, address, or λ) or an irreducible coercion applied to a raw value: $V_r \langle c \rangle$, where there is no M such that $V_r \langle c \rangle \longrightarrow M$. The definition of irreducible coercion is in Figure 3.6.

The reduction rules in Figure 3.5 apply a coercion to a value, yielding a value or triggering blame. The *cast* rule normalizes the coercion \bar{c} on the security label. If it normalizes to a failure coercion, the rule *cast-blame* triggers blame. We reduce identity coercions using rule *cast-id*. Finally, if the value is wrapped with an irreducible coercion, we compose the coercion with the coercion being applied (rule *cast-comp*). The composition operator \circ is also defined in Figure 3.6; the intuition of the composition operator is that $V_r \langle c \rangle \langle d \rangle$ must be contextual equivalent to $V_r \langle c \circ d \rangle$. There are no reduction rules specific to reference coercions $\mathbf{Ref} \ c \ d$ or function coercions $(\bar{d}, c \rightarrow d)$ because they are irreducible coercions that wrap a value. Their action occurs when the value is used in an elimination form such as in a function call or a read or write to the reference, which we explain in the next section.

variables	x, y, z	
constants	k	$\in \{\text{unit}, \text{true}, \text{false}\}$
terms	L, M, N	$::= x \mid \$ k \mid \text{addr } n \mid \lambda x. N \mid \text{app } L M A B \ell \mid \text{app}^* L M A T$ $\mid \text{if } L A \ell M N \mid \text{if}^* L T M N \mid \text{let } x=M:A \text{ in } N$ $\mid \text{ref } \ell M \mid \text{ref}^? \ell M \mid ! M A \ell \mid !^* M T$ $\mid \text{assign } L M T \hat{\ell} \ell \mid \text{assign}^? \ell M T \hat{g}$ $\mid \text{prot } e \ell M A \mid M \langle c \rangle \mid \text{blame } p$
raw values	V_r, W_r	$::= \$ k \mid \text{addr } n \mid \lambda x. N$
values	V, W	$::= V_r \mid V_r \langle c \rangle$, where Irreducible c

Figure 3.7: Syntax of the cast calculus λ_{IFC}^c

3.4 The Cast Calculus λ_{IFC}^c : An Intermediate Language For Gradual IFC

In this section, we define the cast calculus λ_{IFC}^c by presenting its syntax in Section 3.4.1, its type system in Section 3.4.2, and its operational semantics in Section 3.4.3.

3.4.1 Syntax of λ_{IFC}^c

As usual, the cast calculus λ_{IFC}^c is a statically-typed language that includes an explicit term for casts, written $M \langle c \rangle$, where M is a term and c is a coercion to be applied to the value of M . Furthermore, many of the operators in λ_{IFC}^c have two variants, a “static” one for when the pertinent security label is statically known and the “dynamic” one for when the security label is statically unknown. The operational semantics of the “dynamic” variants involve runtime checking. The syntax and typing rules for λ_{IFC}^c are shown in Figure 3.8 and described in the following paragraphs.

A value is a raw value (constant, address or λ) or an irreducible coercion applied to a raw value.

3.4.2 Type System of λ_{IFC}^c

The typing rules are syntax-directed. The typing judgment is of the form $\Gamma; \Sigma; g; \ell \vdash M \Leftarrow A$, which says that we are type-checking λ_{IFC}^c term M against the expected type A ,

$$\boxed{\Gamma; \Sigma; g; \ell \vdash M \Leftarrow A}$$

$$\begin{array}{c}
\vdash \text{var} \frac{\Gamma \ni x : A}{\Gamma; \Sigma; g; \ell \vdash x \Leftarrow A} \quad \vdash \text{const} \frac{k : \iota}{\Gamma; \Sigma; g; \ell \vdash \$ k \Leftarrow \iota_\ell} \\
\vdash \text{addr} \frac{\Sigma(\hat{\ell}, n) = T}{\Gamma; \Sigma; g; \ell' \vdash \text{addr } n \Leftarrow (\text{Ref } T_{\hat{\ell}})_\ell} \\
\vdash \text{lam} \frac{\forall \ell''. (\Gamma, x : A); \Sigma; g; \ell'' \vdash N \Leftarrow B}{\Gamma; \Sigma; g'; \ell' \vdash \lambda x. N \Leftarrow (A \xrightarrow{g} B)_\ell} \quad \vdash \text{let} \frac{\Gamma; \Sigma; g; \ell \vdash M \Leftarrow A \quad \forall \ell'. (\Gamma, x : A); \Sigma; g; \ell' \vdash N \Leftarrow B}{\Gamma; \Sigma; g; \ell \vdash \text{let } x = M : A \text{ in } N \Leftarrow B} \\
\vdash \text{app} \frac{\Gamma; \Sigma; \ell'; \ell'' \vdash L \Leftarrow (A \xrightarrow{\ell' \vee \ell} B)_\ell \quad \Gamma; \Sigma; \ell'; \ell'' \vdash M \Leftarrow A \quad C = \text{stamp } B \ell}{\Gamma; \Sigma; \ell'; \ell'' \vdash \text{app } L M A B \ell \Leftarrow C} \quad \vdash \text{app}^\star \frac{\Gamma; \Sigma; g; \ell \vdash L \Leftarrow (A \xrightarrow{\star} (T_\star))_\star \quad \Gamma; \Sigma; g; \ell \vdash M \Leftarrow A}{\Gamma; \Sigma; g; \ell \vdash \text{app}^\star L M A T \Leftarrow T_\star} \\
\vdash \text{if} \frac{\Gamma; \Sigma; \ell'; \ell'' \vdash L \Leftarrow \text{Bool}_\ell \quad \forall \ell_1. \Gamma; \Sigma; \ell' \vee \ell; \ell_1 \vdash M \Leftarrow A \quad \forall \ell_2. \Gamma; \Sigma; \ell' \vee \ell; \ell_2 \vdash N \Leftarrow A \quad B = \text{stamp } A \ell}{\Gamma; \Sigma; \ell'; \ell'' \vdash \text{if } L A \ell M N \Leftarrow B} \quad \vdash \text{if}^\star \frac{\Gamma; \Sigma; g; \ell \vdash L \Leftarrow \text{Bool}_\star \quad \forall \ell_1. \Gamma; \Sigma; \star; \ell_1 \vdash M \Leftarrow T_\star \quad \forall \ell_2. \Gamma; \Sigma; \star; \ell_2 \vdash N \Leftarrow T_\star}{\Gamma; \Sigma; g; \ell \vdash \text{if}^\star L T M N \Leftarrow T_\star} \\
\vdash \text{ref} \frac{\Gamma; \Sigma; \ell'; \ell'' \vdash M \Leftarrow T_\ell \quad \ell' \leq \ell}{\Gamma; \Sigma; \ell'; \ell'' \vdash \text{ref } \ell M \Leftarrow (\text{Ref } T_\ell)_{\text{low}}} \\
\vdash \text{ref}^\star \frac{\Gamma; \Sigma; \star; \ell' \vdash M \Leftarrow T_\ell}{\Gamma; \Sigma; \star; \ell' \vdash \text{ref}^{?P} \ell M \Leftarrow (\text{Ref } T_\ell)_{\text{low}}} \\
\vdash \text{deref} \frac{\Gamma; \Sigma; g; \ell' \vdash M \Leftarrow (\text{Ref } A)_\ell \quad B = \text{stamp } A \ell}{\Gamma; \Sigma; g; \ell' \vdash ! M A \ell \Leftarrow B} \quad \vdash \text{deref}^\star \frac{\Gamma; \Sigma; g; \ell \vdash M \Leftarrow (\text{Ref } (T_\star))_\star}{\Gamma; \Sigma; g; \ell \vdash !^\star M T \Leftarrow T_\star} \\
\vdash \text{assign} \frac{\Gamma; \Sigma; \ell'; \ell'' \vdash L \Leftarrow (\text{Ref } T_{\hat{\ell}})_\ell \quad \Gamma; \Sigma; \ell'; \ell'' \vdash M \Leftarrow T_{\hat{\ell}} \quad \ell' \vee \ell \leq \hat{\ell}}{\Gamma; \Sigma; \ell'; \ell'' \vdash \text{assign } L M T \hat{\ell} \ell \Leftarrow \text{Unit}_{\text{low}}} \\
\vdash \text{assign}^\star \frac{\Gamma; \Sigma; g; \ell \vdash L \Leftarrow (\text{Ref } T_{\hat{g}})_\star \quad \Gamma; \Sigma; g; \ell \vdash M \Leftarrow T_{\hat{g}}}{\Gamma; \Sigma; g; \ell \vdash \text{assign}^{?P} L M T \hat{g} \Leftarrow \text{Unit}_{\text{low}}} \\
\vdash \text{prot} \frac{\Gamma; \Sigma; g'; |PC| \vdash M \Leftarrow A \quad \vdash PC \Leftarrow g' \quad \ell' \vee \ell \leq |PC| \quad B = \text{stamp } A \ell}{\Gamma; \Sigma; g; \ell' \vdash \text{prot } PC \ell M A \Leftarrow B} \\
\vdash \text{cast} \frac{\Gamma; \Sigma; g; \ell \vdash M \Leftarrow A \quad \vdash \mathbf{c} : A \Rightarrow B}{\Gamma; \Sigma; g; \ell \vdash M \langle \mathbf{c} \rangle \Leftarrow B} \quad \vdash \text{blame} \frac{}{\Gamma; \Sigma; g; \ell \vdash \text{blame } p \Leftarrow A}
\end{array}$$

Figure 3.8: Typing rules of the cast calculus λ_{IFC}^c . The side conditions that enforce the heap policy statically during memory write operations are highlighted

where Γ is the typing context, Σ is the heap typing context, g is the security label that PC is typed at, and ℓ is the security level of PC. Both Σ and the security level ℓ play a role during runtime. The security level of the PC is constrained in rule \vdash_{prot} , which is for the protection term that arises during reduction (we are going to discuss this rule momentarily). In the premises for sub-terms that do not immediately reduce, such as the body of a λ and the branches of an `if`, we universally quantify the security level (as in $\forall \ell$), which helps us prove that compilation preserves types (Lemma 6). The heap context Σ is mostly standard: looking up $\Sigma(\hat{\ell}, n)$, where n is the index in part of the heap with security $\hat{\ell}$, gives us a raw type. Each memory cell is associated with a specific security label $\hat{\ell}$, which is specified by the programmer when that cell is created.

As the typing rules always stay in checking mode, constants, addresses, and λ s do not carry any label. The security of these raw values is in their types: for example, `addr $n \Leftarrow (\text{Ref } T_{\hat{\ell}})_{\ell}$` says that the security of the address n itself is ℓ and it points to a memory cell labeled $\hat{\ell}$.

The typing rules for the static variants are similar to the typing rules in a static security type system. For example, in the static function application rule \vdash_{app} , both top-level labels on the function type (ℓ and $\ell' \vee \ell$) as well as the security label that the current PC is typed at (ℓ') are static and the rule mirrors one in a static system. On the other hand, in the dynamic version of application rule \vdash_{app^*} , both top-level labels as well as the label of the co-domain type are \star and PC is allowed to be typed at \star , indicating the presence of injections. As another example, in the static version of memory assignment rule \vdash_{assign} , all labels to perform the heap policy check, including the security of the memory address itself (ℓ), the security of the memory cell that the address references ($\hat{\ell}$), and the security of PC (ℓ') are known statically and satisfy $\ell' \vee \ell \leq \hat{\ell}$. At runtime, this static assignment can happen directly without any runtime overhead (as is shown in the example of Section 2.2.2). Its dynamic counterpart rule $\vdash_{assign?}$ does not maintain these static security invariants and thus requires runtime NSU checking, which is implemented as a projection on PC.

As we shall see in the reduction rules, the semantics of the protection term `prot` performs two things: (1) `prot` promotes the security of the value reduced from its body by level ℓ (2) it uses a new PC to reduce its body. However, the new PC cannot be any label expression. It has to be one with higher security than both the current PC and the security level ℓ . We capture this invariant in side condition $\ell' \vee \ell \leq |PC|$ in rule $\vdash\textit{prot}$, where ℓ' is the security of the current PC and $|PC|$ is the security for the new PC. The invariant is used in the proof of noninterference (case *prot-ctx* in the proof of Lemma 41, which is used by Theorem 45).

3.4.3 Operational Semantics for λ_{IFC}^c

We show the the operational semantics of λ_{IFC}^c in Figure 3.9 and 3.10. The reduction relation takes the form $M \mid \mu \mid PC \longrightarrow N \mid \mu'$, which reduces the configuration of term M and heap μ under the label expression PC to another configuration N and μ' . The heap is a map $(\ell, n) \mapsto V$, where a cell is indexed by its security level ℓ and by index n among the cells of ℓ . The predicate $(n \text{ FreshIn } \mu(\ell))$ means that the index n is fresh (not already in use) among all cells with security ℓ ; when performing a lookup, $\mu(\ell, n) = V$ retrieves the value V at index n whose security level is ℓ .

Protection terms. A protection term is of the form `prot $PC' \ell M A$` . Following standard approaches to IFC, a protection term has two functionalities: (1) it ensures that the reduction inside M does not leak information through heap write operations (2) it promotes the security level of the computation result of M to at least level ℓ . The first functionality is achieved by switching to PC' from the current PC when reducing the body M (rule *prot-ctx*). Recall Section 3.4.2, the typing of `prot` makes sure that PC' has the correct security that is at least as secure as both PC and ℓ . The second functionality is achieved by stamping the value produced by the body of `prot` (rule *prot-val*). The stamping of values in λ_{IFC}^c (Figure 3.11) is analogous to stamping of label expressions and turns to the stamping

$$\boxed{M \mid \mu \mid PC \longrightarrow N \mid \mu'}$$

$$\begin{array}{c}
\xi \frac{M \mid \mu \mid PC \longrightarrow M' \mid \mu'}{\text{plug } M F \mid \mu \mid PC \longrightarrow \text{plug } M' F \mid \mu'} \quad \xi\text{-blame} \frac{}{\text{plug } (\text{blame } p) F \mid \mu \mid PC \longrightarrow \text{blame } p \mid \mu} \\
\text{prot-ctx} \frac{M \mid \mu \mid PC' \longrightarrow M' \mid \mu'}{\text{prot } PC' \ell M A \mid \mu \mid PC \longrightarrow \text{prot } PC' \ell M' A \mid \mu'} \\
\text{prot-val} \frac{}{\text{prot } PC' \ell V A \mid \mu \mid PC \longrightarrow \text{stamp-val } V A \ell \mid \mu} \\
\text{prot-blame} \frac{}{\text{prot } PC' \ell (\text{blame } p) A \mid \mu \mid PC \longrightarrow \text{blame } p \mid \mu} \quad \text{cast} \frac{V \langle c \rangle \longrightarrow M}{V \langle c \rangle \mid \mu \mid PC \longrightarrow M \mid \mu} \\
\beta \frac{}{\text{app } (\lambda x. N) V A B \ell \mid \mu \mid PC \longrightarrow \text{prot } (\text{stamp } PC \ell) \ell (N[x := V]) B \mid \mu} \\
\text{app-cast} \frac{\mathbf{NF} \bar{c} \quad (\text{stamp } PC \ell) \langle \bar{d} \rangle \longrightarrow^* PC' \quad V \langle c \rangle \longrightarrow^* W}{\text{app } (\lambda x. N \langle \bar{d}, c \rightarrow d, \bar{c} \rangle) V C D \ell \mid \mu \mid PC \longrightarrow \text{prot } PC' \ell ((N[x := W]) \langle d \rangle) D \mid \mu} \\
\text{app-blame-pc} \frac{\mathbf{NF} \bar{c} \quad (\text{stamp } PC \ell) \langle \bar{d} \rangle \longrightarrow^* \text{blame } p}{\text{app } (\lambda x. N \langle \bar{d}, c \rightarrow d, \bar{c} \rangle) V C D \ell \mid \mu \mid PC \longrightarrow \text{blame } p \mid \mu} \\
\text{app-blame} \frac{\mathbf{NF} \bar{c} \quad (\text{stamp } PC \ell) \langle \bar{d} \rangle \longrightarrow^* PC' \quad V \langle c \rangle \longrightarrow^* \text{blame } p}{\text{app } (\lambda x. N \langle \bar{d}, c \rightarrow d, \bar{c} \rangle) V C D \ell \mid \mu \mid PC \longrightarrow \text{blame } p \mid \mu} \\
\text{app}\star\text{-cast} \frac{\mathbf{NF} \bar{c} \quad (\text{stamp! } PC |\bar{c}|) \langle \bar{d} \rangle \longrightarrow^* PC' \quad V \langle c \rangle \longrightarrow^* W}{\text{app}\star (\lambda x. N \langle \bar{d}, c \rightarrow d, \bar{c} \rangle) V C T \mid \mu \mid PC \longrightarrow \text{prot } PC' |\bar{c}| ((N[x := W]) \langle d \rangle) (T_\star) \mid \mu} \\
\text{app}\star\text{-blame-pc} \frac{\mathbf{NF} \bar{c} \quad (\text{stamp! } PC |\bar{c}|) \langle \bar{d} \rangle \longrightarrow^* \text{blame } p}{\text{app}\star (\lambda x. N \langle \bar{d}, c \rightarrow d, \bar{c} \rangle) V C T \mid \mu \mid PC \longrightarrow \text{blame } p \mid \mu} \\
\text{app}\star\text{-blame} \frac{\mathbf{NF} \bar{c} \quad (\text{stamp! } PC |\bar{c}|) \langle \bar{d} \rangle \longrightarrow^* PC' \quad V \langle c \rangle \longrightarrow^* \text{blame } p}{\text{app}\star (\lambda x. N \langle \bar{d}, c \rightarrow d, \bar{c} \rangle) V C T \mid \mu \mid PC \longrightarrow \text{blame } p \mid \mu} \\
\text{if-true} \frac{}{\text{if } (\$ \text{true}) A \ell M N \mid \mu \mid PC \longrightarrow \text{prot } (\text{stamp } PC \ell) \ell M A \mid \mu} \\
\text{if-true-cast} \frac{}{\text{if } (\$ \text{true} \langle \text{id}(\text{Bool}), \text{id}(\text{low}); \uparrow \rangle) A \text{high} M N \mid \mu \mid PC \longrightarrow \text{prot } (\text{stamp } PC \text{high}) \text{high} M A \mid \mu} \\
\text{if}\star\text{-true-cast} \frac{\mathbf{NF} \bar{c}}{\text{if}\star (\$ \text{true} \langle \text{id}(\text{Bool}), \bar{c} \rangle) T M N \mid \mu \mid PC \longrightarrow \text{prot } (\text{stamp! } PC |\bar{c}|) |\bar{c}| M (T_\star) \mid \mu} \\
\text{if-false} \frac{}{\text{if } (\$ \text{false}) A \ell M N \mid \mu \mid PC \longrightarrow \text{prot } (\text{stamp } PC \ell) \ell N A \mid \mu} \\
\text{if-false-cast} \frac{}{\text{if } (\$ \text{false} \langle \text{id}(\text{Bool}), \text{id}(\text{low}); \uparrow \rangle) A \text{high} M N \mid \mu \mid PC \longrightarrow \text{prot } (\text{stamp } PC \text{high}) \text{high} N A \mid \mu} \\
\text{if}\star\text{-false-cast} \frac{\mathbf{NF} \bar{c}}{\text{if}\star (\$ \text{false} \langle \text{id}(\text{Bool}), \bar{c} \rangle) T M N \mid \mu \mid PC \longrightarrow \text{prot } (\text{stamp! } PC |\bar{c}|) |\bar{c}| N (T_\star) \mid \mu} \\
\text{let} \frac{}{\text{let } x=V:A \text{ in } N \mid \mu \mid PC \longrightarrow N[x := V] \mid \mu}
\end{array}$$

Figure 3.9: Operational semantics of λ_{IFC}^c (Part I).

$$\boxed{M \mid \mu \mid PC \longrightarrow N \mid \mu'}$$

$$\begin{array}{c}
\text{ref} \frac{n \text{ FreshIn } \mu(\ell)}{\text{ref } \ell V \mid \mu \mid PC \longrightarrow \text{addr } n \mid (\mu, \ell \mapsto n \mapsto V)} \\
\text{ref?} \frac{n \text{ FreshIn } \mu(\ell) \quad PC \langle \star \Rightarrow^p \ell \rangle \longrightarrow^* PC'}{\text{ref?}^p \ell V \mid \mu \mid PC \longrightarrow \text{addr } n \mid (\mu, \ell \mapsto n \mapsto V)} \\
\text{ref?-blame} \frac{PC \langle \star \Rightarrow^p \ell \rangle \longrightarrow^* \text{blame } q}{\text{ref?}^p \ell V \mid \mu \mid PC \longrightarrow \text{blame } q \mid \mu} \\
\text{assign} \frac{}{\text{assign } (\text{addr } n) V T \hat{\ell} \ell \mid \mu \mid PC \longrightarrow \$ \text{unit} \mid [\hat{\ell} \mapsto n \mapsto V] \mu} \\
\text{assign-cast} \frac{\mathbf{NF} \bar{c} \quad \vdash c : T_{\hat{\ell}_2} \Rightarrow S_{\hat{\ell}_1} \quad \vdash d : S_{\hat{\ell}_1} \Rightarrow T_{\hat{\ell}_2} \quad V \langle c \rangle \longrightarrow^* W}{\text{assign } (\text{addr } n \langle \mathbf{Ref} \ c \ d, \bar{c} \rangle) V T \hat{\ell}_2 \ell \mid \mu \mid PC \longrightarrow \$ \text{unit} \mid [\hat{\ell}_1 \mapsto n \mapsto W] \mu} \\
\text{assign-blame} \frac{\mathbf{NF} \bar{c} \quad \vdash c : T_{\hat{\ell}_2} \Rightarrow S_{\hat{\ell}_1} \quad \vdash d : S_{\hat{\ell}_1} \Rightarrow T_{\hat{\ell}_2} \quad V \langle c \rangle \longrightarrow^* \text{blame } p}{\text{assign } (\text{addr } n \langle \mathbf{Ref} \ c \ d, \bar{c} \rangle) V T \hat{\ell}_2 \ell \mid \mu \mid PC \longrightarrow \text{blame } p \mid \mu} \\
\text{assign?-cast} \frac{\mathbf{NF} \bar{c} \quad \vdash c : T_g \Rightarrow S_{\hat{\ell}} \quad \vdash d : S_{\hat{\ell}} \Rightarrow T_g \quad (stamp! PC \mid \bar{c} \mid) \langle \star \Rightarrow^p \hat{\ell} \rangle \longrightarrow^* PC' \quad V \langle c \rangle \longrightarrow^* W}{\text{assign?}^p (\text{addr } n \langle \mathbf{Ref} \ c \ d, \bar{c} \rangle) V T g \mid \mu \mid PC \longrightarrow \$ \text{unit} \mid [\hat{\ell} \mapsto n \mapsto W] \mu} \\
\text{assign?-cast-blame-pc} \frac{\mathbf{NF} \bar{c} \quad \vdash c : T_g \Rightarrow S_{\hat{\ell}} \quad \vdash d : S_{\hat{\ell}} \Rightarrow T_g \quad (stamp! PC \mid \bar{c} \mid) \langle \star \Rightarrow^p \hat{\ell} \rangle \longrightarrow^* \text{blame } q}{\text{assign?}^p (\text{addr } n \langle \mathbf{Ref} \ c \ d, \bar{c} \rangle) V T g \mid \mu \mid PC \longrightarrow \text{blame } q \mid \mu} \\
\text{assign?-cast-blame} \frac{\mathbf{NF} \bar{c} \quad \vdash c : T_g \Rightarrow S_{\hat{\ell}} \quad \vdash d : S_{\hat{\ell}} \Rightarrow T_g \quad (stamp! PC \mid \bar{c} \mid) \langle \star \Rightarrow^p \hat{\ell} \rangle \longrightarrow^* PC' \quad V \langle c \rangle \longrightarrow^* \text{blame } q}{\text{assign?}^p (\text{addr } n \langle \mathbf{Ref} \ c \ d, \bar{c} \rangle) V T g \mid \mu \mid PC \longrightarrow \text{blame } q \mid \mu} \\
\text{deref} \frac{\mu(\hat{\ell}, n) = V}{!(\text{addr } n) T_{\hat{\ell}} \ell \mid \mu \mid PC \longrightarrow \text{prot } _ \ell V T_{\hat{\ell}} \mid \mu} \\
\text{deref-cast} \frac{\mathbf{NF} \bar{c} \quad \vdash c : A \Rightarrow T_{\hat{\ell}} \quad \vdash d : T_{\hat{\ell}} \Rightarrow A \quad \mu(\hat{\ell}, n) = V}{!(\text{addr } n \langle \mathbf{Ref} \ c \ d, \bar{c} \rangle) A \ell \mid \mu \mid PC \longrightarrow \text{prot } _ \ell (V \langle d \rangle) A \mid \mu} \\
\text{deref*-cast} \frac{\mathbf{NF} \bar{c} \quad \vdash c : S_{\star} \Rightarrow T_{\hat{\ell}} \quad \vdash d : T_{\hat{\ell}} \Rightarrow S_{\star} \quad \mu(\hat{\ell}, n) = V}{! \star (\text{addr } n \langle \mathbf{Ref} \ c \ d, \bar{c} \rangle) S \mid \mu \mid PC \longrightarrow \text{prot } _ \mid \bar{c} \mid (V \langle d \rangle) (S_{\star}) \mid \mu}
\end{array}$$

Figure 3.10: Operational semantics of λ_{IFC}^c (Part II). NSU checks that are represented using label expressions are highlighted

$$\boxed{\text{coerce-id } T = c_r \text{ and } \text{coerce-id } A = \mathbf{c}}$$

$$\begin{aligned} \text{coerce-id } \iota &= \mathbf{id}(\iota) \\ \text{coerce-id } (\mathbf{Ref } A) &= \mathbf{Ref} (\text{coerce-id } A) (\text{coerce-id } A) \\ \text{coerce-id } (A \xrightarrow{g} B) &= (\mathbf{id}(g), \text{coerce-id } A \rightarrow \text{coerce-id } B) \\ \text{coerce-id } T_g &= (\text{coerce-id } T, \mathbf{id}(g)) \end{aligned}$$

$$\boxed{\text{stamp } V A \ell = W}$$

$$\begin{aligned} \text{stamp } V - \mathbf{low} &= V \\ \text{stamp } V T_{\mathbf{low}} \mathbf{high} &= V \langle (\text{coerce-id } T), \mathbf{id}(\mathbf{low}); \uparrow \rangle \\ \text{stamp } V T_{\mathbf{high}} \mathbf{high} &= V \\ \text{stamp } (V \langle c_r, \bar{c} \rangle) - \ell &= V \langle c_r, \text{stamp } \bar{c} \ell \rangle \end{aligned}$$

Figure 3.11: Stamping on values of λ_{IFC}^c

operation for coercions on labels (Figure 3.2) in a similar way. The value is either (1) a raw value or (2) a coercion-wrapped value. Suppose the value is a raw value, if its type has \mathbf{low} security and is stamped with \mathbf{high} , the value becomes wrapped with a subtype coercion (the second case of *stamp* in Figure 3.11); otherwise, the value stays unchanged (the first and the third cases of *stamp*). If the value is wrapped with an irreducible coercion, we stamp the top-level coercion sequence (the fourth case of *stamp*).

Function Application. The β rule is standard for IFC languages. It generates a prot term with the specific security label ℓ that comes from the label on the λ , preventing implicit flow from the function being applied through both the computation result and the heap. The *app-cast* rule applies a function wrapped in a function coercion to a value V . The application is “static”, so the security level of prot comes from the function type just like β . The function coercion is distributed into its domain coercion \mathbf{c} , its co-domain coercion \mathbf{d} , and the coercion on PC \bar{d} . The coercion \bar{c} is not used because the function type is fully static, so its security is already indicated by its type. The domain coercion \mathbf{c} casts the input of the function V to W and W is substituted into the body of the λ . The substituted body

goes through the co-domain cast d , and is then protected by ℓ using prot . The stamped PC casts to PC' by \bar{d} and PC' is used as the PC for prot . The rule $\text{app}\star\text{-cast}$ is similar to $\text{app}\text{-cast}$ except for two things: (1) the PC is stamped and then injected using stamp! to preserve types (2) the security level of the function proxy used in protection is indicated by $|\bar{c}|$ instead, because the top-level security label of the function is statically unknown (\star).

If-conditional. The static rule $\text{if}\text{-true}$ is standard; the if term reduces a prot whose security ℓ comes from the type of the branch condition, guarding against implicit flow. The rationale of $\text{if}\star\text{-true}\text{-cast}$ follows that of $\text{app}\star\text{-cast}$: (1) a stamp! is generated to stamp and then inject the PC and (2) the security of prot is retrieved from the coercion in the branch condition.

NSU and heap operations. Let us consider reference creation first. A “static” reference creation is secure because its typing (rule $\vdash\text{-ref}$, Figure 3.8) already enforces the heap policy. Consequently, the allocation can happen directly (rule ref) without any runtime checking. Rule ref? does the same reference creation but with NSU checking, by casting the current PC to the security ℓ of the newly created memory cell. The coerce function $(- \Rightarrow^- -)$ takes two security labels and a blame label to generate a coercion on labels. In this case, $\star \Rightarrow^p \ell$ generates $\text{id}(\star); \ell?^p$, which performs a projection whose target is ℓ . If the projected PC reduces to a blame, it means that NSU checking fails so we lift the blame to λ_{IFC}^c . Assignment follows the same pattern: a static assignment can happen directly, while assign? requires NSU checking, by stamping the current PC with the security indicated in the coercion and then projecting to $\hat{\ell}$, where $\hat{\ell}$ is the security of the heap cell. The input coercion c is applied before the value is stored into the cell.

The rule for static dereferencing deref looks up index n in all memory cells with security level $\hat{\ell}$. The value from the lookup is protected with ℓ , the top-level security label of the reference type. The PC of the prot does not matter, because V is already a value and will not reduce. The rule $\text{deref}\star\text{-cast}$ dereferences a reference proxy. It looks up the

$$\boxed{M \mid \mu_1 \mid PC \longrightarrow^* N \mid \mu_2}$$

$$\frac{}{M \mid \mu \mid PC \longrightarrow^* M \mid \mu} \quad \frac{L \mid \mu_1 \mid PC \longrightarrow M \mid \mu_2 \quad M \mid \mu_2 \mid PC \longrightarrow^* N \mid \mu_3}{L \mid \mu_1 \mid PC \longrightarrow^* N \mid \mu_3}$$

Figure 3.12: Multi-step reduction of λ_{IFC}^c

value V in the heap, applies the output coercion \mathbf{d} , and generates a prot with the security of the coercion $|\bar{c}|$. The PC of prot does not matter in this case either, because applying a coercion is pure and does not produce side effects.

The multi-step reduction of λ_{IFC}^c is defined as the reflexive transitive closure of single-step reduction (Figure 3.12).

Finally, we define the evaluation of λ_{IFC}^c using the reduction relation:

$$\begin{aligned}
M \mid \mu_1 \Downarrow V \mid \mu_2 &\triangleq M \mid \mu_1 \mid \mathbf{low} \longrightarrow^* V \mid \mu_2 \\
M \mid \mu_1 \Downarrow \mathbf{blame } p \mid \mu_2 &\triangleq M \mid \mu_1 \mid \mathbf{low} \longrightarrow^* \mathbf{blame } p \mid \mu_2 \\
M \Uparrow \mid \mu_1 &\triangleq \forall L \mu_2. M \mid \mu_1 \mid \mathbf{low} \longrightarrow^* L \mid \mu_2 \text{ and } \exists N \mu_3. L \mid \mu \mid \mathbf{low} \longrightarrow N \mid \mu_3
\end{aligned}$$

The λ_{IFC}^c term M evaluates to a value V if it reduces to V in zero or more steps from the initial heap μ_1 and \mathbf{low} PC. It is the same for blame. Term M diverges if for all L that M reduces to, L can always take an additional step.

CHAPTER 4

COMPILING FROM λ_{IFC}^* TO λ_{IFC}^c

In this section, I define the coerce functions that produce coercions between security labels and types (Section 4.1). After that, I define the compilation function from λ_{IFC}^* to λ_{IFC}^c (Section 4.2), so that the semantics of λ_{IFC}^* can be given by λ_{IFC}^c .

4.1 Coerce Functions

The coerce function between security labels is defined in Figure 4.1. Function $g_1 \Rightarrow^p g_2 = \bar{c}$ takes two security labels that satisfies $g_1 \lesssim g_2$ and a blame label, and generates a coercion sequence. The source of the coercion sequence is g_1 , and the target is g_2 . The blame label goes to projections inside the generated coercion sequence.

The coerce functions between security types is also defined in Figure 4.1. Function $S \Rightarrow^p T = c_r$ takes two raw types satisfying $S \lesssim T$ and generates a raw coercion from S to T . Function $A \Rightarrow^p B = c$ takes two types satisfying $A \lesssim B$ and generates a coercion on values from A to B . The coerce functions of types call the coerce function of security labels on each pair of labels inside the two types. For example

$$\text{Bool}_{\text{low}} \Rightarrow^p \text{Bool}_* = \text{id}(\text{Bool}), \text{id}(\text{low}); \text{low}!$$

and

$$\begin{aligned} & (\text{Bool}_* \xrightarrow{\text{low}} \text{Bool}_*)_{\text{low}} \Rightarrow^p (\text{Bool}_{\text{low}} \xrightarrow{\text{low}} \text{Bool}_{\text{high}})_{\text{high}} = \\ & \text{id}(\text{low}), (\text{id}(\text{Bool}), \text{id}(\text{low}); \text{low}!) \rightarrow (\text{id}(\text{Bool}), \text{id}(*); \text{high}^{?^p}), \text{id}(\text{low}); \uparrow \end{aligned}$$

$$\begin{array}{l}
\star \Rightarrow^p \star = \mathbf{id}(\star) \quad \boxed{g \Rightarrow^p g = \bar{c}} \\
\ell \Rightarrow^p \star = \mathbf{id}(\ell); \ell! \\
\star \Rightarrow^p \ell = \mathbf{id}(\star); \ell ?^p \\
\mathbf{low} \Rightarrow^p \mathbf{low} = \mathbf{id}(\mathbf{low}) \\
\mathbf{low} \Rightarrow^p \mathbf{high} = \mathbf{id}(\mathbf{low}); \uparrow \\
\mathbf{high} \Rightarrow^p \mathbf{high} = \mathbf{id}(\mathbf{high}) \\
\iota \Rightarrow^p \iota = \mathbf{id}(\iota) \quad \boxed{S \Rightarrow^p T = c_r} \\
(\mathbf{Ref} A) \Rightarrow^p (\mathbf{Ref} B) = \mathbf{Ref} (B \Rightarrow^p A) (A \Rightarrow^p B) \\
(A \xrightarrow{g_1} B) \Rightarrow^p (C \xrightarrow{g_2} D) = (g_2 \Rightarrow^p g_1, (C \Rightarrow^p A) \rightarrow (B \Rightarrow^p D)) \\
S_{g_1} \Rightarrow^p T_{g_2} = (S \Rightarrow^p T, g_1 \Rightarrow^p g_2) \quad \boxed{A \Rightarrow^p B = c}
\end{array}$$

Figure 4.1: Coerce functions of security labels and types

4.2 Compilation

The compile function takes the form $\mathcal{C} M = M'$, where the input M is a λ_{IFC}^* program and the output M' is a λ_{IFC}^c term. We discuss six interesting cases: function application, if-conditional, annotation, reference creation, dereferencing, and reference assignment.

Function Application. If the top-level labels of the function type, g and gc , as well as the current PC (g') are all specific, the constraint on PC during function application can be statically justified. As a result, we recursively compile L and M and generate app , the static variant of function application. We cast compiled L using c_1 , which adjusts the PC part of the function type. We then cast compiled M using c_2 , which goes from A' (the type of M) to A (the domain type of the function). Otherwise, if at least one of the three security labels is \star , we cannot statically justify the constraint on PC. Consequently, we recursively compile L and M and generate $\text{app}\star$, the dynamic variant of function application. We insert three casts: the first cast (c_1) converts both top-level labels of the function type and the top-level label of the co-domain type to \star . The second cast (c_2) goes from A' (the type of M) to the domain type A . The third cast (d) casts the result of the function application,

$$\boxed{\mathcal{C} M = N}$$

$$\mathcal{C} (\$ k)_\ell = \$ k$$

$$\mathcal{C} x = x$$

$$\mathcal{C} (\lambda^g x : A. N)_\ell = \lambda x. \mathcal{C} N$$

Let L, M be well-typed $\Gamma; g' \vdash L : (A \xrightarrow{gc} B)_g, \Gamma; g' \vdash M : A'$

$$\mathcal{C} (LM)^P = \begin{cases} \text{app} ((\mathcal{C} L) \langle \mathbf{c}_1 \rangle) ((\mathcal{C} M) \langle \mathbf{c}_2 \rangle) A B g & \text{if } g, g' \text{ and } gc \text{ are all specific} \\ \text{where } \mathbf{c}_1 = (A \xrightarrow{gc} B)_g \Rightarrow^P (A \xrightarrow{g' \vee g} B)_g, \mathbf{c}_2 = A' \Rightarrow^P A & \\ \text{app}^* ((\mathcal{C} L) \langle \mathbf{c}_1 \rangle) ((\mathcal{C} M) \langle \mathbf{c}_2 \rangle) AT \langle \mathbf{d} \rangle & \text{otherwise} \\ \text{where } B = T_{g''}, \mathbf{c}_1 = (A \xrightarrow{gc} T_{g''})_g \Rightarrow^P (A \xrightarrow{*} T_{*})_* & \\ \mathbf{c}_2 = A' \Rightarrow^P A, \mathbf{d} = T_{*} \Rightarrow^P (\text{stamp } T_{g''} g) & \end{cases}$$

Let L, M, N be well-typed $\Gamma; g' \vdash L : \text{Bool}_g, \Gamma; g' \tilde{\vee} g \vdash M : A, \Gamma; g' \tilde{\vee} g \vdash N : B$

$$\mathcal{C} (\text{if } L \text{ then } M \text{ else } N)^P = \begin{cases} \text{if} ((\mathcal{C} L) (A \tilde{\vee} B) g) ((\mathcal{C} M) \langle \mathbf{c}_1 \rangle) ((\mathcal{C} N) \langle \mathbf{c}_2 \rangle) & \text{if } g, g' \text{ are both specific} \\ \text{if}^* ((\mathcal{C} L) \langle \mathbf{d}_1 \rangle) T ((\mathcal{C} M) \langle \mathbf{c}_1 \rangle \langle \mathbf{d}_2 \rangle) ((\mathcal{C} N) \langle \mathbf{c}_2 \rangle \langle \mathbf{d}_3 \rangle) \langle \mathbf{d}_4 \rangle & \\ \text{where } A \tilde{\vee} B = T_{g''}, \mathbf{d}_1 = \text{Bool}_g \Rightarrow^P \text{Bool}_*, \mathbf{d}_2 = T_{g''} \Rightarrow^P T_* & \\ \mathbf{d}_3 = T_{g''} \Rightarrow^P T_*, \mathbf{d}_4 = (\text{stamp } (A \tilde{\vee} B) *) \Rightarrow^P (\text{stamp } (A \tilde{\vee} B) g) & \\ \text{where } \mathbf{c}_1 = A \Rightarrow^P A \tilde{\vee} B, \mathbf{c}_2 = B \Rightarrow^P A \tilde{\vee} B & \end{cases}$$

Let M be well-typed $\Gamma; g' \vdash M : A'$

$$\mathcal{C} (M : A)^P = (\mathcal{C} M) \langle A' \Rightarrow^P A \rangle$$

Let M, N be well-typed $\Gamma; g' \vdash M : A, (\Gamma, x:A); g' \vdash N : B$

$$\mathcal{C} (\text{let } x = M \text{ in } N) = \text{let } x = (\mathcal{C} M) : A \text{ in } (\mathcal{C} N)$$

Let M be well-typed $\Gamma; g' \vdash M : T_g$

$$\mathcal{C} (\text{ref } \ell M)^P = \begin{cases} \text{ref } \ell ((\mathcal{C} M) \langle T_g \Rightarrow^P T_\ell \rangle) & \text{if } g' \text{ is specific} \\ \text{ref}^? \ell ((\mathcal{C} M) \langle T_g \Rightarrow^P T_\ell \rangle) & \text{otherwise} \end{cases}$$

Let M be well-typed $\Gamma; g' \vdash M : (\text{Ref } A)_g$

$$\mathcal{C} (!^P M) = \begin{cases} ! (\mathcal{C} M) A g & \text{if } g \text{ is specific} \\ !^* ((\mathcal{C} M) \langle \mathbf{c} \rangle) T & \text{otherwise} \\ \text{where } A = T_{g''}, \mathbf{c} = (\text{Ref } T_{g''})_g \Rightarrow^P (\text{Ref } T_{*})_* & \end{cases}$$

Let L, M be well-typed $\Gamma; g' \vdash L : (\text{Ref } T_{\hat{g}})_g, \Gamma; g' \vdash M : A$

$$\mathcal{C} (L := M)^P = \begin{cases} \text{assign} (\mathcal{C} L) ((\mathcal{C} M) \langle \mathbf{c}_2 \rangle) T \hat{g} g & \text{if } g, g' \text{ and } \hat{g} \text{ are all specific} \\ \text{assign}^? ((\mathcal{C} L) \langle \mathbf{c}_1 \rangle) ((\mathcal{C} M) \langle \mathbf{c}_2 \rangle) T \hat{g} & \text{otherwise} \\ \text{where } \mathbf{c}_1 = (\text{Ref } T_{\hat{g}})_g \Rightarrow^P (\text{Ref } T_{\hat{g}})_*, \mathbf{c}_2 = A \Rightarrow^P T_{\hat{g}} & \end{cases}$$

Figure 4.2: Compilation from λ_{IFC}^* to λ_{IFC}^c

from T_\star (where T is the raw type part of the co-domain type) to the result of stamping the label of the function type onto the co-domain.

If-conditional. If the label of the branch condition (g) and the current PC (g') are both specific, we recursively compile the branch condition as well as the two branches and generate a static `if`. We insert one cast into each of the compiled branches, from the type of that branch to the consistent join of the types of the two branches ($A \tilde{\vee} B$). If at least one label is \star , we recursively compile the sub-terms and generate `if \star` . In this case, we need to insert four additional casts: the first cast d_1 converts the branch condition from Bool_g to Bool_\star . The second (d_2) and the third cast (d_3) convert their respective branches into T_\star (where T is the raw type part of $A \tilde{\vee} B$). The fourth cast (d_4) casts the type of the `if-conditional` to the result of stamping g onto $A \tilde{\vee} B$ to preserve types.

Annotation. We recursively compile the annotated term and insert one cast, which goes from the type of the term to the type annotation.

Reference Creation. Recall that the security level of the newly-created memory location is given in the syntax of `ref` (always specific). If the current PC (g') is specific, we can statically justify the heap-policy check. Consequently, we recursively compile the sub-term (M) and generate a static `ref`. We only need to insert one cast, which goes from M 's type (T_g) to the type of the memory location (T_ℓ). Otherwise, if the current PC is \star , we generate the dynamic version of reference creation (`ref?`), thus incurring an NSU check, which enforces the heap policy at runtime.

Dereferencing. We case on g , which is the label on the type of the sub-term (M). If g is specific, we recursively compile M and generate the static version of dereference. On the other hand, if g is unknown (\star), we recursively compile M and generate the dynamic version of dereference (`! \star`). In addition, we cast the labels on both the reference type and

the referenced type to \star .

Reference Assignment. If g , g' , and \hat{g} are specific, the check for heap policy can be statically justified. We recursively compile both L and M and generate a static `assign`. We cast M' using coercion c_2 , which casts from the type of M' to the type of the memory cell. If at least one of the three security labels is \star , the typing information is insufficient to justify the assignment. The compilation produces an `assign?` instead, whose semantics performs NSU checking at runtime.

CHAPTER 5

TYPE SAFETY OF λ_{IFC}^*

In this chapter, I prove type safety for λ_{IFC}^* (Theorem 9). Type safety says that untrapped errors (or “undefined behaviors”) never occur in λ_{IFC}^* . In other words, the evaluation of a λ_{IFC}^* program will never reach the `stuck` case of `eval`. In Section 5.1, I prove that the cast calculus λ_{IFC}^c satisfies progress and preservation. In Section 5.2, I prove that the compilation from λ_{IFC}^* to λ_{IFC}^c preserves types. Finally, in Section 5.3, I prove the type safety theorem for λ_{IFC}^* .

The Agda proofs cited in the section are available at:

<https://github.com/Gradual-Typing/LambdaIFCStar>

5.1 Type Safety of λ_{IFC}^c by Progress and Preservation

We first prove that substitution preserves types:

Lemma 3 (Substitution preserves types). *If $(\Gamma, x:A); \Sigma; g; \ell \vdash N \Leftarrow B$ and $\forall g' \ell'. \Gamma; \Sigma; g'; \ell' \vdash V \Leftarrow A$, then $\Gamma; \Sigma; g \ell \vdash N[x := V] \Leftarrow B$.*

Proof. The proof is fully mechanized in `/src/CC2/SubstPreserve.agda`. □

Heap well-typedness of λ_{IFC}^c is defined point-wise (in `/src/Memory/HeapTyping.agda`):

$$\Sigma \vdash \mu \triangleq \forall n \ell T. \text{if } \Sigma(\ell, n) = T, \text{ then } \mu(\ell, n) = V$$

and $\emptyset; \Sigma; \text{low}; \text{low} \vdash V \Leftarrow T_\ell$ for some V

We show that λ_{IFC}^c is type safe by proving progress and preservation. Progress says that a well-typed λ_{IFC}^c term does not get stuck. The term is either a value or a blame, which does

not reduce, or the term takes one reduction step.

Lemma 4 (Progress of λ_{IFC}^c). *Suppose PC is well-typed: $\vdash PC \Leftarrow g$, M is well-typed:*

$$\emptyset; \Sigma; g; |PC| \vdash M \Leftarrow A$$

and the heap μ is also well-typed: $\Sigma \vdash \mu$. Then either (1) M is a value or (2) M is a blame: $M = \text{blame } p$ or (3) M can take a reduction step: $M \mid \mu \mid PC \longrightarrow N \mid \mu' \mid PC$ for some N and μ' .

Proof. The proof is fully mechanized in progress of `/src/CC2/Progress.agda`. \square

The operation semantics of λ_{IFC}^c preserves types and the well-typedness of heap:

Lemma 5 (Preservation of λ_{IFC}^c). *Suppose PC is well-typed: $\vdash PC \Leftarrow g$, M is well-typed:*

$\emptyset; \Sigma; g; |PC| \vdash M \Leftarrow A$ and the heap μ is also well-typed: $\Sigma \vdash \mu$. If $M \mid \mu \mid PC \longrightarrow N \mid \mu' \mid PC$, there exists Σ' s.t $\Sigma' \supseteq \Sigma$, $\emptyset; \Sigma'; g; |PC| \vdash N \Leftarrow A$, and $\Sigma' \vdash \mu'$.

Proof. The proof is fully mechanized in pres of `/src/CC2/Preservation.agda`. \square

5.2 Compilation From λ_{IFC}^* to λ_{IFC}^c Preserves Types

We prove that the compilation from λ_{IFC}^* to λ_{IFC}^c (defined in Chapter 4) preserves types:

Lemma 6 (Compilation from λ_{IFC}^* to λ_{IFC}^c preserves types). *If M is a well-typed λ_{IFC}^* term:*

$\Gamma; g \vdash M : A$, then the compiled λ_{IFC}^c term is also well-typed: $\Gamma; \emptyset; g; \text{low} \vdash \mathcal{C} M \Leftarrow A$.

Proof. The proof is fully mechanized in `/src/Compile/CompilationPresTypes.agda`. \square

5.3 Type Safety of λ_{IFC}^*

Leading to the type safety for λ_{IFC}^* , we first prove that multi-step reduction of λ_{IFC}^c preserves types, which is a direct corollary of Lemma 5:

$\boxed{\vdash r : A}$

$$\begin{array}{c}
\text{WT-const} \frac{k : \iota}{\vdash k : \iota_\ell} \qquad \text{WT-addr} \frac{}{\vdash \text{addr} : (\text{Ref } A)_g} \\
\text{WT-fun} \frac{}{\vdash \text{fun} : (A \xrightarrow{g_2} B)_{g_1}} \qquad \text{WT-diverge} \frac{}{\vdash \text{diverge} : A}
\end{array}$$

Figure 5.1: Well-typed evaluation result

Lemma 7. *Suppose $\emptyset; \Sigma; \text{low}; \text{low} \vdash M \Leftarrow A$ and $\Sigma \vdash \mu$. If $M \mid \mu \mid \text{low} \longrightarrow^* M' \mid \mu'$, then there exists Σ' s.t $\Sigma' \supseteq \Sigma$, $\emptyset; \Sigma'; \text{low}; \text{low} \vdash M' \Leftarrow A$, and $\Sigma' \vdash \mu'$.*

Proof. Mechanized in `pres-mult` of `/src/CC2/MultiStep.agda`. By induction on the reduction sequence. \square

We then prove the following lemma about the evaluation of λ_{IFC}^c :

Lemma 8. *If $\emptyset; \Sigma; \text{low}; \text{low} \vdash M \Leftarrow A$ and $\Sigma \vdash \mu$, then either (1) $M \mid \mu \Downarrow V \mid \mu'$ and $\emptyset; \Sigma'; \text{low}; \text{low} \vdash V \Leftarrow A$ for some Σ', μ' , or (2) $M \mid \mu \Downarrow \text{blame } p \mid \mu'$ for some p, μ' , or (3) $M \mid \mu \Uparrow$.*

Proof. By law of excluded middle, there are four cases: (1) M reduces to a value, (2) M reduces to a blame, (3) M diverges, or (4) M gets stuck.

- **M reduces to some value V :** $M \mid \mu \Downarrow V \mid \mu'$. By definition of $- \mid - \Downarrow - \mid -$, $M \mid \mu \mid \text{low} \longrightarrow^* V \mid \mu'$. By Lemma 7 (multi-step reduction of λ_{IFC}^c preserves types), we have $\emptyset; \Sigma'; \text{low}; \text{low} \vdash V \Leftarrow A$ for some Σ' .
- **M reduces to a blame:** $M \mid \mu \Downarrow \text{blame } p \mid \mu'$. The lemma is trivially true.
- **M diverges:** $M \mid \mu \Uparrow$. The lemma is also trivially true.
- **Otherwise, M gets stuck:** $M \mid \mu \mid \text{low} \longrightarrow^* L \mid \mu'$, L is neither a value nor a blame, and there is no N such that $L \mid \mu' \mid \text{low} \longrightarrow N \mid \mu''$ for some μ'' , but that contradicts Lemma 4 (progress of λ_{IFC}^c).

□

Finally, we prove type safety for λ_{IFC}^* . Recall that a λ_{IFC}^* program may evaluate to a constant, an address, or a function, or the program may diverge, or the program may get stuck (Figure 2.7). The well-typedness of evaluation results is defined in Figure 5.1. The evaluation result being well-typed means that a λ_{IFC}^* program never gets stuck:

Theorem 9 (Type safety of λ_{IFC}^*). *If M is a λ_{IFC}^* program and $\text{eval}(M, b) = r$, then the evaluation result is well-typed: $\vdash r : \text{Bool}_{\text{low}}$.*

Proof. By Definition 1 (λ_{IFC}^* programs), $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M : \text{Bool}_{\text{low}}$. By Lemma 6 (compilation preserves types) and Lemma 3 (substitution preserves types), $\emptyset; \emptyset; \text{low}; \text{low} \vdash (\mathcal{C} M)[x := \$ b] \Leftarrow \text{Bool}_{\text{low}}$. Empty heap is trivially well-typed: $\emptyset \vdash \emptyset$. Applying Lemma 8 yields three cases:

- $(\mathcal{C} M)[x := b] \mid \emptyset \Downarrow V \mid \mu$ and $\emptyset; \Sigma'; \text{low}; \text{low} \vdash V \Leftarrow \text{Bool}_{\text{low}}$ for some Σ' .
By the canonical form of a value of Bool_{low} , $V = \$ b'$ for some b' . As a result, $r = \text{eval}(M, b) = \text{obs}(\$ b') = b'$. By rule *WT-const* of Figure 5.1, $\vdash b' : \text{Bool}_{\text{low}}$, so $\vdash r : \text{Bool}_{\text{low}}$.
- $(\mathcal{C} M)[x := b] \mid \mu \Downarrow \text{blame } p \mid \mu'$. By definition, $\text{eval}(M, b) = \text{diverge}$. By rule *WT-diverge*, $\vdash r : \text{Bool}_{\text{low}}$.
- $(\mathcal{C} M)[x := b] \mid \mu \Uparrow$. By definition, $\text{eval}(M, b) = \text{diverge}$. By rule *WT-diverge*, $\vdash r : \text{Bool}_{\text{low}}$.

□

CHAPTER 6

GRADUAL GUARANTEE OF λ_{IFC}^*

In this chapter, I prove that λ_{IFC}^* satisfies the gradual guarantee (Theorem 21). The gradual guarantee says that if a λ_{IFC}^* program successfully evaluates to some value, then a variant of the program where the type annotations are less precise must also evaluate to the same value. In Section 6.1, I prove a simulation lemma between more and less precise coercion sequences: if the more precise side takes one step, the less precise side can take zero or more steps and be related by precision again. After that, in Section 6.2, I prove a similar simulation lemma for the cast calculus λ_{IFC}^c . Finally, in Section 6.3, I prove the gradual guarantee for λ_{IFC}^* as a corollary of the simulation lemma for λ_{IFC}^c .

The Agda proofs cited in the section are available at:

<https://github.com/Gradual-Typing/LambdaIFCStar>

6.1 Simulation Between More and Less Precise Coercion Sequences

Our end goal is to prove the gradual guarantee for λ_{IFC}^* . The proof depends on a simulation lemma between more and less precise terms of the cast calculus λ_{IFC}^c . We use coercion sequences as the IFC monitor in λ_{IFC}^c . Reducing a coercion sequence can result in a blame which errors the program. So we would like to prove that the simulation lemma holds for the coercion calculus on security labels. The precision relation on security coercions is defined in Figure 6.1. The precision relation between two coercion sequences \bar{c}, \bar{d} takes the form $\vdash \bar{c} \sqsubseteq \bar{d}$. Recall that the gradual guarantee states that replacing type annotations with \star (decreasing type precision) should result in the same value for a correctly running program while adding annotations (increasing type precision) may trigger more runtime errors. The precision relation is a syntactical characterization of the runtime behaviors of

$\boxed{\vdash c \sqsubseteq d, \vdash c \sqsubseteq g, \text{ and } \vdash g \sqsubseteq d}$

$$\sqsubseteq\text{-}c \frac{g_1 \sqsubseteq g'_1 \quad g_2 \sqsubseteq g'_2 \quad \vdash c : g_1 \Rightarrow g_2 \quad \vdash d : g'_1 \Rightarrow g'_2}{\vdash c \sqsubseteq d}$$

$$\sqsubseteq\text{-}cl \frac{g_1 \sqsubseteq g \quad g_2 \sqsubseteq g \quad \vdash c : g_1 \Rightarrow g_2}{\vdash c \sqsubseteq g} \quad \sqsubseteq\text{-}cr \frac{g \sqsubseteq g_1 \quad g \sqsubseteq g_2 \quad \vdash d : g_1 \Rightarrow g_2}{\vdash g \sqsubseteq d}$$

$\boxed{\vdash \bar{c} \sqsubseteq \bar{d}}$

$$\begin{aligned} \sqsubseteq\text{-}id & \frac{g \sqsubseteq g'}{\vdash \mathbf{id}(g) \sqsubseteq \mathbf{id}(g')} & \sqsubseteq\text{-}cast & \frac{\vdash \bar{c} \sqsubseteq \bar{d} \quad \vdash c \sqsubseteq d}{\vdash \bar{c}; c \sqsubseteq \bar{d}; d} \\ \sqsubseteq\text{-}castl & \frac{\vdash \bar{c} \sqsubseteq \bar{d} \quad \vdash c \sqsubseteq g_2 \quad \vdash \bar{d} : g_1 \Rightarrow g_2}{\vdash \bar{c}; c \sqsubseteq \bar{d}} & \sqsubseteq\text{-}castr & \frac{\vdash \bar{c} \sqsubseteq \bar{d} \quad \vdash g_2 \sqsubseteq d \quad \vdash \bar{c} : g_1 \Rightarrow g_2}{\vdash \bar{c} \sqsubseteq \bar{d}; d} \\ \sqsubseteq\text{-}\perp & \frac{g_1 \sqsubseteq g_3 \quad g_2 \sqsubseteq g_4 \quad \vdash \bar{c} : g_1 \Rightarrow g_2}{\vdash \bar{c} \sqsubseteq \perp^P g_3 g_4} \end{aligned}$$

$\boxed{\vdash \bar{c} \sqsubseteq_l g}$

$$\sqsubseteq_l\text{-}id \frac{g \sqsubseteq g'}{\vdash \mathbf{id}(g) \sqsubseteq_l g'} \quad \sqsubseteq_l\text{-}cast \frac{\vdash \bar{c} \sqsubseteq_l g \quad \vdash c \sqsubseteq_l g}{\vdash \bar{c}; c \sqsubseteq_l g}$$

$\boxed{\vdash g \sqsubseteq_r \bar{d}}$

$$\begin{aligned} \sqsubseteq_r\text{-}id & \frac{g \sqsubseteq g'}{\vdash g \sqsubseteq_r \mathbf{id}(g')} & \sqsubseteq_r\text{-}cast & \frac{\vdash g \sqsubseteq_r \bar{d} \quad \vdash g \sqsubseteq_r d}{\vdash g \sqsubseteq_r \bar{d}; d} \\ \sqsubseteq_r\text{-}\perp & \frac{g \sqsubseteq g_1 \quad g \sqsubseteq g_2}{\vdash g \sqsubseteq_r \perp^P g_1 g_2} \end{aligned}$$

Figure 6.1: Precision relation of coercions on security labels

programs of different type precision. We explain the intuition with two examples.

Example 10. Consider the following two programs that are related by precision because the first one has a \star annotation where the second one has $high$.

$$\text{true}_{low} : \text{Bool}_{\star} : \text{Bool}_{\star} \quad \text{and} \quad \text{true}_{low} : \text{Bool}_{high} : \text{Bool}_{\star}$$

At runtime, the less precise program on the left produces value $(\text{true} \langle \text{id}(low); low! \rangle)$ and the more precise program on the right produces $(\text{true} \langle \text{id}(low); \uparrow; high! \rangle)$. The trues are straightforwardly related; we need to show the two coercion sequences are also related:

$$\vdash \text{id}(low); low! \sqsubseteq \text{id}(low); \uparrow; high!$$

Starting at the beginning of the two sequences, we have $\text{id}(low) \sqsubseteq \text{id}(low)$ because \sqsubseteq is reflexive. Next we have $low! \sqsubseteq \uparrow$, which makes sense because the source and targets of the two coercions are related by precision: $low \sqsubseteq low$ and $\star \sqsubseteq high$. Finally, the coercion $high!$ can be added to the end of the more precise sequence because both its source and target type are more precise than the target of the left-hand sequence. That is, $\star \sqsubseteq high$ and $\star \sqsubseteq \star$. The injections at the ends of the two sequences, $low!$ and $high!$, cannot be directly related via precision because $low \not\sqsubseteq high$. Instead, $low!$ is related with \uparrow . This underlines the indispensability of explicit subtype coercion \uparrow for the purposes of proving the gradual guarantee.

Example 11. Next we consider an example where the less precise program produces a value but the more precise program encounters an error. This situation is allowed by the gradual guarantee, but the opposite one is not. We extend the example with a cast to low on the more precise side.

$$\text{true}_{low} : \text{Bool}_{\star} : \text{Bool}_{\star} : \text{Bool}_{\star} \quad \text{and} \quad \text{true}_{low} : \text{Bool}_{high} : \text{Bool}_{\star} : \text{Bool}_{low}$$

The first program again produces value $(\text{true} \langle \text{id}(\text{low}); \text{low}! \rangle)$. The second, on the other hand, reduces to $(\text{true} \langle \text{id}(\text{low}); \uparrow; \text{high}!; \text{low}^p \rangle) \longrightarrow^* (\text{true} \langle \perp^p \text{low low} \rangle)$, because of the contradicting annotations high and low (note that high is in the middle of the sequence and both the source and target labels of the blame coercion are low so that types are preserved). The failure is then propagated out and the term further reduces to $\text{blame } p$. The precision of coercion sequences relates \perp on the right-hand side to any coercion sequence on the left so long as the respective source and target types are related via precision, in this case $\text{low} \sqsubseteq \text{low}$ and $\star \sqsubseteq \text{low}$.

Consider the security levels (Definition 2) of both sides of Example 10, which are low on the less precise side and high on the more precise side. We observe that λ_{IFC}^* terms related by precision may produce values of different security: a less precise value may have lower security than a more precise value. Indeed, we prove the following for coercions in normal form:

Lemma 12 (Security is monotonic with respect to precision). *Suppose $\text{NF } \bar{c}$ and $\text{NF } \bar{d}$. If $\vdash \bar{c} \sqsubseteq \bar{d}$, then $|\bar{c}| \leq |\bar{d}|$.*

Proof. The proof is mechanized in `/src/CoercionExpr/SecurityLevel.agda`. \square

Next we prove a catch-up lemma for coercion sequences, where the less precise side catches up with a more precise sequence that is in normal form. The proof is by casing on $\text{NF } \bar{d}$ first and then performing induction on the precision relation in each case.

Lemma 13 (Catching up to a more precise coercion sequence). *If $\text{NF } \bar{d}$ and $\vdash \bar{c} \sqsubseteq \bar{d}$, there exists \bar{c}' such that $\bar{c} \longrightarrow^* \bar{c}'$, $\text{NF } \bar{c}'$, and $\vdash \bar{c}' \sqsubseteq \bar{d}$.*

Proof. The proof is mechanized in `/src/CoercionExpr/CatchUp.agda`. \square

Using Lemma 13, we then prove the following simulation lemma for coercion sequences:

Lemma 14 (Simulation between related coercion sequences). *If $\vdash \bar{c} \sqsubseteq \bar{d}$ and $\bar{d} \longrightarrow \bar{d}'$, there exists \bar{c}' such that $\bar{c} \longrightarrow^* \bar{c}'$ and $\vdash \bar{c}' \sqsubseteq \bar{d}'$.*

$\boxed{c \sqsubseteq d}$

$$\begin{array}{c} \sqsubseteq\text{-base} \frac{\vdash \bar{c} \sqsubseteq c'}{(\mathbf{id}(\iota), \bar{c}) \sqsubseteq (\mathbf{id}(\iota), \bar{c}')} \\ \sqsubseteq\text{-ref} \frac{c \sqsubseteq c' \quad d \sqsubseteq d' \quad \vdash \bar{c} \sqsubseteq c'}{(\mathbf{Ref} \ c \ d, \bar{c}) \sqsubseteq (\mathbf{Ref} \ c' \ d', \bar{c}')} \quad \sqsubseteq\text{-fun} \frac{\vdash \bar{d} \sqsubseteq \bar{d}' \quad \vdash \bar{c} \sqsubseteq c' \quad c \sqsubseteq c' \quad d \sqsubseteq d'}{(\bar{d}, c \rightarrow d, \bar{c}) \sqsubseteq (\bar{d}', c' \rightarrow d', \bar{c}')} \end{array}$$

 $\boxed{c \sqsubseteq A}$

$$\begin{array}{c} \sqsubseteq\text{-base} \frac{\vdash \bar{c} \sqsubseteq_l g}{(\mathbf{id}(\iota), \bar{c}) \sqsubseteq \iota_g} \\ \sqsubseteq\text{-ref} \frac{c \sqsubseteq A \quad d \sqsubseteq A \quad \vdash \bar{c} \sqsubseteq_l g}{(\mathbf{Ref} \ c \ d, \bar{c}) \sqsubseteq (\mathbf{Ref} \ A)_g} \quad \sqsubseteq\text{-fun} \frac{\vdash \bar{d} \sqsubseteq_l g_1 \quad \vdash \bar{c} \sqsubseteq_l g_2 \quad c \sqsubseteq A \quad d \sqsubseteq B}{(\bar{d}, c \rightarrow d, \bar{c}) \sqsubseteq (A \xrightarrow{g_1} B)_{g_2}} \end{array}$$

 $\boxed{A \sqsubseteq c}$

$$\begin{array}{c} \sqsubseteq\text{-base} \frac{\vdash g \sqsubseteq_r \bar{c}}{\iota_g \sqsubseteq (\mathbf{id}(\iota), \bar{c})} \\ \sqsubseteq\text{-ref} \frac{A \sqsubseteq c \quad A \sqsubseteq d \quad \vdash g \sqsubseteq_r \bar{c}}{(\mathbf{Ref} \ A)_g \sqsubseteq (\mathbf{Ref} \ c \ d, \bar{c})} \quad \sqsubseteq\text{-fun} \frac{\vdash g_1 \sqsubseteq_r \bar{d} \quad \vdash g_2 \sqsubseteq_r \bar{c} \quad A \sqsubseteq c \quad B \sqsubseteq d}{(A \xrightarrow{g_1} B)_{g_2} \sqsubseteq (\bar{d}, c \rightarrow d, \bar{c})} \end{array}$$

Figure 6.2: Precision relation of coercions on values

Proof. The proof is mechanized in `sim` of `/src/CoercionExpr/Simulation.agda`. \square

We also prove that stamping on coercion sequences preserves precision:

Lemma 15 (Stamping preserves precision of coercion sequences). *If $\vdash \bar{c} \sqsubseteq \bar{d}$, then $\vdash \text{stamp } \bar{c} \ell \sqsubseteq \text{stamp } \bar{d} \ell$ and $\vdash \text{stamp! } \bar{c} \ell_1 \sqsubseteq \text{stamp! } \bar{d} \ell_2$ and $\vdash \text{stamp! } \bar{c} \ell_1 \sqsubseteq \text{stamp } \bar{d} \ell_2$ if $\ell_1 \leq \ell_2$.*

Proof. The proof is mechanized in `/src/CoercionExpr/Stamping.agda`. \square

6.2 Simulation Between λ_{IFC}^c Terms of Different Precision

The main simulation lemma says that if two terms are related by *precision* and the more precise side takes one step, then the less precise side is able to multi-step and get back in

$$\boxed{\vdash e \sqsubseteq e' \Leftarrow g \sqsubseteq g'}$$

$$\begin{array}{c}
\sqsubseteq\text{-}l \frac{}{\vdash \ell \sqsubseteq \ell \Leftarrow \ell \sqsubseteq \ell} \quad \sqsubseteq\text{-}cast \frac{\vdash e \sqsubseteq e' \Leftarrow g_1 \sqsubseteq g'_1 \quad \vdash \bar{c} \sqsubseteq \bar{c}' \quad \vdash \bar{c} : g_1 \Rightarrow g_2 \quad \vdash \bar{c}' : g'_1 \Rightarrow g'_2}{\vdash e \langle \bar{c} \rangle \sqsubseteq e' \langle \bar{c}' \rangle \Leftarrow g_2 \sqsubseteq g'_2} \\
\sqsubseteq\text{-}castl \frac{\vdash e \sqsubseteq e' \Leftarrow g_1 \sqsubseteq g' \quad \vdash \bar{c} \sqsubseteq_l g' \quad \vdash \bar{c} : g_1 \Rightarrow g_2}{\vdash e \langle \bar{c} \rangle \sqsubseteq e' \Leftarrow g_2 \sqsubseteq g'} \quad \sqsubseteq\text{-}castr \frac{\vdash e \sqsubseteq e' \Leftarrow g \sqsubseteq g'_1 \quad \vdash g \sqsubseteq_r \bar{c}' \quad \vdash \bar{c}' : g'_1 \Rightarrow g'_2}{\vdash e \sqsubseteq e' \langle \bar{c}' \rangle \Leftarrow g \sqsubseteq g'_2} \\
\sqsubseteq\text{-}blame \frac{\vdash e \Leftarrow g \quad g \sqsubseteq g'}{\vdash e \sqsubseteq \text{blame } p \Leftarrow g \sqsubseteq g'}
\end{array}$$

Figure 6.3: Precision relation of label expressions

sync. The precision relation is in form $\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'$, where $\Gamma; \Sigma; g; \ell$ corresponds to the typing context, heap context, type of PC , and security of PC of the less precise term M and $\Gamma'; \Sigma'; g'; \ell'$ is for those of the more precise term M' . The types of the two terms, A and A' , are related by precision between types. The intuition between this precision relation is that casts are allowed to appear in different places between the more precise and the less precise λ_{IFC}^c terms. Moreover, the casts must be in shapes that preserve the precision of λ_{IFC}^* (more or fewer static type annotations provided by the programmer). According to the gradual guarantee, the more precise side is allowed to signal more blames, so there is a rule ($\sqsubseteq\text{-}blame$) that relates $\text{blame } p$ to any term M on the less precise side as long as their types are in sync. We list the precision rules for the cast calculus λ_{IFC}^c in Figure 6.4, 6.5, 6.6, and 6.7. The precision rules for λ_{IFC}^c use the precision relations for coercions on values and label expressions, which are defined in Figure 6.2 and Figure 6.3 respectively.

With the precision relation of λ_{IFC}^c defined, we first state the catch-up lemma, which catches up to a more-precise value by multi-stepping on the less-precise side:

Lemma 16 (Catching up to more precise). *If term M and value V' are related by precision:*

$$\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq V' \Leftarrow A \sqsubseteq A'$$

$$\boxed{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'}$$

$$\sqsubseteq\text{-var} \frac{\Gamma \ni x : A \quad \Gamma' \ni x : A'}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash x \sqsubseteq x \Leftarrow A \sqsubseteq A'}$$

$$\sqsubseteq\text{-const} \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash \$ k \sqsubseteq \$ k \Leftarrow \iota_\ell \sqsubseteq \iota_\ell}$$

$$\sqsubseteq\text{-addr} \frac{\Sigma(\hat{\ell}, n) = T \quad \Sigma'(\hat{\ell}, n) = T'}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash \text{addr } n \sqsubseteq \text{addr } n \Leftarrow (\text{Ref } T_{\hat{\ell}})_\ell \sqsubseteq (\text{Ref } T'_{\hat{\ell}})_\ell}$$

$$\sqsubseteq\text{-lam} \frac{\begin{array}{c} g_2 \sqsubseteq g'_2 \qquad A \sqsubseteq A' \\ \forall \ell \ell'. (\Gamma, x:A); (\Gamma', x:A'); \Sigma; \Sigma'; g_2; g'_2; \ell; \ell' \vdash N \sqsubseteq N' \Leftarrow B \sqsubseteq B' \end{array}}{\Gamma; \Gamma'; \Sigma; \Sigma'; g_1; g'_1; \ell_1; \ell'_1 \vdash \lambda x. N \sqsubseteq \lambda x. N' \Leftarrow (A \xrightarrow{g_2} B)_{\ell_2} \sqsubseteq (A' \xrightarrow{g'_2} B')_{\ell_2}}$$

$$\sqsubseteq\text{-app} \frac{\begin{array}{c} \Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell'_2 \vdash L \sqsubseteq L' \Leftarrow (A \xrightarrow{\ell_1 \vee \ell_3} B)_{\ell_3} \sqsubseteq (A' \xrightarrow{\ell_1 \vee \ell_3} B')_{\ell_3} \\ \Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell'_2 \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \\ C = \text{stamp } B \ell_3 \qquad C' = \text{stamp } B' \ell_3 \end{array}}{\Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell'_2 \vdash \text{app } L M A B \ell_3 \sqsubseteq \text{app } L' M' A' B' \ell_3 \Leftarrow C \sqsubseteq C'}$$

$$\sqsubseteq\text{-app}\star \frac{\begin{array}{c} \Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash L \sqsubseteq L' \Leftarrow (A \xrightarrow{\star} T_\star)_\star \sqsubseteq (A' \xrightarrow{\star} T'_\star)_\star \\ \Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \end{array}}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash \text{app}\star L M A T \sqsubseteq \text{app}\star L' M' A' T' \Leftarrow T_\star \sqsubseteq T'_\star}$$

$$\sqsubseteq\text{-app}\star l \frac{\begin{array}{c} \Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell'_2 \vdash L \sqsubseteq L' \Leftarrow (A \xrightarrow{\star} T_\star)_\star \sqsubseteq (A' \xrightarrow{\ell_1 \vee \ell_3} B')_{\ell_3} \\ \Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell'_2 \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \\ C' = \text{stamp } B' \ell_3 \end{array}}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell'_2 \vdash \text{app}\star L M A T \sqsubseteq \text{app } L' M' A' B' \ell_3 \Leftarrow T_\star \sqsubseteq C'}$$

$$\sqsubseteq\text{-let} \frac{\begin{array}{c} \Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell'_1 \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \\ \forall \ell_2 \ell'_2. (\Gamma, x:A); (\Gamma', x:A'); \Sigma; \Sigma'; g; g'; \ell_2; \ell'_2 \vdash N \sqsubseteq N' \Leftarrow B \sqsubseteq B' \end{array}}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell'_1 \vdash \text{let } x=M : A \text{ in } N \sqsubseteq \text{let } x=M' : A' \text{ in } N' \Leftarrow B \sqsubseteq B'}$$

Figure 6.4: Precision rules of λ_{IFC}^c (Part I)

$$\boxed{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'}$$

$$\begin{array}{c}
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell_2' \vdash L \sqsubseteq L' \Leftarrow \text{Bool}_{\ell_3} \sqsubseteq \text{Bool}_{\ell_3} \\
\forall \ell \ell'. \Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1 \vee \ell_3; \ell_1 \vee \ell_3; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \\
\forall \ell \ell'. \Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1 \vee \ell_3; \ell_1 \vee \ell_3; \ell; \ell' \vdash N \sqsubseteq N' \Leftarrow A \sqsubseteq A' \\
B = \text{stamp } A \ell_3 \qquad B' = \text{stamp } A' \ell_3
\end{array} \\
\sqsubseteq\text{-if} \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell_2' \vdash \text{if } L \ A \ \ell_3 \ M \ N \sqsubseteq \text{if } L' \ A' \ \ell_3 \ M' \ N' \Leftarrow B \sqsubseteq B'} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell_1' \vdash L \sqsubseteq L' \Leftarrow \text{Bool}_* \sqsubseteq \text{Bool}_* \\
\forall \ell_2 \ell_2'. \Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_2; \ell_2' \vdash M \sqsubseteq M' \Leftarrow T_* \sqsubseteq T_*' \\
\forall \ell_2 \ell_2'. \Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_2; \ell_2' \vdash N \sqsubseteq N' \Leftarrow T_* \sqsubseteq T_*'
\end{array} \\
\sqsubseteq\text{-if*} \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell_1' \vdash \text{if*} \ L \ T \ M \ N \sqsubseteq \text{if*} \ L' \ T' \ M' \ N' \Leftarrow T_* \sqsubseteq T_*'} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell_2' \vdash L \sqsubseteq L' \Leftarrow \text{Bool}_* \sqsubseteq \text{Bool}_{\ell_3} \\
\forall \ell \ell'. \Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_1 \vee \ell_3; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow T_* \sqsubseteq T_*' \\
\forall \ell \ell'. \Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_1 \vee \ell_3; \ell; \ell' \vdash N \sqsubseteq N' \Leftarrow T_* \sqsubseteq T_*' \\
B' = \text{stamp } A' \ell_3
\end{array} \\
\sqsubseteq\text{-if*} \ell \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell_2' \vdash \text{if*} \ L \ T \ M \ N \sqsubseteq \text{if} \ L' \ A' \ \ell_3 \ M' \ N' \Leftarrow T_* \sqsubseteq B'} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell_2' \vdash M \sqsubseteq M' \Leftarrow T_{\ell_3} \sqsubseteq T'_{\ell_3} \quad \ell_1 \leq \ell_3
\end{array} \\
\sqsubseteq\text{-ref} \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell_2' \vdash \text{ref } \ell_3 \ M \sqsubseteq \text{ref } \ell_3 \ M' \Leftarrow (\text{Ref } T_{\ell_3})_{\text{low}} \sqsubseteq (\text{Ref } T'_{\ell_3})_{\text{low}}} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_1; \ell_1' \vdash M \sqsubseteq M' \Leftarrow T_{\ell_2} \sqsubseteq T'_{\ell_2}
\end{array} \\
\sqsubseteq\text{-ref?} \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_1; \ell_1' \vdash \text{ref}^p \ell_2 \ M \sqsubseteq \text{ref}^q \ell_2 \ M' \Leftarrow (\text{Ref } T_{\ell_2})_{\text{low}} \sqsubseteq (\text{Ref } T'_{\ell_2})_{\text{low}}} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_1; \ell_2; \ell_3 \vdash M \sqsubseteq M' \Leftarrow T_{\ell} \sqsubseteq T'_{\ell} \quad \ell_1 \leq \ell
\end{array} \\
\sqsubseteq\text{-ref?} \ell \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; *, *; \ell_1; \ell_2; \ell_3 \vdash \text{ref}^p \ell \ M \sqsubseteq \text{ref} \ell \ M' \Leftarrow (\text{Ref } T_{\ell})_{\text{low}} \sqsubseteq (\text{Ref } T'_{\ell})_{\text{low}}} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell_1' \vdash M \sqsubseteq M' \Leftarrow (\text{Ref } A)_{\ell_2} \sqsubseteq (\text{Ref } A')_{\ell_2} \\
B = \text{stamp } A \ell_2 \qquad B' = \text{stamp } A' \ell_2
\end{array} \\
\sqsubseteq\text{-deref} \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell_1' \vdash ! \ M \ A \ \ell_2 \sqsubseteq ! \ M' \ A' \ \ell_2 \Leftarrow B \sqsubseteq B'} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow (\text{Ref } T_*)_* \sqsubseteq (\text{Ref } T'_*)_* \\
\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash ! * \ M \ T \sqsubseteq ! * \ M' \ T' \Leftarrow T_* \sqsubseteq T'_*
\end{array} \\
\sqsubseteq\text{-deref*} \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash ! * \ M \ T \sqsubseteq ! * \ M' \ T' \Leftarrow T_* \sqsubseteq T'_*} \\
\\
\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell_1' \vdash M \sqsubseteq M' \Leftarrow (\text{Ref } T_*)_* \sqsubseteq (\text{Ref } A')_{\ell_2} \\
B' = \text{stamp } A' \ell_2
\end{array} \\
\sqsubseteq\text{-deref*} \ell \frac{}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell_1; \ell_1' \vdash ! * \ M \ T \sqsubseteq ! \ M' \ A' \ \ell_2 \Leftarrow T_* \sqsubseteq B'}
\end{array}$$

Figure 6.5: Precision rules of λ_{IFC}^c (Part II)

$$\boxed{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'}$$

$$\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell_2' \vdash L \sqsubseteq L' \Leftarrow (\text{Ref } T_{\hat{\ell}})_{\ell} \sqsubseteq (\text{Ref } T'_{\hat{\ell}})_{\ell} \\
\Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell_2' \vdash M \sqsubseteq M' \Leftarrow T_{\hat{\ell}} \sqsubseteq T'_{\hat{\ell}} \\
\ell_1 \leq \hat{\ell} \qquad \qquad \qquad \ell \leq \hat{\ell} \\
\hline
\text{\(\sqsubseteq\)-assign} \quad \Gamma; \Gamma'; \Sigma; \Sigma'; \ell_1; \ell_1; \ell_2; \ell_2' \vdash \text{assign } L M T \hat{\ell} \ell \sqsubseteq \text{assign } L' M' T' \hat{\ell} \ell \Leftarrow \\
\text{Unit}_{\text{low}} \sqsubseteq \text{Unit}_{\text{low}}
\end{array}$$

$$\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash L \sqsubseteq L' \Leftarrow (\text{Ref } T_{\hat{g}})_{\star} \sqsubseteq (\text{Ref } T'_{\hat{g}'})_{\star} \\
\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow T_{\hat{g}} \sqsubseteq T'_{\hat{g}'} \\
\hline
\text{\(\sqsubseteq\)-assign?} \quad \Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash \text{assign}^? L M T \hat{g} \sqsubseteq \text{assign}^? L' M' T' \hat{g}' \Leftarrow \\
\text{Unit}_{\text{low}} \sqsubseteq \text{Unit}_{\text{low}}
\end{array}$$

$$\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell_2' \vdash L \sqsubseteq L' \Leftarrow (\text{Ref } T_{\hat{g}})_{\star} \sqsubseteq (\text{Ref } T'_{\hat{\ell}})_{\ell} \\
\Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell_2' \vdash M \sqsubseteq M' \Leftarrow T_{\hat{g}} \sqsubseteq T'_{\hat{\ell}} \\
\ell_1 \leq \hat{\ell} \qquad \qquad \qquad \ell \leq \hat{\ell} \\
\hline
\text{\(\sqsubseteq\)-assign?!} \quad \Gamma; \Gamma'; \Sigma; \Sigma'; g; \ell_1; \ell_2; \ell_2' \vdash \text{assign}^? L M T \hat{g} \sqsubseteq \text{assign } L' M' T' \hat{\ell} \ell \Leftarrow \\
\text{Unit}_{\text{low}} \sqsubseteq \text{Unit}_{\text{low}}
\end{array}$$

$$\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g_1; g'_1; |PC|; |PC'| \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \quad \vdash PC \sqsubseteq PC' \Leftarrow g_1 \sqsubseteq g'_1 \\
\ell_1 \vee \ell_2 \leq |PC| \quad \ell'_1 \vee \ell_2 \leq |PC'| \quad B = \text{stamp } A \ell_2 \quad B' = \text{stamp } A' \ell_2 \\
\hline
\text{\(\sqsubseteq\)-prot} \quad \Gamma; \Gamma'; \Sigma; \Sigma'; g_2; g'_2; \ell_1; \ell'_1 \vdash \text{prot } PC \ell_2 M A \sqsubseteq \text{prot } PC' \ell_2 M' A' \Leftarrow B \sqsubseteq B'
\end{array}$$

$$\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g_1; g'_1; |PC|; |PC'| \vdash M \sqsubseteq M' \Leftarrow T_{\star} \sqsubseteq T'_{\star} \quad \vdash PC \sqsubseteq PC' \Leftarrow g_1 \sqsubseteq g'_1 \\
\ell_1 \vee \ell_2 \leq |PC| \quad \ell'_1 \vee \ell'_2 \leq |PC'| \quad \ell_2 \leq \ell'_2 \\
\hline
\text{\(\sqsubseteq\)-prot!} \quad \Gamma; \Gamma'; \Sigma; \Sigma'; g_2; g'_2; \ell_1; \ell'_1 \vdash \text{prot } PC \ell_2 M T_{\star} \sqsubseteq \text{prot } PC' \ell'_2 M' T'_{\star} \Leftarrow T_{\star} \sqsubseteq T'_{\star}
\end{array}$$

$$\begin{array}{c}
\Gamma; \Gamma'; \Sigma; \Sigma'; g_1; g'_1; |PC|; |PC'| \vdash M \sqsubseteq M' \Leftarrow T_{\star} \sqsubseteq A \quad \vdash PC \sqsubseteq PC' \Leftarrow g_1 \sqsubseteq g'_1 \\
\ell_1 \vee \ell_2 \leq |PC| \quad \ell'_1 \vee \ell'_2 \leq |PC'| \quad B = \text{stamp } A \ell'_2 \quad \ell_2 \leq \ell'_2 \\
\hline
\text{\(\sqsubseteq\)-prot!!} \quad \Gamma; \Gamma'; \Sigma; \Sigma'; g_2; g'_2; \ell_1; \ell'_1 \vdash \text{prot } PC \ell_2 M T_{\star} \sqsubseteq \text{prot } PC' \ell'_2 M' A \Leftarrow T_{\star} \sqsubseteq B
\end{array}$$

Figure 6.6: Precision rules of λ_{IFC}^c (Part III)

$$\boxed{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'}$$

$$\begin{aligned}
& \sqsubseteq\text{-cast} \frac{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \quad c \sqsubseteq c' \quad \vdash c : A \Rightarrow B \quad \vdash c' : A' \Rightarrow B'}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \langle c \rangle \sqsubseteq M' \langle c' \rangle \Leftarrow B \sqsubseteq B'} \\
& \sqsubseteq\text{-castl} \frac{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \quad c \sqsubseteq c' \quad \vdash c : A \Rightarrow B}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \langle c \rangle \sqsubseteq M' \Leftarrow B \sqsubseteq A'} \\
& \sqsubseteq\text{-castr} \frac{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A' \quad A \sqsubseteq c' \quad \vdash c' : A' \Rightarrow B'}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq M' \langle c' \rangle \Leftarrow A \sqsubseteq B'} \\
& \sqsubseteq\text{-blame} \frac{\Gamma; \Sigma; g; \ell \vdash M \Leftarrow A \quad A \sqsubseteq A'}{\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash M \sqsubseteq \text{blame } p \Leftarrow A \sqsubseteq A'}
\end{aligned}$$

Figure 6.7: Precision rules of λ_{IFC}^c (Part IV)

then there exists value V s.t $M \mid \mu \mid PC \longrightarrow^* V \mid \mu$ and $\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash V \sqsubseteq V' \Leftarrow A \sqsubseteq A'$.

Proof. The proof is mechanized in `/src/Simulation/CatchUp.agda`. \square

It is straightforward to show that substitution preserves precision for λ_{IFC}^c (typing context precision is defined point-wise):

Lemma 17 (Substitution preserves precision). *Suppose the typing contexts are related by precision: $\Gamma \sqsubseteq \Gamma'$ and $\Sigma \sqsubseteq \Sigma'$. If*

$$(\Gamma, x:A); (\Gamma', x:A'); \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash N \sqsubseteq N' \Leftarrow B \sqsubseteq B'$$

and

$$\forall g' \ell' \ell. \Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash V \sqsubseteq V' \Leftarrow A \sqsubseteq A'$$

then $\Gamma; \Gamma'; \Sigma; \Sigma'; g; g'; \ell; \ell' \vdash N[x := V] \sqsubseteq N'[x := V'] \Leftarrow B \sqsubseteq B'$.

Proof. The proof is mechanized in `/src/CC2/SubstPrecision.agda`. \square

We then prove the main simulation lemma using Lemma 16 (“catch-up”) and Lemma 17 (substitution preserves precision). Heap precision is defined point-wise, similar to the definition of heap well-typedness.

Lemma 18 (Simulation between more precise and less precise λ_{IFC}^c terms). *Suppose PC, PC' are related by precision: $\vdash PC \sqsubseteq PC' \Leftarrow g \sqsubseteq g'$. Moreover suppose M, M' are related by precision:*

$$\emptyset; \emptyset; \Sigma_1; \Sigma'_1; g; g'; |PC|; |PC'| \vdash M \sqsubseteq M' \Leftarrow A \sqsubseteq A'$$

heap contexts Σ_1, Σ'_1 are related by precision: $\Sigma_1 \sqsubseteq \Sigma'_1$, the initial heaps μ_1, μ'_1 are also related by precision: $\Sigma_1; \Sigma'_1 \vdash \mu_1 \sqsubseteq \mu'_1$.

If $M' \mid \mu'_1 \mid PC' \longrightarrow N' \mid \mu'_2$, there exists $\Sigma_2, \Sigma'_2, N, \mu_2$ s.t $\Sigma_2 \supseteq \Sigma_1, \Sigma'_2 \supseteq \Sigma'_1, \Sigma_2 \sqsubseteq \Sigma'_2$,

$$M \mid \mu_1 \mid PC \longrightarrow^* N \mid \mu_2$$

the resulting terms are related by precision: $\emptyset; \emptyset; \Sigma_2; \Sigma'_2; g; g'; |PC|; |PC'| \vdash N \sqsubseteq N' \Leftarrow A \sqsubseteq A'$ and the resulting heaps are also related by precision: $\Sigma_2; \Sigma'_2 \vdash \mu_2 \sqsubseteq \mu'_2$.

Proof. The proof is mechanized in `/src/Simulation/Simulation.agda`. □

6.3 The Gradual Guarantee of λ_{IFC}^*

The definition of term precision for λ_{IFC}^* is in Figure 6.8. We first prove that the compilation from λ_{IFC}^* to λ_{IFC}^c preserves types:

Lemma 19 (Compilation preserves precision). *Suppose the tying contexts are related by precision: $\Gamma \sqsubseteq \Gamma'$, and the PC labels are also related: $g \sqsubseteq g'$. If well-typed λ_{IFC}^* terms M, M' are related by precision: $\vdash M \sqsubseteq M'$ and $\Gamma; g \vdash M : A$ and $\Gamma'; g' \vdash M' : A'$, then*

$$\forall \ell \ell'. \Gamma; \Gamma'; \emptyset; \emptyset; g; g'; \ell; \ell' \vdash \mathcal{C} M \sqsubseteq \mathcal{C} M' \Leftarrow A \sqsubseteq A'$$

$$\boxed{\vdash M_1 \sqsubseteq M_2}$$

$$\begin{array}{c}
\sqsubseteq^G\text{-const} \frac{}{\vdash (\$ k)_\ell \sqsubseteq (\$ k)_\ell} \quad \sqsubseteq^G\text{-var} \frac{}{\vdash x \sqsubseteq x} \\
\sqsubseteq^G\text{-lam} \frac{g_1 \sqsubseteq g_2 \quad A_1 \sqsubseteq A_2 \quad \vdash N_1 \sqsubseteq N_2}{\vdash (\lambda^{g_1} x : A_1 . N_1)_\ell \sqsubseteq (\lambda^{g_2} x : A_2 . N_2)_\ell} \quad \sqsubseteq^G\text{-app} \frac{\vdash L_1 \sqsubseteq L_2 \quad \vdash M_1 \sqsubseteq M_2}{\vdash (L_1 M_1)^p \sqsubseteq (L_2 M_2)^p} \\
\sqsubseteq^G\text{-if} \frac{\vdash L_1 \sqsubseteq L_2 \quad \vdash M_1 \sqsubseteq M_2 \quad \vdash N_1 \sqsubseteq N_2}{\vdash (\text{if } L_1 \text{ then } M_1 \text{ else } N_1)^p \sqsubseteq (\text{if } L_2 \text{ then } M_2 \text{ else } N_2)^p} \\
\sqsubseteq^G\text{-ann} \frac{\vdash M_1 \sqsubseteq M_2 \quad A_1 \sqsubseteq A_2}{\vdash (M_1 : A_1)^p \sqsubseteq (M_2 : A_2)^p} \\
\sqsubseteq^G\text{-let} \frac{\vdash M_1 \sqsubseteq M_2 \quad \vdash N_1 \sqsubseteq N_2}{\vdash \text{let } x = M_1 \text{ in } N_1 \sqsubseteq \text{let } x = M_2 \text{ in } N_2} \\
\sqsubseteq^G\text{-ref} \frac{\vdash M_1 \sqsubseteq M_2}{\vdash (\text{ref } \ell M_1)^p \sqsubseteq (\text{ref } \ell M_2)^p} \quad \sqsubseteq^G\text{-deref} \frac{\vdash M_1 \sqsubseteq M_2}{\vdash !^p M_1 \sqsubseteq !^p M_2} \\
\sqsubseteq^G\text{-assign} \frac{\vdash L_1 \sqsubseteq L_2 \quad \vdash M_1 \sqsubseteq M_2}{\vdash (L_1 := M_1)^p \sqsubseteq (L_2 := M_2)^p}
\end{array}$$

Figure 6.8: Precision rules of λ_{IFC}^*

Proof. The proof is mechanized in `/src/Compile/Precision/CompilePrecision.agda`. □

We prove the following lemma about λ_{IFC}^* , which is a corollary of Lemma 18:

Lemma 20. *Suppose M and M' are well-typed terms in λ_{IFC}^* that are related by precision, that is $\vdash M \sqsubseteq M'$ and $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M : A$ and $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M' : A'$, so $A \sqsubseteq A'$. If the compilation of M' substituted with input b reduces to a value:*

$$(\mathcal{C} M')[x := \$ b] \mid \emptyset \mid \text{low} \longrightarrow^* V' \mid \mu'$$

there exists some value V and heap μ s.t. the compilation of M substituted with b reduces to V :

$$(\mathcal{C} M)[x := \$ b] \mid \emptyset \mid \text{low} \longrightarrow^* V \mid \mu$$

and the resulting values are related by precision for some Σ, Σ' :

$$\emptyset; \emptyset; \Sigma; \Sigma'; \text{low}; \text{low}; \text{low}; \text{low} \vdash V \sqsubseteq V' \Leftarrow A \sqsubseteq A'$$

Proof. By Lemma 19 (compilation preserves precision), we have

$$(x:\text{Bool}_{\text{high}}); (x:\text{Bool}_{\text{high}}); \emptyset; \emptyset; \text{low}; \text{low}; \text{low}; \text{low} \vdash \mathcal{C} M \sqsubseteq \mathcal{C} M' \Leftarrow A \sqsubseteq A'$$

Substitution preserves precision (Lemma 17), so

$$\emptyset; \emptyset; \emptyset; \emptyset; \text{low}; \text{low}; \text{low}; \text{low} \vdash (\mathcal{C} M)[x := \$ b] \sqsubseteq (\mathcal{C} M')[x := \$ b] \Leftarrow A \sqsubseteq A'$$

We then proceed by induction on the reduction of $(\mathcal{C} M')[x := \$ b]$ to a value V' , using Lemma 18 to show that $(\mathcal{C} M)[x := \$ b]$ reduces to a corresponding term at each step. So we have $(\mathcal{C} M)[x := \$ b] \longrightarrow^* N$ where $N \sqsubseteq V'$ for some N . We then apply Lemma 16 to show that N reduces to a value V where $V \sqsubseteq V'$. \square

Finally, we state and prove the gradual guarantee of λ_{IFC}^* as a corollary of Lemma 20:

Theorem 21 (Gradual guarantee of λ_{IFC}^*). *Suppose M and M' are λ_{IFC}^* programs related by precision: $\vdash M \sqsubseteq M'$. If $\text{eval}(M', b_1) = b_2$, then $\text{eval}(M, b_1) = b_2$.*

Proof. By Definition 1 (whole programs of λ_{IFC}^*), $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M : \text{Bool}_{\text{low}}$ and $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M' : \text{Bool}_{\text{low}}$. By $\text{eval}(M', b_1) = b_2$ and the definition of eval (Figure 2.7), we have $(\mathcal{C} M')[x := \$ b_1] \mid \emptyset \mid \text{low} \longrightarrow^* \$ b_2 \mid \mu'$ for some μ' . By Lemma 20, there exists some V, μ such that

$$(\mathcal{C} M)[x := \$ b_1] \mid \emptyset \mid \text{low} \longrightarrow^* V \mid \mu$$

and

$$\emptyset; \emptyset; \Sigma; \Sigma'; \text{low}; \text{low}; \text{low}; \text{low} \vdash V \sqsubseteq \$ b_2 \Leftarrow \text{Bool}_{\text{low}} \sqsubseteq \text{Bool}_{\text{low}}$$

By inversion on the precision of λ_{IFC}^c and by the canonical form of a value of Bool_{low} ,
 $V = \$ b_2$. By the definition of *eval*, $\text{eval}(M, b_1) = b_2$. □

CHAPTER 7

NONINTERFERENCE OF λ_{IFC}^*

In this chapter, I prove that λ_{IFC}^* satisfies noninterference (Theorem 45). Noninterference says that high-security input never flows into low-security output.

I show that security checks can be modeled by reducing security coercion sequences to their normal form in Section 7.1. After that, I present the full proof of noninterference for λ_{IFC}^* . This proof employs a three-step approach.

First, in Section 7.2, I prove that $\lambda_{\text{IFC}}^{\text{DYN}}$ (the dynamic extreme of λ_{IFC}^*) satisfies noninterference. This proof is a straightforward adaptation of the noninterference proof from Chen and Siek [65]. The proof (Lemma 28) uses a standard erasure-based approach [47, 23, 50, 59, 51], where the high-security parts of a program are erased to an opaque value.

Second, in Section 7.3, I prove a simulation lemma between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$ (Lemma 41). The simulation relation is defined in Figures 7.2 and 7.3; the intuition is that a λ_{IFC}^c term always produces a value that is as secure as the one produced by its related $\lambda_{\text{IFC}}^{\text{DYN}}$ term. I translate λ_{IFC}^c terms to $\lambda_{\text{IFC}}^{\text{DYN}}$ by (1) getting rid of all the casts and (2) converting static heap enforcement (`ref` and `assign`) to dynamic enforcement (`NSU`). The translation is defined in Figure 7.4. The noninterference property of λ_{IFC}^c (Lemma 43) follows directly from the multi-step simulation lemma (Lemma 42) and the noninterference result of $\lambda_{\text{IFC}}^{\text{DYN}}$.

Third, in Section 7.4, I prove the noninterference theorem of λ_{IFC}^* (Theorem 45) as a corollary of the noninterference property of λ_{IFC}^c .

Similar to GSL_{Ref} and GLIO , the statements of noninterference for both $\lambda_{\text{IFC}}^{\text{DYN}}$ and λ_{IFC}^c are *termination-insensitive*. Thus, I will only consider successful executions that produce values. In other words, I will not consider reduction rules that trigger IFC monitor failures, such as `NSU` errors in $\lambda_{\text{IFC}}^{\text{DYN}}$ or cast errors in λ_{IFC}^c . As I explained in Section 2.2.1, the programming language runtime forces the program to diverge whenever blame is detected,

possibly sending a private error message to the software developer.

The Agda proofs cited in the section are available at:

<https://github.com/Gradual-Typing/LambdaIFCStar>

7.1 The Normalization of Coercions Checks Information Flow

I show that reducing coercion sequences to normal form models IFC checks because the normalization either succeeds or fails. If a coercion sequences successfully reduces to normal form, the IFC check succeeds and the flow is justified. If it reduces to a failure, then an illegal flow is detected and the program errors.

Lemma 22 (Strong normalization of coercion sequences). *If $\vdash \bar{c} : g_1 \Rightarrow g_2$, then either (1) $\bar{c} \longrightarrow^* \bar{d}$ and $\mathbf{NF} \bar{d}$ or (2) $\bar{c} \longrightarrow^* \perp^p g_1 g_2$.*

Proof. The proof is in `cexpr-sn` of `/src/CoercionExpr/CoercionExpr.agda` □

Normalization of coercion sequences is deterministic:

Lemma 23 (Reduction of coercion sequences is deterministic). *If $\bar{c} \longrightarrow \bar{d}_1$ and $\bar{c} \longrightarrow \bar{d}_2$, then $\bar{d}_1 = \bar{d}_2$.*

Proof. The proof is in `det` of `/src/CoercionExpr/CoercionExpr.agda`. □

Lemma 24 (Normalization of coercion sequences is deterministic). *Suppose $\bar{c} \longrightarrow^* \bar{d}_1$ and $\bar{c} \longrightarrow^* \bar{d}_2$. If $\mathbf{NF} \bar{d}_i$ or $\bar{d}_i = \perp^p g_1 g_2$, then $\bar{d}_1 = \bar{d}_2$.*

Proof. The proof is in `det-mult` of `/src/CoercionExpr/CoercionExpr.agda`. □

Recall that in Section 3.2.2, we mention that coercion composition models explicit flows, while coercion stamping models implicit flow. Here we prove lemmas that formalize that intuition.

We reason about explicit flows first. If we compose one coercion sequence with another and then reduce the result to normal form, the security of the resulting coercion sequence should be greater than or equal to that of the first sequence:

Lemma 25 (Composition models explicit flow).

If $\mathbf{NF} \bar{c}$ and $\bar{c} \bar{d} \longrightarrow^* \bar{c}'$ and $\mathbf{NF} \bar{c}'$, then $|\bar{c}| \leq |\bar{c}'|$.

Proof. The proof is in `comp-security` of `/src/CoercionExpr/SecurityLevel.agda`. □

Next we show that stamping models implicit flow correctly, promoting the security of the stamped coercion by joining it with the stamped label:

Lemma 26 (Stamping models implicit flow).

If $\mathbf{NF} \bar{c}$, then $|\mathit{stamp} \bar{c} \ell| = |\bar{c}| \vee \ell$ and $|\mathit{stamp}! \bar{c} \ell| = |\bar{c}| \vee \ell$.

Proof. The proof is in `stamp1-security` of `/src/CoercionExpr/Stamping.agda`. □

7.2 Noninterference of $\lambda_{\text{IFC}}^{\text{DYN}}$

We obtain the big-step semantics in Figure 7.1 by a mechanical conversion from the successful (non-erroring) cases of the small-step semantics of $\lambda_{\text{IFC}}^{\text{DYN}}$ in Figure 2.9. I conjecture that small-step and big-step semantics for $\lambda_{\text{IFC}}^{\text{DYN}}$ coincide; in particular, multi-stepping to a value should imply big-step:

Lemma 27. If $M \mid \mu \mid pc \longrightarrow^* V \mid \mu'$, then $\mu \mid pc \vdash M \Downarrow V \mid \mu'$

The proof technique for Lemma 27 should be standard. We could first follow Streicher [77] (the first exercise about PCF on p. 17) and prove a lemma that if $M \mid \mu_1 \mid pc \longrightarrow N \mid \mu_2$ and $\mu_2 \mid pc \vdash N \Downarrow V \mid \mu_3$, then $\mu_1 \mid pc \vdash M \Downarrow V \mid \mu_3$. We then perform induction on $M \mid \mu \mid pc \longrightarrow^* V \mid \mu'$: if M is already a value, the goal is proved directly; otherwise, M takes at least one reduction step, so we use the induction hypothesis and apply the aforementioned lemma.

Lemma 28 (Noninterference of $\lambda_{\text{IFC}}^{\text{DYN}}$). If $\emptyset \mid \text{low} \vdash M[x := (\$ b_1)_{\text{high}}] \Downarrow (\$ b_3)_{\text{low}} \mid \mu_1$ and $\emptyset \mid \text{low} \vdash M[x := (\$ b_2)_{\text{high}}] \Downarrow (\$ b_4)_{\text{low}} \mid \mu_2$ then $b_3 = b_4$.

$$\boxed{\mu \mid pc \vdash M \Downarrow V \mid \mu'}$$

$$\begin{array}{c}
\Downarrow\text{-val} \frac{}{\mu \mid pc \vdash V \Downarrow V \mid \mu} \quad \Downarrow\text{-app} \frac{\mu \mid pc \vdash L \Downarrow (\lambda x. N)_\ell \mid \mu_1 \quad \mu_1 \mid pc \vdash M \Downarrow V \mid \mu_2 \quad \mu_2 \mid pc \vee \ell \vdash N[x := V] \Downarrow W \mid \mu_3}{\mu \mid pc \vdash L M \Downarrow W \vee \ell \mid \mu_3} \\
\Downarrow\text{-if-true} \frac{\mu \mid pc \vdash L \Downarrow (\$ \text{true})_\ell \mid \mu_1 \quad \mu_1 \mid pc \vee \ell \vdash M \Downarrow V \mid \mu_2}{\mu \mid pc \vdash \text{if } L M N \Downarrow V \vee \ell \mid \mu_2} \quad \Downarrow\text{-if-false} \frac{\mu \mid pc \vdash L \Downarrow (\$ \text{false})_\ell \mid \mu_1 \quad \mu_1 \mid pc \vee \ell \vdash N \Downarrow V \mid \mu_2}{\mu \mid pc \vdash \text{if } L M N \Downarrow V \vee \ell \mid \mu_2} \\
\Downarrow\text{-ref?} \frac{\mu \mid pc \vdash M \Downarrow V \mid \mu_1 \quad n \text{ FreshIn } \mu_1(\ell) \quad pc \leq \ell}{\mu \mid pc \vdash \text{ref? } \ell M \Downarrow (\text{addr } n_\ell)_{\text{low}} \mid (\mu_1, \ell \mapsto n \mapsto (V \vee \ell))} \\
\Downarrow\text{-deref} \frac{\mu \mid pc \vdash M \Downarrow (\text{addr } n_{\hat{\ell}})_\ell \mid \mu_1 \quad \mu_1(\hat{\ell}, n) = V}{\mu \mid pc \vdash ! M \Downarrow V \vee \ell \mid \mu_1} \\
\Downarrow\text{-assign?} \frac{\mu \mid pc \vdash L \Downarrow (\text{addr } n_{\hat{\ell}})_\ell \mid \mu_1 \quad \mu_1 \mid pc \vdash M \Downarrow V \mid \mu_2 \quad pc \vee \ell \leq \hat{\ell}}{\mu \mid pc \vdash L :=? M \Downarrow (\$ \text{unit})_{\text{low}} \mid [\hat{\ell} \mapsto n \mapsto (V \vee \hat{\ell})] \mu_2}
\end{array}$$

Figure 7.1: Big-step operational semantics (successful cases) of $\lambda_{\text{IFC}}^{\text{DYN}}$

Proof. The proof is fully mechanized in `/src/Dyn/Noninterference.agda`. The structure of the proof directly follows the noninterference proof of $\lambda_{\text{SEC}}^{\Rightarrow}$ [65]. \square

7.3 Simulation Between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$

Definition 29 (Simulation between heaps). $\Sigma \vdash \mu' \leq \mu \triangleq \forall \ell, n. \text{if } \mu(\ell, n) = V, \text{ then there exists } V' \text{ s.t. } \mu'(\ell, n) = V' \text{ and } \text{low} \vdash V' \leq V \Leftarrow T_\ell \text{ where } T = \Sigma(\ell, n).$

Lemma 30 (Casting preserves simulation). *If $gc \vdash W' \leq V \Leftarrow A, \vdash c : A \Rightarrow B,$ and $V \langle c \rangle \longrightarrow^* W,$ then $gc \vdash W' \leq W \Leftarrow B.$*

Proof. By induction on the multi-step cast reduction.

Zero step Directly proved by applying rule $\leq\text{-cast}$.

One or more steps Casing on the first reduction step yields three sub-cases:

cast

$$gc \vdash W' \leq V_r \Leftarrow A \tag{7.1}$$

$$V_r \langle c_r, \bar{c} \rangle \longrightarrow V_r \langle c_r, \bar{d} \rangle \longrightarrow^* W \tag{7.2}$$

By induction hypothesis, $gc \vdash W' \leq W \Leftarrow B.$

cast-id

$$gc \vdash W' \leq V_r \Leftarrow A \tag{7.3}$$

$$V_r \langle \text{id}(\iota), \text{id}(g) \rangle \longrightarrow V_r \tag{7.4}$$

The goal is proved directly.

$$\boxed{gc \vdash M \leq N \Leftarrow A}$$

$$\begin{array}{c}
\leq\text{-var} \frac{}{gc \vdash x \leq x \Leftarrow A} \quad \leq\text{-const} \frac{\ell' \leq \ell}{gc \vdash (\$ k)_{\ell'} \leq \$ k \Leftarrow \iota_{\ell}} \\
\leq\text{-wrapped-const} \frac{\ell' \leq |\bar{c}| \quad \vdash \bar{c} : \ell \Rightarrow g \quad \mathbf{NF} \bar{c} \quad \ell \neq g}{gc \vdash (\$ k)_{\ell'} \leq \$ k \langle \mathbf{id}(\iota), \bar{c} \rangle \Leftarrow \iota_g} \\
\leq\text{-lam} \frac{g \vdash N' \leq N \Leftarrow B \quad \ell' \leq \ell}{gc \vdash (\lambda x. N')_{\ell'} \leq \lambda x. N \Leftarrow (A \xrightarrow{g} B)_{\ell}} \\
\leq\text{-wrapped-lam} \frac{g_3 \vdash N' \leq N \Leftarrow D \quad \ell \leq |\bar{c}| \quad \vdash \bar{d} : g_1 \Rightarrow g_3 \quad \vdash \mathbf{c} : A \Rightarrow C \quad \vdash \mathbf{d} : D \Rightarrow B \quad \mathbf{NF} \bar{c}}{gc \vdash (\lambda x. N')_{\ell} \leq (\lambda x. N) \langle \bar{d}, \mathbf{c} \rightarrow \mathbf{d}, \bar{c} \rangle \Leftarrow (A \xrightarrow{g_1} B)_{g_2}} \\
\leq\text{-addr} \frac{\ell' \leq \ell}{gc \vdash (\mathbf{addr} n_{\hat{\ell}})_{\ell'} \leq \mathbf{addr} n \Leftarrow \mathbf{Ref} (T_{\hat{\ell}})_{\ell}} \\
\leq\text{-wrapped-addr} \frac{\ell \leq |\bar{c}| \quad \vdash \mathbf{c} : T_{g_1} \Rightarrow S_{\hat{\ell}} \quad \vdash \mathbf{d} : S_{\hat{\ell}} \Rightarrow T_{g_1} \quad \mathbf{NF} \bar{c}}{gc \vdash (\mathbf{addr} n_{\hat{\ell}})_{\ell} \leq (\mathbf{addr} n) \langle \mathbf{Ref} \mathbf{c} \mathbf{d}, \bar{c} \rangle \Leftarrow (\mathbf{Ref} (T_{g_1}))_{g_2}} \\
\leq\text{-app} \frac{\ell^c \vdash M' \leq M \Leftarrow (A \xrightarrow{\ell^c \vee \ell} B)_{\ell} \quad \ell^c \vdash N' \leq N \Leftarrow A}{\ell^c \vdash M' N' \leq \mathbf{app} M N A B \ell \Leftarrow C} \\
\leq\text{-app} \star \frac{gc \vdash M' \leq M \Leftarrow (A \xrightarrow{\star} T_{\star})_{\star} \quad gc \vdash N' \leq N \Leftarrow A}{gc \vdash M' N' \leq \mathbf{app} \star M N A T \Leftarrow T_{\star}} \\
\leq\text{-if} \frac{\ell^c \vdash L' \leq L \Leftarrow \mathbf{Bool}_{\ell} \quad \ell^c \vee \ell \vdash M' \leq M \Leftarrow A \quad \ell^c \vee \ell \vdash N' \leq N \Leftarrow A}{\ell^c \vdash \mathbf{if} L' M' N' \leq \mathbf{if} L A \ell M N \Leftarrow B} \\
\leq\text{-if} \star \frac{gc \vdash L' \leq L \Leftarrow \mathbf{Bool}_{\star} \quad \star \vdash M' \leq M \Leftarrow T_{\star} \quad \star \vdash N' \leq N \Leftarrow T_{\star}}{gc \vdash \mathbf{if} L' M' N' \leq \mathbf{if} \star L T M N \Leftarrow T_{\star}} \\
\leq\text{-cast} \frac{gc \vdash M \leq N \Leftarrow A \quad \vdash \mathbf{c} : A \Rightarrow B}{gc \vdash M \leq N \langle \mathbf{c} \rangle \Leftarrow B} \\
\leq\text{-prot} \frac{\ell' \leq \ell \quad g_2 \vdash M' \leq M \Leftarrow A \quad \vdash PC \Leftarrow g_2}{g_1 \vdash \mathbf{prot} \ell' M' \leq \mathbf{prot} PC \ell M A \Leftarrow B}
\end{array}$$

Figure 7.2: Simulation relation between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$ (Part I)

$$\boxed{gc \vdash M \leq N \Leftarrow A}$$

$$\begin{aligned} & \leq\text{-ref} \frac{\ell^c \vdash M' \leq M \Leftarrow T_\ell}{\ell^c \vdash \text{ref}^? \ell M' \leq \text{ref} \ell M \Leftarrow (\text{Ref } T_\ell)_{\text{low}}} \\ & \leq\text{-ref}^? \frac{\star \vdash M' \leq M \Leftarrow T_\ell}{\star \vdash \text{ref}^? \ell M' \leq \text{ref}^? \ell M \Leftarrow (\text{Ref } T_\ell)_{\text{low}}} \\ & \leq\text{-deref} \frac{gc \vdash M' \leq M \Leftarrow (\text{Ref } A)_\ell}{gc \vdash ! M' \leq ! M A \ell \Leftarrow B} \quad \leq\text{-deref} \star \frac{gc \vdash M' \leq M \Leftarrow (\text{Ref } T_\star)_\star}{gc \vdash ! M' \leq ! \star M T \Leftarrow T_\star} \\ & \leq\text{-assign} \frac{\ell^c \vdash L' \leq L \Leftarrow (\text{Ref } T_\ell)_\ell \quad \ell^c \vdash M' \leq M \Leftarrow T_{\hat{\ell}}}{\ell^c \vdash L' :=^? M' \leq \text{assign } L M T \hat{\ell} \ell \Leftarrow \text{Unit}_{\text{low}}} \\ & \leq\text{-assign}^? \frac{gc \vdash L' \leq L \Leftarrow (\text{Ref } T_g)_\star \quad gc \vdash M' \leq M \Leftarrow T_g}{gc \vdash L' :=^? M' \leq \text{assign}^? L M T g \Leftarrow \text{Unit}_{\text{low}}} \end{aligned}$$

Figure 7.3: Simulation relation between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$ (Part II)

cast-comp

$$gc \vdash W' \leq V_r \langle \mathbf{c} \rangle \Leftarrow A \tag{7.5}$$

$$V_r \langle \mathbf{c} \rangle \langle \mathbf{d} \rangle \longrightarrow V_r \langle \mathbf{c} \ ; \ \mathbf{d} \rangle \longrightarrow^* W \tag{7.6}$$

We further reason about $V_r \langle \mathbf{c} \ ; \ \mathbf{d} \rangle \longrightarrow^* W$. The coercion after composition $\mathbf{c} \ ; \ \mathbf{d}$ is reducible and not identity, so the reduction must take one step by *cast* and reduce the top-level coercion sequence to its normal form \bar{c}_n :

$$V_r \langle c_{r1}, \bar{c} \rangle \langle c_{r2}, \bar{d} \rangle \longrightarrow V_r \langle c_r, \bar{c} \ ; \ \bar{d} \rangle \longrightarrow V_r \langle c_r, \bar{c}_n \rangle \longrightarrow^* W$$

We know $|\bar{c}| \leq |\bar{c}_n|$ because composition models explicit flow (Lemma 25).

- If $V_r = \$ k$ and $\bar{c}_n = \text{id}(\ell)$. $W' = (\$ k)_{\ell'}$ and $\ell' \leq |\bar{c}|$ (Lemma 32). We know $gc \vdash (\$ k)_{\ell'} \leq \$ k \Leftarrow \iota_\ell$ by rule $\leq\text{-const}$, because $\ell' \leq |\bar{c}| \leq |\bar{c}_n| = |\text{id}(\ell)| = \ell$.
- Otherwise, $V_r \langle c_r, \bar{c}_n \rangle$ is already a value. Apply rule $\leq\text{-wrapped-const}$,

\leq -wrapped-lam, or \leq -wrapped-addr depending on whether V_r is a constant, a λ , or an address.

□

Lemma 31 (Substitution preserves simulation). *If $g \vdash N' \leq N \Leftarrow A, (\Gamma, x : B); \Sigma; g; \ell \vdash N \Leftarrow A$, and $\forall g.g \vdash M' \leq M \Leftarrow B$, then $g \vdash N'[x := M'] \leq N[x := M] \Leftarrow A$.*

Proof. The proof is fully mechanized in `/src/Security/SubstPres.agda`. The simulation relation is defined in `/src/Security/SimRel.agda`. □

Lemma 32 (Simulation with wrapped constant). *If $gc \vdash M \leq (\$ k) \langle \mathbf{id}(\iota), \bar{c} \rangle \Leftarrow \iota_g$ and $(\mathbf{id}(\iota), \bar{c})$ is irreducible, then there exists ℓ s.t $M = (\$ k)_\ell$, and $\ell \leq |\bar{c}|$.*

Proof. Inversion on the simulation relation:

Related by \leq -wrapped-const The goal is proved directly.

Related by \leq -cast We know $gc \vdash M \leq \$ k \Leftarrow \iota_\ell$ and $\bar{c} : \ell \Rightarrow g$. Note that $(\mathbf{id}(\iota), \bar{c})$ is irreducible, so \bar{c} can be \uparrow , $\ell!$, or \uparrow ; **high!**, so $\ell \leq |\bar{c}|$. By rule \leq -const, we know $M = (\$ k)_{\ell'}$ for some ℓ' and $\ell' \leq \ell$. Thus $\ell' \leq \ell \leq |\bar{c}|$.

□

Lemma 33 (Simulation with reference proxy). *If $gc \vdash M \leq (\mathbf{addr} n) \langle \mathbf{Ref} c d, \bar{c} \rangle \Leftarrow (\mathbf{Ref} T_{g_1})_{g_2} \vdash c : T_{g_1} \Rightarrow S_{\hat{\iota}}, \vdash d : S_{\hat{\iota}} \Rightarrow T_{g_1}$, and $\mathbf{NF} \bar{c}$, then there exists ℓ s.t $M = (\mathbf{addr} n_{\hat{\iota}})_\ell$, and $\ell \leq |\bar{c}|$.*

Proof. Analogous to Lemma 32. The only extra case to consider is $\bar{c} = \mathbf{id}(\ell)$, which also satisfies $\ell \leq |\bar{c}|$. □

Lemma 34 (Simulation with function proxy). *If $gc \vdash M \leq (\lambda x. N) \langle \bar{d}, c \rightarrow d, \bar{c} \rangle \Leftarrow (A \xrightarrow{g_1} B)_{g_2}, \vdash \bar{d} : g_1 \Rightarrow g_3$, and $\mathbf{NF} \bar{c}$, then there exists N', ℓ s.t $M = (\lambda x. N')_\ell, g_3 \vdash N' \leq N \Leftarrow B$, and $\ell \leq |\bar{c}|$.*

Proof. Analogous to Lemma 32 and Lemma 33. □

Lemma 35 (Simulation with λ_{IFC}^c value). *If $gc \vdash M \leq V \Leftarrow A$, then M is a value.*

Proof. Inversion on the value V and the simulation relation $gc \vdash M \leq V \Leftarrow A$ yields 6 cases. We consider the two cases for constants; the cases for λ s and addresses are analogous.

Related by \leq -const We know $M = (\$ k)_\ell$ for some k, ℓ , so M is a value.

Related by \leq -wrapped-const By Lemma 32, $M = (\$ k)_\ell$ for some k, ℓ , so M is a value. □

Lemma 36 (Stamping preserves simulation). *If $gc \vdash V \leq W \Leftarrow A$ and $\ell' \leq \ell$, then $gc \vdash V \vee \ell' \leq \text{stamp } W \ell \Leftarrow \text{stamp } A \ell$.*

Proof. Casing on $\ell' \leq \ell$:

Case 1 $\ell' = \text{low}, \ell = \text{low}$: Straightforward because stamping low returns the same value.

Case 2 $\ell' = \text{low}, \ell = \text{high}$: Casing on value W and the simulation relation $gc \vdash V \leq W \Leftarrow A$ produces 6 sub-cases. We consider the two cases for constants below (corresponding to \leq -const and \leq -wrapped-const), because the cases for λ s and addresses are analogous:

- $gc \vdash (\$ k)_{\ell_1} \leq \$ k \Leftarrow \iota_{\ell_1}$
 - If $\ell_1 = \text{low}$, $gc \vdash (\$ k)_{\ell_1} \leq \$ k \langle \text{id}(\iota), \uparrow \rangle \Leftarrow \iota_{\text{high}}$ because $\ell_1 \leq | \uparrow | = \text{high}$ (rule \leq -wrapped-const).
 - If $\ell_1 = \text{high}$, $gc \vdash (\$ k)_{\ell_1} \leq \$ k \Leftarrow \iota_{\text{high}}$ because $\ell_1 \leq \text{high}$ (rule \leq -const).
- $gc \vdash (\$ k)_{\ell_1} \leq \$ k \langle \text{id}(\iota), \bar{c} \rangle \Leftarrow -$

- $gc \vdash (\$ k)_{\ell_1} \leq k \langle \text{id}(\iota), \text{stamp } \bar{c} \text{ high} \rangle \Leftarrow -$ because
 $\ell_1 \leq |\text{stamp } \bar{c} \text{ high}| = |\bar{c}| \vee \text{high} = \text{high}$ by Lemma 26 (Stamping models implicit flow) and rule \leq -wrapped-const.

Case 3 $\ell' = \text{high}, \ell = \text{high}$: Analogous to Case 2; the only difference is that instead of $(\$ k)_{\ell_1}$, the left side is always $(\$ k)_{\text{high}}$ because it is stamped with $\ell' = \text{high}$.

□

Lemma 37 (Casting a label expression models explicit flow). *If $\text{NF } e_1$ and $e_1 \langle \bar{c} \rangle \longrightarrow^* e_2$ and $\text{NF } e_2, |e_1| \leq |e_2|$.*

Proof. The proof is in cast-security of /src/LabelExpr/Security.agda. The proof is by inversion on the multi-step reduction. □

Lemma 38 (Stamping a label expression models implicit flow). *If $\text{NF } e, |\text{stamp } e \ell| = |e| \vee \ell$ and $|\text{stamp! } e \ell| = |e| \vee \ell$.*

Proof. The proof is fully mechanized in stamp_e-security and stamp!_e-security of /src/LabelExpr/Security.agda. The proof is by casing on $\text{NF } e$. □

Lemma 39. *Suppose $\Gamma; \Sigma; g; \ell \vdash V \Leftarrow S_{\ell_1}$ and $\vdash c : S_{\ell_1} \Rightarrow T_{\ell_2}$. If $V \langle c \rangle \longrightarrow^* W$, then $\ell_1 \leq \ell_2$.*

Proof. By induction on the multi-step reduction.

Zero step $c = (c_r, \bar{c})$ is irreducible, so $\bar{c} : \ell_1 \Rightarrow \ell_2$ is in its normal form. Thus $\ell_1 \leq \ell_2$.

One or more steps Case on the first reduction step:

- Rule *cast*: $V_r \langle c_r, \bar{c} \rangle \longrightarrow V_r \langle c_r, \bar{d} \rangle \longrightarrow^* W$. By preservation of coercion sequences, $\bar{d} : \ell_1 \Rightarrow \ell_2$. By induction hypothesis, $\ell_1 \leq \ell_2$.
- Rule *cast-id*: $V_r \langle \text{id}(\iota), \text{id}(g) \rangle \longrightarrow V_r$. We directly know $\ell_1 = g = \ell_2$.

- **Rule *cast-comp*:** $V_r \langle \mathbf{c} \rangle \langle \mathbf{d} \rangle \longrightarrow V_r \langle \mathbf{c} \ ; \ \mathbf{d} \rangle \longrightarrow^* W$. Suppose $\mathbf{c} = (-, \bar{c}), \bar{c} : \ell_0 \Rightarrow \ell_1$ and $\mathbf{d} = (-, \bar{d}), \bar{d} : \ell_1 \Rightarrow \ell_2$. We know $V_r \langle -, \bar{c} \ ; \ \bar{d} \rangle \longrightarrow V_r \langle \bar{c}_n \rangle \longrightarrow^* W$. By Lemma 25 (Composition models explicit flow), $\ell_1 = |\bar{c}| \leq |\bar{c}_n| = \ell_2$.

□

Lemma 40. *If $g_1 \vdash M \leq V \Leftarrow A$ then $g_2 \vdash M \leq V \Leftarrow A$.*

Proof. The proof is fully mechanized in Agda and can be found in the supplementary material. The proof is by casing on value and the simulation relation. □

Lemma 41 (Simulation between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$). *Suppose M_1 is a well-typed λ_{IFC}^c term: $\Gamma; \Sigma_1; gc; |PC| \vdash M_1 \Leftarrow A$, PC is a well-typed label expression: $\vdash PC \Leftarrow gc$, and μ_1 is a well-typed heap: $\Sigma_1 \vdash \mu_1$. Suppose $gc \vdash M'_1 \leq M_1 \Leftarrow A$, $\Sigma_1 \vdash \mu'_1 \leq \mu_1$, and $\ell \leq |PC|$. If $M_1 \mid \mu_1 \mid PC \longrightarrow M_2 \mid \mu_2$, then there exists M_3, μ_3, M'_2, μ'_2 s.t $M_2 \mid \mu_2 \mid PC \longrightarrow^* M_3 \mid \mu_3$, $M'_1 \mid \mu'_1 \mid \ell \longrightarrow^* M'_2 \mid \mu'_2$, $gc \vdash M'_2 \leq M_3 \Leftarrow A$, and $\Sigma_2 \vdash \mu'_2 \leq \mu_3$ for some Σ_2 .*

Proof. The proof is by induction on the reduction relation $M_1 \mid \mu_1 \mid PC \longrightarrow M_2 \mid \mu_2$ and inversion on the simulation relation $\vdash M'_1 \leq M_1 \Leftarrow A$. We only consider successful cases of the reduction (which do not produce errors), because the theorem statement of noninterference (Theorem 45) for λ_{IFC}^* is termination-insensitive.

Case ξ :

$$N_1 \mid \mu_1 \mid PC \longrightarrow N_2 \mid \mu_2 \tag{7.7}$$

$$\text{plug } N_1 F = M_1 \tag{7.8}$$

$$\text{plug } N_2 F = M_2 \tag{7.9}$$

We case on the frame F . We consider $F = \text{app } \square M A B \ell$ below; the cases for other frames all follow the same pattern. By inversion of the simulation relation, the

left side must be a function application:

$$\ell^c \vdash N'_1 M' \leq \text{app } N_1 M A B \ell \Leftarrow C$$

We know $\ell^c \vdash N'_1 \leq N_1 \Leftarrow (A \xrightarrow{\ell^c \vee \ell} B)_\ell$ and $\ell^c \vdash M' \leq M \Leftarrow A$. By the induction hypothesis, there exists N_3, μ_3, N'_2, μ'_2 such that $N_2 \mid \mu_2 \mid PC \longrightarrow^* N_3 \mid \mu_3, N'_1 \mid \mu'_1 \mid \ell \longrightarrow^* N'_2 \mid \mu'_2, \ell^c \vdash N'_2 \leq N_3 \Leftarrow (A \xrightarrow{\ell^c \vee \ell} B)_\ell$, and $\Sigma_2 \vdash \mu'_2 \leq \mu_3$ for some Σ_2 . Choose $M_3 = \text{plug } N_3 F = \text{app } N_3 M A B \ell, \mu_3 = \mu_3, M'_2 = N'_2 M',$ and $\mu'_2 = \mu'_2$. We know $\text{app } N_2 M A B \ell \mid \mu_2 \mid PC \longrightarrow^* \text{app } N_3 M A B \ell \mid \mu_3, N'_1 M' \mid \mu'_1 \mid \ell \longrightarrow^* N'_2 M' \mid \mu'_2,$ and $\Sigma_2 \vdash \mu'_2 \leq \mu_3$. We still need to show $\ell^c \vdash N'_2 M' \leq \text{app } N_3 M A B \ell \Leftarrow C$, which is proved by applying rule $\leq\text{-app}$.

Case *prot-val*: By inversion on the simulation relation, the left side must be a protection term. By Lemma 35, the body V' must be a value:

$$g_1 \vdash \text{prot } \ell' V' \leq \text{prot } PC \ell V A \Leftarrow B \quad (7.10)$$

$$g_2 \vdash V' \leq V \Leftarrow A \quad (7.11)$$

$$\ell' \leq \ell \quad (7.12)$$

$$\vdash PC \Leftarrow g_2 \quad (7.13)$$

We take a step by *prot-val* on the left side, we need to relate:

$$g_1 \vdash V' \vee \ell' \leq \text{stamp } V \ell \Leftarrow B$$

which is proved by applying Lemma 36 on (7.11) (with Lemma 40 applied) and (7.12).

Case *prot-ctx*:

$$M \mid \mu_1 \mid PC_2 \longrightarrow N \mid \mu_2$$

By inversion on the simulation relation, the left side must be protection:

$$g_1 \vdash \text{prot } \ell' M' \leq \text{prot } PC_2 \ell M A \Leftarrow B \quad (7.14)$$

$$g_2 \vdash M' \leq M \Leftarrow A \quad (7.15)$$

$$\ell' \leq \ell \quad (7.16)$$

$$\vdash PC_2 \Leftarrow g_2 \quad (7.17)$$

By inversion on the typing of the protection term on the right:

$$|PC_1| \vee \ell \leq |PC_2| \quad (7.18)$$

where PC_1 is the original PC to reduce the protection term on the right. To get the induction hypothesis, we need to show $\ell'' \vee \ell' \leq |PC_2|$ where ℓ'' is the PC to reduce the protection term on the left. We know $\ell'' \leq |PC_1|$, thus:

$$\ell'' \vee \ell' \leq \ell' \vee \ell \leq |PC_1| \vee \ell \leq |PC_2|$$

The induction hypothesis shows that there exists L, μ_3, N', μ'_2 s.t $N \mid \mu_2 \mid PC_2 \longrightarrow^* L \mid \mu_3, M' \mid \mu'_1 \mid \ell'' \vee \ell' \longrightarrow^* N' \mid \mu'_2, g_2 \vdash N' \leq L \Leftarrow A$, and $\Sigma_2 \vdash \mu'_2 \leq \mu_3$ for some Σ_2 . We step the left side to $\text{prot } \ell' N'$ and step the right side to $\text{prot } PC_2 \ell L A$ using the congruence of *prot-ctx*. We then relate the protection terms on both sides using rule $\leq\text{-prot}$.

Case *cast*: There are three sub-cases: *cast*, *cast-id*, and *cast-comp*.

Sub-case *cast*: We know $gc \vdash M' \leq V_r \langle c_r, \bar{c} \rangle \Leftarrow B$. $V_r \langle c_r, \bar{c} \rangle$ is not a value, so $gc \vdash M' \leq V_r \Leftarrow A$. We need to relate $gc \vdash M' \leq V_r \langle c_r, \bar{d} \rangle \Leftarrow B$, which is directly proved by applying $\leq\text{-cast}$.

Sub-case *cast-id*: We know $gc \vdash M' \leq V_r \langle \text{id}(\iota), \text{id}(g) \rangle \Leftarrow A$. Again, note that

$V_r \langle \text{id}(\iota), \text{id}(g) \rangle$ is not a value, so $gc \vdash M' \leq V_r \Leftarrow A$.

Sub-case *cast-comp*: We know $gc \vdash M' \leq V_r \langle c \rangle \langle d \rangle \Leftarrow B$. By inversion using rule $\leq\text{-cast}$, $gc \vdash M' \leq V_r \langle c \rangle \Leftarrow A$; c is irreducible, by Lemma 35 M' is a value. We need to show there exists M such that $V_r \langle c \mathbin{\text{\$}} d \rangle \mid \mu_1 \mid PC \longrightarrow^* M \mid \mu_1$ and $gc \vdash M' \leq M \Leftarrow B$. Casing on V_r :

- If $V_r = \$ k$. Suppose $c \mathbin{\text{\$}} d = d_r, \bar{c} \mathbin{\text{\$}} \bar{d}$ for some d_r . We take a step using *cast* by reducing the composed coercion sequence to its normal form: $\bar{c} \mathbin{\text{\$}} \bar{d} \longrightarrow^+ \bar{c}_n$. By Lemma 25 (Composition models explicit flow), $|\bar{c}| \leq |\bar{c}_n|$. By Lemma 32, we know $M' = (\$ k)_{\ell'}$ and $\ell' \leq |\bar{c}|$. By transitivity, $\ell' \leq |\bar{c}_n|$. (1) If \bar{c}_n is not an identity coercion, we relate $gc \vdash (\$ k)_{\ell'} \leq \$ k \langle d_r, \bar{c}_n \rangle \Leftarrow B$ directly using rule $\leq\text{-wrapped-const}$. (2) If $\bar{c}_n = \text{id}(\ell)$, we take one additional step by *cast-id*. We know $\ell' \leq |\text{id}(\ell)| = \ell$, thus $gc \vdash (\$ k)_{\ell'} \leq \$ k \Leftarrow \iota_{\ell}$ by rule $\leq\text{-const}$.
- If $V_r = \lambda x. N$. Similar to the constant case above. The only difference is that we do not need to specially handle $\bar{c}_n = \text{id}(\ell)$, because the function coercion $(\bar{d}, c \rightarrow d, \text{id}(\ell))$ is already irreducible.
- If $V_r = \text{addr } n$. Same as the lambda case above.

Case β : By inversion on the simulation relation, the left side must be a function application:

$$\ell^c \vdash L M \leq \text{app } (\lambda x. N) V A B \ell \Leftarrow C$$

where $C = \text{stamp } B \ell$. We know that L must be a λ by rule $\leq\text{-lam}$ and M must be a

value by Lemma 35:

$$\ell^c \vdash ((\lambda x. N')_{\ell'}) V' \leq \text{app } (\lambda x. N) V A B \ell \Leftarrow C \quad (7.19)$$

$$\ell' \leq \ell \quad (7.20)$$

$$\ell^c \vee \ell \vdash N' \leq N \Leftarrow B \quad (7.21)$$

$$\ell^c \vdash V' \leq V \Leftarrow A \quad (7.22)$$

We take one step by β on the left side of the simulation relation:

$$((\lambda x. N')_{\ell'}) V' \mid \mu' \mid \ell'' \longrightarrow \text{prot } \ell' (N'[x := V']) \mid \mu'$$

Now we need to show

$$\ell^c \vdash \text{prot } \ell' (N'[x := V']) \leq \text{prot } (\text{stamp } PC \ell) \ell (N[x := V]) B \Leftarrow C$$

Applying $\leq\text{-prot}$ yields three sub-goals: (1) $\ell' \leq \ell$ is directly proved by (7.20) (2) $\ell^c \vee \ell \vdash N'[x := V'] \leq N[x := V] \Leftarrow B$ is proved by applying Lemma 31 on (7.21) and (7.22). (3) $\vdash \text{stamp } PC \ell \Leftarrow \ell^c \vee \ell$ because stamping is well-typed.

Case *app-cast*:

$$(\text{stamp } PC_1 \ell) \langle \bar{d} \rangle \longrightarrow^* PC_2 \quad (7.23)$$

$$V \langle \mathbf{c} \rangle \longrightarrow^* W \quad (7.24)$$

By inversion on the simulation relation, the left side must be a function application:

$$\ell^c \vdash L M \leq \text{app } (\lambda x. N \langle \bar{d}, \mathbf{c} \rightarrow \mathbf{d}, \bar{c} \rangle) V C D \ell \Leftarrow E$$

where $E = \text{stamp } D \ell, \vdash \bar{d} : \ell^c \vee \ell \Rightarrow gc, \vdash \mathbf{c} : C \Rightarrow A$, and $\vdash \mathbf{d} : B \Rightarrow D$.

We know that L must be a λ by Lemma 34 and M must be a value by Lemma 35:

$$\ell^c \vdash ((\lambda x. N')_{\ell'}) V' \leq \text{app } (\lambda x. N \langle \bar{d}, \mathbf{c} \rightarrow \mathbf{d}, \bar{c} \rangle) V C D \ell \Leftarrow E \quad (7.25)$$

$$\ell' \leq |\bar{c}| \quad (7.26)$$

$$gc \vdash N' \leq N \Leftarrow B \quad (7.27)$$

$$\ell^c \vdash V' \leq V \Leftarrow C \quad (7.28)$$

We take a step by β on the left side of simulation relation:

$$((\lambda x. N')_{\ell'}) V' \mid \mu' \mid \ell'' \longrightarrow \text{prot } \ell' (N'[x := V']) \mid \mu'$$

Now we need to show

$$\ell^c \vdash \text{prot } \ell' (N'[x := V']) \leq \text{prot } PC_2 \ell ((N[x := W]) \langle \mathbf{d} \rangle) D \Leftarrow E$$

By rule $\leq\text{-prot}$, it is equivalent to showing:

$$\ell' \leq \ell \quad (7.29)$$

$$gc \vdash N'[x := V'] \leq (N[x := W]) \langle \mathbf{d} \rangle \Leftarrow D \quad (7.30)$$

$$\vdash PC_2 \Leftarrow gc \quad (7.31)$$

We know $\vdash PC_2 \Leftarrow gc$ because reducing label expressions preserves types. By (7.26) and $|\bar{c}| = \ell$ we prove (7.29). Apply Lemma 30 on (7.28) and (7.24), we get $\ell^c \vdash V' \leq W \Leftarrow A$. By Lemma 31, Lemma 40, and (7.27), $gc \vdash N'[x := V'] \leq N[x := W] \Leftarrow B$. Apply rule $\leq\text{-cast}$ and we prove (7.30).

Case $\text{app}\star\text{-cast}$: Similar to $\text{app}\text{-cast}$. The only noteworthy difference is that on the right side, we use $|\bar{c}|$ instead of the ℓ from the syntax of the function application, both in the protection term and when stamping the PC.

Cases β -if-true and β -if-false: By inversion on the simulation relation, the left side must also be an if-conditional:

$$\ell^c \vdash \text{if } (\$ \text{ true})_{\ell'} M' N' \leq \text{if } \$ \text{ true } A \ell M N \Leftarrow B$$

We know:

$$\ell' \leq \ell \tag{7.32}$$

$$\ell^c \vee \ell \vdash M' \leq M \Leftarrow A \tag{7.33}$$

$$\ell^c \vee \ell \vdash N' \leq N \Leftarrow A \tag{7.34}$$

Take one step on the left side. We need to relate

$$\ell^c \vdash \text{prot } \ell' M' \leq \text{prot } (\text{stamp } PC \ell) \ell M A \Leftarrow B$$

The goal is directly proved by applying rule \leq -prot. The case for β -if-false is analogous.

Cases if-true-cast and if-false-cast: Note that the label partial order trivially holds because $\ell' \leq \text{high}$ for any ℓ' . The rest is analogous to β -if-true and β -if-false.

Cases if*-true-cast and if*-false-cast: By Lemma 32 we know $\ell' \leq |\bar{c}|$. The rest is analogous to β -if-true and β -if-false.

Case ref: From the typing derivation (rule \vdash -ref), we also know:

$$\vdash PC \Leftarrow \ell^c \tag{7.35}$$

$$\ell^c \leq \ell \tag{7.36}$$

which implies

$$|PC| \leq \ell \quad (7.37)$$

We know $\ell' \leq |PC| \leq \ell$, where ℓ' is the PC that the left side is reduced with. We take a step by *ref?-ok* on the left side by choosing the same fresh address n . We need to show:

$$\ell^c \vdash (\text{addr } n_\ell)_{\text{low}} \leq \text{addr } n \Leftarrow (\text{Ref } T_\ell)_{\text{low}} \quad (7.38)$$

$$(\Sigma_1, \ell \mapsto n \mapsto T) \vdash (\mu', \ell \mapsto n \mapsto (V' \vee \ell)) \leq (\mu, \ell \mapsto n \mapsto V) \quad (7.39)$$

(7.38) is proved directly by $\leq\text{-addr}$. To prove (7.39) we need to show:

$$\text{low} \vdash V' \vee \ell \leq V \Leftarrow T_\ell \quad (7.40)$$

By inversion on the simulation relation and Lemma 40, we know $\text{low} \vdash V' \leq V \Leftarrow T_\ell$. From the definition of stamping we know $\text{stamp } V \ell = V$ if $V \Leftarrow T_\ell$; (7.40) is thus proved by Lemma 36.

Case *ref?*:

$$PC_1 \langle \star \Rightarrow^P \ell \rangle \longrightarrow^* PC_2$$

Reducing label expressions preserves types, so $\vdash PC_2 \Leftarrow \ell$. We know $|PC_2| = \ell$ and $\ell' \leq |PC_1|$, where ℓ' is the PC that the left side is reduced with. Casting models explicit flow (Lemma 37), so $|PC_1| \leq |PC_2|$. We thus have $\ell' \leq |PC_1| \leq |PC_2| = \ell$. Take one step by *ref?-ok* on the left side by choosing the same fresh address n . The rest of the proof follows that of *ref*.

Case *deref*:

$$\mu(\hat{\ell}, n) = V$$

By $\Sigma \vdash \mu' \leq \mu$ and Definition 29, there exists V' s.t $\mu'(\hat{\ell}, n) = V'$ and $\mathbf{low} \vdash V' \leq V \Leftarrow T_{\hat{\ell}}$ where $T = \Sigma(\hat{\ell}, n)$. By inversion on the simulation relation, the left side must be a dereference:

$$gc \vdash ! (\text{addr } n_{\hat{\ell}})_{\ell'} \leq ! (\text{addr } n) T_{\hat{\ell}} \Leftarrow B$$

where $\ell' \leq \ell$. Take one step on the left using rule *deref*, we need to relate:

$$gc \vdash \text{prot } \ell' V' \leq \text{prot } \mathbf{high} \ell V T_{\hat{\ell}} \Leftarrow B$$

which is directly proved by rule $\leq\text{-prot}$ and Lemma 40.

Case *deref*-cast*:

$$\mu(\hat{\ell}, n) = V$$

By $\Sigma \vdash \mu' \leq \mu$ and Definition 29, there exists V' s.t $\mu'(\hat{\ell}, n) = V'$ and $\mathbf{low} \vdash V' \leq V \Leftarrow S_{\hat{\ell}}$ where $S = \Sigma(\hat{\ell}, n)$. By inversion on the simulation relation, the left side must be a dereference:

$$gc \vdash ! (\text{addr } n_{\hat{\ell}})_{\ell} \leq ! \star (\text{addr } n \langle \mathbf{Ref } \mathbf{c } \mathbf{d}, \bar{c} \rangle) T \Leftarrow T_{\star}$$

where $\vdash \mathbf{c} : T_{\star} \Rightarrow S_{\hat{\ell}}, \vdash \mathbf{d} : S_{\hat{\ell}} \Rightarrow T_{\star}$. By Lemma 33, $\ell \leq |\bar{c}|$. We take a step on the left side using *deref*. We need to relate:

$$gc \vdash \text{prot } \ell V' \leq \text{prot } \mathbf{high} |\bar{c}| (V \langle \mathbf{d} \rangle) T_{\star} \Leftarrow B$$

which is proved directly by applying Lemma 40, $\leq\text{-prot}$, and then $\leq\text{-cast}$.

Case *deref-cast*: Analogous to *deref*-cast*.

Case β -assign: By inversion on the simulation relation, the left side is also an assignment:

$$\ell^c \vdash (\text{addr } n_{\hat{\ell}})_{\ell'} :=^? V' \leq \text{assign } (\text{addr } n) V T \hat{\ell} \ell \Leftarrow \text{Unit}_{\text{low}}$$

where $\ell' \leq \ell$. From the typing derivation we know $\ell^c \vee \ell \leq \hat{\ell}, \vdash PC \Leftarrow \ell^c$. We know $\ell'' \leq |PC| = \ell^c$ where ℓ'' is the PC on the left, thus $\ell'' \vee \ell' \leq \hat{\ell}$. We take one step on the left side using *assign?-ok*. The units on both sides relate straightforwardly. We need to relate:

$$\Sigma \vdash [\hat{\ell} \mapsto n \mapsto (V' \vee \hat{\ell})]_{\mu'} \leq [\hat{\ell} \mapsto n \mapsto V]_{\mu}$$

Given $\Sigma(\hat{\ell}, n) = T$, we need to show:

$$\text{low} \vdash V' \vee \hat{\ell} \leq V \Leftarrow T_{\hat{\ell}}$$

which can be proved similar to (7.40) because $\vdash V \Leftarrow T_{\hat{\ell}}$.

Case *assign-cast*:

$$V \langle c \rangle \longrightarrow^* W$$

where $\vdash c : T_{\hat{\ell}_2} \Rightarrow S_{\hat{\ell}_1}$. By Lemma 39, $\hat{\ell}_2 \leq \hat{\ell}_1$. From the typing derivation, $\vdash PC \Leftarrow \ell^c$ and $\ell^c \vee \ell \leq \hat{\ell}_2$. By inversion on the simulation relation, the left side is also an assignment:

$$\ell^c \vdash (\text{addr } n_{\hat{\ell}_1})_{\ell'} :=^? V' \leq \text{assign } (\text{addr } n \langle \text{Ref } c \mathbf{d}, \bar{c} \rangle) V T \hat{\ell}_2 \ell \Leftarrow \text{Unit}_{\text{low}}$$

We know $\ell' \leq |\bar{c}| = \ell$. $\ell'' \leq |PC| = \ell^c$ where ℓ'' is the PC that the left side is reduced with. Thus we have $\ell'' \vee \ell' \leq \hat{\ell}_2 \leq \hat{\ell}_1$. The check succeeds so we take one step on the left side by *assign?-ok*. The units on both sides relate straightforwardly. We need

to relate:

$$\Sigma \vdash [\hat{\ell}_1 \mapsto n \mapsto (V' \vee \hat{\ell}_1)]\mu' \leq [\hat{\ell}_1 \mapsto n \mapsto W]\mu$$

Need to show:

$$\text{low} \vdash V' \vee \hat{\ell}_1 \leq W \Leftarrow S_{\hat{\ell}_1}$$

which can be proved similar to (7.40).

Case *assign?-cast*:

$$(\text{stamp! } PC_1 \mid \bar{c}) \langle \star \Rightarrow^p \hat{\ell} \rangle \longrightarrow^* PC_2 \quad (7.41)$$

$$V \langle \mathbf{c} \rangle \longrightarrow^* W \quad (7.42)$$

By inversion on the simulation relation, the left side is also an assignment:

$$gc \vdash (\text{addr } n_{\hat{\ell}})_{\ell} :=^? V' \leq \text{assign?}^p (\text{addr } n \langle \mathbf{Ref } \mathbf{c } \mathbf{d}, \bar{c} \rangle) V T g \Leftarrow \text{Unit}_{\text{low}}$$

where $\vdash \mathbf{c} : T_g \Rightarrow S_{\hat{\ell}}, \vdash \mathbf{d} : S_{\hat{\ell}} \Rightarrow T_g$. By Lemma 33, $\ell \leq |\bar{c}|$. Stamping models implicit flow (Lemma 38), so $|\text{stamp } PC_1 \mid \bar{c}| = |PC_1| \vee |\bar{c}|$; casting models explicit flow (Lemma 37), so $|PC_1| \vee |\bar{c}| \leq |PC_2|$. Reduction of label expressions preserves types, so $\vdash PC_2 \Leftarrow \hat{\ell}$. Thus $|PC_2| = \hat{\ell}$, $|PC_1| \vee |\bar{c}| \leq \hat{\ell}$. We know $\ell' \leq |PC_1|$ where ℓ' is the PC the left side reduces with. Thus $\ell' \vee \ell \leq \hat{\ell}$; the check succeeds so we take one step on the left side by *assign?-ok*. The units on both sides relate straightforwardly. We need to relate:

$$\Sigma \vdash [\hat{\ell} \mapsto n \mapsto (V' \vee \hat{\ell})]\mu' \leq [\hat{\ell} \mapsto n \mapsto W]\mu$$

Need to show:

$$\text{low} \vdash V' \vee \hat{\ell} \leq W \Leftarrow S_{\hat{\ell}}$$

which again can be proved similar to (7.40).

□

Lemma 42 (Multi-step simulation). *Suppose M is a well-typed λ_{IFC}^c term $\Gamma; \Sigma; gc; |PC| \vdash M \Leftarrow A$, PC is a well-typed label expression: $\vdash PC \Leftarrow gc$, and μ_1 is a well-typed heap: $\Sigma \vdash \mu_1$. Suppose $gc \vdash M' \leq M \Leftarrow A$, $\Sigma \vdash \mu'_1 \leq \mu_1$, and $\ell \leq |PC|$. If $M \mid \mu_1 \mid PC \longrightarrow^* V \mid \mu_2$, then there exists V', μ'_2 such that $M' \mid \mu'_1 \mid \ell \longrightarrow^* V' \mid \mu'_2$ and $gc \vdash V' \leq V \Leftarrow A$.*

Proof. By induction on multi-step reduction $M \mid \mu_1 \mid PC \longrightarrow^* V \mid \mu_2$.

Zero step If M is already a value, choose V' to be M' . By Lemma 35, M' is also a value.

The values are in sync because $gc \vdash M' \leq M \Leftarrow A$.

One or more steps

$$M \mid \mu_1 \mid PC \longrightarrow N \mid \mu_3 \tag{7.43}$$

$$N \mid \mu_3 \mid PC \longrightarrow^* V \mid \mu_2 \tag{7.44}$$

Apply Lemma 41 and then use the induction hypothesis.

□

We then prove that λ_{IFC}^c satisfies termination-insensitive noninterference. The statement of termination-insensitive noninterference says that if we run a program with different high-security inputs in two executions, then their low-security output values should be related (e.g the same boolean):

Lemma 43 (Noninterference for λ_{IFC}^c). *If M is well-typed: $(x:\text{Bool}_{\text{high}}); \emptyset; \text{low}; \text{low} \vdash M \Leftarrow \text{Bool}_{\text{low}}$ and*

$$M[x := \$ b_1] \mid \emptyset \mid \text{low} \longrightarrow^* V_1 \mid \mu_1 \quad \text{and} \quad M[x := \$ b_2] \mid \emptyset \mid \text{low} \longrightarrow^* V_2 \mid \mu_2$$

$$\boxed{\epsilon M A = M'}$$

$$\begin{aligned}
& \epsilon x - = x \\
& \epsilon (\$ k) (\iota \ell) = (\$ k)_{\ell} \\
& \epsilon (\lambda x. N) ((A \xrightarrow{-} B)_{\ell}) = (\lambda x. \epsilon N B)_{\ell} \\
& \epsilon (\text{addr } n) (\text{Ref } (T_{\hat{\ell}})_{\ell}) = (\text{addr } n_{\hat{\ell}})_{\ell} \\
& \epsilon (\text{app } M N A B \ell) - = (\epsilon M (A \xrightarrow{*} B)_{\ell}) (\epsilon N A) \\
& \epsilon (\text{app}^* M N A T) - = (\epsilon M (A \xrightarrow{*} T_{*})_{*}) (\epsilon N A) \\
& \epsilon (\text{if } L A \ell M N) - = \text{if } (\epsilon L \text{Bool}_{\ell}) (\epsilon M A) (\epsilon N A) \\
& \epsilon (\text{if}^* L T M N) - = \text{if } (\epsilon L \text{Bool}_{*}) (\epsilon M T_{*}) (\epsilon N T_{*}) \\
& \epsilon (\text{ref } \ell M) (\text{Ref } T_{\ell})_{\text{low}} = \text{ref}^? \ell (\epsilon M T_{\ell}) \\
& \epsilon (\text{ref}^{?P} \ell M) (\text{Ref } T_{\ell})_{\text{low}} = \text{ref}^? \ell (\epsilon M T_{\ell}) \\
& \epsilon (! M A \ell) - = ! (\epsilon M (\text{Ref } A)_{\ell}) \\
& \epsilon (!^* M T) - = ! (\epsilon M (\text{Ref } T_{*})_{*}) \\
& \epsilon (\text{assign } L M T \hat{\ell} \ell) - = (\epsilon L (\text{Ref } T_{\hat{\ell}})_{\ell}) :=^? (\epsilon M T_{\hat{\ell}}) \\
& \epsilon (\text{assign}^{?P} L M T \hat{g}) - = (\epsilon L (\text{Ref } T_{\hat{g}})_{*}) :=^? (\epsilon M T_{\hat{g}}) \\
& \epsilon (N \langle c \rangle) - = \epsilon N A \quad , \text{ where } \vdash c : A \Rightarrow B \\
& \epsilon (\text{prot } PC \ell M A) - = \text{prot } \ell (\epsilon M A)
\end{aligned}$$

Figure 7.4: Erasure from λ_{IFC}^c to $\lambda_{\text{IFC}}^{\text{DYN}}$

then $V_1 = V_2$.

Proof. From the definition of ϵ (Figure 7.4), we know $\text{low} \vdash \epsilon(M)[x := (\$ b_i)_{\text{high}}] \leq M[x := \$ b_i] \Leftarrow \text{Bool}_{\text{low}}$. By Lemma 42, there exists V'_i, μ'_i s.t. $\epsilon(M)[x := (\$ b_i)_{\text{high}}] \mid \emptyset \mid \text{low} \longrightarrow^* V'_i \mid \mu'_i$ and $\text{low} \vdash V'_i \leq V_i \Leftarrow \text{Bool}_{\text{low}}$. The simulation relation must be of form $\text{low} \vdash (\$ a_i)_{\text{low}} \leq \$ a_i \Leftarrow \text{Bool}_{\text{low}}$ where a_i is the output boolean. By Lemma 27 and Lemma 28, $a_1 = a_2$, thus $V_1 = \$ a_1 = \$ a_2 = V_2$. \square

7.4 Noninterference of λ_{IFC}^*

The noninterference lemma of λ_{IFC}^* is a straightforward corollary of the noninterference lemma of λ_{IFC}^c and compilation preserves types:

Lemma 44. *Suppose a λ_{IFC}^* term M is well-typed:*

$$(x:\text{Bool}_{\text{high}}); \text{low} \vdash M : \text{Bool}_{\text{low}}$$

If for any boolean inputs b_1, b_2

$$(\mathcal{C} M)[x := \$ b_1] \mid \emptyset \mid \text{low} \longrightarrow^* V_1 \mid \mu_1 \quad \text{and} \quad (\mathcal{C} M)[x := \$ b_2] \mid \emptyset \mid \text{low} \longrightarrow^* V_2 \mid \mu_2$$

then the resulting values $V_1 = V_2$.

Proof. By Lemma 6 (compilation preserves types), $(x:\text{Bool}_{\text{high}}); \emptyset; \text{low}; \text{low} \vdash M' \Leftarrow \text{Bool}_{\text{low}}$. By Lemma 43 (noninterference for λ_{IFC}^c), $V_1 = V_2$. \square

Finally, we prove noninterference for λ_{IFC}^* as a direct corollary of Lemma 44. Noninterference says that for any high-security, sensitive user input b_1, b_2 , the low-security, publicly visible output will always be the same:

Theorem 45 (Noninterference for λ_{IFC}^*). *If M is a λ_{IFC}^* program and $\text{eval}(M, b_1) = b'_1$ and $\text{eval}(M, b_2) = b'_2$, then $b'_1 = b'_2$.*

Proof. By Definition 1 (whole programs of λ_{IFC}^*), $(x:\text{Bool}_{\text{high}}); \text{low} \vdash M : \text{Bool}_{\text{low}}$. By the definition of *eval* and the canonical form of a value of Bool_{low} , $(\mathcal{C} M)[x := \$ b_1] \mid \emptyset \mid \text{low} \longrightarrow^* \$ b'_1 \mid \mu_1$ and $(\mathcal{C} M)[x := \$ b_2] \mid \emptyset \mid \text{low} \longrightarrow^* \$ b'_2 \mid \mu_2$. Applying Lemma 44, $b'_1 = b'_2$. □

CHAPTER 8

FUTURE WORK AND CONCLUSION

I explore future directions of λ_{IFC}^* and gradual IFC by presenting four research questions in Section 8.1. After that, I summarize the entire dissertation in Section 8.2.

8.1 Future Work

I plan to augment λ_{IFC}^* with numbers, strings, tuples, vectors, and recursive types. I also plan to support arbitrary security label lattices. Let us call the extended language λ_{IFC}^*+ . There are four main research questions about the theory and practice of $\lambda_{\text{IFC}}^*/\lambda_{\text{IFC}}^*+$ that I would like to answer:

RQ1 (theory): How to mechanize the noninterference proof for λ_{IFC}^* ? In Chapter 7 of this dissertation, I present an on-paper noninterference proof for λ_{IFC}^* by simulation with its dynamic extreme. It remains an open question how to mechanize the noninterference proof for λ_{IFC}^* .

RQ2 (theory and practice): Is it possible to make λ_{IFC}^* space efficient? The space efficiency problem in gradual typing was first recognized and solved (in theory) for the gradually-typed lambda calculus (GTLC) by Herman, Tomb, and Flanagan [78, 69] and (in practice) by Kuhlenschmidt, Almahallawi, and Siek [79]. The space efficiency problem of gradual IFC remains to be investigated.

RQ3 (practice): Is λ_{IFC}^*+ expressive? It remains an open question whether λ_{IFC}^*+ , or gradual IFC in general, is expressive enough to support real-world use cases such as blockchain, e-voting, mobile app development, and digital education systems. It is also interesting to

explore how the gradual transition between static and dynamic IFC in λ_{IFC}^*+ benefits the development and security enhancement of such applications.

RQ4 (practice): Is it possible to implement λ_{IFC}^*+ with high performance? Grift [79] is an efficient compiler for a gradually-typed language with structural types called GTLC+. However, currently there is no IFC support in Grift. It is worth investigating whether it is possible to implement λ_{IFC}^*+ in a similarly efficient manner.

For the rest of this section, I will describe the research plan and the expected outcome for each research question in its own subsection.

8.1.1 RQ1: How to mechanize the noninterference proof for λ_{IFC}^* ?

To answer RQ1, we plan to mechanize the noninterference proof for λ_{IFC}^* by extending the Agda model of λ_{IFC}^* . In prior work [73], we modeled the semantics of λ_{IFC}^* in Agda and mechanized the proofs of type safety and the gradual guarantee. We also mechanized the noninterference proof for the dynamic extreme of λ_{IFC}^* called $\lambda_{\text{IFC}}^{\text{DYN}}$. We proved on paper that λ_{IFC}^* satisfies noninterference by a simulation between λ_{IFC}^* and $\lambda_{\text{IFC}}^{\text{DYN}}$. Given the sensitive nature of information-flow control, we plan to verify that λ_{IFC}^* is indeed secure by mechanizing its noninterference proof.

Following our pen-and-paper proof, we plan to take a three-step approach. In the first step, we define a relation between terms in the λ_{IFC}^c cast calculus and terms in $\lambda_{\text{IFC}}^{\text{DYN}}$. This relation captures the semantic invariant that the value produced by a λ_{IFC}^c term is always at least as secure as the one produced by its related term in $\lambda_{\text{IFC}}^{\text{DYN}}$. We define this simulation relation as a datatype in Agda. Second, we prove a simulation lemma between λ_{IFC}^c and $\lambda_{\text{IFC}}^{\text{DYN}}$: if a λ_{IFC}^c term and a $\lambda_{\text{IFC}}^{\text{DYN}}$ term are related and the λ_{IFC}^c side takes one step, the two sides are able to step to terms that are related again. Finally in the last step, we prove the noninterference lemma of λ_{IFC}^c as a corollary of the simulation lemma, and then the noninterference theorem of λ_{IFC}^* as a corollary of both the noninterference lemma of λ_{IFC}^c

and that compilation from λ_{IFC}^* to λ_{IFC}^c preserves types.

8.1.2 RQ2: Is it possible to make λ_{IFC}^* space efficient?

Siek, Thiemann, and Wadler [80] design two coercion-based cast calculi for GTLC, $\lambda\mathbf{C}$ and $\lambda\mathbf{S}$, and establish a bisimulation between the two. $\lambda\mathbf{C}$ is not space efficient, while $\lambda\mathbf{S}$ is space efficient. The key design that enables space efficiency for $\lambda\mathbf{S}$ is that the semantics compresses coercions into a compact form using the composition operator. However, the IFC cast calculus λ_{IFC}^c does not compress security coercions, so the current semantics of λ_{IFC}^* is not space efficient.

We plan to develop a space-efficient semantics for λ_{IFC}^* , thereby presenting a positive answer to the research question. Inspired by $\lambda\mathbf{S}$, we will define a composition operator on security coercions and use it to define a space-efficient semantics. In addition, we plan to mechanize our space efficiency proof by extending the Agda model of λ_{IFC}^* . Our mechanization is expected to follow the one in our prior work about GTLC [81], by proving that the size of any cast is bounded.

8.1.3 RQ3: Is λ_{IFC}^* expressive?

We plan to show that λ_{IFC}^* is expressive by (1) implementing λ_{IFC}^* in the Grift compiler and (2) exploring software prototypes of smart contracts, mobile applications, and university education systems using this compiler. The λ_{IFC}^* language will augment λ_{IFC}^* with numbers, strings, tuples, vectors, and recursive types.

Compiler development In the first step, we plan to build a compiler for λ_{IFC}^* , namely GriftIFC, by modifying the compiler passes of Grift to incorporate security types and security coercions. First, we extend the typechecker of Grift to accept programs with security type annotations. For the cast insertion pass, we are going to implement the cast insertion algorithm in Figure 4.2 of Chapter 4 of this dissertation. The algorithm inserts security

coercions whenever there is insufficient static information to enforce IFC. After this step, the source program is translated to an intermediate representation where all casts are made explicit. For the cast lowering pass, we are going to modify Grift to expose runtime functions that implement the semantics of security coercions. We plan to use the same closure conversion and memory allocation / reclamation techniques as Grift.

Case studies In the second step, we plan to demonstrate the expressiveness of λ_{IFC}^+ via four case studies using GriftIFC. In each case study, we will experiment with multiple versions of the same program that vary in precision, to demonstrate that gradual typing is indeed able to enable programmer-controlled migration between static and dynamic IFC in real-world applications.

Smart contracts To support interactions between both users and smart contracts on the Ethereum blockchain [82], the Solidity programming language features two unique mechanisms, fallback functions and delegate calls [83]. Unfortunately, security flaws may arise from the sophisticated interaction between those two mechanisms. For example, the Parity Multisig wallet bug [84] enables the attacker to take full control of the victim’s contract by setting its ownership through carefully crafted function calls. Recently, Yao et al. [85] study the security challenges of smart contracts through the lens of IFC; they observe that in an open blockchain system, public callable functions cannot ensure the privilege of the caller statically, so dynamic checking is necessary in addition to static IFC. We are going to implement a prototype in λ_{IFC}^+ that simulates the Parity Wallet smart contract attack and demonstrate that λ_{IFC}^+ is able to protect the integrity of important state variables. We will also explore the prevention techniques against common smart contract security threats such as confused deputy attacks and re-entrancy attacks, by encoding the defense against the two types of attacks as information-flow properties in λ_{IFC}^+ following Cecchetti et al. [86] and Yao et al. [85].

E-voting We plan to reimplement part of the Civitas e-voting system [87] in λ_{IFC}^+ . This case study will demonstrate that λ_{IFC}^+ is expressive enough for existing programs that take advantage of static and dynamic IFC to enforce their security policies.

Mobile computing We plan to implement the two case studies of Lifty [88], “AirBnB” and “Instagram”. “AirBnB” models an information leak of AirBnB which displays uncensored phone numbers in message previews. We will follow Lifty and implement the information-flow policies that (1) unredacted phone numbers are only visible to message senders and system administrators (2) redacted phone numbers are visible to message recipients using λ_{IFC}^+ . “Instagram” models the information leakage of Instagram that exposes the “following” relation of private accounts through followers whose accounts are public. We plan to enforce the information-flow policy that the “following” relation between A and B is visible to C if the accounts of both A and B are visible to C.

Education We plan to implement a prototype student information management system in λ_{IFC}^+ that is compliant with FERPA. The system should protect confidential information, such as a student’s GPA, transcript, SSN, and banking information, against public disclosure. On the other hand, the student’s “directory information” such as name, enrollment status, campus, and field of study are publicly visible by default, unless restricted by the student.

8.1.4 RQ4: Is it possible to implement λ_{IFC}^+ with high performance?

We plan to demonstrate that it is possible to implement λ_{IFC}^+ with high performance. We are going to evaluate the performance of GriftIFC by (1) measuring the execution time of versions of a program that vary in type precision and (2) comparing the runtime overhead of GriftIFC against fully static IFC programming languages, such as Jif [67] and Flow Caml [89], and fully dynamic systems, such as LIO [51].

Benchmark suite and sampling To evaluate the implementation of gradually-typed languages, one needs to consider many versions of the same program obtained by adding or removing type annotations [90]. We plan to build a benchmark suite for gradual IFC using excerpts from the case study programs of Section 8.1.3. After that, we plan to extend the type sampling algorithm of Grift. The type sampling algorithm takes a statically-typed program and generates gradual versions of the original program, by inserting gradual types at the source locations where static types were originally found. The algorithm constrains the overall program’s type precision, so that the percentage of the unknown type falls in a certain range. The algorithm selects an equal number of samples for each range until the desired number of samples have been generated.

Runtime cost of gradual IFC under different type precision We will explore the runtime overhead of gradual IFC with respect to type precision. For each program in the benchmark suite, we plan to measure the execution time of configurations that vary in precision and show the result in a scatter graph, where x-axis is type precision and y-axis is program execution time.

Gradual IFC overhead compared to static and dynamic IFC To measure the overhead of gradual IFC compared with fully static IFC, we plan to compare the execution time of programs in λ_{IFC}^+ to the same programs adapted to Jif and Flow Caml. To measure the overhead compared with fully dynamic IFC, we will compare λ_{IFC}^+ to LIO. In both experiments, we plan to display the results as bar charts where x-axis is different benchmark programs in different IFC languages and y-axis is their execution time.

8.2 Conclusion

In my dissertation, I presented the design of a gradual information-flow language λ_{IFC}^* , which satisfies both noninterference and the gradual guarantee, while maintaining the principle of type-based reasoning. The key to the design of λ_{IFC}^* is to walk back the decision in GSL_{Ref} to include the unknown label \star among the runtime security labels. So λ_{IFC}^* takes a more standard approach to gradually-typed IFC: the \star label can be used in type annotations but not as the security level of a runtime value. The λ_{IFC}^* language is defined by translation to a cast calculus λ_{IFC}^c . This intermediate language employs a coercion calculus to express the implicit conversions between more-or-less precise parts of the program; the security coercion calculus is able to capture both explicit and implicit information flows. I proved that λ_{IFC}^* satisfies termination-insensitive noninterference. I also proved that λ_{IFC}^* satisfies type safety as well as the gradual guarantee and mechanized those results in Agda.

In summary, my dissertation shows that it is possible to design a gradual IFC programming language that satisfies noninterference and the gradual guarantee while supporting type-based reasoning, by excluding the unknown label from run-time security labels and using security coercions to represent casts.

REFERENCES

- [1] Carole Cadwalladr. “Facebook suspends data firm hired by Vote Leave over alleged Cambridge Analytica ties”. In: *The Guardian* (2018).
- [2] S Kitchgaessner. “Cambridge Analytica used data from Facebook and Politico to help Trump”. In: *The Guardian* 26 (2017).
- [3] Felipe González et al. “Global reactions to the cambridge analytica scandal: A cross-language social media study”. In: *Companion Proceedings of the 2019 world wide web conference*. 2019, pp. 799–806.
- [4] Joanne Hinds, Emma J Williams, and Adam N Joinson. ““It wouldn’t happen to me”: Privacy concerns and perspectives following the Cambridge Analytica scandal”. In: *International Journal of Human-Computer Studies* 143 (2020), p. 102498.
- [5] Adil Hussain Seh et al. “Healthcare data breaches: insights and implications”. In: *Healthcare*. Vol. 8. 2. MDPI. 2020, p. 133.
- [6] Jürgen Pfeffer, Katja Mayer, and Fred Morstatter. “Tampering with Twitter’s sample API”. In: *EPJ Data Science* 7.1 (2018), p. 50.
- [7] Masarah Paquet-Clouston, Olivier Bilodeau, and David Décary-Héту. “Can we trust social media data? social network manipulation by an iot botnet”. In: *Proceedings of the 8th international conference on social media & society*. 2017, pp. 1–9.
- [8] Yelena Smirnova and Victoriano Travieso-Morales. “Understanding challenges of GDPR implementation in business enterprises: a systematic literature review”. In: *International Journal of Law and Management* (2024).

- [9] Sean Sirur, Jason RC Nurse, and Helena Webb. “Are we there yet? Understanding the challenges faced in complying with the General Data Protection Regulation (GDPR)”. In: *Proceedings of the 2nd International Workshop on Multimedia Privacy and Security*. 2018, pp. 88–95.
- [10] M da C Freitas and Miguel Mira da Silva. “GDPR Compliance in SMEs: There is much to be done”. In: *Journal of Information Systems Engineering & Management* 3.4 (2018), p. 30.
- [11] Ralf Christian Härting et al. “Impacts of the New General Data Protection Regulation for small-and medium-sized enterprises”. In: *Proceedings of Fifth International Congress on Information and Communication Technology: ICICT 2020, London, Volume 1*. Springer. 2021, pp. 238–246.
- [12] Andrei Sabelfeld and Andrew C Myers. “Language-based information-flow security”. In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19.
- [13] Kenneth J Biba et al. “Integrity considerations for secure computer systems”. In: (1977).
- [14] Hala Assal and Sonia Chiasson. “Security in the software development lifecycle”. In: *Fourteenth symposium on usable privacy and security (SOUPS 2018)*. 2018, pp. 281–296.
- [15] Anuradha Sharma and Praveen Kumar Misra. “Aspects of enhancing security in software development life cycle”. In: *Advances in Computational Sciences and Technology* 10.2 (2017), pp. 203–210.
- [16] Curtis Steward Jr et al. “Software security: The dangerous afterthought”. In: *2012 Ninth International Conference on Information Technology-New Generations*. IEEE. 2012, pp. 815–818.
- [17] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. “A sound type system for secure flow analysis”. In: *Journal of computer security* 4.2-3 (1996), pp. 167–187.

- [18] Andrew C. Myers and Barbara Liskov. “A Decentralized Model for Information Flow Control”. In: *SIGOPS Oper. Syst. Rev.* 31.5 (1997), pp. 129–142.
- [19] Andrew C Myers. “JFlow: Practical mostly-static information flow control”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 228–241.
- [20] Aslan Askarov and Andrei Sabelfeld. “Tight Enforcement of Information-Release Policies for Dynamic Languages”. In: *2009 22nd IEEE Computer Security Foundations Symposium*. 2009, pp. 43–59.
- [21] Thomas H Austin and Cormac Flanagan. “Efficient purely-dynamic information flow analysis”. In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. 2009, pp. 113–124.
- [22] Dominique Devriese and Frank Piessens. “Noninterference through Secure Multi-execution”. In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 109–124.
- [23] Deian Stefan et al. “Flexible dynamic information flow control in Haskell”. In: *Proceedings of the 4th ACM symposium on Haskell*. 2011, pp. 95–106.
- [24] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. “Multiple Facets for Dynamic Information Flow with Exceptions”. In: *ACM Trans. Program. Lang. Syst.* 39.3 (2017).
- [25] Jian Xiang and Stephen Chong. “Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages”. In: *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. Piscataway, NJ, USA: IEEE Press, May 2021.
- [26] Gurvan Le Guernic and Thomas Jensen. “Monitoring information flow”. In: *Proc. Workshop on Foundations of Computer Security*. 2005, pp. 19–30.
- [27] Gurvan Le Guernic. “Automaton-based confidentiality monitoring of concurrent programs”. In: *20th IEEE Computer Security Foundations Symposium (CSF’07)*. IEEE. 2007, pp. 218–232.

- [28] Deepak Chandra and Michael Franz. “Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine”. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007, pp. 463–475.
- [29] Paritosh Shroff, Scott Smith, and Mark Thober. “Dynamic Dependency Monitoring to Secure Information Flow”. In: *20th IEEE Computer Security Foundations Symposium (CSF’07)*. 2007, pp. 203–217.
- [30] Alejandro Russo and Andrei Sabelfeld. “Dynamic vs. static flow-sensitive security analysis”. In: *2010 23rd IEEE Computer Security Foundations Symposium*. IEEE. 2010, pp. 186–199.
- [31] Scott Moore and Stephen Chong. “Static analysis for efficient hybrid information-flow control”. In: *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE. 2011, pp. 146–160.
- [32] Joseph A Goguen and José Meseguer. “Security policies and security models”. In: *1982 IEEE Symposium on Security and Privacy*. IEEE. 1982, pp. 11–11.
- [33] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. “HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 289–301. ISBN: 9781450336697.
- [34] D Elliott Bell and Leonard J La Padula. *Secure computer system: Unified exposition and multics interpretation*. Tech. rep. MITRE CORP BEDFORD MA, 1976.
- [35] Martin Lester, Luke Ong, and Max Schäfer. “Information flow analysis for a dynamically typed language with staged metaprogramming”. In: *Journal of Computer Security* 24.5 (2016), pp. 541–582.
- [36] Dorothy E Denning. “A lattice model of secure information flow”. In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

- [37] Dorothy E Denning and Peter J Denning. “Certification of programs for secure information flow”. In: *Communications of the ACM* 20.7 (1977), pp. 504–513.
- [38] Stephan Arthur Zdancewic. “Programming languages for information security”. PhD thesis. 2002.
- [39] Nevin Heintze and Jon G Riecke. “The SLam calculus: programming with secrecy and integrity”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1998, pp. 365–377.
- [40] Gilles Barthe and Tamara Rezk. “Non-interference for a JVM-like language”. In: *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. 2005, pp. 103–112.
- [41] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. “A logic for information flow in object-oriented programs”. In: *ACM SIGPLAN Notices* 41.1 (2006), pp. 91–102.
- [42] Aaron Bohannon et al. “Reactive noninterference”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. 2009, pp. 79–90.
- [43] François Pottier and Vincent Simonet. “Information flow inference for ML”. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2002, pp. 319–330.
- [44] Vincent Simonet and Inria Rocquencourt. “Flow Caml in a nutshell”. In: *Proceedings of the first APPSEM-II workshop*. 2003, pp. 152–165.
- [45] Jeffrey S Fenton. “Memoryless subsystems”. In: *The Computer Journal* 17.2 (1974), pp. 143–147.
- [46] Peng Li and Steve Zdancewic. “Encoding information flow in Haskell”. In: *19th IEEE Computer Security Foundations Workshop (CSFW’06)*. IEEE. 2006, 12–pp.

- [47] Peng Li and Steve Zdancewic. “Arrows for secure information flow”. In: *Theoretical Computer Science* 411.19 (2010). Mathematical Foundations of Programming Semantics (MFPS 2006), pp. 1974–1994.
- [48] Thomas H Austin and Cormac Flanagan. “Permissive dynamic information flow analysis”. In: *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. 2010, pp. 1–12.
- [49] Thomas H Austin and Cormac Flanagan. “Multiple facets for dynamic information flow”. In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2012, pp. 165–178.
- [50] Deian Stefan et al. “Flexible dynamic information flow control in the presence of exceptions”. In: *arXiv preprint arXiv:1207.1457* (2012).
- [51] Deian Stefan et al. “Flexible dynamic information flow control in the presence of exceptions”. In: *Journal of Functional Programming* 27 (2017).
- [52] Petros Efstathopoulos et al. “Labels and event processes in the Asbestos operating system”. In: *ACM SIGOPS Operating Systems Review* 39.5 (2005), pp. 17–30.
- [53] Nikolai Zeldovich et al. “Making information flow explicit in HiStar”. In: *Communications of the ACM* 54.11 (2011), pp. 93–101.
- [54] Maxwell Krohn et al. “Information flow control for standard OS abstractions”. In: *ACM SIGOPS Operating Systems Review* 41.6 (2007), pp. 321–334.
- [55] Steve Vandeboogart et al. “Labels and event processes in the Asbestos operating system”. In: *ACM Transactions on Computer Systems (TOCS)* 25.4 (2007), 11–es.
- [56] Jeremy G. Siek and Walid Taha. “Gradual typing for functional languages”. In: *Scheme and Functional Programming Workshop*. 2006, pp. 81–92.
- [57] Jeremy G. Siek and Walid Taha. “Gradual Typing for Objects”. In: *European Conference on Object-Oriented Programming*. Vol. 4609. LCNS. 2007, pp. 2–27.

- [58] Tim Disney and Cormac Flanagan. “Gradual Information Flow Typing”. In: *Workshop on Script to Program Evolution*. 2011.
- [59] L. Fennell and P. Thiemann. “Gradual Security Typing with References”. In: *2013 IEEE 26th Computer Security Foundations Symposium*. 2013, pp. 224–239.
- [60] Luminous Fennell and Peter Thiemann. “LJGS: Gradual Security Types for Object-Oriented Languages”. In: *Workshop on Foundations of Computer Security*. FCS. 2015.
- [61] Jeremy G. Siek et al. “Refined Criteria for Gradual Typing”. In: *SNAPL: Summit on Advances in Programming Languages*. LIPIcs: Leibniz International Proceedings in Informatics. 2015.
- [62] Matías Toro, Ronald Garcia, and Éric Tanter. “Type-Driven Gradual Security with References”. In: *ACM Trans. Program. Lang. Syst.* 40.4 (Dec. 2018), 16:1–16:55.
- [63] Ronald Garcia, Alison M. Clark, and Éric Tanter. “Abstracting Gradual Typing”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 429–442. ISBN: 978-1-4503-3549-2.
- [64] Arthur Azevedo de Amorim, Matt Fredrikson, and Limin Jia. “Reconciling noninterference and gradual typing”. In: *Logic in Computer Science*. LICS. 2020.
- [65] Tianyu Chen and Jeremy G. Siek. *Mechanized Noninterference for Gradual Security*. 2022. arXiv: [2211.15745](https://arxiv.org/abs/2211.15745) [cs.PL].
- [66] Abhishek Bichhawat, McKenna McCall, and Limin Jia. “Gradual Security Types and Gradual Guarantees”. In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE. 2021, pp. 1–16.
- [67] Andrew C. Myers et al. *Jif 3.0: Java information flow*. 2006.

- [68] Fritz Henglein. “Dynamic typing: syntax and proof theory”. In: *Science of Computer Programming* 22.3 (1994), pp. 197–230.
- [69] David Herman, Aaron Tomb, and Cormac Flanagan. “Space-efficient gradual typing”. In: *Higher-Order and Symbolic Computation* 23.2 (2010), pp. 167–189.
- [70] Robert Bruce Findler and Matthias Felleisen. *Contracts for Higher-Order Functions*. Tech. rep. NU-CCS-02-05. Northeastern University, 2002.
- [71] Sam Tobin-Hochstadt and Matthias Felleisen. “The Design and Implementation of Typed Scheme”. In: *Symposium on Principles of Programming Languages*. 2008.
- [72] Preston Tunnell Wilson et al. “The Behavior of Gradual Types: A User Study”. In: *Dynamic Languages Symposium*. 2018.
- [73] Tianyu Chen and Jeremy G Siek. “Quest Complete: The Holy Grail of Gradual Security”. In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024), pp. 1609–1632.
- [74] Philip Wadler and Robert Bruce Findler. “Well-typed programs can’t be blamed”. In: *European Symposium on Programming*. ESOP. 2009, pp. 1–16.
- [75] Philip Wadler. “Theorems for free!” In: *FPCA ’89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*. Imperial College, London, United Kingdom: ACM, 1989, pp. 347–359. ISBN: 0-89791-328-0.
- [76] Michael Greenberg. “Space-Efficient Manifest Contracts”. In: *CoRR* abs/1410.2813 (2014).
- [77] Thomas Streicher. *Domain-theoretic foundations of functional programming*. World Scientific Publishing Co., Inc., 2006.
- [78] David Herman, Aaron Tomb, and Cormac Flanagan. “Space-Efficient Gradual Typing”. In: *Trends in Functional Prog. (TFP)*. 2007, p. XXVIII.

- [79] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. “Toward Efficient Gradual Typing for Structural Types via Coercions”. In: *Conference on Programming Language Design and Implementation*. PLDI. ACM, 2019.
- [80] Jeremy G. Siek, Peter Thiemann, and Philip Wadler. “Blame and coercion: Together again for the first time”. In: *Conference on Programming Language Design and Implementation*. PLDI. 2015.
- [81] Jeremy G. Siek and Tianyu Chen. “Parameterized cast calculi and reusable meta-theory for gradually typed lambda calculi”. In: *Journal of Functional Programming* 31 (2021), e30.
- [82] Gavin Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper 151.2014* (2014), pp. 1–32.
- [83] Chris Dannen and Chris Dannen. “Solidity programming”. In: *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners* (2017), pp. 69–88.
- [84] Lorenz Breidenbach et al. *An In-Depth Look at the Parity Multisig Bug*. July 2017.
- [85] Siqiu Yao et al. “SCIF: A Language for Compositional Smart Contract Security”. In: *arXiv preprint arXiv:2407.01204* (2024).
- [86] Ethan Cecchetti et al. “Compositional security for reentrant applications”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1249–1267.
- [87] Michael R Clarkson, Stephen Chong, and Andrew C Myers. “Civitas: Toward a secure voting system”. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 354–368.
- [88] Nadia Polikarpova et al. “Liquid information flow control”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2020), pp. 1–30.

- [89] Vincent Simonet. *The Flow Caml System: documentation and user's manual*. Technical Report 0282. ©INRIA. Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.
- [90] Asumu Takikawa et al. “Is Sound Gradual Typing Dead?” In: *Principles of Programming Languages*. POPL. ACM, 2016.

VITA

Tianyu Chen

CONTACT INFORMATION

3025B. Luddy Hall, Indiana University
700 N. Woodlawn Ave.
Bloomington, IN 47408

E-mail: chen512@iu.edu

Website: homes.luddy.indiana.edu/chen512

EDUCATION

- *Ph.D.*, Computer Science, Indiana University **August 2016 – May 2025**
 - Advisor: Professor Jeremy G. Siek
- *B.Eng.*, Tsinghua University **August 2012 – July 2016**
 - Major: Computer Science and Technology

EXPERIENCE

Indiana University, Bloomington, Indiana USA

Instructor-of-Record (IOR)

August 2024 – Present

Research Assistant and Associate Instructor

August 2016 – July 2024

CertiK, Remote, New York USA

Software Verification Intern on Coq and CompCert

May 2021 – August 2021

Columbia University, New York City, New York USA

Visiting Student on Transparent Paxos

August 2015 – October 2015

PUBLICATIONS

Quest Complete: the Holy Grail of Gradual Security

Tianyu Chen, Jeremy G. Siek.

In Proceedings of the 45th ACM SIGPLAN Conference on Programming Language Design and Implementation (**PLDI 2024**).

Parameterized Cast Calculi and Reusable Meta-theory for Gradually Typed Lambda Calculi

Jeremy G. Siek, Tianyu Chen.

In Journal of Functional Programming (**JFP**), November 2021.

Mechanized Type Safety for Gradual Information Flow

Tianyu Chen, Jeremy G. Siek.

In the 7th Workshop on Language-Theoretic Security (**LangSec 2021**).

Racing in Hyperspace: Closing Hyper-threading Side Channels on SGX with Contrived Data Races

Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, Dongdai Lin.

In Proceedings of 2018 IEEE Symposium on Security and Privacy (**Oakland 2018**).

Characterizing Smartwatch Usage in the Wild

Xing Liu, Tianyu Chen, Feng Qian, Zhixiu Guo, Felix Xiaozhu Lin, Xiaofeng Wang, Kai Chen.

In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (**MobiSys 2017**).

Paxos Made Transparent

Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, Junfeng Yang.

In Proceedings of the 25th Symposium on Operating Systems Principles (**SOSP 2015**).

DRAFTS AND POSTERS

Mechanized Noninterference for Gradual Security

Tianyu Chen, Jeremy G. Siek.

Draft, November 2022.

Generic Blame-Subtyping Theorem in Agda Using Abstract Binding Trees

Tianyu Chen.

In POPL 2022 Student Research Competition (**POPL SRC 2022**).

TEACHING

System Programming With C and Unix (CSCI-C291 and ENGR-E111)

Instructor

Spring 2025, Fall 2024

Data Structures (CSCI-C343, 200 students)

Lead Associate Instructor

Spring 2024

Data Structures, Honors (CSCI-H343)

Associate Instructor

Fall 2023

Secure Protocols (CSCI-B433 and INFO-I433)

Associate Instructor

Spring 2020, Spring 2019

Malware: Threat and Defense (CSCI-B546 and INFO-I521)

Associate Instructor

Fall 2019

AWARDS

Luddy PhD Instructor Award

2024 – 2025

Research Assistant of the Year (Computer Science)

2023 – 2024

SERVICE

PL Wonks

Organizer, Video Chair

Fall 2023 – Present

The PL Reading Group (PLRG)

Organizer

Fall 2021 – Spring 2023

SOFTWARE

Mechanized Gradual Information-Flow Control

December 2023

- Designed two gradual information-flow control programming languages, with and without type-guided classification.
- Mechanized the proofs of type safety, noninterference, and the gradual guarantee in Agda.

Gradual Typing in Agda

August 2020

- Contributed to the mechanized compendium of the gradually-typed λ -calculus (GTLC) and a variety of cast calculi.

Mechanization of GLIO

May 2020

- Implemented a definitional interpreter for GLIO, an experimental gradual security programming language proposed by researchers from Carnegie Mellon University.
- Proved type safety for GLIO and mechanized the proof in Agda.

PROGRAMMING SKILLS

Languages: C, Java, Python, Racket, Agda, Coq

Tools: Emacs, Shell scripting, Unix sysadmin, virtualization, Linux containers