CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Type System for Purpose Limitation

Master's thesis in Computer science and engineering - Algorithms, Languages and Logic

Chengyuan Wang

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Master's thesis 2022

A Type System for Purpose Limitation

Chengyuan Wang

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A Type System for Purpose Limitation

Chengyuan Wang

Supervisor: Sandro Stucki, Department of Computer Science and Engineering
Examiner: David Sands, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

A Type System for Purpose Limitation

Chengyuan Wang
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Purpose Limitation is a GDPR [1] principle that restricts software to only collect and process personal data for specified purposes to which the user consented. However, there are few works that implement automatic purpose limitation check in software. Fortunately, there is a similar policy called confidentiality with lots of previous works. In practice, to ensure this principle, the policy checks are usually implemented by an Information-Flow Control (IFC) framework. This technique restricts how data flows within the software to prevent policy violations. Previously, Stefan et al. have done a series of works to use IFC to ensure confidentiality and present a concrete implementation as a Haskell Library [2, 3]. In this project, we present a static IFC system as a type system which checks the purpose limitation statically. Additionally, we embed this type system into Haskell and formalize it in Agda. The former implementation presents a concrete example of how our type system would behave in an industrial language, while the latter proves our type system's correctness.

Keywords: Type system, Security & Privacy, Information-flow control, Noninterference

Acknowledgements

I would like to thank my supervisor Sandro Stucki for his patience and guidance through the project. I would also like to thank Mr. Yinjian Liu for his love. Lastly, I want to thank my family and friends for their support and encouragement.

Chengyuan Wang, Gothenburg, August 2022

Contents

List of Figures

1
Introduction

Nowadays, computer software plays a vital part in our society. On the one hand, it brings great convenience to our daily life. On the other hand, it increases the risk of personal privacy leaks and leads to a growing concern about protecting personal privacy.

This concern results in the adoption of purpose limitation, a privacy protection principle of the General Data Protection Regulation (GDPR) [1]. Under GDPR, purpose limitation restricts personal data from being collected and processed for specified purposes to which the user consented. For example, social media such as Facebook collect and analyze personal data, such as IP addresses and email addresses, for personalized advertisements. However, with purpose limitations, these personal data can only be used for necessary identification, such as login, unless the user agrees with further processing.

To implement a purpose limitation check for software, an essential part is to implement regulations to data structures in code. Fortunately, similar work has been done in the security field for confidentiality. Confidentiality is a security policy which protects sensitive data from leaking. For example, a password should not flow to a public channel such as the standard output.

In previous work, the confidentiality policy was modeled as a security lattice, indicating different security levels in the software. Subsequently, the policy check is ensured via Information-Flow Control (IFC) [4], a technique that labels all the data with a security label and traces them through execution, alerting the programmer when policy violation occurs. Since both confidentiality and purpose limitation are policies restricting how the software uses certain data, security and purpose label behaviour shall have some similarities. Based on this, we can learn from the previous works in the security field and build our own IFC framework for purpose limitation.

The IFC framework often comes in two flavours, dynamic and static. The dynamic approach checks policy during runtime and is much easier to implement. The static approach does the policy check during compile time and needs extra structures to embed policy labels to data and computations. To avoid runtime overhead introduced by the dynamic approach, we implement the static IFC approach by building a new type system.

Overall, we build a static IFC framework for purpose limitation check and embed this framework into a type system. Then, the type checker can check whether the purpose limitation has been obeyed. Additionally, proof of correctness is given to ensure our type system does the right thing. The overall pipeline is shown in Figure 1.1.

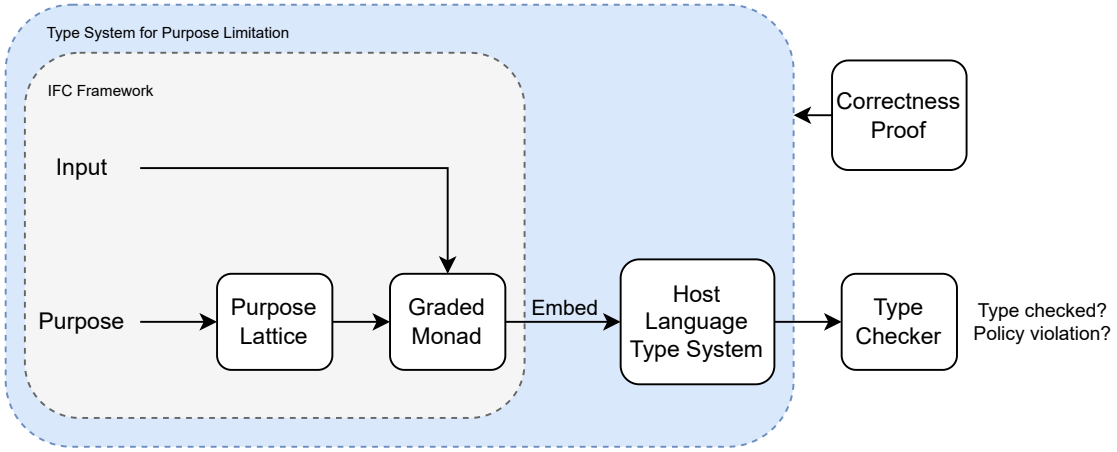

Figure 1.1: Overall pipeline of this project

1.1 Research Questions

This project aims to present a static purpose limitation check via a type system and address the following research questions.

1. What kind of structure are we using to abstract purpose? How can we embed this purpose label with data and computation statically?
2. How would the type system for purpose limitation behave in an industrial language? How would this new feature affect the usability of the host language?
3. How can we ensure the correctness of the presented type system?

1.2 Main Contribution

The main contribution of this project is divided into the following points.

1. We model purpose limitation with data and computation using a mathematical concept called Graded Monads [5, 6]. (Chapter 3)
2. We encode the purpose label as a side effect in Haskell and write an example program as a case study to evaluate our approach. (Chapter 4)
3. We mechanize the formal semantics of our type system in Agda, and come up with a proof of noninterference to show the correctness of our type system. Additionally, we extend the semantics of our type system to support the write operation. (Chapter 5)

At the end of this thesis, we also included a discussion about ethical issues in this project in Chapter 7.
The full version of the Haskell microbenchmark and the proof of noninterference in Agda can be found in the Appendices A.2 and A.1, respectively.

2
Background

In this Section, we provide the necessary background for understanding this project. In Section 2.1 we discuss about confidentiality and purpose limitation and discuss the similarity and differences between these two policies. Section 2.2 includes the main idea of IFC and how it is related to purpose limitation. Section 2.3 introduces the graded monads, a mathematical concept that we use to encode the purpose label in our type system.

2.1 Confidentiality and Purpose Limitation

Confidentiality is a security policy that protects data from unauthorized viewers. Usually, the authorization will be abstracted as security levels. For example `L` for a public viewer and `H` for an admin viewer. An admin account can see all security data and public data in the system, while the public viewer can only see the public data.
Intuitively, we can see that there exists a partial order between different security labels [7]. In previous work, this partial order was abstract as a lattice with partial order $\sqsubseteq$. Admin viewer `H` can see all public data with level `L` ($\texttt{L} \sqsubseteq \texttt{H}$), but not vice versa. For example, the function that writes directly to the standard output channel will have a low-security level `L`. When it tries to print a secured data with a high-security level `H`, there would be a policy violation because label `H` can not flow to label `L` ($\texttt{H} \not\sqsubseteq \texttt{L}$). More advanced, we can have multi-level security labels ($\texttt{H}_{1...n}$) to represent the security data for each independent user and draw a Hasse diagram to describe the relationship between these labels as shown in Figure 2.1.
A Hasse diagram illustrates a finite partially ordered set. Each vertex represents an element in the set, the lines between vertexes represent the partial order $\sqsubseteq$. The partial order always goes upwards, which means the elements live in the upper level "larger than" the elements in the lower level. In Figure 2.1, we have three different users who have their own secure data with security level from $\texttt{H}_1$ to $\texttt{H}_3$, a public level

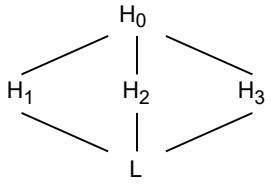

Figure 2.1: Security label in Confidentiality

$\texttt{L}$ and an administrator with level $\texttt{H}_0$ can see all data in this system.

On the other hand, purpose limitation is a privacy policy that ensures that private data can only be processed for a specific purpose to which the user consented. For example, in a login process with purpose `Login`, the user IP address is usually allowed to use to verify potential account hacking. If the login IP is not the same as the previous one, an additional identification email will be sent to the user's email address. Here, we mark this usage of data as purpose `Verify`, so the procedure with purpose `Login` can access the user IP address ($\texttt{Login} \sqsubseteq \texttt{Verify}$). However, this IP address is also often used for location-based advertisement. Here, we call this purpose `Marketing`, which the user does not allow. With purpose limitation, any use of IP address other than login verification results in a policy violation ($\texttt{Marketing} \not\sqsubseteq \texttt{Verify}$).

2.2 Information-Flow Control and Noninterference

As described above, the policy violation is specified by the partial order between policy labels; IFC uses certain mathematical concepts to attach these labels to data and computation, then tracks how the data propagates through execution to ensure that no policy violation occurs. In practice, information flow control usually comes in two flavours, dynamic and static. The dynamic approach can support richer policies while introducing more runtime overhead. The static method has less runtime overhead but is harder to implement. Hendin and Sabelfeld present an excellent survey paper [4] to introduce the motivation of IFC and how it ensures confidentiality in a theoretical aspect. Furthermore, they include an introduction to the basic concept of noninterference.

One of the mathematical concepts people are using in the security field to attach labels is the Dependency Core Calculus (DCC) [8]. In DCC, the security label is considered a dependency on the datatype. Once the data is attached with a security label l means this piece of data is protected at security level l. Moreover, DCC checks whether the data flow follows the allowed direction in the binding operator.

Subsequently, Stefan and Russo have presented the labeled IO monads (LIO) as a dynamic IFC library in Haskell [2], the security labels are attached to the data via a state monads. In the static aspect, MAC [9] produces a static IFC framework in Haskell and includes more discussion about advanced programming features such as exceptions and concurrency. Similarly, Hybrid LIO (HLIO) [3] and Lifty [10] have also encoded the security labels in the Haskell type system for a static policy check, the former uses the Haskell type family [11] while the latter uses the liquid type [12] via Liquid Haskell [13].

Noninterference is a semantics notion in the security field that demands public output does not depend on secret input [14]. While in privacy policies like purpose limitation, the definition of noninterference shall vary, we will talk more about this in Section 5.2.

$$return :: A \to M\ I\ A$$
$$>>= :: M\ l_1\ A \to (A \to M\ l_2\ B) \to M\ (l_1\ \bullet\ l_2)\ B$$

Figure 2.2: Basic monadic functions in graded monads

2.3 Effect Systems and Graded Monads

The effect systems [5] indicates all computational effects in a software and monads is a mathematical concept as well as a tool in functional programming language to abstract effects. However, monads only provides a binary view: effectful or pure, and the effect systems provides more fine-grained information about effects [15]. To combine these two concepts and provide functional programming with a more powerful tool to abstract effects, the graded monads [15, 6] is presented.
Unlike DCC, which regards the policy labels as dependencies and adds additional constraint to the binding operator, graded monads regard the policy labels as computational effects and model them as a partially ordered monoid $(F, I, \bullet, \sqsubseteq)$, where I represents pure computation, F is the set of effects, $\bullet$ is the composition between labels, and $\sqsubseteq$ is the partial order operator between labels.
When using the graded monad in an IFC framework, the $\bullet$ operator will produce the least upper bound between two labels. In this way, the bind operator $>>=$ always computes data from the lower level in the Hasse diagram to upper or equal level and makes the data flow always satisfies the policy.
In this project, we will use graded monads to encode our purpose label.

3
Modeling Purpose Limitation

In this section, we will discuss how we model the purpose limitation similarly to confidentiality in previous work. Section 3.1 first describes how we model the purpose label as a lattice and the differences between purpose labels and security labels. In Section 3.2, we model the purpose labels as a computational effect and encode them in graded monads.

3.1 Purpose as Lattice

As described in Section 2.1, similarly to confidentiality, purpose limitation is also a policy that restricts how the software uses the data. The "purpose" can also be modeled as a lattice based on this. Unlike security labels in confidentiality, purpose labels in purpose limitation indicate the purpose for which the data can be used. Use the previous login process as an example, and we can draw a Hasse diagram to make this clear.
In confidentiality, each label represents a security level, the most strict label that contains all labels has the most strict security level and lives at the top of the Hasse diagram (Figure 2.1). However, the powerset of label turns up side and down for purpose label. In Figure 3.1, each purpose label represents the purpose for which the attached data can be used. In this case, the most strict label shall be the empty set `Nil`, which means that the attached data cannot be used for any purpose. Subsequently, we defined all the basic functionalities in the system, like read database and send email as singleton labels indexed with a unique natural number; we called them basic purpose labels. A more complex purpose label shall

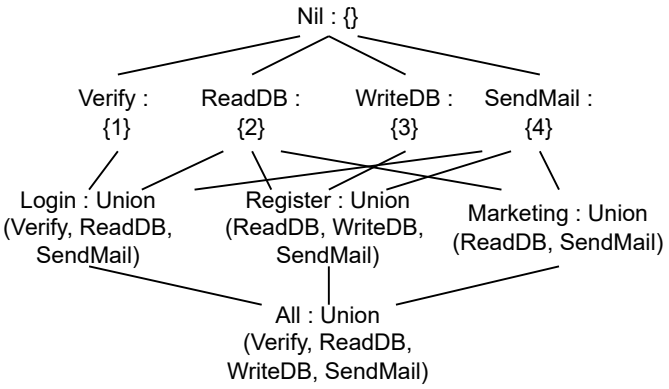

Figure 3.1: Purpose label in Purpose Limitation

be the union of multiple basic purpose labels. At the bottom of the Hasse diagram is the purpose `All`, which contains all basic purposes in the system, and represents that the attached data can be used for all purposes.
Additionally, to ensure purpose limitation, the data can only flow from a more general purpose to a more strict purpose. In other words, it flows from the bottom of the Hasse diagram to the top. Therefore, our purpose lattice is a bounded join-semilattice, a partially ordered set with a join operator to produce the least upper bound of two elements, and was bounded by a minimal element (in our case, the purpose `All`).

3.2 Purpose as Effects

After modeling the purpose as a lattice, we need to go one step further to embed them into computations so that the IFC framework can trace the labels during execution.
For all pure computations, they are nothing more than a lookup table and have not really used the data for any purpose, so we tag them with the most general purpose `All`. For effectful computations, we need to ask the graded monads for help. In the basic definition of graded monads, the effects have two related operators; one is the composition operator $\bullet$, and the other is the partial order relation $\sqsubseteq$. Since we define our purpose label as a lattice, and more general label contains more elements. The composition operator $\bullet$ will be our join operator and behave the same as the set intersection. Meanwhile, the partial order relation $\sqsubseteq$ checks whether the right-hand side is a subset of the left-hand side.

4

Haskell Library for Purpose Limitation

In this chapter, we present a Haskell approach for our purpose limitation check as well as a microbenchmark to evaluate our approach. Section 4.1 introduces the Haskell approach of graded monads. In Section 4.2, we embed a type level purpose limitation check into the Haskell type system. A registration program is presented as a micro-benchmark in Section 4.3 to evaluate our approach. The code in this chapter is open sourced on Github (https://github.com/ericwang385/DAT085/tree/poly).

4.1 Graded Monads in Haskell

The graded monads introduces a formal way to embed the effect systems into monad [6]. Based on this, Orchard and Petricek have successfully implemented the graded monads in Haskell [15]. Figure 4.1 shows their definition of the graded monads in Haskell.

In this work, they build the effect composition • as a type level operator via Haskell type family `Plus`. The `Unit` type is label for pure computation, and `Inv` is a Haskell type level constraint to add additional restriction to the instance of `Effect`. By default it is empty.

In our project, we will follow the step of Orchard and Petricek and build our own graded monads for purpose limitation.

```
class Effect (m :: k -> Type -> Type) where
   type Unit m :: k
   type Plus m (f :: k) (g :: k) :: k
   type Inv m (f :: k) (g :: k) :: Constraint

   return :: a -> m (Unit m) a
   (>>=) :: (Inv m f g) => m f a -> (a -> m g b) -> m (Plus m f g) b
   (>>) :: (Inv m f g) => m f a -> m g b -> m (Plus m f g) b
```

Figure 4.1: The graded monads in Haskell

```
type All         = Set '[Natural 0]

type SendMail    = Set '[Natural 1]
type ReadDB      = Set '[Natural 2]
type WriteDB     = Set '[Natural 3]
type Verify      = Set '[Natural 4]

type Login = Union ReadDB (Union SendMail Verify)
type Marketing = Union ReadDB SendMail
type Register = Union ReadDB (Union WriteDB (Union SendMail Verify))
```

Figure 4.2: Purpose label declaration

```
newtype Labeled p a = MkLabeled { val :: a }

instance Effect Labeled where
  type Unit Labeled = All
  type Inv Labeled a b = (IsSet a, IsSet b)
  type Plus Labeled a b = Join a b

  return = MkLabeled
  (MkLabeled a) >>= f = MkLabeled (val (f a))

sub :: (CanFlowTo p1 p2) => Labeled p1 a -> Labeled p2 a
sub (MkLabeled a) = MkLabeled a

tag :: a -> Labeled p a
tag = MkLabeled
```

Figure 4.3: Labeled data type in our Haskell approach

4.2 Purpose Limitation in Haskell

As stated in Section 3.1, the purpose labels are actually a partially ordered set. To embed this definition into Haskell type system, we use a type level set [15] to define our purpose label. As shown in Figure 4.2, each basic purpose were indexed with a unique natural number. More complex purpose label was formalized as the union of multiple basic purposes via a type level operator `Union`.

Meanwhile, the `All` purpose is hard to define. Since our library allows the user to define their own purpose label and we do not know what kind of purpose will be in the label `All`. To solve this, we treat this label as a special case and use the natural number 0 to index the label `All`. In the following definition of the partial order relation shown in Figures 4.4 and 4.5, we also give the label `All` a special case.

For the next step, we can define our `Labeled` type as an instance of the graded monads type class `Effect`. As shown in Figure 4.3, the effect for pure computation

```
type Intersection s t = Dup (Sort (s :++ t))

type family Dup t where
    Dup '[]           = '[]
    Dup '[e]          = '[]
    Dup (e ': e ': s) = e ': Dup s
    Dup (e ': f ': s) = Dup (f ': s)

type family Join a b where
  Join All (Set b) = Set b
  Join (Set a) All = Set a
  Join (Set a) (Set b) = Set (Intersection a b)
```

Figure 4.4: Join operator and type level set intersection

```
type family CanFlowTo a b :: Constraint where
  CanFlowTo All a = a ~ a
  CanFlowTo (Set a) (Set b) = Union a b ~ a
```

Figure 4.5: Type level partial order in graded monads

is `All`, and the constraint `Inv` to ensure that two purpose labels in the bind operator are all type level sets. The effect composition is defined as a `Join` operator, which is a type level set intersection, it contains a type level quicksort function `Sort` and a type family `Dup` to collect the duplicate elements in the set (Figure 4.4). The `sub` function is the lifting operation for our labeled types. It lifts the purpose label from a more general purpose to a more strict one. The type family `CanFlowTo` (Figure 4.5) ensures this flow is always in the correct direction, the operator means the type equality in Haskell. The function `tag` is only defined for mocked test data in the microbenchmark.

4.3 Case Study

In this section, we present a registration program as a case study to evaluate our Haskell approach, noticing in our Haskell example program, we use the bind operator >>= explicitly rather than the do notation. This mostly because the bind for graded monad rise a conflict with the original definition in Haskell, this .
The basic functionality of this registration program is as follows:

- *User Registration:* add a new user and its password to the database.
- *Existing User Detect:* detect whether the provided username is already in the database.
- *Send Advertisements:* After registration, send an advertisement to the user email without using the IP address.

To fulfill these functions, we use the same hierarchy as shown in Figure 3.1 for our purpose labels in this case study. Then we can reuse the purpose label definition in

```
sendAds :: Labeled Marketing Bool
sendAds = usermail >>= \mail -> userIP >>= \ip -> tag True
```

Figure 4.6: Ill-typed function in purpose limitation

Figure 4.2 and present the main registration process as an example.
In Figure 4.7, we define a function `register` for the registration process. Noticing that the purpose of the return value is the empty set purpose `Nil`. Because in this microbenchmark, there is no extra call for function `register`, which means its result shall not be used for any purpose. In a real-world registration program, there should be a more strict purpose like `ShowResult`, then the result of `register` will have a nonempty purpose to flow to.
Returning to the microbenchmark, the `register` function first checks whether the user already exists with a `searchDB` function; if it exists, then it returns false; otherwise, it updates the database and sends an email without using the userIP. Unfortunately, this example program does not support mutable reference and IO action yet, so the function `searchDB` is a trivial function always return True. Here we only show a fragment of the benchmark; the complete version is listed in Appendix A.2.
Since the userIP has purpose `Verify`, which means it can not be used for sending location based advertising emails. If we try to write code as in Figure 4.6, the Haskell typechecker will throw an error because the label `Join Verify SendMail` is not compatible with the label `Marketing`.

```
type DB = Labeled All
    [(Labeled Register String, Labeled Login String)]

username :: Labeled Register String
username = tag "TestName1"

usermail :: Labeled Login String
usermail = tag "Testmail1"

userIP :: Labeled Verify String
userIP = tag "TestIP"

password :: Labeled Login String
password = tag "TestPassword"

database :: DB
database = tag [(username, password)]

register :: Labeled (Set '[]) Bool
register = do
            ans <- userExist username
            if ans then tag False :: Labeled (Set '[]) Bool
            else do
                _ <- updateDB database username password
                ans3 <- sendmail usermail
                if ans3 then tag True :: Labeled (Set '[]) Bool
                else tag False :: Labeled (Set '[]) Bool

userExist :: Labeled Register String
            -> Labeled Verify Bool

updateDB :: DB -> Labeled Register String
            -> Labeled Login String -> DB

sendmail :: Labeled p String -> Labeled SendMail Bool
```

Figure 4.7: Well-typed function in purpose limitation

5

Correctness

In Section 5.1, we first come up with the formal syntax for our type system in Agda. Then we discuss multiple ways of interpreting the syntax and present a formal semantics of our type system using denotational semantics. In Section 5.3, we further extend this formal syntax to interact with output channels. In Section 5.2 various interpretations of noninterference in purpose limitation are presented, we choose two of them and give a proof in Agda. The code in this chapter is open sourced on Github (https://github.com/ericwang385/Purpose-Limitation).

5.1 Purpose Limitation in Agda

5.1.1 Syntax

Agda is a dependently typed functional programming language. Based on the Curry–Howard correspondence, Agda can formalize a proof as a program written in a functional programming style. Data types in Agda are usually introduced as generalized algebraic datatype (GADT). Each function in Agda will be regarded as a mathematical proof.

To formalize our type system in Agda, we first need to define the basic syntax. `Context` is defined as a snoc list of types, and all variable inside the `Context` is represented by the De Bruijn index ∋, which represents terms of lambda calculus without naming the bound variables [16]. In the meanwhile, the syntax of types contains natural numbers, booleans, functions, and a labeled data type (Figure 5.1 and 5.2). In Agda, we can separate a constructor with _, this is called the Mixfix operator [17]. For example, in the function type ⇒, the following representations are equal:

```
_⇒_ a b
(a ⇒_) b
a ⇒ b
```

The labeled data type represents data with a purpose label. In the Agda standard library, there is a data structure called BoundedJoinSemiLattice, which is the exact definition of our purpose label presented previously. With these definitions, we can define terms as shown in Figure 5.3.

Other than basic terms such as `lit` for natural number and `true` for boolean, we also have a `unit` term to represent the unit type. The η and $\uparrow$ terms are related to the purpose label. η creates a labeled data term with the bottom label. $\uparrow$ takes an allowed flow and lift the purpose label to a more strict one.

```
infixl 5 _,_
data Ctx : Set c where
  ∅   : Ctx
  _,_ : Ctx → Type → Ctx

infix  4 _∋_
data _∋_ : Ctx → Type → Set where
    Z  : ∀ {Γ A} → Γ , A ∋ A
    S_ : ∀ {Γ A B} → Γ ∋ A → Γ , B ∋ A
```

Figure 5.1: Context in our type system

```
data Type : Set c where
    Unit : Type
    Nat  : Type
    Bool : Type
    _⇒_  : (a b : Type) → Type      -- Function Type
    ⟨_⟩_ : Label → Type → Type      -- Labeled data with purpose
```

Figure 5.2: Basic types in our type system

```
infix  4 _⊢_
data _⊢_ (Γ : Ctx) : Type → Set (c ⊔ ℓ₂) where
    unit           : Γ ⊢ Unit
    true           : Γ ⊢ Bool
    false          : Γ ⊢ Bool
    lit            : ℕ → Γ ⊢ Nat
    case_of[zero⇒_|suc⇒_] : Γ ⊢ Nat → Γ ⊢ a → Γ , Nat ⊢ a → Γ ⊢ a

    var_           : Γ ∋ a → Γ ⊢ a
    ƛ_             : Γ , a ⊢ b → Γ ⊢ (a ⇒ b)
    _•_            : Γ ⊢ (a ⇒ b) → Γ ⊢ a → Γ ⊢ b
    plus           : Γ ⊢ Nat → Γ ⊢ Nat → Γ ⊢ Nat
    If_Then_Else_  : Γ ⊢ Bool → Γ ⊢ a → Γ ⊢ a → Γ ⊢ a

    η_             : Γ ⊢ a → Γ ⊢ ⟨ ⊥ ⟩ a
    _↑_            : l₁ ⊑ l₂ → Γ ⊢ ⟨ l₁ ⟩ a → Γ ⊢ ⟨ l₂ ⟩ a
    label          : (l : Label) → Γ ⊢ a → Γ ⊢ ⟨ l ⟩ a
    Let⇐_In_       : Γ ⊢ (⟨ l₁ ⟩ a) →
                     Γ , a ⊢ ⟨ l₂ ⟩ b → Γ ⊢ ⟨ l₁ ∘ l₂ ⟩ b
```

Figure 5.3: Terms in our type system

```
Value : Type → Set v
Value Nat        = ℕ
Value Bool       = B
Value Unit       = ⊤
Value (a ⇒ b)    = Value a → Value b
Value (< l > a) = M l (Value a)
```

Figure 5.4: Mapping from our Type System to Agda Type

5.1.2 Semantics

Usually, there are two choices for formal semantics; one is operational semantics and the other is denotational semantics [18].
The operational semantics regard the evaluation as a sequence of operations mapping an input term to an output term. Usually, people first define the reduction between two different terms as the small-step semantics, and often a big-step semantics maps from a term directly to the value. The small-step semantics illustrates each computation step in a program, while the big-step semantics assumes there is an underlying small-step semantics and ensures the program terminates. For the denotational semantics, it translates the syntax directly into mathematical objects. In this project, we choose to prove the noninterference through a distinguishability relation, where only the computation with the right purpose can "see" the data. In this case, there is not much difference between a denotational semantics and a big-step operational semantics. Additionally, Agda provides a termination check so that our denotational semantics ensures termination for free.
In our case, the mathematical object for the denotational semantics is the Agda type system. While the labeled data type need to map into a graded monads structure as shown in Figure 5.5. We define the mapping from our type system to the Agda type system using a `Value` function. For `Nat`, `Bool` and `Unit` we simply map them to the corresponding Agda type. The function type from `a` to `b` is formed as an Agda function from `Value a` to `Value b`. The type of labeled data is defined by our graded monads in Agda (Figure 5.5).

```
record GMonad : Set (c l⊔ ℓ₂) where
    field
        M : Label → Set → Set
        return : A → M ⊥ A
        _>>=_ : M l₁ A → (A → M l₂ B) → M (l₁ ∘ l₂) B
        sub : l₁ ⊑ l₂ → M l₁ A → M l₂ A
```

Figure 5.5: Graded monads in Agda

To evaluate our formal semantics in Agda, we also need an environment to store all the local variables (Figure 5.6). The `lookupVar` taken a environment and a De Bruijn index as input, produces the value stored in the environment. Then we can have our evaluation process taking a term and an environment to produce a value

```
data Env : Ctx → Set v where
    ∅   : Env ∅
    _,_ : Env Γ → Value a → Env (Γ , a)

lookupVar : Env Γ → Γ ∋ a → Value a
lookupVar (ρ , v) Z     = v
lookupVar (ρ , v) (S x) = lookupVar ρ x
```

Figure 5.6: Environment in Agda

`eval : Γ ⊢ a → Env Γ → Value a`
Here, we only give the type signature of `eval`, the complete version is listed in Appendix A.1

5.2 Noninterference proof

Non-interference is a concept originating from the security field. Its original version says "The public output of the program must not depend on sensitive inputs", which ensures that the sensitive input will not leak [14].
In a privacy policy like purpose limitation, this definition can be translated as "Input with specific purpose must produce outputs with a more specific purpose" this sentence can also be interpreted differently. On the one hand, we can say "For all computations, erasing the inputs that are invisible to the current observer before/after execution should produce the same output". On the other hand, we can also say, "For all computations, during execution only data with purpose label that is visible to the current observer can be distinguished". These two different interpretations give us two possible ways to prove noninterference.
The first interpretation was based on the term erasure method [19] and was used in a series of articles on information flow libraries [20, 2, 3, 21]. The technique uses a separate erasure function on terms, which essentially rewrites the data according to the user's security level u. If the user level is lower than the data security level p, the function will erase the sensitive data to a trivial value (in the equation, use $\top$ to represent).

$$M_p A = \begin{cases} A, & p \sqsubseteq u \\ \top, & p \not\sqsubseteq u \end{cases} \tag{5.1}$$

Usually, operational semantics often use this approach by combining the separate erasure function with the small-step semantics. In our denotational semantics approach, the rewrite step is incorporated into the graded monads rather than written as a separate function. Gaboardi et al. show a way to combine the graded monads with the erasure step [22]. A discussion of the relation between using a separate erasure function and embed the erasure step in the evaluation step will be regard as a future work for our project.
The condition in Equation 5.1 can be regarded as either a logic implication (Figure

```
M : Label → Set ℓ₂ → Set ℓ₂
M l a = l ⊑ u → a
```

Figure 5.7: Erasure method as a logic implication

```
M : Label → Set ℓ₂ → Set ℓ₂
M l a with (l ⊑? u)
...   | yes p = a
...   | no ¬p = ⊤
```

Figure 5.8: Erasure method as a decidable relation

5.7) or pattern matching with a decidable relation. (Figure 5.8). Here we only show the monad constructor, the complete version can be found in Appendix A.1.

For the second interpretation, the idea comes from Reynolds' concept of parametricity [23]. The method can be easily expressed as the distinguishability relation R. If the user level u is lower than the data security level p, it will return a trivial relation; otherwise, it returns a relation on the underlying type A without the label. A concrete approach that uses this relation is the Dependency Core Calculus (DCC) [8]. In this work, Abadi et al. present a way to combine this distinguishability relation with monads. This led to a series of papers that embed this distinguishability relation in Haskell using DCC [24, 25, 3].

$$R_{\langle p \rangle} A = \begin{cases} R_A, & p \sqsubseteq u \\ |A| \times |A|, & p \not\sqsubseteq u \end{cases} \tag{5.2}$$

In this case, the graded monad is simply an identity monads, and the basic type without labels (natural number/boolean) will produce an equivalence relation. For function type, if the input is visible, then the output should also be visible. For the labeled type, it can be formalized as a logic implication as shown in Figure 5.9.

To formalize the noninterference proof for both erasure based and relational-based interpretation, we can say – for the current viewer, given a term and two related environments, our type system shall produce two values in the same relation. Since we already have a distinguishability relation on `Value`, with a environment distinguishability relation shown in 5.10, we can formalize the noninterference proof as follows (complete version in Appendix A.1):

```
[_]_~_ : (a : Type) → Rel (Value a) (c l⊔ ℓ₂)
[ ⟨ l₁ ⟩ t ] x ~ y = l₁ ⊑ u  → ([ t ] x ~ y)
[ a ⇒ b ] f ~ g = ∀ {x y : Value a}  → [ a ] x ~ y → [ b ] f x ~ g y
[ t ] x ~ y = x ≡ y
```

Figure 5.9: Distinguishability relation

```
⟨_⟩_~_ : (Γ : Ctx) → Rel (Env Γ) ℓ₂
⟨ ∅ ⟩ ea ~ eb  = ⊤
⟨ ctx , a ⟩ (ea , va) ~ (eb , vb)
    = (⟨ ctx ⟩ ea ~ eb) × ([ a ] va ~ vb)
```

Figure 5.10: Environment equivalence

```
noninterference : {t : Type} → (term : Γ ⊢ t)
    → (env₁ env₂ : Env Γ) → ⟨ Γ ⟩ env₁ ~ env₂
    → [ t ] (eval term env₁) ~ (eval term env₂)
```

Figure 5.11: Noninterference statement

5.3 IO extension

To enrich the syntax of our type system, we also add a write extension. First, we add the syntax level definition of our IO extension as shown in Figure 5.12. The labeled IO type $IO\langle\_\rangle$ models the output channel with a purpose label. In this way, we get a series of IO channels, each with a different purpose label. The `write` term represents the writing process in our type system. Due to the lack of time, the `read` term is not implemented. It takes a labeled data and an output channel with the same purpose label, producing the updated output channel.

```
IO⟨_⟩_ : Label → Type → Type

write : Γ ⊢ ⟨ l ⟩ a → Γ ⊢ IO⟨ l ⟩ a → Γ ⊢ IO⟨ l ⟩ a
```

Figure 5.12: Syntax of IO extension

Secondly, to evaluate our IO extension, on the semantics level, we define the semantics of the IO extension in Figure 5.13. The IO channel is interpreted as an Agda list of graded monads in the `Value` function. In this way, the evaluation step of `write` term would append the input value to the end of the output channel.

```
Value (IO⟨ l ⟩ a) = List (M l (Value a))

eval (write x io flow) ρ = (eval x ρ) :: eval io ρ
```

Figure 5.13: Semantics of IO extension

Lastly, for the noninterference proof, we need to define the distinguishability relation for the IO channels, which checks whether all values inside a channel are with the same relation. In the `noninterference` function, the proof is given by the noninterference propriety of the term to write and the io channel.

```
[ IO⟨ l₁ ⟩ t ] (x :: xs) ~ (y :: ys)
    = l₁ ⊑ u → ([ t ] x ~ y) × [ IO⟨ l₁ ⟩ t ] xs ~ ys

noninterference {Γ} (write term io) e1 e2 enveq
    = λ flow → noninterference {Γ} term e1 e2 enveq flow
             ,' noninterference {Γ} io e1 e2 enveq
```

Figure 5.14: Noninterference proof of IO extension

6
Related Works

6.1 IFC for Confidentiality

The IFC technique is widely used for confidentiality check to protect data from leaking. In the dynamic approach LIO [2] and the static approach MAC [9] Stefan et al. and Russo et al. have modeled the security level as a label attached to data and computations. Both works have been presented as a Haskell library. HLIO [3], as a follow up work of LIO has changed the policy check from dynamic to hybrid, which has gain some performance increase. Comparing with our Haskell example program, these approaches has more advanced features such as IO and concurrency. However, these approaches only work for confidentiality and can not extend to purpose limitation. While our project makes some trade-offs, we drop some advanced feature and reform the IFC methods in previous works to fit the purpose limitation check perfectly.
Furthermore, from a static policy check perspective, Algehed model the security level as type level labels and presents a Haskell implementation of DCC [24] and a simpler calculus, which shares the same denotation domain as DCC [25].
In our project, we also model the purpose as a label attached to data, our labeled data structure in the Haskell example program is heavily inspired by the Haskell embedding of DCC[24]. However, their policy check is done in a DCC style – regarding the security label as a status of the data and adding constraints to the bound operator to ensure the data flows in the correct order, while in our approach we regard the purpose label as a computational effect and use an effect composition operator to ensure the data always flow from a general purpose to a more strict purpose. From our perspective, the effect systems approach is more neat than the DCC approach.

6.2 Graded Monads

Wadler and Thiemann have presented a formal way to combine the computational effect with monads [5], which receives a new name as the graded monads in the following works [26]. Katsumata formalized the parametric effect monads [6] and includes a discussion of denotational semantics of graded monads and the soundness of it. To go one step further, Orchard and Petricek have embedded this formal representation into Haskell by using a type-level set to represent the effects [15]. The graded monads definition in our Haskell program is mainly based on this approach and modifies it to fit the purpose label.

6.3 Noninterference proof

For a more theoretical aspect, rather than implementing the information flow control into certain languages, people more care about high-level abstraction of this method and its correctness. Heintze and Riecke use the logical relations to proof noninterference in their work SLam [27], and we also use this proving idea in our implementation. In Hunt and Sands' work [21], they has discussed information flow control and its soundness in an untyped language, while we are presenting the soundness of a type system based on simple typed lambda calculus. Moreover, as an extension of the simple typed lambda calculus, *A Core Calculus of Dependence* (DCC) [8] interpreters policy check as a distinguishability relation to enforce the visibility between computation and data at different security levels. This approach directly inspires our distinguishability relation, however, as stated in Section 5.2, the interpretation of noninterference in confidentiality and purpose limitation are different. In our approach, we come up with our own distinguishability relation based noninterference proof for purpose limitation.
In follow-up paper of MAC by Vassena et al. [20] they formalize MAC into Agda and use computational semantics to interpret their typed language. In this work, due to the sequential steps in operational semantics, they use a separate erasure function and prove the noninterference for each evaluation step, this erasure based proof inspires our erasure-based proof for purpose limitation. However, in purpose limitation, since the interaction between labeled data and computation is the only thing that matters. We choose to use denotational semantics to interpret our language based on simple-typed lambda calculus. Moreover, rather than using a separate erasure function to map from simply typed terms to the corresponding untyped terms, we embed the erasure step into our denotational semantics. The noninterference propriety can be proved by comparing the evaluation result between two terms with the same purpose label.
In Algehed and Bernardy's work, the same as our approach, they use a denotational semantics to formalize their language, but the mathematics concept they are working with is DCC and they use parametricity to show that the noninterference is a consequence of "free theorem" [28]. Again, because of the semantics of DCC and graded monads are different, we can not use this work to prove the correctness of our type system. Gaboardi et al. use an example of their calculus as an erasure method with graded monads in the denotational semantics [22], this work is more theoretical than practical, our approach shares the same underlying spirit of noninterference in information flow, but focusing more on specific policies such as purpose limitation.

7

Discussion & Conclusion

7.1 Benchmark in Haskell

We have presented a Haskell program to evaluate our type system. It illustrates the overall behavior of static purpose limitation check via a type system. Comparing with the dynamic policy check, our approach avoids some runtime overhead, however, due to the way Haskell handling type classes and constraints, there are still remaining some runtime overhead. Additionally, there are still several shortages that need to be improved in future work. Firstly, the graded monads is not fully embedded into the Haskell monad system, so we need to write the bind operator explicitly rather than using a do notation, and this decreased the usability of our library. Secondly, the labeled mutable data structure and IO actions is not supported, which make some functions in our example trivial.

7.2 Formal Semantic in Agda

In the Agda approach, we discuss the relation between different semantics in purpose limitation. And explore various interpretation of the noninterference propriety. We believe there must be an underlying equivalence between noninterference propriety in denotational semantics and operational semantics, this would mark as a possible future work.

7.3 Ethical Issues

In this project, we provide a type system for privacy policy call noninterference. The type system itself can only perform as a framework for policy check, the policy details shall be customized by the user. During the customization, our type system can only see the purpose that the software is meant to use the data and our type system will not process or collect any other user data. Additionally, the source code is open sourced on Github (https://github.com/ericwang385/Purpose-Limitation), we believe this can benefit the whole community. With this approach, people can use it as a library in Haskell and write program that follows the purpose limitation policy, or build a static analysis tool to detect the potential policy violation.

7.4 Conclusion

In this project, we have explored a possible way to model the purpose limitation in a type system, from purpose lattice to effect system, then using a graded monad to embed the policy check into Haskell type system and build up a concrete example program. Moreover, we have presented a formal semantic in Agda for our type system. At last, we have come up with a noninterference proof to show the correctness of our type system.

Bibliography

[1] 2018 reform of eu data protection rules. European Commission. [Online]. Available: https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf

[2] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *Proceedings of the 4th ACM Symposium on Haskell*, 2011, pp. 95–106.

[3] P. Buiras, D. Vytiniotis, and A. 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*, 2015, pp. 289–301.

[4] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[5] P. Wadler, "The marriage of effects and monads," in *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, 1998, pp. 63–74.

[6] S.-y. Katsumata, "Parametric effect monads and semantics of effect systems," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014, pp. 633–645.

[7] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[8] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp. 147–160.

[9] A. Russo, "Functional pearl: two can keep a secret, if one of them uses haskell," *ACM SIGPLAN Notices*, vol. 50, no. 9, pp. 280–288, 2015.

[10] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, "Liquid information flow control," 2020.

[11] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann, "Type checking with open type functions," in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, 2008, pp. 51–62.

[12] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 159–169.

[13] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, 2014, pp. 269–282.

[14] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*. IEEE, 1982, pp. 11–11.

[15] D. Orchard and T. Petricek, "Embedding effect systems in haskell," in *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 2014, pp. 13–24.

[16] N. G. De Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem," in *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5. Elsevier, 1972, pp. 381–392.

[17] Mixfix operators. Mixfix Operators - Agda 2.6.2 documentation. [Online]. Available: https://agda.readthedocs.io/en/v2.6.2/language/mixfix-operators.html

[18] P. Wadler, W. Kokke, and J. G. Siek, *Programming Language Foundations in Agda*, Jul. 2020. [Online]. Available: http://plfa.inf.ed.ac.uk/20.07/

[19] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theoretical computer science*, vol. 411, no. 19, pp. 1974–1994, 2010.

[20] M. Vassena, A. Russo, P. Buiras, and L. Waye, "Mac a verified static information-flow control library," *Journal of logical and algebraic methods in programming*, vol. 95, pp. 148–180, 2018.

[21] S. Hunt and D. Sands, "On flow-sensitive security types," *ACM SIGPLAN Notices*, vol. 41, no. 1, pp. 79–90, 2006.

[22] M. Gaboardi, S.-y. Katsumata, D. Orchard, F. Breuvart, and T. Uustalu, "Combining effects and coeffects via grading," *ACM SIGPLAN Notices*, vol. 51, no. 9, pp. 476–489, 2016.

[23] J. C. Reynolds, "Types, abstraction and parametric polymorphism," in *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*, 1983, pp. 513–523.

[24] M. Algehed and A. Russo, "Encoding dcc in haskell," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017, pp. 77–89.

[25] M. Algehed, "A perspective on the dependency core calculus," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, 2018, pp. 24–28.

[26] A. Smirnov, "Graded monads and rings of polynomials," *Journal of Mathematical Sciences*, vol. 151, no. 3, pp. 3032–3051, 2008.

[27] N. Heintze and J. 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.

[28] M. Algehed and J.-P. Bernardy, "Simple noninterference from parametricity," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–22, 2019.

A

Appendix 1

A.1 Agda Definition

```
record Functor (F : Set v → Set v) : Set (lsuc v) where
    field
        fmap : (A → B) → F A → F B
    _<$>_ = fmap
    infixl 4 _<$>_

record GMonad : Set (c l⊔ ℓ₂ l⊔ lsuc v) where
    field
        M : Label → Set v → Set v
        return : A → M ⊥ A
        _>>=_ : M l₁ A → (A → M l₂ B) → M (l₁ ∘ l₂) B
        sub : l₁ ⊑ l₂ → M l₁ A → M l₂ A

    join : (M l₁ (M l₂ A)) → M (l₁ ∘ l₂) A
    join ma = ma >>= λ x → x

    fmap : (A → B) → M l₁ A → M l₁ B
    fmap f ma = sub (⊑-reflexive (identityʳ _)) (ma >>= λ x → return (f x))

    functor : Functor (M l)
    functor = record {fmap = fmap}

    _>>_ : M l₁ A → M l₂ B → M (l₁ ∘ l₂) B
    ma >> mb = ma >>= λ a → mb
```

Figure A.1: Graded monad definition

```
eval : Γ ⊢ a → Env Γ → Value a
eval true ρ          = B.true
eval false ρ         = B.false
eval unit ρ          = tt
eval (lit n) ρ       = n
eval case x of[zero⇒ expr1 |suc⇒ expr2 ] ρ with (eval x ρ)
...        |  ℕ.zero = eval expr1 ρ
...        |  ℕ.suc n = eval expr2 (ρ , n)

eval (var x) ρ       = lookupVar ρ x
eval (ƛ x) ρ         = λ y → eval x (ρ , y)
eval (f • x) ρ       = eval f ρ (eval x ρ)
eval (plus x y) ρ    = (eval x ρ) + (eval y ρ)
eval (If cond Then e1 Else e2) ρ with (eval cond ρ)
...        | B.true  = eval e1 ρ
...        | B.false = eval e2 ρ

eval (η x) ρ         = return (eval x ρ)
eval (flow ↑ x) ρ    = sub flow (eval x ρ)
eval (label l x) ρ   = sub ⊥-⊑? (return (eval x ρ))
eval (Let⇐ ma In mb) ρ = eval ma ρ >>= λ v → eval mb (ρ , v)

eval (write x io flow) ρ = (sub flow (eval x ρ)) :: eval io ρ
```

Figure A.2: Evaluation Steps

```
suc-injective : {ℓ : Level} {x y : ℕ} → _≡_ {ℓ} (ℕ.suc x) (ℕ.suc y) → _≡_ {ℓ} x y
suc-injective refl = refl

plus-≡ : {ℓ : Level} {a b c d : ℕ} → _≡_ {ℓ} a b → _≡_ {ℓ} c d →  _≡_ {ℓ} (a + c) (b + d)
plus-≡ refl refl = refl

noninterference : {t : Type} → (term : Γ ⊢ t)
    → (env₁ env₂ : Env Γ) → ⟨ Γ ⟩ env₁ ~ env₂ → [ t ] (eval term env₁) ~ (eval term env₂)
noninterference unit e1 e2 enveq = refl
noninterference true e1 e2 enveq = refl
noninterference false e1 e2 enveq = refl
noninterference (lit x) e1 e2 enveq = refl

noninterference {Γ ,⍰ a} (var x) e1 e2 enveq with e1 | e2 | enveq | x
... | (e1 , x) | (e2, y) | (p ,' v) | Z = v
... | (e1 , _) | (e2 , _) | (p ,' v) | (S x) = noninterference {Γ} (var x) e1 e2 p

noninterference {Γ} (case term of[zero⇒ term₁ |suc⇒ term₂ ])
                e1 e2 enveq with (noninterference {Γ} term e1 e2 enveq)
... | p  with (eval term e1) | (eval term e2)
...      | ℕ.zero | ℕ.zero = noninterference {Γ} term₁ e1 e2 enveq
...      | ℕ.suc x | ℕ.suc y = noninterference {Γ ,⍰ Nat} term₂ (e1 , x) (e2 , y) (enveq ,' (suc-injective {lzero} p))

noninterference {Γ} {a ⇒ b} (ƛ term) e1 e2 enveq {x} {y} inputeq
                = (noninterference {Γ ,⍰ a} term (e1 , x) (e2 , y) (enveq ,' inputeq))

noninterference {Γ} (_·_ {a} term term₁) e1 e2 enveq
                = (noninterference {Γ} term e1 e2 enveq) (noninterference {Γ} (term₁) e1 e2 enveq)

noninterference {Γ} (plus term term₁) e1 e2 enveq
                = plus-≡ (noninterference {Γ} term e1 e2 enveq) (noninterference {Γ} (term₁) e1 e2 enveq)

noninterference {Γ} (If term Then term₁ Else term₂) e1 e2 enveq
                with (noninterference {Γ} term e1 e2 enveq)
... | p with (eval term e1) | (eval term e2)
...      | B.true | B.true = noninterference {Γ} term₁ e1 e2 enveq
...      | B.false | B.false = noninterference {Γ} term₂ e1 e2 enveq

noninterference {Γ} (η term) e1 e2 enveq = λ _ → noninterference term e1 e2 enveq

noninterference {Γ} (flow ↑ term) e1 e2 enveq = λ x → noninterference term e1 e2 enveq (⊑-trans flow x)

noninterference {Γ} (label l term) e1 e2 enveq = λ x → noninterference {Γ} term e1 e2 enveq

noninterference {Γ} (Let←_In_ {l₁} {a} {l₂} term1 term2) e1 e2 enveq with noninterference {Γ} term1 e1 e2 enveq
... | p = λ x → noninterference {Γ ,⍰ a} term2 (e1 , eval term1 e1) (e2 , eval term1 e2) (enveq ,' p (flow1 x)) (flow2 x)
    where flow1 : {l₁ l₂ u : Label} → l₁ ∘ l₂ ⊑ u → l₁ ⊑ u
          flow1 {l₁} {l₂} x = (⊑-trans (x≤x∨y l₁ l₂) x)
          flow2 : {l₁ l₂ u : Label} → l₁ ∘ l₂ ⊑ u → l₂ ⊑ u
          flow2 {l₁} {l₂} x = (⊑-trans (y≤x∨y l₁ l₂) x)

noninterference {Γ} (write term io flow) e1 e2 enveq
    = λ x → noninterference {Γ} term e1 e2 enveq ((⊑-trans flow x)) ,' noninterference {Γ} io e1 e2 enveq
```

Figure A.3: Noninterference proof for relational based method

A.2 Haskell Case Study

```
register :: Labeled (Set '[]) Bool
register = do
            ans <- userExist username
            if ans then tag False :: Labeled (Set '[]) Bool
            else do
                _ <- updateDB database username password
                ans3 <- sendmail usermail
                if ans3 then tag True :: Labeled (Set '[]) Bool
                else tag False :: Labeled (Set '[]) Bool

login :: Labeled (Set '[]) Bool
login = do
         ans1 <- verifyIP userIP
         if not ans1 then tag False
         else do
            ans2 <- checkPass username password
            if ans2 then tag True :: Labeled (Set '[]) Bool
            else tag False :: Labeled (Set '[]) Bool

checkPass :: Labeled Register String -> Labeled Login String
            -> Labeled Verify Bool
checkPass _ pass = do
                p <- password
                p' <- pass
                if p == p' then tag True :: Labeled Verify Bool
                else tag False :: Labeled Verify Bool

sendmail :: Labeled Login String -> Labeled SendMail Bool
sendmail ma = do
            _ <- ma
            tag True :: Labeled SendMail Bool

userExist :: Labeled Register String -> Labeled Verify Bool
userExist name = do
                ans <- searchDB name
                if ans then tag True :: Labeled Verify Bool
                else tag False :: Labeled Verify Bool

verifyIP :: Labeled Verify String -> Labeled Verify Bool
verifyIP ma = ma >>= \a -> tag True :: Labeled Verify Bool

searchDB :: Labeled Register String -> Labeled Login Bool
searchDB _ = tag True :: Labeled Login Bool

updateDB :: DB -> Labeled Register String -> Labeled Login String -> DB
updateDB db name pass = do
                    rawdb <- db
                    tag (rawdb ++ [(name, pass)]) :: DB
```

Figure A.4: Example code in Haskell