DEPARTMENT OF COMPUTER SCIENCE

Formal Proofs of Non-Interference for Modal Calculi

[redacted]

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering.

Wednesday 3$^{\text{rd}}$ May, 2023

Abstract

We prove an important correctness property (known as non-interference) for a particular type of modal lambda calculi that is of recent interest. This has been done via a recent technique based on bisimulation that is remarkably simpler than older techniques, allowing us to formalize our results in the proof assistant Agda.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

[redacted], Wednesday 3rd May, 2023

Contents

List of Figures

List of Tables

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Dr. Alex Kavvos.

Chapter 1

Introduction

A lambda calculus enhanced with a unary modal type constructor (often denoted as T, or here, $\Box$) is known as a modal lambda calculus. Intuitively, the type $\Box A$ represents a restricted version of the type A (with those restrictions depending on the exact calculus). In particular, we focus on *Fitch-style* modal calculi (popularised by [4] in 2018). This introduces two new term-formers **box** and **unbox**. The former is the introduction rule for modal types; taking a term of type A to produce a term of type $\Box A$, but eliminating a *lock* (denoted 🔒) from the context — meaning the context is locked "inside" the box. These locks place a heavy restriction on the context, as the variable rule is prevented from accessing the context left of locks. Essentially, this means that terms in locked contexts are closed with respect to variables "protected" by locks, and so terms inside a **box** constructor cannot use variables bound outside the constructor. Dually, the **unbox** constructor eliminates $\Box A$ to produce A, but in turn introduces a lock into the context.

As a practical example, we see the term $\vdash \boldsymbol{\lambda} x.\, \mathbf{box}\ x : A \to \Box A$ cannot be typed (see Fig 1.1), but the modal axiom $\vdash \boldsymbol{\lambda} f.\, \boldsymbol{\lambda} x.\, \mathbf{box}\ (\mathbf{unbox}\ f)(\mathbf{unbox}\ x) : \Box(A \to B) \to \Box A \to \Box B$ is fine (see Fig 1.2). This corresponds nicely to our intuition (developed below) that the first term would "leak" information.

Such calculi can be understood (under the Curry-Howard tradition) to correspond to certain modal logics. However, we often want to use such a system for practical purposes. Modal calculi have seen a wide array of uses, from older uses in managing effects in programming languages [14], to more recent innovations in resource management [5] and functional reactive programming [12].

One promising usage is for modelling distributed ("cloud") computing in the lambda calculus [15]. We do not develop this idea here, but simply point this out to demonstrate the importance of correctness, and the secure flow of information, in modal calculi. Such information flow properties have been well studied, starting from the early days of the field [13], up to the current day.

Non-interference properties originate from the field of security [9] and in particular security calculi [16]. These properties control information flow within a system. Recently, it has been pointed out by [11] that non-interference can be applied to modal calculi, as a powerful approach to proving language-based properties about information flow.

We develop this notion for the Fitch-style modal lambda calculi and formalise this result in the proof assistant Agda (giving much-needed reassurance that these import security properties do in fact hold) using a fairly recent technique based on bisimulations (developed by [3]) that is radically simpler than the original approach. We also discuss the usage and restrictions of this technique for other calculi.

Figure 1.1: An invalid typing derivation.

$$\dfrac{\dfrac{\dfrac{?}{x : A, \text{🔒} \vdash x : A}\ (???)}{x : A \vdash \mathbf{box}\ x : \Box A}\ (\textsc{TBox})}{\vdash \boldsymbol{\lambda} x.\, \mathbf{box}\ x : A \to \Box A}\ (\textsc{TLam})$$

Note that ($\textsc{TVar}$) cannot apply because of the presence of the lock.

Figure 1.2: A valid typing derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \Box(A \to B), x : \Box A \vdash f : \Box(A \to B)}\,(\textsc{TVar})}{f : \Box(A \to B), x : \Box A, 🔒 \vdash \mathbf{unbox}\ f : A \to B}\,(\textsc{TUnbox}) \quad \dfrac{\dfrac{}{f : \Box(A \to B), x : \Box A \vdash x : \Box A}\,(\textsc{TVar})}{f : \Box(A \to B), x : \Box A, 🔒 \vdash \mathbf{unbox}\ x : A}\,(\textsc{TUnbox})}{f : \Box(A \to B), x : \Box A, 🔒 \vdash (\mathbf{unbox}\ f)(\mathbf{unbox}\ x) : B}\,(\textsc{TApp})}{\dfrac{f : \Box(A \to B), x : \Box A \vdash \mathbf{box}\ (\mathbf{unbox}\ f)(\mathbf{unbox}\ x) : \Box B}{\vdash \boldsymbol{\lambda} f.\, \boldsymbol{\lambda} x.\, \mathbf{box}\ (\mathbf{unbox}\ f)(\mathbf{unbox}\ x) : \Box(A \to B) \to \Box A \to \Box B}\,\text{2x}\ (\textsc{TLam})}\,(\textsc{TBox})$$

1.1 Outline

We begin with a short background section introducing the specific calculi used. We will then provide a proof of the main non-interference result, before detailing the formalisation approach and result.

Chapter 2

Background

2.1 A modal lambda calculus

As an example throughout this document, we use the following lambda calculus in the style of [4], extended for the purposes of demonstration with a base type (`Nat`) and a product type construction. We list here its typing system (in figure 2.2) and transition system (in figure 2.3). We call such a calculus *Fitch-style* after [8]. In this setting, working "inside" a modal term or *box* (or, monad, etc.) requires **unbox**ing it (eliminating $\Box A$ to get A), locking the context until we **box** it again (transforming our conclusion B into a modal one $\Box B$). This is opposed to the, perhaps more familiar, monad-like **bind** and **return** based paradigm.

2.2 Non-interference

Non-interference is an important security property for modal (or similar, e.g. [1, 14]) lambda calculi, which essentially states that observers from a lower level (in our case, outside of boxes) cannot tell the difference between values of a higher level (inside boxes). We prove this using the syntactic approach pioneered by [3], in which we introduce an *indistinguishability* relation (introduced as $\sim$ in figure 2.4) and prove that it is a (bi)simulation. This relation captures the intuitive notion of two terms being identical from an "unprivileged" point of view (i.e. without the ability to look into the contents of boxed subterms), and by proving it is maintained under reduction, we show non-interference. Looking inside a box introduces a lock into the context (as in the (SimBox) rule), and all terms under a lock are considered indistinguishable by (SimLock).

Figure 2.1: The shape of valid types and contexts.

types	A, B	$::=$	Nat	base type
			$A \times B$	product type
			$A \to B$	function type
			$\Box A$	modal type
contexts	Γ	$::=$	$\cdot$	empty context
			$\Gamma, x : A$	extended context
			$\Gamma,$ 🔒	locked context

Figure 2.2: Typing rules for our calculus, with a base type and products.

Core typing rules.

$$\textsc{TVar}\quad \frac{🔒 \notin \Gamma_2}{\Gamma_1, x : A, \Gamma_2 \vdash x : A} \qquad \textsc{TLam}\quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \boldsymbol{\lambda} x.\, t : A \to B} \qquad \textsc{TApp}\quad \frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \qquad \textsc{TBox}\quad \frac{\Gamma, 🔒 \vdash t : A}{\Gamma \vdash \textbf{box}\ t : \Box A}$$

$$\textsc{TUnbox}\quad \frac{\Gamma_1 \vdash t : \Box A}{\Gamma_1, 🔒, \Gamma_2 \vdash \textbf{unbox}\ t : A}$$

Additional typing rules.

$$\textsc{TNum}\quad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathrm{Nat}} \qquad \textsc{TProd}\quad \frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B} \qquad \textsc{TProj1}\quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \boldsymbol{\pi}_1(t) : A} \qquad \textsc{TProj2}\quad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \boldsymbol{\pi}_2(t) : B}$$

Figure 2.3: Small-step dynamics for our calculus.

Core dynamics.

$$\textsc{DBeta}\quad \frac{}{(\boldsymbol{\lambda} x.\, t_1) t_2 \longmapsto t_1[t_2/x]} \qquad \textsc{DBoxUnbox}\quad \frac{}{\textbf{unbox}\ (\textbf{box}\ t) \longmapsto t} \qquad \textsc{DAppL}\quad \frac{t_1 \longmapsto t_1'}{t_1 t_2 \longmapsto t_1' t_2} \qquad \textsc{DAppR}\quad \frac{t_2 \longmapsto t_2'}{t_1 t_2 \longmapsto t_1 t_2'}$$

$$\textsc{DUnbox}\quad \frac{t \longmapsto t'}{\textbf{unbox}\ t \longmapsto \textbf{unbox}\ t'}$$

Additional dynamics.

$$\textsc{DProj1}\quad \frac{}{\boldsymbol{\pi}_1(\langle t_1, t_2 \rangle) \longmapsto t_1} \qquad \textsc{DProj2}\quad \frac{}{\boldsymbol{\pi}_1(\langle t_1, t_2 \rangle) \longmapsto t_2} \qquad \textsc{DProd1}\quad \frac{t_1 \longmapsto t_1'}{\langle t_1, t_2 \rangle \longmapsto \langle t_1', t_2 \rangle} \qquad \textsc{DProd2}\quad \frac{t_2 \longmapsto t_2'}{\langle t_1, t_2 \rangle \longmapsto \langle t_1, t_2' \rangle}$$

Figure 2.4: Indistinguishability relation $\sim$ over terms of our calculus.

Core simulation rules.

$$\textsc{SimLock}\ \frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : A \qquad \text{🔒} \in \Gamma}{\Gamma \vdash t_1 \sim t_2 : A} \qquad \textsc{SimVar}\ \frac{\text{🔒} \notin \Gamma_2}{\Gamma_1, x : A, \Gamma_2 \vdash x \sim x : A}$$

$$\textsc{SimApp}\ \frac{\Gamma \vdash t_1 \sim t_1' : A \to B \qquad \Gamma \vdash t_2 \sim t_2' : A}{\Gamma \vdash t_1 t_2 \sim t_1' t_2' : B} \qquad \textsc{SimLam}\ \frac{\Gamma, x : A \vdash t \sim t' : B}{\Gamma \vdash \boldsymbol{\lambda} x.\, t \sim \boldsymbol{\lambda} x.\, t' : A \to B} \qquad \textsc{SimBox}\ \frac{\Gamma, \text{🔒} \vdash t \sim t' : A}{\Gamma \vdash \mathbf{box}\ t \sim \mathbf{box}\ t' : \Box A}$$

Additional simulation rules.

$$\textsc{SimNum}\ \frac{n \in \mathbb{N}}{\Gamma \vdash n \sim n : \mathrm{Nat}} \qquad \textsc{SimProd}\ \frac{\Gamma \vdash t_1 \sim t_1' : A \qquad \Gamma \vdash t_2 \sim t_2' : B}{\Gamma \vdash \langle t_1, t_2 \rangle \sim \langle t_1', t_2' \rangle : A \times B} \qquad \textsc{SimProj1}\ \frac{\Gamma \vdash t \sim t' : A \times B}{\Gamma \vdash \boldsymbol{\pi}_1(t) \sim \boldsymbol{\pi}_1(t') : A}$$

$$\textsc{SimProj2}\ \frac{\Gamma \vdash t \sim t' : A \times B}{\Gamma \vdash \boldsymbol{\pi}_2(t) \sim \boldsymbol{\pi}_2(t') : B}$$

Chapter 3

A proof of non-interference

We will first need the following lemmas.

Lemma 1. *(Substitution lemma) If $\Gamma, x : B \vdash t_1 : A$ and $\Gamma \vdash t_2 : B$ then $\Gamma \vdash t_1[t_2/x] : A$.*

Lemma 2. *(Variable weakening) If $\Gamma_1, \Gamma_2 \vdash t : A$ then $\Gamma_1, x : B, \Gamma_2 \vdash t : A$.*

Proof. A proof of both of these can be found in [4]. □

Lemma 3. *(Simulation implies typing) If $\Gamma \vdash t_1 \sim t_2 : A$, then $\Gamma \vdash t_1 : A$ and $\Gamma \vdash t_2 : A$.*

Proof. By induction on $\Gamma \vdash t_1 \sim t_2 : A$. The base cases (SimVar) and (SimBox) both guarantee $\Gamma \vdash t_1, t_2 : A$, the other cases require a simple application of the induction hypothesis. □

3.1 Indistinguishability under substitution

To finish our proof of non-interference (in particular, for the (DBeta) case), we require the following lemma.

Theorem 1 (Indistinguishability under substitution)**.** *If $\Gamma, x : B \vdash t_1 \sim t_2 : A$ and $\Gamma \vdash a_1 \sim a_2 : B$, then $\Gamma \vdash t_1[a_1/x] \sim t_2[a_2/x] : A$.*

Proof. By induction on the derivation of $\Gamma, x : B \vdash t_1 \sim t_2 : A$. We proceed by cases.

Case(SimVar). If $\Gamma, x : B \vdash t_1 \sim t_2 : A$ via the (SimVar) rule, then we obtain $\Gamma, x : B \equiv \Gamma_1, y : B, \Gamma_2$ (where $\text{🔒} \notin \Gamma_2$), $A \equiv B$, $t_1 \equiv y$ and $t_2 \equiv y$.

If $x \neq y$ we see that $t_1[a_1/x] = t_1$ (and similarly for t_2, a_2), and the required proof is obtained by strengthening the first premise (noting that x does not occur freely in y). On the other hand, if $x = y$ then $t_1[a_1/x] = a_1$ (similarly, $t_2[a_2/x] = a_2$), and the required proof is instead the second premise.

Case(SimLock). If $\Gamma, x : B \vdash t_1 \sim t_2 : A$ via the (SimLock) rule then we see that $\text{🔒} \in \Gamma$ and obtain proofs of $\Gamma \vdash t_1 : A$, $\Gamma \vdash t_2 : A$. Because substitution preserves typing (Lemma 1), we can then provide the following proof.

$$\dfrac{\Gamma \vdash t_1[a_1/x] : A \qquad \Gamma \vdash t_2[a_2/x] : A \qquad \text{🔒} \in \Gamma}{\Gamma \vdash t_1[a_1/x] \sim t_2[a_2/x] : A}\ (\textsc{SimLock})$$

Case(SimBox). If $\Gamma, x : B \vdash t_1 \sim t_2 : A$ via the (SimBox) rule, we obtain $t_1 \equiv \textbf{box}\ b_1$, $t_2 \equiv \textbf{box}\ b_2$, $A \equiv \Box\alpha$, and a proof of $\Gamma, x : B, \text{🔒} \vdash b_1 \sim b_2 : \alpha$. We see that $t_1[a_1/x] \equiv \textbf{box}\ b_1[a_1/x]$ and $t_2[a_2/x] \equiv \textbf{box}\ b_2[a_2/x]$, and can immediately provide the following proof tree (noting again that simulation implies typing, and substitution preserves typing, to provide the first two hypotheses).

$$\dfrac{\dfrac{\Gamma, \text{🔒} \vdash t_1[a_1/x] : \alpha \qquad \Gamma, \text{🔒} \vdash t_2[a_2/x] : \alpha \qquad \text{🔒} \in \Gamma, \text{🔒}}{\Gamma, \text{🔒} \vdash b_1[a_1/x] \sim b_2[a_2/x] : \alpha}\ (\textsc{SimLock})}{\Gamma \vdash \underbrace{\textbf{box}\ b_1[a_1/x]}_{t_1[a_1/x]} \sim \underbrace{\textbf{box}\ b_2[a_2/x]}_{t_2[a_2/x]} : \underbrace{\Box\alpha}_{A}}\ (\textsc{SimApp})$$

Case(SimApp). If $\Gamma, x : B \vdash t_1 \sim t_2 : A$ via the (SimApp) rule, we obtain $t_1 \equiv t_1' t_1''$, $t_2 \equiv t_2' t_2''$, $A \equiv T \to U$ and proofs of (1) $\Gamma, x : B \vdash t_1' \sim t_2' : T \to U$ and (2) $\Gamma, x : B \vdash t_1'' \sim t_2'' : T$.

Now, $t_1[a_1/x] = t_1'[a_1/x]t_1''[a_1/x]$ and $t_2[a_2/x] = t_2'[a_2/x]t_2''[a_2/x]$. We apply the induction hypothesis on the statements (1) and (2) (along with the original second premise) to obtain proofs of (3) $\Gamma \vdash t_1'[a_1/x] \sim t_2'[a_2/x] : T \to U$ and (4) $\Gamma \vdash t_1''[a_1/x] \sim t_2''[a_2/x] : T$, then provide the following proof tree.

$$\dfrac{\dfrac{(3)}{\Gamma \vdash t_1'[a_1/x] \sim t_2'[a_2/x] : T \to U} \qquad \dfrac{(4)}{\Gamma \vdash t_1''[a_1/x] \sim t_2''[a_2/x] : T}}{\Gamma \vdash t_1'[a_1/x]t_1''[a_1/x] \sim t_2'[a_2/x]t_2''[a_2/x] : T \to U}\ (\textsc{SimApp})$$

Cases(SimLam, SimProd, SimProj1, SimProj2) are all similar or even simpler congruence cases, consisting of similar applications of the inductive hypothesis.

□

3.2 Non-interference

With this we are now well-equipped to prove the main result of this chapter. Note that we require Γ to be lock-free. This is required to prevent the relation of a value and a non-value via the (SimLock) rule, which would break non-interference.

Theorem 2 ($\sim$ is a bisimulation)**.** *If $\Gamma \vdash t_1 \sim t_2 : A$, 🔒 $\notin \Gamma$, and $t_1 \longmapsto t_1'$ then there is a t_2' such that $t_2 \longmapsto t_2'$ and $\Gamma \vdash t_1' \sim t_2' : A$.*

Proof. By induction on the transition relation $t_1 \longmapsto t_1'$. We again proceed by cases.

Case(DBoxUnbox). For $t_1 \longmapsto t_1'$ to hold via the rule (DBoxUnbox) we require $t_1 \equiv$ **unbox** (**box** b_1). By simple inversion on the evidence that $\Gamma \vdash t_1 : A$ (obtained by Lemma 3), we see that 🔒 $\in \Gamma$, which would contradict the second premise.

Case(DUnbox). Essentially similar to the above; we observe that it is required that $t_1 \equiv$ **unbox** b_1 for some b_1, and inversion on the typing derivation shows that the context cannot be lock-free.

Case(DBeta). If $t_1 \longmapsto t_1'$ via the (DBeta) rule, we have $t_1 \equiv (\boldsymbol{\lambda} x.\, a_1)b_1$ and $t_1' \equiv a_1[b_1/x]$. Via inversion on $\Gamma \vdash t_1 \sim t_2 : A$ we obtain the following.

$$\dfrac{\dfrac{(1)\ \dfrac{\vdots}{\Gamma, x : T \vdash a_1 \sim a_2 : U}}{\Gamma \vdash \boldsymbol{\lambda} x.\, a_1 \sim \boldsymbol{\lambda} x.\, a_2 : T \to U}\ (\textsc{SimLam}) \qquad (2)\ \dfrac{\vdots}{\Gamma \vdash b_1 \sim b_2 : T}}{\Gamma \vdash \underbrace{(\boldsymbol{\lambda} x.\, a_1)b_1}_{t_1} \sim \underbrace{(\boldsymbol{\lambda} x.\, a_2)b_2}_{t_2} : \underbrace{U}_{A}}\ (\textsc{SimApp})$$

We provide $a_2[b_2/x]$ as the value of t_2'. Constructing the required indistinguishability proof requires use of Theorem 1, combining the second premise with (1) to produce a proof of $\Gamma \vdash a_1[b_1/x] \sim a_2[b_2/x] : U$ (which is also our proof obligation).

Case(DAppR). If $t_1 \longmapsto t_1'$ via the (DAppR) rule, we have $t_1 \equiv a_1 b_1$, $t_1' \equiv a_1 b_1'$ and additionally a proof of (1) $b_1 \longmapsto b_1'$. Via inversion on $\Gamma \vdash t_1 \sim t_2 : A$ we obtain the following.

$$\dfrac{(2)\ \dfrac{\vdots}{\Gamma \vdash a_1 \sim a_2 : T \to U} \qquad (3)\ \dfrac{\vdots}{\Gamma \vdash b_1 \sim b_2 : T}}{\Gamma \vdash \underbrace{a_1 b_1'}_{t_1} \sim \underbrace{a_2 b_2'}_{t_2} : \underbrace{U}_{A}}\ (\textsc{SimApp})$$

With this, we can apply the induction hypothesis along with (1) and (3) to obtain a b_2' such that $b_2 \longmapsto b_2'$ and (4) $\Gamma \vdash b_1' \sim b_2' : T$. Finally, we provide $t_2' \equiv a_2 b_2'$ along with the following proof.

$$\dfrac{\dfrac{(2)}{\Gamma \vdash a_1 \sim a_2 : T \to U} \qquad \dfrac{(4)}{\Gamma \vdash b_1' \sim b_2' : T}}{\Gamma \vdash \underbrace{a_1 b_1'}_{t_1} \sim \underbrace{a_2 b_2'}_{t_2} : \underbrace{U}_{A}} \ (\textsc{SimApp})$$

Cases(DAppL, DProd1, DProd2) are similar congruence cases to the above.

Case(DProj1). If $t_1 \longmapsto t_1'$ via the (DProj1) rule then $t_1 \equiv \boldsymbol{\pi}_1(\langle a_1, b_1 \rangle)$ and it is easy to show $t_2 \equiv \boldsymbol{\pi}_1(\langle a_2, b_2 \rangle)$ via inversion (we obtain $A \equiv T \times U$ and so $\Gamma \vdash a_1, a_2 : T$). We then provide a_2 as the value of t_2', and the required indistinguishability proof $\Gamma \vdash t_1' \sim t_2' : T$ can be found in the derivation of the second premise.

Case(DProj2). Similar to the above.

□

Chapter 4

Formalising non-interference

We now cover the formalisation part of our proof. Note that a full proof listing can be found in Appendix A. We first briefly cover how we formalise the definition of the calculus (modules `Base`, `Terms`, `LFExt` and `Trans`), before discussing the core indistinguishability relation (module `Simul`) and describing in-detail the core proof terms provided (module `NonInterference`).

4.1 Basic definitions

To begin with, we define contexts and types (A.1), almost identically to the definitions given in Figures 2.1 and 2.2.

Terms use an intrinsically well-typed and well-scoped representation [2], as is common when using dependently-typed languages. With this, we do away with the concept of pre-terms — it is impossible to specify a term without also specifying its type, and the context with which it is typed with. However, the terms are constructed in such a way that it is simply not possible to construct a term whose type is invalid. For example, there is simply no way to construct a term of shape `ƛ t` that does not have a type of shape `A ⇒ B`. Similarly, all other constructors are defined by their resulting types (and the type of their subterms). The same is true for variables and contexts; using a variable (as per the `var` constructor) requires providing a de Bruijn index [6] that witnesses the presence of the variable inside of the context. The definition of these indices is shown in Figure 4.1. We see that it is impossible to witness a variable in an empty context (and this is easy enough to prove as a lemma within Agda, see `¬A∈∅`), and furthermore that this ensures well-typing, we cannot construct a witness that `A ∈ Γ` unless a variable of type `A` is actually in the context.

Additionally, defining terms requires some manipulation of contexts, as in the (TUnbox) rule. There are many ways to represent this conveniently. We have chosen to define and use the type of *lock-free extensions* (a technique borrowed from [17]), which are proofs that a context Γ has shape Γ_1, Γ_2 (where Γ_2 has no locks). We can then allow Agda to "see" the structure of contexts by pattern matching on these extensions, revealing whether the extension contains additional variables, or is empty (in this case, we learn that Γ and Γ_1 are in fact identical, and can unify the two). We see the utility of this type in the `unbox` constructor, where an extension is used to partition a context around its "topmost" lock, matching the definition in (TUnbox) and allowing us to build weakening into the rule in a way that is amenable to formalisation.

Figure 4.1: Typed de Bruijn indices.

```
-- Typed de Bruijn indices.
data _∈_ : Type → Context → Set where
  Z : A ∈ Γ , A
  S : A ∈ Γ → A ∈ Γ , B

-- Nothing can be a member of an empty context.
¬A∈∅ : ¬ (A ∈ ∅)
¬A∈∅ ()
```

Figure 4.2: Lock-free extensions.

```
-- The lock-free extension relation.
-- Relates contexts extended to the right with lock free contexts.
-- By pattern matching on this, the structure of Γ is revealed.
data _is_∷_ : Context → Context → Context → Set where
  is-nil : Γ is Γ ∷ ∅
  is-ext : Γ is Γ₁ ∷ Γ₂ → Γ , A is Γ₁ ∷ Γ₂ , A
```

Figure 4.3: Intrinsically typed modal terms.

```
-- The type of well-typed and scoped terms.
data _⊢_ : Context → Type → Set where
  nat : ℕ   → Γ ⊢ Nat
  var : A ∈ Γ → Γ ⊢ A

  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  box : Γ ■ ⊢ A   → Γ ⊢ □ A
  -- Without weakening, this reads something like...
  -- unbox : Γ ⊢ □ A → Γ ■ ⊢ A
  -- However, this definition builds in weakening.
  unbox : {ext : Γ is Γ₁ ■ ∷ Γ₂} → Γ₁ ⊢ □ A → Γ ⊢ A

  _•_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
```

The core bisimulation relation, which we use to relate indistinguishable terms, comes next. This is also essentially identical to the rules we have in Figure 2.4. We capture the notion that all boxes are indistinguishable to an unprivileged observer with a rule that equates *all terms* when the context is locked (and equating **box** t_1 with **box** t_2 iff t_1 is related to t_2, as usual).

Figure 4.4: The formalised simulation relation.

```
data _⊢_~_:_ : (Γ : Context) → Γ ⊢ A → Γ ⊢ A → (A : Type) → Set where
  sim-nat : (n : ℕ)
          ---------
          → Γ ⊢ nat n ~ nat n : Nat

  sim-lock : Γ is Γ₁ ■ :: Γ₂
           → (t : Γ ⊢ A)
           → (t′ : Γ ⊢ A)
           --------------
           → Γ ⊢ t ~ t′ : A

  sim-var : (x : A ∈ Γ)
          --------------------
          → Γ ⊢ var x ~ var x : A

  sim-app : Γ ⊢ t₁    ~ t₁′ : A ⇒ B
          → Γ ⊢ t₂    ~ t₂′ : A
          --------------------------
          → Γ ⊢ t₁ • t₂ ~ t₁′ • t₂′ : B

  sim-lam : Γ , A ⊢ t ~ t′ : B
          -----------------------
          → Γ ⊢ ƛ t ~ ƛ t′ : A ⇒ B

  sim-box : Γ ■ ⊢ t   ~ t′ : A
          -----------------------
          → Γ ⊢ box t ~ box t′ : □ A
```

We also must define the β-reduction relation before we can complete the proof. This is done in a completely standard manner. However, it is important to note that we treat boxes as values, i.e. we do not permit a rule that allows us to conclude **box** $t \longmapsto$ **box** t' from $t \longmapsto t'$. In fact, this rule breaks non-interference! The idea here is to look at the relation $\Gamma \vdash$ **box** $(\boldsymbol{\lambda} x.\, x)1 \sim$ **box** $2 : \Box\text{Nat}$ (or anything similar); we see that the first term can reduce to **box** 1, so we expect **box** 2 to be able to reduce to a related term. However, no such reduction exists (in fact, this term is a value), so indistinguishability would not a simulation.

4.2 Non-interference

Somewhat surprisingly, it is fairly easy to prove the main result if we allow use of the indistinguishability under substituion lemma. The proof takes the form of a term `ni : ¬■ Γ → Γ ⊢ t₁ ~ t₂ : A → t₁ →β t₁′ → Σ[ t₂′ ∈ Γ ⊢ A ] ((t₂ →β t₂′) × (Γ ⊢ t₁′ ~ t₂′ : A))`. The premise `¬■ Γ` (which witnesses $🔒 \notin \Gamma$) might seem surprising, but it is necessary: otherwise a relation like $🔒 \vdash (\boldsymbol{\lambda} x.\, x)1 \sim 2 : \text{Nat}$ breaks the simulation, as the left hand side can reduce but the right hand side cannot. However, this extra constraint shouldn't change the essence of what is being proved — the theorem will continue to hold for closed terms; the only way a lock can be introduced into the context is via unboxing.

This makes the proof surprisingly tractable, however — when we perform case-analysis on the second argument, we see that the simulation cannot hold via (SimBox) (as boxes are values) or (SimLock) (as this requires a locked context). Of the remaining cases, the results for (SimApp) is a simple congruence cases requiring only one invocation of the inductive hypothesis, meaning (SimLam) is the only real problem. This requires using the indistinguishability under substitution lemma we have just shown, the required proof that the substitutions are related can be constructed by extending the proof that identical substitutions are related (`sim-refl`). Figure 4.5 shows the formal proof of this case — note that we rewrite with the `sit` (simulation implies typing) lemma to allow Agda to see that $\Gamma \vdash t_1 \sim t_2 : A$ implies $\Gamma \vdash t_1 : A$ and

$\Gamma \vdash t_2 : A$.

Figure 4.5: The β-reduction case of the non-interference proof.

```
ni prf sim@(sim-app {t₁ = f₁} {f₂} {t₂ = x₁} {x₂} simƛ simᵣ) βƛ
                                    with sit _ _ sim | sit _ _ simƛ | sit _ _ simᵣ
... | refl | refl | refl            with simƛ
... | sim-lock ext _ _ = ⊥-elim (¬■-■ prf ext)
... | sim-lam {t = b₁} {b₂} sim∘ with sit _ _ sim∘
... | refl = b₂ [ x₂ ]
             , βƛ
             , ius (¬■, prf) prf b₁ b₂ (sub-refl • x₁) (sub-refl • x₂) sim∘ (simσ-• simσ-refl simᵣ)
```

4.3 Indistinguishability under substitution

As mentioned prior, this is a fairly tricky lemma to prove. The main reason for this is it seems to require strengthening the notion of substitution (to allow parallel substitution of multiple terms in one step). The definition of these substitutions are shown in Figure 4.6. Additionally, instead of simply requiring the substituted terms to be related, we must extend the definition of the simulation relation pointwise to substitutions.

Figure 4.6: Substitutions and simulations between them.

```
-- (Subst.agda) Substitutions between contexts.
data _⇉_ : Context → Context → Set where
  -- Base substitution - we can always substitute every variable in an empty context.
  ε : Γ ⇉ ∅

  -- Weaken a substitution.
  wk : Γ ⊆ Γ′         → Γ′ ⇉ Γ
  -- Under locks.
  _•■ : Γ ⇉ Δ         → Γ ■ ⇉ Δ ■
  -- Extend a substitution with a term.
  _•_ : Γ ⇉ Δ → Γ ⊢ A → Γ ⇉ Δ , A


-- (Simul.agda) Simulation extended pointwise to substitutions.
data _,_⊢_~_ : (Γ Δ : Context) → Γ ⇉ Δ → Γ ⇉ Δ → Set where
  simσ-ε : Γ , ∅ ⊢ ε ~ ε

  simσ-p : (w : Γ ⊆ Γ′)
         --------------
         → Γ′ , Γ ⊢ wk w ~ wk w

  simσ-■ : Γ , Δ ⊢ σ ~ τ
         -------------------------
         → Γ ■ , Δ ■ ⊢ σ •■ ~ τ •■

  simσ-• : Γ , Δ ⊢ σ ~ τ
         → Γ ⊢ t₁ ~ t₂ : A
         -----------------------------------
         → Γ , (Δ , A) ⊢ (σ • t₁) ~ (τ • t₂)
```

Because non-interference requires a proof of 🔒 $\notin \Gamma$, our job will be easier if we require this here too (as we will see there is no need to show the `unbox` case). The (SimLock) case then simply consists of propagating this rule, and the variable case requires only doing a substitution and using a small lemma or two.

Most inductive cases are simple congruences, but trouble occurs in cases where we must modify the context. The case for (SimBox) is straightforward, as the context "inside" the box is locked. When dealing with lambdas, however, we must extend the context with a new type (as we are using de Bruijn indexing). It turns out to be rather difficult to show that this preserves the indistinguishability of the substitutions. From our experience, this requires use of the inductive hypothesis, and Agda (for whatever reason) has trouble seeing that this terminates (and requires a little coaxing).

Chapter 5

Concluding thoughts

We conclude with some finishing thoughts. First, we note that by formalising this proof, we discovered several subtleties in the definition of the calculus. Without this, it would be very easy to e.g. include a rule that allows reduction under boxes, subsequently breaking the non-interference property! This speaks to the accomplishments of this project, and the use of formalisation in general.

Secondly, we would like to point out that this technique for proving secure information flow seems to be much simpler and more amenable to formalisation than other approaches (perhaps this comes at some cost for the calculi it is able to represent). It would have been nice to demonstrate this via providing (formal) proofs for other calculi, but this could not be included in this dissertation due to time constraints.

Bibliography

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160, New York, NY, USA, 1999. Association for Computing Machinery.

[2] Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. 05 1999.

[3] Pritam Choudhury, Harley Eades, and Stephanie Weirich. A dependent dependency calculus. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 403–430, Cham, 2022. Springer International Publishing.

[4] Ranald Clouston. *Fitch-Style Modal Lambda Calculi*, pages 258–275. Springer International Publishing, 04 2018.

[5] Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. A theory of effects and resources: Adjunction models and polarised calculi. *SIGPLAN Not.*, 51(1):44–56, jan 2016.

[6] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, Jan 1972.

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

[8] Frederic Brenton Fitch. Symbolic logic, an introduction. *American Journal of Physics*, 21:237–237, 1953.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982.

[10] 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*, POPL '98, pages 365–377, New York, NY, USA, 1998. Association for Computing Machinery.

[11] G. A. Kavvos. Modalities, cohesion, and information flow. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[12] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 221–232, New York, NY, USA, 2013. Association for Computing Machinery.

[13] Leonard J. LaPadula and D. Elliott Bell. Mitre technical report 2547, volume ii. *Journal of Computer Security*, 4:239–263, 1996. 2-3.

[14] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[15] T. Murphy, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pages 286–295, 2004.

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

[17] Nachiappan Valliappan, Fabian Ruch, and Carlos Tomé Cortiñas. Normalization for fitch-style modal calculi. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.

Appendix A

Agda proof source

A.1 Basic definitions

```
module Base where
open import Relation.Nullary
open import Relation.Nullary.Decidable
open import Data.Bool
open import Data.Empty
open import Data.Unit
open import Relation.Binary.PropositionalEquality
open import Data.Product renaming (_,_ to _،_)

infixr 7 _⇒_
-- Modal type constructors.
data Type : Set where
  Nat : Type
  □_  : Type → Type
  _⇒_ : Type → Type → Type

infixl 5 _,_
-- Contexts with locks.
data Context : Set where
  ∅   : Context
  _,_ : Context → Type → Context
  _■ : Context → Context

private variable
  A B : Type
  Γ Γ′ Δ Δ′ Γ₁ Γ₂ Γ₃ : Context

infixl 4 _∷_
-- -- Context combination.
_∷_ : Context → Context → Context
Γ ∷ ∅   = Γ
Γ ∷ Δ , x = (Γ ∷ Δ) , x
Γ ∷ Δ ■ = (Γ ∷ Δ) ■

-- Lock-free contexts
data ¬■ : Context → Set where
  ¬■∅ : ¬■ ∅
  ¬■, : ¬■ Γ → ¬■ (Γ , A)

infix 4 _∈_
-- Witnesses the membership of a variable with a given type in a context.
data _∈_ : Type → Context → Set where
```

```
  Z : A ∈ Γ , A
  S : A ∈ Γ → A ∈ Γ , B

-- Nothing can be a member of an empty context
¬A∈∅ : ¬ (A ∈ ∅)
¬A∈∅ ()

-- Elements left of the leftmost lock
←■ : Context → Context
←■ ∅     = ∅
←■ (Γ , A) = ←■ Γ
←■ (Γ ■) = Γ

-- Elements right of the rightmost lock
■→ : Context → Context
■→ ∅     = ∅
■→ (Γ , A) = ■→ Γ , A
■→ (Γ ■) = ∅

infix 4 _⊆_
-- Subcontexts, for weakening
data _⊆_ : Context → Context → Set where
  ⊆-empty : ∅            ⊆ ∅
  ⊆-drop : Γ ⊆ Δ → Γ    ⊆ Δ , A
  ⊆-keep : Γ ⊆ Δ → Γ , A ⊆ Δ , A
  ⊆-lock : Γ ⊆ Δ → Γ ■ ⊆ Δ ■

⊆∅ : Γ ⊆ ∅ → Γ ≡ ∅
⊆∅ {∅} wk = refl
⊆∅ {Γ , x} ()
⊆∅ {Γ ■} ()

⊆-assoc : Γ₁ ⊆ Γ₂ → Γ₂ ⊆ Γ₃ → Γ₁ ⊆ Γ₃
⊆-assoc ⊆-empty wk₂ = wk₂
⊆-assoc (⊆-drop wk₁) (⊆-drop wk₂) = ⊆-drop (⊆-assoc (⊆-drop wk₁) wk₂)
⊆-assoc (⊆-drop wk₁) (⊆-keep wk₂) = ⊆-drop (⊆-assoc wk₁ wk₂)
⊆-assoc (⊆-keep wk₁) (⊆-drop wk₂) = ⊆-drop (⊆-assoc (⊆-keep wk₁) wk₂)
⊆-assoc (⊆-keep wk₁) (⊆-keep wk₂) = ⊆-keep (⊆-assoc wk₁ wk₂)
⊆-assoc (⊆-lock wk₁) (⊆-drop wk₂) = ⊆-drop (⊆-assoc (⊆-lock wk₁) wk₂)
⊆-assoc (⊆-lock wk₁) (⊆-lock wk₂) = ⊆-lock (⊆-assoc wk₁ wk₂)

⊆-refl : Γ ⊆ Γ
⊆-refl {Γ = Γ , x} = ⊆-keep ⊆-refl
⊆-refl {Γ = Γ ■} = ⊆-lock ⊆-refl
⊆-refl {Γ = ∅}   = ⊆-empty

⊆-←■ : Γ ⊆ Δ → ←■ Γ ⊆ ←■ Δ
⊆-←■ ⊆-empty = ⊆-empty
⊆-←■ (⊆-drop wk) = ⊆-←■ wk
⊆-←■ (⊆-keep wk) = ⊆-←■ wk
⊆-←■ (⊆-lock wk) = wk

Γ-weak : Γ ⊆ Δ → A ∈ Γ → A ∈ Δ
Γ-weak (⊆-drop rest) x = S (Γ-weak rest x)
Γ-weak (⊆-keep rest) (S x) = S (Γ-weak rest x)
Γ-weak (⊆-keep rest) Z = Z
```

A.2 Intrinsically-typed terms

```
module Terms where
open import Base
open import LFExt
open import Data.Nat
open import Data.Unit
open import Data.Empty
open import Function.Base
open import Relation.Binary.PropositionalEquality
open import Data.Product renaming (_,_ to _‚_)

private variable
  A B T U : Type
  Γ Δ Γ₁ Γ₂ : Context

infixl 6 _•_
infix 5 ƛ_
infix 3 _⊢_
-- The type of well-typed and scoped terms.
data _⊢_ : Context → Type → Set where
  nat : ℕ   → Γ ⊢ Nat
  var : A ∈ Γ → Γ ⊢ A

  ƛ_  : Γ , A ⊢ B → Γ ⊢ A ⇒ B
  box : Γ ■ ⊢ A   → Γ ⊢ □ A
  -- Ideally, this reads something like
  -- unbox : Γ     ⊢ □ A → Γ ■ ⊢ A
  -- However we have to deal with weakening etc
  unbox : {ext : Γ is Γ₁ ■ ∷ Γ₂} → Γ₁ ⊢ □ A → Γ ⊢ A

  _•_ : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B


weakening : Γ ⊆ Δ → Γ ⊢ A → Δ ⊢ A
weakening wk (nat n) = nat n
weakening wk (var x) = var (Γ-weak wk x)
weakening wk (l • r) = (weakening wk l) • (weakening wk r)
weakening wk (ƛ t) = ƛ weakening (⊆-keep wk) t
weakening wk (box t) = box (weakening (⊆-lock wk) t)
weakening wk (unbox {ext = e} t)
  = unbox {ext = is∷-Δweak e wk} (weakening (is∷-←■weak e wk) t)
```

A.3 Lock-free extensions

```
module LFExt where
open import Base
open import Relation.Nullary
open import Relation.Nullary.Decidable
open import Data.Bool
open import Data.Empty
open import Data.Unit
open import Relation.Binary.PropositionalEquality
open import Data.Product renaming (_,_ to _‚_)

private variable
  A B : Type
  Γ Δ Γ₁ Γ₂ Γ₃ : Context

infix 3 _is_∷_
-- Lock free extension relation
-- Relates contexts extended to the right with lock free contexts
```

```
data _is_∷_ : Context → Context → Context → Set where
  is-nil : Γ is Γ ∷ ∅
  is-ext : Γ is Γ₁ ∷ Γ₂ → Γ , A is Γ₁ ∷ Γ₂ , A

-- Lock free extensions are equivalences
is∷-≡ : Γ is Γ₁ ∷ Γ₂ → Γ ≡ (Γ₁ ∷ Γ₂)
is∷-≡ is-nil = refl
is∷-≡ (is-ext ext) rewrite is∷-≡ ext = refl

-- Lock free extensions are equivalences, inverse direction with lock-freeness
≡-is∷ : ¬■ Γ₂ → Γ ≡ (Γ₁ ∷ Γ₂) → Γ is Γ₁ ∷ Γ₂
≡-is∷ {Γ₂ = ∅} prf refl = is-nil
≡-is∷ {Γ₂ = Γ₂ , x} (¬■, prf) refl = is-ext (≡-is∷ prf refl)

-- Γ is Γ₁ if Γ₂ is empty
is∷-Γ₂≡∅ : Γ is Γ₁ ∷ ∅ → Γ ≡ Γ₁
is∷-Γ₂≡∅ is-nil = refl

-- Left-of-lock is Γ₁ if it ends with a lock
is∷-←■ : Γ is Γ₁ ■ ∷ Γ₂ → (←■ Γ) ≡ Γ₁
is∷-←■ (is-ext ext) = is∷-←■ ext
is∷-←■ is-nil = refl

-- Lock free extensions mean lock free contexts.
is∷-¬■Γ : Γ is Γ₁ ∷ Γ₂ → ¬■ Γ₂
is∷-¬■Γ is-nil    = ¬■∅
is∷-¬■Γ (is-ext ext) = ¬■, (is∷-¬■Γ ext)

-- Contexts with locks in aren't lock free.
¬■-■ : ¬■ Γ → ¬ (Γ is Γ₁ ■ ∷ Γ₂)
¬■-■ (¬■, prf) (is-ext ext) = ¬■-■ prf ext

-- If supercontext of context with a lock, the context also has a lock.
■⊆′ : Γ ■ ⊆ Δ → Σ[ Δ₁ ∈ Context ] Σ[ Δ₂ ∈ Context ] Δ is Δ₁ ■ ∷ Δ₂
■⊆′ {_} {(Δ ■)} (⊆-lock wk) = Δ , ∅ , is-nil
■⊆′ {_} {(Δ , B)} (⊆-drop wk) with ■⊆′ wk
... | Δ₁ , Δ₂ , ext = Δ₁ , Δ₂ , B , is-ext ext

-- Extensions are congruent under the left-of-lock operation ←■
is∷-←■weak : Γ is Γ₁ ■ ∷ Γ₂ → Γ ⊆ Δ → Γ₁ ⊆ ←■ Δ
is∷-←■weak ext     (⊆-drop wk) = is∷-←■weak ext wk
is∷-←■weak (is-ext ext) (⊆-keep wk) = is∷-←■weak ext wk
is∷-←■weak is-nil (⊆-lock wk) = wk

-- Apply a weakening to an entire extension
is∷-Δweak : Γ is Γ₁ ■ ∷ Γ₂ → Γ ⊆ Δ → Δ is ((←■ Δ) ■) ∷ (■→ Δ)
is∷-Δweak ext     (⊆-drop wk) = is-ext (is∷-Δweak ext wk)
is∷-Δweak (is-ext ext) (⊆-keep wk) = is-ext (is∷-Δweak ext wk)
is∷-Δweak is-nil (⊆-lock wk) = is-nil
```

A.4 Well-typed substitution

```
module Subst where
open import Base
open import Terms
open import LFExt
open import Relation.Binary.PropositionalEquality hiding ([_])
open import Data.Bool
open import Data.Unit
open import Data.Empty
open import Data.Product renaming (_,_ to _،_)

private variable
  A B : Type
  Γ Δ θ Γ₁ Γ₂ Γ′ Δ₁ Δ₂ Δ′ : Context

infixr 3 _⇉_
infixr 4 _•■
infixr 4 _•_

data _⇉_ : Context → Context → Set where
  -- Base substitution
  ε : Γ ⇉ ∅

  -- Weaken a substitution
  wk : Γ ⊆ Γ′        → Γ′ ⇉ Γ
  -- Under locks
  _•■ : Γ ⇉ Δ        → Γ ■ ⇉ Δ ■
  -- Extend a substitution with a term
  _•_ : Γ ⇉ Δ → Γ ⊢ A → Γ ⇉ Δ , A

p : Γ , A ⇉ Γ
p = wk (⊆-drop ⊆-refl)

-- Strengthen the domain of a substitution
⇉-st : Γ ⇉ Δ → Δ′ ⊆ Δ → Γ ⇉ Δ′
⇉-st _      ⊆-empty = ε
⇉-st (wk x) w        = wk (⊆-assoc w x)
⇉-st (σ •■) (⊆-lock w) = ⇉-st σ w •■
⇉-st (σ • x) (⊆-drop w) = ⇉-st σ w
⇉-st (σ • x) (⊆-keep w) = ⇉-st σ w • x

⇉-refl : Γ ⇉ Γ
⇉-refl {∅}   = ε
⇉-refl {Γ ■} = ⇉-refl •■
⇉-refl {Γ , x} = p • var Z

-- Useful lemma for proofs involving the unbox constructor.
-- ... since extensions of this type are produced by it,
-- and we need one in order to put everyhting back together again.
is∷-Δsub : Δ is Δ₁ ■ ∷ Δ₂ → Γ ⇉ Δ → Γ is (←■ Γ) ■ ∷ (■→ Γ)
is∷-Δsub is-nil (wk (⊆-drop w)) = is-ext (is∷-Δsub is-nil (wk w))
is∷-Δsub is-nil (wk (⊆-lock w)) = is-nil
is∷-Δsub is-nil (σ •■)          = is-nil
is∷-Δsub (is-ext ext) (wk (⊆-drop w)) = is-ext (is∷-Δsub (is-ext ext) (wk w))
is∷-Δsub (is-ext ext) (wk (⊆-keep w)) = is-ext (is∷-Δsub ext (wk w))
is∷-Δsub (is-ext ext) (σ • t)    = is∷-Δsub ext σ

-- Much like before. This gives us a substitution that...
-- ... only works left of a lock, from one produced by unbox.
-- Γ   is   (Γ₁) ■ ∷ Γ₂
-- ↓          ↓↓
-- Δ   is   (Δ₁) ■ ∷ Γ₂
sub-←■ : Δ is Δ₁ ■ ∷ Δ₂ → Γ ⇉ Δ → (←■ Γ) ⇉ Δ₁
sub-←■ is-nil (wk w) with ■⊆′ w
... | Γ₁ ، Γ₂ ، ext with is∷-←■ ext
... | refl          = wk (⊆-←■ w)
sub-←■ is-nil (σ •■) = σ
```

```
sub-←■ (is-ext ext) (wk w) with is∷-←■ ext
... | refl = wk (⊆-←■ w)
sub-←■ (is-ext ext) (σ • x) = sub-←■ ext σ

-- This is fairly theoretically grounded
{-# TERMINATING #-}
mutual
  _◦_ : Δ ⇉ θ → Γ ⇉ Δ → Γ ⇉ θ
  sub : Γ ⇉ Δ → Δ ⊢ A → Γ ⊢ A

  σ+ : Γ ⇉ Δ → Γ , A ⇉ Δ , A
  σ+ σ = (σ ◦ p) • (var Z)

  -- Parallel substitution, defined mutually recursively with composition
  sub σ (nat x) = nat x
  --------------------------
  sub (wk w) (var Z)  = var (Γ-weak w Z)
  sub (σ • t) (var Z) = t
  sub σ      (var (S x)) = sub (p ◦ σ) (var x)
  ----------------------------------------
  sub σ (ƛ t) = ƛ sub (σ+ σ) t
  sub σ (box t) = box (sub (σ •■) t)
  sub σ (l • r) = sub σ l • sub σ r
  ---------------------------------------------------
  sub σ (unbox {ext = e} t) = unbox {ext = is∷-Δsub e σ} (sub (sub-←■ e σ) t)

  -- Compose substitutions
  ε      ◦ τ = ε
  wk w   ◦ τ = ⇉-st τ w
  (σ • t) ◦ τ        = (σ ◦ τ) • sub τ t
  (σ •■) ◦ wk w with ■⊆′ w | w
  ... | Γ₁ ‚ ∅ ‚ is-nil | ⊆-lock w = (σ ◦ wk w) •■
  ... | _  ‚ _ ‚ is-ext ext | w = (σ •■) ◦ wk w
  (σ •■) ◦ (τ •■) = (σ ◦ τ) •■

sub-refl : Γ ⇉ Γ
sub-refl {∅}   = ε
sub-refl {Γ , x} = σ+ sub-refl
sub-refl {Γ ■} = sub-refl •■

infix 5 _[_]
-- Single variable substitution on the first free variable. Used in β.
_[_] : Γ , B ⊢ A → Γ ⊢ B → Γ ⊢ A
t₁ [ t₂ ] = sub (sub-refl • t₂) t₁
```

A.5 The transition relation

```
module Trans where
open import Base
open import Terms
open import Subst
open import LFExt

private variable
  t l r t₁ t₂ : _ ⊢ _
  t′ l′ r′ t₁′ t₂′ : _ ⊢ _
  A B : Type
  Γ Γ₁ Γ₂ : Context
```

```
    □ext : Γ is Γ₁ ■ ∷ Γ₂

  infix 4 _→β_
  -- Transitiion relation.
  data _→β_ : Γ ⊢ A → Γ ⊢ A → Set where
    β■ : unbox {ext = □ext} (box t) →β t
    βƛ : (ƛ t) • r →β t [ r ]

    ξappl : l →β l′ → l • r →β l′ • r
    ξappr : r →β r′ → l • r →β l • r′

    -- No β under boxes, we should treat boxes as values.
    -- Otherwise box 1 ~ box ((λx. x) 1), and NI is broken.
    -- ξbox   : t →β t′ → box   t →β box   t′
    ξunbox : t →β t′ → unbox {ext = □ext} t →β unbox {ext = □ext} t′
```

A.6 The bisimulation relation

```
  module Simul where
  open import Base
  open import LFExt
  open import Terms
  open import Trans
  open import Relation.Binary.PropositionalEquality hiding ([_])
  open import Function
  open import Data.Bool
  open import Data.Nat
  open import Data.Product renaming (_,_ to _▯_)
  open import Subst

  private variable
    t t′ t₁ t₂ t₁′ t₂′ a a₁ a₂ a′ b b₁ b₂ b′ : _ ⊢ _
    A B : Type
    Γ Γ′ Δ Γ₁ Γ₂ θ : Context
    □ext : Γ is Γ₁ ■ ∷ Γ₂
    σ σ′ σ₁ σ₂ τ τ′ : _ ⇉ _

  infix 2 _⊢_~_:_
  data _⊢_~_:_ : (Γ : Context) → Γ ⊢ A → Γ ⊢ A → (A : Type) → Set where
    sim-nat : (n : ℕ)
            ----------
            → Γ ⊢ nat n ~ nat n : Nat

    sim-lock : Γ is Γ₁ ■ ∷ Γ₂
             → (t : Γ ⊢ A)
             → (t′ : Γ ⊢ A)
             ------------------
             → Γ ⊢ t ~ t′ : A

    sim-var : (x : A ∈ Γ)
            ----------------------------------
            → Γ ⊢ var x ~ var x : A

    sim-app : Γ ⊢ t₁      ~ t₁′ : A ⇒ B
            → Γ ⊢ t₂      ~ t₂′ : A
            -----------------------------------
            → Γ ⊢ t₁ • t₂ ~ t₁′ • t₂′ : B
    sim-lam : Γ , A ⊢ t   ~ t′ : B
            ---------------------------------
            → Γ   ⊢ ƛ t ~ ƛ t′ : A ⇒ B

    sim-box : Γ ■ ⊢ t     ~ t′ : A
            ----------------------------
            → Γ ⊢ box t ~ box t′ : □ A

    sim-unbox : Γ ⊢ t ~ t′ : □ A
              ----------------------------------------------------------
              → Γ ■ ⊢ unbox {ext = □ext} t ~ unbox {ext = □ext} t′ : A

  sim-refl : (t : Γ ⊢ A) → Γ ⊢ t ~ t : A
  sim-refl (nat n) = sim-nat n
  sim-refl (var x) = sim-var x
  sim-refl (ƛ t) = sim-lam (sim-refl t)
  sim-refl (box t) = sim-box (sim-refl t)
  sim-refl (l • r) = sim-app (sim-refl l) (sim-refl r)
  sim-refl (unbox {ext = e} t)
    = sim-lock e (unbox t) (unbox t)

  -- Simulation implies typing, used to coax agda into unifying types of simulations.
  sit : (t₁ t₂ : Γ ⊢ B) → Γ ⊢ t₁ ~ t₂ : A → A ≡ B
  sit t₁        t₂        (sim-nat n)      = refl
  sit t₁        t₂        (sim-lock x _ _) = refl
  sit t₁        t₂        (sim-var x)      = refl
  sit (ƛ t₁)    (ƛ t₂)    (sim-lam sim)      rewrite sit t₁ t₂ sim = refl
  sit (box t₁)  (box t₂)  (sim-box sim)      rewrite sit t₁ t₂ sim = refl
  sit (l₁ • r₁) (l₂ • r₂) (sim-app simₗ simᵣ) with sit l₁ l₂ simₗ
  ... | refl = refl
  sit (unbox t₁) (unbox t₂) (sim-unbox sim) with sit t₁ t₂ sim
  ... | refl = refl

  -- Simulation extended pointwise to substitutions
  infix 2 _,_⊢_~_
  data _,_⊢_~_ : (Γ Δ : Context) → Γ ⇉ Δ → Γ ⇉ Δ → Set where
    simσ-ε : Γ , ∅ ⊢ ε ~ ε

    simσ-p : (w : Γ ⊆ Γ′)
           --------------
           → Γ′ , Γ ⊢ wk w ~ wk w

    simσ-■ : Γ    , Δ ⊢ σ ~ τ
           -------------------------
           → Γ ■ , Δ ■ ⊢ σ •■ ~ τ •■

    simσ-• : Γ , Δ ⊢ σ ~ τ
           → Γ    ⊢ t₁ ~ t₂ : A
           ---------------------------------
           → Γ , (Δ , A) ⊢ (σ • t₁) ~ (τ • t₂)

  simσ-refl : Γ , Δ ⊢ σ ~ σ
  simσ-refl {σ = ε}    = simσ-ε
  simσ-refl {σ = wk x} = simσ-p x
  simσ-refl {σ = σ •■} = simσ-■ simσ-refl
  simσ-refl {σ = σ • t} = simσ-• simσ-refl (sim-refl t)
```

A.7 Proof of non-interference

```
module NonInterference where
open import Base
open import LFExt
open import Terms
open import Trans
open import Relation.Binary.PropositionalEquality hiding ([_])
open import Relation.Nullary
open import Function
open import Data.Bool
open import Data.Empty
open import Data.Nat
open import Data.Product renaming (_,_ to _,_)
open import Subst
open import Simul


private variable
  t t′ t₁ t₂ t₁′ t₂′ a a₁ a₂ a′ b b₁ b₂ b′ : _ ⊢ _
  A B : Type
  Γ Δ Δ′ Γ₁ Γ₂ : Context
  □ext : Γ is Γ₁ ■ :: Γ₂
  σ σ′ σ₁ σ₂ τ τ′ : _ ⇉ _

private module lemmas where

  lemma-st : (w : Δ′ ⊆ Δ)
           → Γ , Δ ⊢ σ ~ τ
           ----------------------------------------
           → Γ , Δ′ ⊢ ⇉-st σ w ~ ⇉-st τ w
  lemma-st ⊆-empty simσ-ε = simσ-ε
  --------------------------------
  lemma-st ⊆-empty (simσ-p w₁) = simσ-ε
  lemma-st (⊆-drop w) (simσ-p w₁) = simσ-p (⊆-assoc (⊆-drop w) w₁)
  lemma-st (⊆-keep w) (simσ-p w₁) = simσ-p (⊆-assoc (⊆-keep w) w₁)
  lemma-st (⊆-lock w) (simσ-p w₁) = simσ-p (⊆-assoc (⊆-lock w) w₁)
  -----------------------------------------------------------------
  lemma-st (⊆-lock w) (simσ-■ simσ) = simσ-■ (lemma-st w simσ)
  -------------------------------------------------------------
  lemma-st (⊆-drop w) (simσ-• simσ x) = lemma-st w simσ
  lemma-st (⊆-keep w) (simσ-• simσ x) = simσ-• (lemma-st w simσ) x

open lemmas

{-# TERMINATING #-}
mutual
  lemma-σ+ : ¬■ Γ
           → ¬■ Δ
           → Γ , Δ                  ⊢ σ ~ τ
           → (Γ , A) , (Δ , A) ⊢ σ+ {A = A} σ ~ σ+ {A = A} τ

  ius : ¬■ Γ
      → ¬■ Δ
      → (t₁ t₂ : Γ ⊢ A)
      → (σ₁ σ₂ : Δ ⇉ Γ)
      ------------------------------------
      → Γ ⊢ t₁    ~ t₂ : A
      → Δ , Γ ⊢ σ₁ ~ σ₂
```

```
   -------------------------------------
   → Δ ⊢ (sub σ₁ t₁) ~ (sub σ₂ t₂) : A

 lemma-σ+ prf₁ prf₂ simσ-ε = simσ-• simσ-ε (sim-var Z)
 lemma-σ+ prf₁ prf₂ (simσ-p w) = simσ-• (lemma-st w (simσ-p (⊆-drop ⊆-refl))) (sim-var Z)
 lemma-σ+ prf₁ (¬■, prf₂) (simσ-• {σ = σ} {t₁ = t₁} {t₂ = t₂} simσ simₜ) with lemma-σ+ prf₁ prf₂ simσ
 ... | simσ-• simσ′ r = simσ-• (simσ-• simσ′ (ius prf₁ (¬■, prf₁) t₁ t₂ p p simₜ simσ-refl)) r


 -- The indistinguishability under substitution lemma.
 ius _ _ t₁ t₂ σ τ (sim-nat n) simσ = sim-nat n
 ---------------------------------------------------
 ius _ _ t₁ t₂ σ τ (sim-lock ext _ _) simσ = sim-lock (is∷-Δsub ext σ) (sub σ t₁) (sub τ t₂)
 -------------------------------------------------------------------------------------------
 ius prf₁ prf₂   t₁ t₂ (wk w) (wk w) (sim-var Z) (simσ-p w) = sim-var (Γ-weak w Z)
 ius (¬■, prf₁) prf₂ t₁ t₂ (wk w) (wk w) (sim-var (S x)) (simσ-p w) =
   let w′ = ⊆-assoc (⊆-drop ⊆-refl) w
   in ius prf₁ prf₂ (var x) (var x) (wk w′) (wk w′) (sim-var x) (simσ-p w′)
 --------------------------------------------------------------------
 ius prf₁ prf₂ t₁ t₂ (σ • _) (τ • _) (sim-var Z) (simσ-• simσ simᵤ) = simᵤ
 ius (¬■, prf₁) prf₂ t₁ t₂ (σ • _) (τ • _) (sim-var (S x)) (simσ-• simσ simᵤ) =
   ius prf₁ prf₂ (var x) (var x) (⇉-st σ ⊆-refl) (⇉-st τ ⊆-refl) (sim-var x) (lemma-st ⊆-refl simσ)
 ---------------------------------------------------------------------------------------
 ius prf₁ prf₂ (l₁ • r₁) (l₂ • r₂) σ τ (sim-app simₗ simᵣ) simσ with sit _ _ simₗ | sit _ _ simᵣ
 ... | refl | refl = sim-app (ius prf₁ prf₂ l₁ l₂ σ τ simₗ simσ) (ius prf₁ prf₂ r₁ r₂ σ τ simᵣ simσ)
 ---------------------------------------------------------------------------------
 ius prf₁ prf₂ (ƛ b₁) (ƛ b₂) σ τ (sim-lam simₜ) simσ with sit b₁ b₂ simₜ
 ... | refl = sim-lam (ius (¬■, prf₁) (¬■, prf₂) b₁ b₂ (σ+ σ) (σ+ τ) simₜ (lemma-σ+ prf₂ prf₁ simσ))
 ----------------------------------------------------------------------------------
 ius prf₁ prf₂ (box b₁) (box b₂) σ τ (sim-box simₜ) simσ with sit b₁ b₂ simₜ
 ... | refl = sim-box (sim-lock is-nil (sub (σ •■) b₁) (sub (τ •■) b₂))

-- Non-interference for the Fitch calculus
ni : ¬■ Γ
  → Γ ⊢ t₁ ~ t₂ : A
  → t₁ →β t₁′
  --------------------------------------------------------
  → Σ[ t₂′ ∈ Γ ⊢ A ] ((t₂ →β t₂′) × (Γ ⊢ t₁′ ~ t₂′ : A))
ni prf (sim-lock ext _ _) _ = ⊥-elim (¬■-■ prf ext)
ni () (sim-unbox _) _
----------------------------------------------------

ni prf sim@(sim-app {t₁ = f₁} {f₂} {t₂ = x₁} {x₂} simƛ simᵣ) βƛ
                with sit _ _ sim | sit _ _ simƛ | sit _ _ simᵣ
... | refl | refl | refl        with simƛ
... | sim-lock ext _ _ = ⊥-elim (¬■-■ prf ext)
... | sim-lam {t = b₁} {b₂} sim∘ with sit _ _ sim∘
... | refl = b₂ [ x₂ ]
      , βƛ
      , ius (¬■, prf) prf b₁ b₂ (sub-refl • x₁) (sub-refl • x₂) sim∘ (simσ-• simσ-refl simᵣ)

ni prf sim@(sim-app {t₁ = l₁} {l₂} {t₂ = r₁} {r₂} simₗ simᵣ) (ξappl step)
             with sit _ _ sim | sit _ _ simₗ | sit _ _ simᵣ
... | refl | refl | refl with ni prf simₗ step
... | l₂′ , βl₂ , ind      with sit _ _ ind
... | refl = l₂′ • r₂
      , ξappl βl₂
      , sim-app ind simᵣ

ni prf sim@(sim-app {t₁ = l₁} {l₂} {t₂ = r₁} {r₂} simₗ simᵣ) (ξappr step)
```

```
                              with sit _ _ sim | sit _ _ simₗ | sit _ _ simᵣ
  ... | refl | refl | refl with ni prf simᵣ step
  ... | r₂′ , βr₂ , ind       with sit _ _ ind
  ... | refl = l₂ • r₂′
             , ξappr βr₂
             , sim-app simₗ ind
```