Paulo Jorge Abreu Duarte Ferreira

Licenciado em Engenharia Informática

Information Flow Analysis using Data-dependent Logical Propositions

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Co-orientadores :	João Costa Seco, Prof. Auxiliar, Universidade Nova de Lisboa
	Carla Ferreira, Prof^{a}. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Doutor João Alexandre Carvalho Pinheiro Leite
Arguente:	Prof. Doutor Simão Melo Patrício de Sousa
Vogal:	Prof. Doutor João Ricardo Viegas da Costa Seco

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2012

Information Flow Analysis using Data-dependent Logical Propositions

Acknowledgements

I would like to thank my thesis' advisers, professor João Seco and professor Carla Ferreira, for providing me with an opportunity to explore an interesting research area, and for their guidance and thorough comments during the elaboration of this thesis.

I would like to thank the members of the PLASTIC research group for their interesting talks and insightful comments that helped dispel some misconceptions in the earlier versions of this work.

I would like to thank my friends Carlos Correia, Diogo Serra and Sérgio Silva, for the insightful discussions throughout the development of this thesis and for sharing their enthusiasm in their own theses and personal projects.

A special thank you to my parents and brothers for their continuous support and inspiration, as well as providing me plenty of opportunities to take a break and get a breath of fresh air :)

Abstract

A significant number of today's software systems are designed around database systems that store business information, as well as data relevant to access control enforcement, such as user profiles and permissions. Thus, the code implementing security mechanisms is scattered across the application code, often replicated at different architectural layers, each one written in its own programming language and with its own data format. Several approaches address this problem by integrating the development of all application layers in a single programming language. For instance, languages like Ur/Web and LiveWeb/λ_{DB} provide static verification of security policies related to access control, ensuring that access control code is correctly placed. However, these approaches provide limited support to the task of ensuring that information is not indirectly leaked because of implementation errors.

In this thesis, we present a type-based information-flow analysis for a core language based in λ_{DB}, whose security levels are logical propositions depending on actual data. This approach allows for an accurate tracking of information throughout a database-backed software system, statically detecting the information leaks that may occur, with precision at the table-cell level. In order to validate our approach, we discuss the implementation of a proof-of-concept extension to the LiveWeb framework and the concerns involved in the development of a medium-sized application in our language.

Keywords: programming language, static verification, security policies, information-flow analysis, type system, data manipulation primitives

Resumo

Um número significativo dos sistemas de software de hoje em dia são suportados por sistemas de base de dados que armazenam tanto informação relevante para o domínio da aplicação, como dados relevantes para a aplicação do controlo de acessos, tais como perfis de utilizador e permissões. Assim, o código que implementa os mecanismos de segurança está espalhado pelo código da aplicação, frequentemente replicado em diferentes camadas arquitecturais, cada uma delas escrita na sua própria linguagem de programação e com o seu próprio formato de dados. Existem várias abordagens que resolvem este problema através da integração do desenvolvimento de todas as camadas numa única linguagem de programação. Por exemplo, linguagens como Ur/Web e LiveWeb/λ_{DB} fazem verificação estática de políticas de segurança relacionadas com o controlo de acessos, garantindo que o código de controlo de acessos está onde é necessário. No entanto, estas abordagens fornecem um suporte limitado à tarefa de garantir que não são causadas fugas de informação devido a errors de implementação.

Nesta tese, apresentamos uma análise de fluxo de informação baseada em tipos para uma linguagem adaptada a partir de λ_{DB}, usando como níveis de segurança proposições lógicas que dependem dos próprios dados. Esta abordagem permite uma monitorização precisa da informação num sistema de software suportado por bases de dados, detectando estaticamente as fugas de informação que podem ocorrer, com precisão ao nível das células de uma tabela. De forma a validar a nossa abordagem, discutimos a implementação de uma extensão à framework LiveWeb como prova de conceito e as preocupações envolvidas no desenvolvimento de uma aplicação de dimensão média na nossa linguagem.

Palavras-chave: linguagem de programação, verificação estática, políticas de segurança, análise de fluxo de informação, sistema de tipos, primitivas de manipulação de dados

Contents

List of Figures

Listings

1

Introduction

1.1 Motivation

Security of data is an important concern in our everyday life since long ago. With the democratisation of Internet access, the need for information security became even more important, as we spend significant amounts of time online, and tend to trust private information to a variety of different websites: social networks (e.g. Facebook, Twitter, YouTube), shopping websites (e.g. Amazon, eBay), search engines (e.g. Google, Yahoo!, Bing) and others. Thus, security concerns are nowadays crucial in the development of software systems. The majority of security mechanisms used in software systems are inspired or have some parallel with their non-digital counterparts, like encryption, security protocols, or access control mechanisms.

Access control in software systems is about having the right permissions to the information we want to access. However, current implementations are still subject to security flaws, often caused by a combination of (i) ad-hoc implementation methods, and (ii) a lack of tool support to ensure all needed verifications are in place. Moreover, a significant number of today's software systems are designed around database systems that store business information as well as data relevant to access control, such as user profiles and permissions. Thus, the code implementing security mechanisms is scattered across the application code, often replicated at different architectural layers, each one written in its own programming language and with its own data format. For instance, in a typical web application there are the database layer, using SQL; the server-side code written in PHP, Java, C#, etc.; and the client-side code written in JavaScript. This setting is even more intricate in the task of maintaining or evolving an application.

Several approaches exist that address this problem by integrating the development of all layers in a single programming language, avoiding incompatibilities between different data formats and allowing the compile-time detection of invalid queries and other system-wide analyses. In particular, some approaches allow specification and static verification of security policies related to access control, such as LiveWeb/λ_{DB} [Dom10, CPS+11] and Ur/Web [Chl10], ensuring that the access control code is correctly placed.

In Ur/Web, an abstract interpretation technique is applied to statically verify the conformance of the code against access control policies, based on the idea of *SQL queries as policies*, where SELECT queries are used to specify an upper bound on the data manipulated by a given database operation. On the other hand, LiveWeb implements the access control model presented in [CPS+11] for λ_{DB}, where policies are defined by arbitrary logical propositions and knowledge is stored in dependent refinement types [GF09], yielding a more flexible and uniform approach.

Although access control provides guarantees about *who* is able to access the information, it is unable to prevent the misuse of access permissions to leak confidential information, because there is no control over *how* information is used. The problem of controlling the use of information is addressed by a technique called *information-flow analysis*, which consists in tracking the various flows of information to ensure confidential information is not leaked or, in general, that security policies are respected.

Information-flow analysis relies on annotating values or store locations with a security level and tracking, either statically or dynamically, if they are used by the program in a secure way. In this setting, program security is tied to a property called *non-interference*, stating that confidential inputs of a program should not *interfere* with the values of its public outputs. However, direct formulations of non-interference are too restrictive to be applied in practice: a login system would be automatically ruled out because it leaks information about whether a password was correct or not. There is a long thread of research on mechanisms to safely *declassify* information (i.e. lower its security level) and their interplay with non-interference, surveyed in [SS05].

Compile-time analyses are usually encoded as type systems where types are annotated with security levels as well, thus guaranteeing that non-interference holds for well-typed programs. Type-based approaches to information security were pioneered by Volpano et al. in [VSI96], and followed by many others [ABHR99, Zda02, CKP05] in the context of both imperative and functional languages of increasing complexity, yielding two pragmatic applications to mainstream programming languages: Jif [Mye99], an extension to Java, and Flow Caml [Sim03], an extension to Objective Caml. None of these approaches directly address the data-centric setting of LiveWeb/λ_{DB} and Ur/Web, whose primitives' information-flow only recently [LC12] got characterized.

Historically, approaches to compile-time analysis were considered to be more fine-grained than those to run-time analysis, because they are able to statically verify all possible executions of a program, instead of only verifying the current one. Even so, run-time analyses are able to accept a larger number of safe programs than compile-time analyses, because they deal directly with the program that is running instead of performing

Listing 1.1: Anonymous comments in LiveWeb/λ_{DB}

```
def entity Comment {
  author : String,
  content : String
}
read author where Auth(author)
read content where true
write author, content where Auth(author)

def action submit(username: {x : String | Auth(x)}, comment: String): Unit =
  let signedComment = comment + " By " + username in
    insert [
      author = username,
      content = signedComment
    ] into Comment
```

approximations. Recent run-time approaches [AF12, YYSL12], are able to be even finer-grained than static approaches – although with increased run-time costs – by computing with *faceted values*, pairs of a confidential value and a public value, during the whole computation. The purpose is to not reveal the confidential values to an untrusted principal, instead yielding the public value, computed in a parallel state.

In the next chapters we describe a necessarily non-complete list of significant works on information-flow in software systems. A more comprehensive survey of language-based approaches to information-flow analysis can be found in [SM03].

Problem Enforcing security policies in database-backed software systems is a challenging task, as access verification code ends up scattered and duplicated not only across the code, but also across various architectural layers. Programming languages like Ur/Web and LiveWeb/λ_{DB}, simplify this problem by integrating the development of the whole system in a single language, allowing them to statically enforce access control policies. However, limited support is given to the task of ensuring that information is not misused.

For instance, consider the well-typed LiveWeb program in Listing 1.1, a fragment of a blog application, where we have a database entity `Comment`, representing anonymous comments, and a function `submit`, used by authenticated users to submit new comments to the database. Anonymity is enforced by the access control policy of the `author` field that states that its value can only be read by a user authenticated as the author of the comment, while the text of the comment is public. Notice that the access control policy is respected, as the `submit` function inserts a new comment whose author is authenticated. In spite of that guarantee, in this case the username of the author is also included in the contents of the comment, which makes it public, breaking the intended anonymity policy.

The specification of security policies as logical propositions is a powerful approach, more so when propositions depend on the actual protected data, allowing for policies to work at the table-cell level. Our goal is to explore their use as security levels in type-based information-flow analysis, in order to address problems as the one above and study the advantages and limitations of this approach in the context of data-centric software systems.

1.2 Proposed Approach

Our solution consists in a proof of concept extension to the LiveWeb/λ_{DB} framework [Dom10, CPS+11], featuring a type-based approach to information-flow analysis whose security levels are first-order logic propositions. In the following, we illustrate the solution with a brief series of small examples based in a simple version of the paper reviewing process, an example application that will be explored further in later chapters.

Consider a software system where registered users, represented by the database entity `User`, are characterized by a unique `id`, a `name` and a `password`. The `id` and `name` fields have security level true(), meaning they are readable by any user of the system (i.e. public), and the `password` field has security level `System()`, modeling the policy that it should only be read by trusted system functions.

```
def entity User {
  id: Int at true()
  name: String at true(),
  password: String at System()
}
```

Authenticated users are able to submit papers to the system, which are then available for any user to read. Papers are represented by the entity `Paper` which is characterized by a unique `id`, a single paper author identified by `author_id` and the `body` of the paper itself, all public.

```
def entity Paper {
  id: Int at true(),
  author_id: Int at true(),
  body: String at true()
}
```

Each paper can have several reviews, but a given review is only readable by the reviewer or the author of the paper, which is expressed in the security level of the `paper_id` and `content` fields. Likewise, the security level of the `reviewer_id` field expresses that reviewers are anonymous. Both security levels are specified in terms of a `User` predicate, representing authenticated users, and a `AuthorOf` predicate, which represents information readable by the author of a specific paper.

```
def entity Review {
  id: Int at true(),
  reviewer_id: Int at User(reviewer_id),
  paper_id: Int at User(reviewer_id) or AuthorOf(paper_id),
  content: String at User(reviewer_id) or AuthorOf(paper_id)
}
```

Information-flow is tracked by associating the security level of each value to its type, just like in the entities' specification, with composite expressions defining their security levels in terms of their sub-expressions' security levels in a conservative way: while the sum of two public values should result in a public value, summing a public value with a private value should yield a value with the most restrictive of both security levels, i.e. private. For instance, consider a select query that selects the ids of all papers whose reviewer is the user with id 3:

```
from r in Review where r.reviewer_id == 3 select r.paper_id
```

From the definition of the `Review` entity, we know the paper ids we are selecting have security level `User(r.reviewer_id)` or `Author(r.paper_id)`. However, *which rows* the query will fetch is dependent on the where condition, composed by a comparison of the `reviewer_id` field, with security level `User(r.reviewer_id)`, and a literal 3, which we consider public. Since the `reviewer_id`'s level is the *most restrictive* of the three, both the condition and the result of the query have security level `User(r.reviewer_id)`, or more specifically `User(3)`, meaning they can be read by an authenticated user with id 3.

On the other hand, if we consider a query that involves *writing* values to the database, it is important to consider that the values we *read* back later must correspond to the declared fields' security levels. Thus, the security level of values we insert cannot be *more restrictive* than the security levels declared in the database. For instance, consider the following insert query:

```
insert
  [ id = 23, author_id = 5, body = "We present a..." ]
into Paper
```

All the values we are writing to the database are literals, which we consider public in this context. Comparing the security levels of each field in the database, we can see the query does not leak information, as we *write* public information that will later be *read* as public information.

Finally, consider an example where we combine both queries, in order to illustrate the detection of an information leak:

```
let paper_ids =
  from r in Review where r.reviewer_id == 3 select r.paper_id
in
  let first_id = head paper_ids in
    insert
      [ id = 23, author_id = 5, body = "" + first_id ]
    into Paper
```

We take the first paper's id from the result of the select query, and use it as the body of the paper in the insert query. Since the results of the first query have security level `User(3)`, we know the first result also has security level `User(3)`. Consequently, when we type the second query and compare the fields' security levels with those of the values we are inserting, we are able to conclude that the value being inserted for the `body` field is not public as required. In other words, we would be *writing* a value with level `User(3)` and later *reading* a value with level `true()`, incurring in an information leak.

1.3 Contributions

The work developed in this thesis provides three main contributions within its primary goal of developing a type-based information flow analysis whose security levels are data-dependent logical propositions, in the setting of data-centric applications.

Type System The definition of a core language and type system featuring type-based information flow analysis in a data-centric setting, whose security levels are logical propositions dependent on the protected data itself, allowing for a expressive characterization of security policies with precision at the table-cell level.

Prototype A prototype implementation of our type system which extends the LiveWeb framework, allowing us to tackle the challenges of a concrete implementation, illustrate the approach in a practical setting and test its limits. Additionally, our use of a *satisfiability modulo theories* (SMT) solver resulted, as a by-product, in the definition of an encoding from the terms of our language to the solver language and an implementation of a SMT solver module that abstracts those details, which could, in principle, be reused with minor modifications in other works.

Example The development of a medium-sized example in our prototype, a manager for conference submissions and reviews rich in confidentiality requirements, targeted at validating our type system and implementation, understanding best practices and discovering common pitfalls.

1.4 Document Structure

The remaining of this document is organized as follows. In Chapter 2, we present a brief introduction to the key concepts of information-flow analysis, followed by a description of a small but significant set of languages that use this technique. The works described are necessarily non-exhaustive given the large and diverse thread of research in the area, but were chosen in such a way that each of them provides a distinct view to type-based information flow analysis. Then, we present type-based access control and give an overview of the λ_{DB} calculus and the LiveWeb framework. In Chapter 3, we discuss three works that relate to ours in terms of setting or approach to information-flow analysis. In Chapter 4, we present our core language and type system, detail the main extensions required to apply it to our prototype and end with a brief overview of how we could approach a proof of *non-interference*. In Chapter 5, we present an example developed in our prototype and discuss its main challenges and insights. In Chapter 6, we discuss the initial implementation of the LiveWeb framework and the implementation of our own prototype on top of it. Finally, in Chapter 7, we make some final remarks and discuss the future work that could be supported by the contributions of this thesis.

2

Background

2.1 Information Flow

In the context of software systems, it is critical to prevent confidential information from being released to the public or, more generally, to prevent information with a given access policy from being accessed by someone whose permissions do not comply with the given policy.

Most systems' information leaks are caused by implementation errors when trying to enforce security policies using ad-hoc methods, even though there are sophisticated attack techniques, such as inferring information from the running time of a program (timing attacks), or monitoring the power consumption of the hardware in which a system is run (power analysis attacks). Several techniques exist to address and prevent these problems, with information-flow analysis being one of the most precise in avoiding information leaks caused by implementation errors.

Information-flow analysis is a technique used to solve the problem of systematically preventing information leaks, which works by tracking the flows of information within a software system, enforcing there is no flow that violates its security policies. We concentrate in (programming) language-based approaches, as they benefit from the use of the language's semantics and type system, thus providing valuable security guarantees about the implementation.

Typically, these approaches consist in annotating all values with a *security level* and then guaranteeing that confidential values are not simply revealed or used to compute public values, thus allowing the inference of their nature and value. In order to compare different security levels in terms of confidentiality, the studied approaches have a central concept of a *lattice* of security levels, used to specify their hierarchy and thus the global information-flow policies of the system.

Security Lattices A lattice is defined by a set of elements L and partial order on those elements $\sqsubseteq$ such that, for any two elements a and b from L, there is a least upper bound (a *join*), denoted $a \sqcup b$, and a greatest lower bound (a *meet*), denoted $a \sqcap b$. In particular, when the lattice is finite, there is a greatest element and a lowest element, also known as top ($\top$) and bottom ($\bot$), respectively.

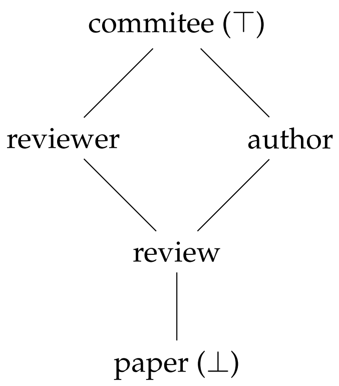


Figure 2.1: Security lattice model for anonymous paper reviews

In the approaches we studied, the greatest element of a security lattice corresponds to the highest security level, and its least element corresponds to the lowest security level, thus it is always safe to restrict information with security level a to a security level b, provided that $a \sqsubseteq b$. Unless specified otherwise, our examples assume the use of a lattice defined by the set $\{\bot, \top\}$, where $\top$ is the security level of confidential information and $\bot$ is the security level of public information, meaning that $\bot \sqsubseteq \top$.

Non-interference Language-based information security defines a secure program in terms of a property called *non-interference*, stating that confidential inputs of a program should not *interfere* with the values of its public outputs. More formally, given a program P and two initial states S_1 and S_2 whose public variables have the same values (denoted $S_1 =_\bot S_2$), running P with S_1 (resp. S_2) should yield a state S'_1 (resp. S'_2) such that $S'_1 =_\bot S'_2$.

For instance, consider a function that takes a confidential boolean x as an argument and returns its logical negation as a public value:

$\lambda\ \mathtt{x}\ :\ \mathsf{bool}_\top.\ \mathsf{if}\ \mathtt{x}\ \mathsf{then}\ \mathsf{false}_\bot\ \mathsf{else}\ \mathsf{true}_\bot$

If we consider the program where we apply this function to the confidential value $\mathsf{true}_\top$, its result will be the public value $\mathsf{false}_\bot$. Conversely, if we apply this function to $\mathsf{false}_\top$, its result will be $\mathsf{true}_\bot$. Thus, by changing the value of the confidential input x, we get different public outputs, meaning that non-interference does not hold for this program.

Implicit flows The most direct way a system can leak information is through *explicit flows* of information, which occur when we explicitly use a confidential value in a place where a public value is expected. However, there exist also *implicit flows* of information such as those resulting from conditional expressions, which leak information about their condition through the control-flow of the system.

In general, implicit flows occur whenever we *deconstruct* a value to know more information about its *identity*. For example, we can deconstruct booleans by using the conditional expression, since we have a branch for when its condition is true and other for when it is false; or we can deconstruct abstractions by applying them to some value, because we can distinguish which abstraction was used by looking at its result (e.g. increment vs decrement).

Outline The remaining of this section presents four language-based approaches to information flow analysis, chosen in such away as to highlight different techniques and points of view. In Subsection 2.1.1 we present λ_{SEC} and λ_{SEC}^{REF}, two functional calculi that follow the typical approach where all values and types have security annotations. Subsection 2.1.2 explores the view of information-flow analysis as a *dependency* analysis by presenting the Dependency Core Calculus. In Subsection 2.1.3 we present the Secure Monadic Calculus, a functional calculus that follows an uncommon approach where only reference types and values have security annotations.

2.1.1 λ_{SEC} and λ_{SEC}^{REF} calculi

The first example we illustrate here is λ_{SEC}, a core functional calculus presented by Zdancewic in his PhD thesis [Zda02] to introduce how one defines a type system that ensures well-typed programs do not leak high security information. We next present one of its extensions, λ_{SEC}^{REF}, which extends the core calculus with references and imperative constructs.

A core calculus

The first calculus, λ_{SEC}, is a variant of the simply-typed lambda calculus [Bar92] featuring boolean and functional values annotated with a security label from an arbitrary lattice. The type system ensures that well-typed programs will never break the security layers, leaking information to undesired contexts. This approach does not require runtime tests to be performed and is supported by the non-annotated semantics of the host language, in this case the standard semantics of the simply-typed lambda calculus.

The type system follows a conservative approach to security, since most typing rules ensure the security level of an expression is at least as restrictive as all its sub-expressions' security levels (see Figure 2.3). The only exception is the typing rule for application (λ_{SEC}-APP), since the security level of e_2 is not explicit in the type of the result.

$$\lambda_{SEC}\text{-APP} \qquad \frac{\Gamma \vdash e_1 : (s_2 \to s)_\ell \qquad \Gamma \vdash e_2 : s_2}{\Gamma \vdash e_1\, e_2 : s \sqcup \ell}$$

$$\begin{array}{lcll}
\ell & \in & \mathcal{L} & \text{(Security labels)} \\
\\
t & ::= & \mathsf{bool} & \text{(Boolean type)} \\
 & | & s \rightarrow s & \text{(Function type)} \\
\\
s & ::= & t_\ell & \text{(Security types)} \\
\\
bv & ::= & \mathsf{true} \mid \mathsf{false} & \text{(Boolean base values)} \\
 & | & \lambda x : s.e & \text{(Functions)} \\
\\
v & ::= & x & \text{(Variables)} \\
 & | & bv_\ell & \text{(Secure Values)} \\
\\
e & ::= & v & \text{(Values)} \\
 & | & e\ e & \text{(Function application)} \\
 & | & e \oplus e & \text{(Primitive operations)} \\
 & | & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e & \text{(Conditional)} \\
\\
\oplus & ::= & \wedge \mid \vee \mid \ldots & \text{(Boolean operations)} \\
\\
\Gamma & ::= & \cdot \mid \Gamma, x : s & \text{(Type environment)}
\end{array}$$

Figure 2.2: λ_{SEC} grammar

Notice also the typing rule for abstraction (λ_{SEC}-FUN), where we can see that if parameter x is used in expression e, the resulting security level must be at least as secure as that of x, since it is a sub-expression (e.g. Listing 2.1). If that is not the case, no information from the parameter will be used in e, proving that the typing rule for application accurately describes the information that may be leaked.

$$\lambda_{SEC}\text{-FUN} \quad \frac{\Gamma, x : s_1 \vdash e : s_2 \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda x : s_1.e)_\ell : (s_1 \rightarrow s_2)_\ell}$$

It is also worth noting, that abstractions are labelled by a security level of their own which provides a lower bound on the security level of the result (see λ_{SEC}-APP), allowing us to constrain the information that is leaked by knowing *what* function is being applied.

Listing 2.1: Simple λ_{SEC} function

```
// Has type (bool⊤ → bool⊤)⊤
(λ h : bool⊤ .
   if h then true⊥ else false⊥
)⊥
```

Non-interference is proved by showing that a user with a low security level cannot distinguish two computations simply by changing the values of high security inputs. Usually, this is accomplished by defining an appropriate behavioural equivalence between programs, parametrized by the security level of the user, as stated in [SM03].

$$\lambda_{SEC}\text{-TRUE} \qquad \frac{}{\Gamma \vdash \mathsf{true}_\ell : \mathsf{bool}_\ell}$$

$$\lambda_{SEC}\text{-FALSE} \qquad \frac{}{\Gamma \vdash \mathsf{false}_\ell : \mathsf{bool}_\ell}$$

$$\lambda_{SEC}\text{-VAR} \qquad \frac{\Gamma(x) = s}{\Gamma \vdash x : s}$$

$$\lambda_{SEC}\text{-FUN} \qquad \frac{\Gamma, x : s_1 \vdash e : s_2 \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda x : s_1.e)_\ell : (s_1 \rightarrow s_2)_\ell}$$

$$\lambda_{SEC}\text{-BINOP} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool}_{\ell_1} \qquad \Gamma \vdash e_2 : \mathsf{bool}_{\ell_2}}{\Gamma e_1 \oplus e_2 : \mathsf{bool}_{\ell_1 \sqcup \ell_2}}$$

$$\lambda_{SEC}\text{-APP} \qquad \frac{\Gamma \vdash e_1 : (s_2 \rightarrow s)_\ell \qquad \Gamma \vdash e_2 : s_2}{\Gamma \vdash e_1\, e_2 : s \sqcup \ell}$$

$$\lambda_{SEC}\text{-COND} \qquad \frac{\Gamma \vdash e : \mathsf{bool}_\ell \qquad \Gamma \vdash e_1 : s \sqcup \ell \qquad \Gamma \vdash e_2 : s \sqcup \ell}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : s \sqcup \ell}$$

$$\lambda_{SEC}\text{-SUB} \qquad \frac{\Gamma \vdash e : s \qquad \Gamma \vdash s \leq s'}{\Gamma \vdash e : s'}$$

Figure 2.3: λ_{SEC} typing rules

Although its simple nature is useful as an introduction to information-flow analysis' techniques, it is also because of its simple nature that it is not a good representative of the challenges faced in mainstream programming languages: as a simple variant of the simply-typed lambda calculus, it still lacks the power of Turing-complete systems.

Coping with side-effects

As a second step to address information-flow analysis in mainstream programming languages, Zdancewic defines λ_{SEC}^{REF} by extending λ_{SEC} with references and recursive functions, thus making it Turing-complete and able to perform side-effects. Consequently, abstraction has a new local variable f that refers to the function itself and a *pc* security label that represents the security level of the program when its body is evaluated.

Type checking in λ_{SEC}^{REF} takes a more fine-grained approach than in λ_{SEC}, tracking the security level of the program at each point by using judgements of the form $\Gamma[pc] \vdash e : s$, stating that e has type s under a type environment Γ at security level pc (for *program counter*). Consequently, the type system is able to deal with the new implicit flows that arise by adding side-effects to the calculus, without being too conservative in its analysis. A complete listing of the typing rules can be found in Figure 2.5.

To understand the type system as a whole, it is important to understand the key mechanism through which the current *pc* influences the security level of an expression, which

$$
\begin{array}{lcll}
\ell, pc & \in & \mathcal{L} & \text{(Security labels)} \\
x, f & \in & \mathcal{V} & \text{(Variables)} \\
\\
t & ::= & \mathsf{unit} & \text{(Unit type)} \\
 & \mid & \mathsf{bool} & \text{(Boolean type)} \\
 & \mid & s\ \mathsf{ref} & \text{(Reference type)} \\
 & \mid & [pc]\ s \to s & \text{(Function type)} \\
\\
s & ::= & t_\ell & \text{(Security types)} \\
\\
bv & ::= & \mathsf{true} \mid \mathsf{false} & \text{(Boolean base values)} \\
 & \mid & \langle\rangle & \text{(Unit value)} \\
 & \mid & \lambda[pc]\ f(x : s).e & \text{(Recursive function)} \\
 & \mid & L^s & \text{(Memory locations)} \\
\\
v & ::= & x & \text{(Variables)} \\
 & \mid & bv_\ell & \text{(Secure Values)} \\
\\
e & ::= & v & \text{(Values)} \\
 & \mid & e\ e & \text{(Function application)} \\
 & \mid & e \oplus e & \text{(Primitive operations)} \\
 & \mid & \mathsf{ref}^s\ e & \text{(Reference creation)} \\
 & \mid & !e & \text{(Dereference)} \\
 & \mid & e := e & \text{(Assignment)} \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e & \text{(Conditional)} \\
\\
\oplus & ::= & \wedge \mid \vee \mid \ldots & \text{(Boolean operations)} \\
\\
\Gamma & ::= & \cdot \mid \Gamma, x : s & \text{(Type environment)}
\end{array}
$$

Figure 2.4: λ_{SEC}^{REF} grammar

is somewhat implicit in the typing rules. The typing rule for values (λ_{SEC}^{REF}-VAL) requires the security level of every value to be at least as restrictive as the current *pc*. Consequently, for an expression to be well-typed, its security level also needs to be at least as restrictive as *pc*, which often ends up requiring the use of subsumption to further restrict the security level.

$$\lambda_{SEC}^{REF}\text{-VAL} \quad \frac{\Gamma \vdash v : s \qquad pc \sqsubseteq \text{label}(s)}{\Gamma[pc] \vdash v : s}$$

In λ_{SEC}, the only source of implicit flows are conditional expressions, here handled by forcing the branches to be typed in a *pc* at least as restrictive as the condition's security level (see λ_{SEC}^{REF}-COND). For instance, if the condition is a high security boolean ($\mathsf{bool}_\top$), both branches will be typed at a high security *pc*, $\Gamma[\top] \vdash e_i : s$, guaranteeing that the result is also high security as a consequence of the typing rule for values. By adding references to the calculus, λ_{SEC}^{REF} now has to deal both with aliasing and function calls that

$$\lambda^{REF}_{SEC}\text{-VAL} \qquad \frac{\Gamma \vdash v : s \qquad pc \sqsubseteq \text{label}(s)}{\Gamma[pc] \vdash v : s}$$

$$\lambda^{REF}_{SEC}\text{-APP} \qquad \frac{\Gamma[pc] \vdash e : ([pc']s' \to s)_\ell \qquad \Gamma[pc] \vdash e' : s' \qquad pc \sqcup \ell \sqsubseteq pc'}{\Gamma[pc] \vdash e\, e' : s \sqcup \ell}$$

$$\lambda^{REF}_{SEC}\text{-REF} \qquad \frac{\Gamma[pc] \vdash e : s}{\Gamma[pc] \vdash \mathsf{ref}^s\ e : s\ \mathsf{ref}_{pc}}$$

$$\lambda^{REF}_{SEC}\text{-DEREF} \qquad \frac{\Gamma[pc] \vdash e : s\ \mathsf{ref}_\ell}{\Gamma[pc] \vdash !e : s \sqcup \ell}$$

$$\lambda^{REF}_{SEC}\text{-ASSN} \qquad \frac{\Gamma[pc] \vdash e_1 : s\ \mathsf{ref}_\ell \qquad \Gamma[pc] \vdash e_2 : s \qquad \ell \sqsubseteq \text{label}(s)}{\Gamma[pc] \vdash e_1 := e_2 : \mathsf{unit}_{pc}}$$

$$\lambda^{REF}_{SEC}\text{-COND} \qquad \frac{\Gamma[pc] \vdash e : \mathsf{bool}_\ell \qquad \Gamma[pc \sqcup \ell] \vdash e_i : s \qquad i \in \{1, 2\}}{\Gamma[pc] \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : s}$$

Figure 2.5: Some λ^{REF}_{SEC} typing rules

write to low security memory locations, so that they cannot be exploited to leak secure information.

Since references are first-class values, they too have a security label of their own. Like the case for functions, this security label is used to avoid leaks caused by knowing *what* reference we are dereferencing (see λ^{REF}_{SEC}-DEREF).

$$\lambda^{REF}_{SEC}\text{-DEREF} \qquad \frac{\Gamma[pc] \vdash e : s\ \mathsf{ref}_\ell}{\Gamma[pc] \vdash !e : s \sqcup \ell}$$

If we did not have this annotation, the program in Listing 2.2 would be well-typed, as there would be no way to distinguish between both references. This notion of the *identity* of a reference is exactly what is compromised by allowing aliasing.

Listing 2.2: Preventing reference aliasing in λ^{REF}_{SEC}

```
// Ill-typed because high cannot have type bool⊥ ref⊥
(λ [⊥] high : bool⊥ ref⊤.
  (λ [⊥] low : bool⊥ ref⊥. !low)⊥ high
)⊥
```

Possibly unsafe function calls are ruled out by the typing rule for function application (λ^{REF}_{SEC}-APP), by requiring the function's program counter to be at least as restrictive as the current *pc*. Combined with the typing rule for assignments (λ^{REF}_{SEC}-ASSN), this guarantees the function being called only writes to references at least as secure as *pc*.

$$\lambda^{REF}_{SEC}\text{-APP} \qquad \frac{\Gamma[pc] \vdash e : ([pc']s' \to s)_\ell \qquad \Gamma[pc] \vdash e' : s' \qquad pc \sqcup \ell \sqsubseteq pc'}{\Gamma[pc] \vdash e\, e' : s \sqcup \ell}$$

$$\lambda_{SEC}^{REF}\text{-ASSN} \quad \frac{\Gamma[pc] \vdash e_1 : s\ \mathsf{ref}_\ell \qquad \Gamma[pc] \vdash e_2 : s \qquad \ell \sqsubseteq \mathrm{label}(s)}{\Gamma[pc] \vdash e_1 := e_2 : \mathsf{unit}_{pc}}$$

Listing 2.3 is an example of a well-typed function that writes high security information and yet can be called in low security contexts, since they will not be able to get any low security information from that reference.

Listing 2.3: Assigning a high security value in a low security context

```
// Has type ([⊥] bool⊤ ref⊤ → unit⊥)⊥
(λ [⊥] high : bool⊤ ref⊤.
  high := not !high
)⊥
```

For this calculus, the proof that non-interference holds is not obtained directly. Instead, after defining its operational semantics and type system, Zdancewic advances to the definition of λ_{SEC}^{CPS}, a variation of λ_{SEC} with first-class continuations, for which non-interference is proven directly. The proof of non-interference for λ_{SEC}^{REF} is then obtained by encoding it in λ_{SEC}^{CPS}.

Besides λ_{SEC}^{CPS}, Zdancewic defines a calculus focused on concurrent programming, λ_{SEC}^{CONCUR}, and its extension, λ_{SEC}^{DISTR}, focused on distributed programming, all of which are out of the scope of this thesis.

2.1.2 Dependency Core Calculus

Non-interference, the fundamental property of information-flow security [Zda04], states that changing the values of a program's high security inputs does not change the value of its low security outputs. In other words, it states that low security values do not *depend* on high security values, which information-flow analysis aims to guarantee by tracking those dependencies.

This idea of tracking the dependencies of a program can be found beyond information-flow analysis, ranging from compiler optimization techniques such as program slicing, used to determine the parts of a program in which its output depends, to side-effect segregation in the style of Haskell [JW93], in which pure values do not depend on impure values.

To capture this broader notion of *dependency*, Abadi et al. introduce the Dependency Core Calculus (DCC) [ABHR99], an extension to the computational lambda calculus that relies in its use of a type constructor T, with the semantics of a monad, to model dependency between values and computations [Mog89]. More precisely, DCC includes such a type constructor for every label in a predefined information lattice, and further extends the calculus with sum types, lifted types and term recursion (see Figure 2.6).

A monad can be defined by a triple (T, *unit*, *bind*) where T is a type constructor, *unit* is a polymorphic function with type $\alpha \rightarrow T(\alpha)$, and *bind* is a polymorphic function with type $T(\alpha) \rightarrow (\alpha \rightarrow T(\beta)) \rightarrow T(\beta)$. Note that there is a close mapping between the

$$
\begin{array}{rcll}
s & ::= & \mathsf{unit} & \text{(Unit type)} \\
 & | & s + s & \text{(Sum types)} \\
 & | & s \times s & \text{(Product types)} \\
 & | & s \to s & \text{(Function types)} \\
 & | & s_{\perp} & \text{(Lifted types)} \\
 & | & T_{\ell}(s) & \text{(Computational types)} \\
\\
v & ::= & () & \text{(Unit value)} \\
 & | & \lambda x : s.e & \text{(Function abstraction)} \\
 & | & \langle e, e \rangle & \text{(Pair values)} \\
 & | & \mathsf{inj}_i\ e & \text{(Sum constructors)} \\
 & | & \eta_{\ell}\ e & \text{(Unit)} \\
 & | & \mathsf{lift}\ e & \text{(Lifted values)} \\
 & | & \mu f : s.e & \text{(Recursive terms)} \\
\\
e & ::= & v & \text{(Values)} \\
 & | & e\ e & \text{(Function application)} \\
 & | & \mathsf{proj}_i\ e & \text{(Projection)} \\
 & | & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inj}_1(x).e \mid \mathsf{inj}_2(x).e & \text{(Case)} \\
 & | & \mathsf{bind}\ x = e\ \mathsf{in}\ e & \text{(Bind)} \\
 & | & \mathsf{seq}\ x = e\ \mathsf{in}\ e & \text{(Sequence)}
\end{array}
$$

Figure 2.6: DCC grammar

monadic functions *unit* and *bind*, and DCC's constructs of the same name, made clearer by analysing the type system (Figure 2.7).

The *unit* function is used to wrap values in the monadic type constructor, corresponding to DCC's $\eta_{\ell}\ e$ construct, which takes an expression e with some type s and wraps it in T_{ℓ} (see *DCC*-UnitM).

$$
\textit{DCC}\text{-UnitM} \qquad \frac{\Gamma \vdash e : s}{\Gamma \vdash \eta_{\ell}\ e : T_{\ell}(s)}
$$

The *bind* function is used to perform computations with wrapped values without unwrapping them back, corresponding to DCC's $\mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$ construct, which takes an expression e_1 with a type $T_{\ell}(s_1)$ and binds it to the name x, to be used by an expression e_2 with a type s_2 (see *DCC*-BindM).

$$
\textit{DCC}\text{-BindM} \qquad \frac{\Gamma \vdash e_1 : T_{\ell}(s_1) \qquad \Gamma, x : s_1 \vdash e_2 : s_2 \qquad s_2 \text{ is protected at level } \ell}{\Gamma \vdash \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2 : s_2}
$$

There is no guarantee that s_2 is a type of the form $T_{\ell}(\beta)$, because DCC's type system uses the concept of *types protected at level* ℓ to model the propagation of dependencies in the typing rule for the bind construct, similarly to what is done in the typing rule for conditionals in security-typed languages. Such types are defined inductively as:

- If $\ell' \sqsubseteq \ell$, then $T_{\ell'}(s)$ is protected at level ℓ;
- If s_1 and s_2 are protected at level ℓ, then $s_1 \times s_2$ is protected at level ℓ;

- If s_2 is protected at level ℓ, then $s_1 \to s_2$ is protected at level ℓ;
- If s is protected at level ℓ, then $T_{\ell'}(\mathrm{s})$ is protected at level ℓ.

By analogy with security-type systems, we can see the first three cases coincide with the typing rules for security restriction, binary operators and function abstraction, respectively, in terms of security labels. The last case reflects the fact that security levels are cumulative, i.e. writing a type as $(s_\ell)_{\ell'}$ is the same as $s_{\ell \sqcup \ell'}$.

$$\textit{DCC}\text{-UnitM} \qquad \frac{\Gamma \vdash e : s}{\Gamma \vdash \eta_\ell\; e : T_\ell(s)}$$

$$\textit{DCC}\text{-BindM} \qquad \frac{\Gamma \vdash e_1 : T_\ell(s_1) \qquad \Gamma, x : s_1 \vdash e_2 : s_2 \qquad s_2 \text{ is protected at level } \ell}{\Gamma \vdash \mathsf{bind}\; x = e_1 \;\mathsf{in}\; e_2 : s_2}$$

Figure 2.7: DCC type rules related to dependency propagation

In the original article, DCC is used to successfully encode call-tracking, program slicing, binding-time analysis and simple security-type systems of both functional and imperative calculi, evidencing that the main difference between various dependency analyses is often the lattice that is used. Although Abadi et al. recognise the similarity between dependency analyses, the article is focused on their encodings in DCC and does not explore whether it would be possible to encode them into each other.

In this thesis, we study DCC from the point of view of information-flow analysis, leading us to observe that it may be possible to use some security-type systems to perform other kinds of analysis directly, even if security is the primary motivation behind them. For instance, we can model program slicing in the λ_{SEC} calculus (Subsection 2.1.1) by following the same approach that DCC follows, giving a distinct label ℓ_i to every value and using the lattice $(\wp(\{\ell_1, \ldots, \ell_n\}), \subseteq)$, as shown for a simple example in Figure 2.8. As expected, since λ_{SEC}'s typing rules for functions coincide with DCC's typing rules when all types are protected at some level, we are able to conclude that the result of the function does not depend on its argument.

$$\frac{\dfrac{\dfrac{}{\Gamma, x : \mathsf{bool}_{\{2\}} \vdash \mathsf{false}_{\{1\}} : \mathsf{bool}_{\{1\}}} \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda x : \mathsf{bool}_{\{2\}}.\mathsf{false}_{\{1\}})_{\{0\}} : (\mathsf{bool}_{\{2\}} \to \mathsf{bool}_{\{1\}})_{\{0\}}} \qquad \dfrac{}{\Gamma \vdash \mathsf{true}_{\{2\}} : \mathsf{bool}_{\{2\}}}}{\Gamma \vdash (\lambda x : \mathsf{bool}_{\{2\}}.\mathsf{false}_{\{1\}})_{\{0\}}\; \mathsf{true}_{\{2\}} : \mathsf{bool}_{\{0,1\}}}$$

Figure 2.8: Performing program slicing in a security-typed calculus

2.1.3 Secure Monadic Calculus

Information-flow analysis can be described as tracking the flows of information between different locations, to determine if they cause a program to leak confidential information. Typically, security-type systems focus directly on what we want to protect, *information*, and base their analysis on annotating all values with a security label. Another approach would be to focus on the places where information stops during a computation, *memory locations*, much like having security checkpoints.

In [CKP05], Crary et al. introduce the Secure Monadic Calculus, which follows this approach by annotating store locations with a security level, instead of values. They designate this as a *store-oriented* view of security, in opposition to the traditional *value-oriented* view of security, and prove it is equally expressive by encoding other languages in their calculus (namely λ_{SEC}^{REF}, presented in Subsection 2.1.1). Besides including labelled first-class references, their calculus (Figure 2.9) builds on the monadic distinction between values and computations which forms the core of the computational lambda calculus [Mog89], denoting them as terms (M) and expressions (E), respectively.

$$
\begin{array}{llll}
r, w, a & \in & \mathcal{L} & \text{(Security labels)} \\
o & \in & \{(r, w) \in \mathcal{L} \times \mathcal{L} \mid r \sqsubseteq w\} & \text{(Operation levels)} \\
\\
A & ::= & & \text{(Types)} \\
 & & 1 & \text{(Unit type)} \\
 & \mid & \mathsf{bool} & \text{(Boolean type)} \\
 & \mid & A \to A & \text{(Function types)} \\
 & \mid & \mathsf{ref}_a\ A \mid \mathsf{refr}_a\ A \mid \mathsf{refw}_a\ A & \text{(Reference types)} \\
 & \mid & \bigcirc_o\ A & \text{(Monadic types)} \\
\\
V & ::= & & \text{(Values)} \\
 & & * & \text{(Unit)} \\
 & \mid & \mathsf{true} \mid \mathsf{false} & \text{(Boolean values)} \\
 & \mid & \lambda x : A.M & \text{(Abstraction)} \\
 & \mid & \ell & \text{(Store location)} \\
 & \mid & \mathsf{val}\ E & \text{(Suspended computation)} \\
\\
M & ::= & & \text{(Terms)} \\
 & & x & \text{(Variables)} \\
 & \mid & V & \text{(Values)} \\
 & \mid & \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M & \text{(Conditional)} \\
 & \mid & MM & \text{(Application)} \\
\\
E & ::= & & \text{(Expressions)} \\
 & & [M] & \text{(Return)} \\
 & \mid & \mathsf{let\ val}\ x = M\ \mathsf{in}\ E & \text{(Sequencing)} \\
 & \mid & \mathsf{ref}_a(M : A) & \text{(Store allocation)} \\
 & \mid & !M & \text{(Store read)} \\
 & \mid & M := M & \text{(Store write)} \\
\\
\Gamma & ::= & \cdot \mid \Gamma, x : A & \text{(Contexts)} \\
\Sigma & ::= & \{\} \mid \Sigma\{\ell : A\} & \text{(Store types)}
\end{array}
$$

Figure 2.9: Secure Monadic Calculus grammar

This distinction is very clear in the type system (Figure 2.10) as well. Terms are typed with judgements of the form $\Sigma;\Gamma \vdash M : A$, which are not concerned with the security level at all, because terms are unable to cause side-effects by definition. On the other hand, expressions can interact with the store, so their typing rules must take the security level into account. In this calculus, an expression is typed in a *operation level* $o = (r, w)$ using judgements of the form $\Sigma;\Gamma \vdash E \div_o A$, where r is a lower bound on the security levels of E's reads and w is an upper bound on its writes. Clearly, r and w must always be such that $r \sqsubseteq w$, since being able to write below what we are able to read is an open door to information leaks (see Listing 2.4).

Listing 2.4: Writing below the reading level

```
// Ill-typed since the operation level would need to be (⊤,⊥)
let val high = val ref⊤ (false : bool) in
let val low = val ref⊥ (false : bool) in
let val h = val !high in
  low := h
```

Interactions with the store are performed through allocation, reading and writing, and have a direct influence in the operation level. Store allocation is typed (*SMC*-31) with operation level $(\bot, \top)$, reflecting the fact that allocating a fresh location from the store is neither a read or a write, leading to no leaks of information[1]. Unsurprisingly, reading from a reference with security level a is typed (*SMC*-32) with operation level $(a, \top)$, reflecting the fact that no write is performed. Conversely, writing to a reference with security level a is typed (*SMC*-33) with operation level $(\bot, a)$, as expected.

The operation level is propagated through the program by the typing rule for the let val construct (*SMC*-30), which corresponds to the monadic bind operation and establishes the bridge between terms and expressions. As it requires both its sub-term and sub-expression to be typed in the same operation level (essentially), there is often the need to use subsumption to restrict both levels to their join, if it exists.

To approach the situation where we have high security computations inside low security ones, Crary et al. introduce the concept of *informativeness* of a type with respect to a security level. Its judgements take the form $\vdash A \nearrow a$, stating that type A only provides information at security level a or above. Intuitively, this means that if we cannot gain any information from type A at our current operation level, we can safely lower it to $(\bot, \top)$. This fine-grained analysis allows us to type programs such as the one in Listing 2.5 as low security computations, since we know that any two values of type unit are indistinguishable from each other. That is not the case in other calculi such as λ^{REF}_{SEC}, for example, where an equivalent program would be typed as a high security computation.

Non-interference results for this calculus guarantee that, if a program is well-typed in an operation level (r, w), no information higher than r is leaked. Their proof is built

[1]At least in the scope of information-flow analysis. Monitoring the memory usage of a program could lead to a side-channel attack.

$$\textit{SMC-27} \quad \frac{\Sigma;\Gamma \vdash E \div_o A}{\Sigma;\Gamma \vdash \mathsf{val}\ E : \bigcirc_o A}$$

$$\textit{SMC-30} \quad \frac{\Sigma;\Gamma \vdash M : \bigcirc_o A \qquad \Sigma;\Gamma, x : A \vdash E \div_o A}{\Sigma;\Gamma \vdash \mathsf{let\ val}\ x = M\ \mathsf{in}\ E \div_o A}$$

$$\textit{SMC-31} \quad \frac{\Sigma;\Gamma \vdash M : A}{\Sigma;\Gamma \vdash \mathsf{ref}_a(M : A) \div_{(\bot,\top)} \mathsf{ref}_a\ A}$$

$$\textit{SMC-32} \quad \frac{\Sigma;\Gamma \vdash M : \mathsf{refr}_a\ A}{\Sigma;\Gamma \vdash\ !M \div_{(a,\top)} A}$$

$$\textit{SMC-33} \quad \frac{\Sigma;\Gamma \vdash M_1 : \mathsf{refw}_a\ A \qquad \Sigma;\Gamma \vdash M_2 : A}{\Sigma;\Gamma \vdash M_1 := M_2 \div_{(\bot,a)} 1}$$

Figure 2.10: Some typing rules for the Secure Monadic Calculus

Listing 2.5: SMC program that benefits from informativeness

```
// Well-typed with operation level (⊥,⊤) instead of requiring (⊤,⊤)
let val high = val ref⊤ false in
let val h = val !high in
  [if h then * else *]
```

on top of two main steps: the definition of an equivalence relation between computation states, parametrized by a security level, and the respective proof that the equivalence is preserved at each evaluation step.

2.2 Type-Based Access Control

Even though software systems exist to process and provide information, either to human users or to other software systems, there is often the need to protect the access to some of their information, either because it should not be accessible to every user of the system, or because its knowledge may directly compromise the established security policies, thus creating the need to control the access specific users have to specific information.

Several *access control* systems were developed to enforce policies that specify whether a given user of the software system has permission to access a given information, more generally referred to as specifying whether a given *subject* has permission to access a given *object*. Depending on the entity to which permissions are associated, subjects or objects, access control systems can be classified in two main classes: those based in capabilities associate permissions with the subjects, while those based in Access Control Lists (ACLs) associate permissions with the objects.

Typical access control mechanisms are purely dynamic in nature and have no tool to support their use and development, forcing the programmer to be extra cautious not to forget any important verification code, which often leads to access control code being scattered everywhere "just in case". However, there are language-based approaches that mitigate this problem by integrating the specification and verification of access control policies at the language level, detecting situations where there is no guarantee that the security policy is being respected.

Ur/Web is an embed domain-specific language for web applications that integrates the database and application layers, allowing the detection of common errors and inconsistencies between the two. In [Chl10], Chlipala presents a tool to statically verify access control policies in Ur/Web without requiring any program annotations besides the policies themselves, based on the idea of *SQL queries as policies*. The intuition is that SELECT queries can be used to specify an upper bound on the data manipulated by a given database operation, so the challenge is to ensure that data manipulated by database operations is a subset of the data allowed by the policies. Given that the analysis also verifies information flows to some extent, it is discussed as a related work in Section 3.1.

In [CPS+11], Caires et al. introduce a calculus with the same goals and setting, where policies are specified as arbitrary logic conditions and knowledge is registered in the types, yielding a different but more flexible approach. In the following, we describe the language in more detail through a series of small examples, given that our language draws on its core database model and type-based approach, and describe the core concepts of LiveWeb [Dom10], a domain specific language for web applications which was extended with the calculus' type system.

2.2.1 λ_{DB} and the LiveWeb framework

Web-based applications alone are enough to show there is a significant number of data-centric software systems that rely on databases to store data, leading to a proliferation of multi-layered software architectures even though they are very challenging to build correctly. Layers are often implemented in different programming languages and interface with each other using ad-hoc methods, difficulting every attempt to maintain coherence between them or enforce system-wide security policies.

The λ_{DB} calculus

In [CPS+11], Caires et al. introduce a language targeting both the database and application layers in a uniform way, in order to shift part of the challenging work from the developer to the development tools. Their language, λ_{DB}, is a functional calculus with records, collections and core database primitives, which uses dependent refinement types to statically enforce access control policies over data. Database policies are specified as read or write permissions, expressed in classical propositional logic extended with equality over terms and the axiom $[\overline{m = v}].m_i = v_i$ to deal with records, which allows for precise access control at the table cell level. We present the calculus' complete syntax in Figure 2.11.

Refinement types, a core concept of this approach, were first introduced in [FP91]

e	$::=$			(*Expressions*)
			v	(Value)
		$\mid$	$e\ (v)$	(Application)
		$\mid$	$[\overline{m = e}]$	(Record)
		$\mid$	$e.m$	(Field Selection)
		$\mid$	$e\ op\ e$	(Operation)
		$\mid$	$e\ ?\ e : e$	(Conditional)
		$\mid$	let $x = e$ in e	(Let)
		$\mid$	$e_1, \ldots, e_k$	(Collection)
		$\mid$	create $t : \beta_{\overline{\rho}}$ in e	(Create)
		$\mid$	from x in t where e select e	(Select)
		$\mid$	update x in t where e with e	(Update)
		$\mid$	append e to t	(Append)
		$\mid$	delete x in t where e	(Delete)
		$\mid$	assume C	(Assume)
		$\mid$	assert C	(Assert)
ρ	$::=$			(*Permissions*)
			rd(m, R)	(Read)
		$\mid$	wr(m, W)	(Write)
C, R, W	$::=$			(*Propositions*)
			$p(\overline{V})$	(Predicate)
		$\mid$	$V = V$	(Equality)
		$\mid$	$C \wedge C$	(Conjunction)
		$\mid$	$C \Longrightarrow C$	(Implication)
u, v	$::=$			(*Values*)
			$()$	(Unit value)
		$\mid$	true	(True)
		$\mid$	false	(False)
		$\mid$	x	(Variable)
		$\mid$	$\lambda x : \tau.e$	(Abstraction)
		$\mid$	$[\overline{m = e}]$	(Record)
		$\mid$	$v_1, \ldots, v_k$	(Collection)
		$\mid$	$\star(v)$	(Classified Value)
V	$::=$			(*Terms*)
			$()$	(Unit value)
		$\mid$	true	(True)
		$\mid$	false	(False)
		$\mid$	x	(Variable)
		$\mid$	$\lambda x : \tau.e$	(Abstraction)
		$\mid$	$[\overline{m = e}]$	(Record)
		$\mid$	$V_1, \ldots, V_k$	(Collection)
		$\mid$	$\star(V)$	(Classified Value)
		$\mid$	$V.m$	(Field Selection)

Figure 2.11: λ_{DB} Grammar

to enhance the type system with more precise type information, allowing compile-time detection of a wider range of errors. In the style of [GF09], refinement types in λ_{DB} take the form $\{x : \tau \mid C\}$, which can be read as the type of "all values x of type τ that satisfy proposition C". For instance, while Int is the type of all integers, the refinement type $\{x : \mathsf{Int} \mid \mathrm{Auth}(x)\}$ is the type of all integers that satisfy predicate Auth, which can be informally taken to mean the type of all authenticated user IDs of the system.

Note that Auth is not a special predicate, in the sense that it has no special meaning within the language, only within a particular system. To enforce the desired semantics, the only source of authenticated IDs would need to be a trusted `authenticate` function like the one in Listing 2.6. By typing its result as $\{x : \mathsf{Bool} \mid x = \mathsf{true} \Rightarrow \mathrm{Auth}(\mathrm{uid})\}$, we reflect the knowledge that if the result equals true then the user ID we supplied as an argument was successfully authenticated, illustrating the potential of refinement types to specify the post-conditions of a function.

Listing 2.6: Trusted function used to authenticate users

```
authenticate ≜ λ uid : Int. λ pwd : String.
  let res = from u in Users where u.id = uid ∧ u.password = pwd select u in
    isEmpty(res) ?
      false
    :
      let _ = assume Auth(uid) in true
```

Database tables are defined by specifying a name and a *table type* $\beta_{\overline{\rho}}$ where β is the type of its rows, typically a record type that defines the table's schema, and a collection $\overline{\rho}$ of read and write permissions of the form $\mathsf{rd}(m, C)$ and $\mathsf{wr}(m, C)$, respectively, where m is a field name and C a logical proposition (as stated above). We are allowed to read (resp. write) a field if, and only if, we are able to prove the disjunction of its read (resp. write) permissions holds. For instance, the `Users` table referred in the example of Listing 2.6, could be defined with table type

$$\begin{aligned}
&[\ \mathrm{id} : \mathsf{Int}, \mathrm{name} : \mathsf{String}, \mathrm{password} : \mathsf{String}, \mathrm{phone} : \mathsf{Int}\ ]_{\overline{\rho}} \\
&\text{where } \overline{\rho} = \{\ \mathsf{rd}(\mathrm{id}, \mathsf{true}), \\
&\qquad\qquad \mathsf{rd}(\mathrm{name}, \mathsf{true}), \\
&\qquad\qquad \mathsf{rd}(\mathrm{password}, \mathsf{true}), \\
&\qquad\qquad \mathsf{rd}(\mathrm{phone}, \mathrm{Auth}(\mathit{this}.\mathrm{id}))\ \}
\end{aligned}$$

informally meaning that we are always able to read the id, name and password of a user, but the only way to read a user's phone number is to be authenticated as that specific user. The type system enforces the read and write policies in all database primitives by taking into account which fields are read and written to, either explicitly (e.g. select clause) or implicitly (e.g. where clause), and requiring us to prove that we are allowed to read or write to those fields based on our current knowledge.

Given that the current knowledge is registered in each type, we can write judgements of the form $\Delta \vdash C$ to mean that C holds based on the knowledge of type environment Δ, allowing us to cumulatively gather knowledge according to the usual scoping rules. For instance, in Listing 2.7 we define a `getPhone` function that given a user ID selects the corresponding phone number from the `Users` table, which is only well-typed because we require the user ID to be authenticated in order to fulfil the table's read policy. Notice no explicit use of `authenticate` is required, because the type system will guarantee the function can only be called if it has the knowledge the user is authenticated.

Listing 2.7: Obtaining a phone number requires authentication

```
getPhone ≜ λ uid : { x : Int | Auth(x) }.
  let res = from u in Users where u.id = uid select u.phone in
     head(res)
```

Knowledge is directly increased with each new declaration, but it is also propagated through the return types of function calls or the results of database primitives, refined with where clause conditions. Other constructs, like the conditional or the assume, increase knowledge implicitly by extending the type environment with mappings of the form $\_ : \{\_ : \mathsf{unit} \mid C\}$. In the example of Listing 2.8, each branch of the conditional is typed under an extended type environment with knowledge about the condition, allowing it to prove the user is authenticated when calling the `getPhone` function. Recall that the return type of the `authenticate` function is $\{x : \mathsf{Bool} \mid x = \mathsf{true} \Rightarrow \mathrm{Auth(uid)}\}$, so the first branch is typed with the additional knowledge $\_ : \{\_ : \mathsf{unit} \mid \mathsf{true} = \mathsf{true} \Rightarrow \mathrm{Auth(user\_id)}\}$, implying that $\mathrm{Auth(user\_id)}$ holds in that scope.

Listing 2.8: Full example with knowledge propagation

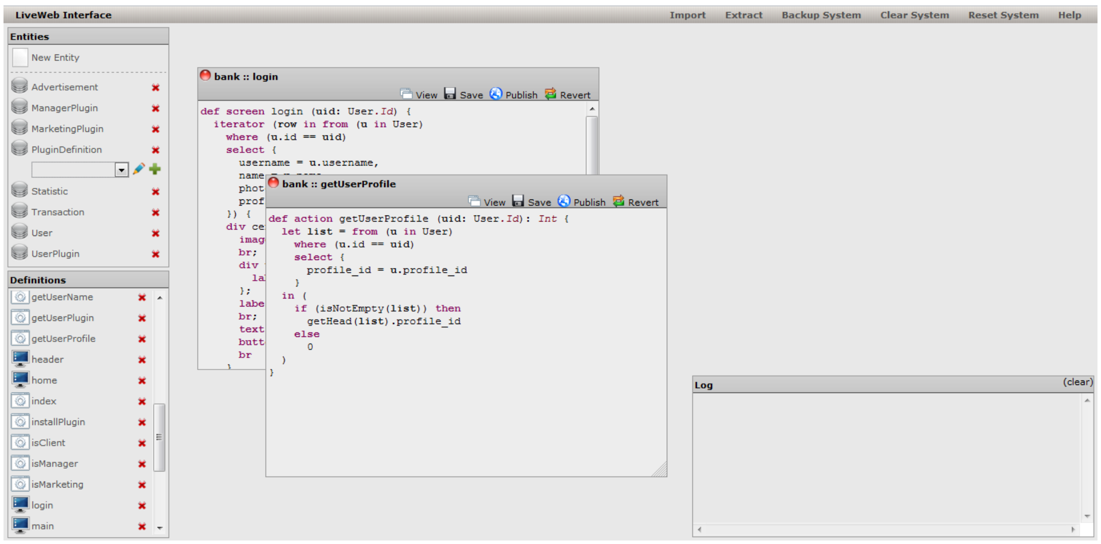
```
λ uname : String. λ pwd : String.
  let res = from u in User where u.name = uname select u.id in
  let user_id = head(res) in
    authenticate(user_id, pwd) ?
      "Phone: " + getPhone(user_id)
    :
      "Access denied"
```

The approach taken in λ_{DB} allows for fine-grained access control policies and provides valuable support to enforce them, rejecting programs for which no static guarantees of access control permissions can be given, thus detecting implementation errors that would otherwise go unnoticed and remain exploitable by ill-intentioned individuals. However, this approach still suffers from the shortcomings of only verifying the access to information, neglecting how it is used after we are granted access. For instance, although it is acceptable for a software system to fetch our password, in order to know if we supplied the right one, it is not acceptable for it to display our password in clear sight afterwards.

The LiveWeb framework

LiveWeb [Dom10] is a statically typed domain-specific language targeted at web applications development, supported by a web-based development environment (Figure 2.12) that allows for the management of successive versions of database entities, web pages and business logic. The framework was designed to serve as a prototype test-bed for various extensions regarding the use of static analysis in data-centric software systems, the first of which was an extension with the λ_{DB}'s type-based access control model.

The language (Figure 2.13) is composed by two major sub-languages: an expression language, whose syntax closely resembles that of λ_{DB}, and a *web page block* language modelled after a fragment of HTML, for presentation purposes. Along with the type language (Figure 2.14), these sub-languages are used in three distinct types of top-level definitions inspired by the components of the Model-View-Controller architectural pattern [Ree79].



```
def screen login (uid: User.Id) {
  iterator (row in from (u in User)
    where (u.id == uid)
    select {
      username = u.username,
```

```
def action getUserProfile (uid: User.Id): Int {
  let list = from (u in User)
    where (u.id == uid)
    select {
      profile_id = u.profile_id
    }
  in (
    if (isNotEmpty(list)) then
      getHead(list).profile_id
    else
      0
  )
}
```

Figure 2.12: LiveWeb's web-based development environment

Entities correspond to database tables and specify their fields and types. Each entity must have a primary key field, indicated by the type Id, but can have several foreign keys by defining fields with types of the form `entity-name`.Id. For instance, in Listing 2.9 we define an entity `User`, with a primary key field `id`, two string fields `username` and `password` and a foreign key `profile_id` referring a `Profile` entity. Each field has a read and write permission of `true`, meaning they can always be read or written.

Listing 2.9: A LiveWeb entity definition

```
def entity User {
  id : Id, username : String, password : String, profile_id : Profile.Id
}
read id, username, password, profile_id where true
write id, username, password, profile_id where true
```

A	$::=$	$\overline{D}$	(Application)
D	$::=$		(*Definition*)
		$\text{def entity}\ t\ \{\ m : \text{Id}, \overline{m : BT}\ \}\ \overline{\rho}$	(Entity)
	$\mid$	$\text{def action}\ a\ (\overline{x : T}) : T\ \{\ e\ \}$	(Action)
	$\mid$	$\text{def screen}\ s\ (\overline{x : T})\ \{\ b\ \}$	(Screen)
ρ	$::=$		(*Entity permissions*)
		$\text{read}\ \overline{x}\ \text{where}\ C$	(Read permission)
	$\mid$	$\text{write}\ \overline{x}\ \text{where}\ C$	(Write permission)
	$\mid$	$\text{invariant}\ C$	(Entity invariant)
e	$::=$		(*Expression*)
		$e\ Op_2\ e$	(Binary operation)
	$\mid$	$Op_1\ e$	(Unary operation)
	$\mid$	v	(Value)
	$\mid$	$\text{let}\ x = e\ \text{in}\ e$	(Variable declaration)
	$\mid$	$\text{if}\ e\ \text{then}\ e\ \text{else}\ e$	(Conditional)
	$\mid$	$a(\overline{e}) \mid s(\overline{e})$	(Action/Screen call)
	$\mid$	$[\ \overline{e}\ ]$	(List)
	$\mid$	$\text{foreach}\ x\ \text{in}\ e\ \text{do}\ e$	(List iterator)
	$\mid$	$\{\ \overline{m = e}\ \}$	(Structure)
	$\mid$	$e.m$	(Field selection)
	$\mid$	$\text{insert}\ e\ \text{in}\ t$	(Entity Insert)
	$\mid$	$\text{update}\ x\ \text{in}\ t\ \text{with}\ e\ \text{where}\ e$	(Entity Update)
	$\mid$	$\text{from}\ (\overline{x\ \text{in}\ t})\ \text{where}\ e\ \text{select}\ e$	(Entity Select)
	$\mid$	$\text{count}(e) \mid \text{max}(e) \mid \text{min}(e)$	(Aggregation functions)
	$\mid$	$\text{assume}\ e$	(Knowledge assumption)
	$\mid$	$\text{assert}\ e$	(Knowledge assertion)
b	$::=$		(*Web Page Block*)
		$\overline{b}$	(Block sequence)
	$\mid$	br	(Line break)
	$\mid$	$\text{label}\ e$	(Label)
	$\mid$	$\text{div}\ \mathit{class}\ \{\ b\ \}$	(Div)
	$\mid$	$\text{image}\ e$	(Image)
	$\mid$	$\text{link}\ \{\ b\ \}\ \text{to}\ e$	(Link)
	$\mid$	$\text{iterator}\ (x\ \text{in}\ e)\ \{\ b\ \}$	(Iterator)
	$\mid$	$\text{textfield}\ x\ \text{with}\ e$	(Text Field)
	$\mid$	$\text{button}\ e\ \text{to}\ e$	(Button)
v	$::=$		(*Value*)
		$\mathit{integer}$	(Number)
	$\mid$	string	(String literal)
	$\mid$	$\text{false} \mid \text{true}$	(Boolean value)
	$\mid$	id	(Variable)

Figure 2.13: LiveWeb base grammar

$$
\begin{array}{lcll}
BT & ::= & & (\textit{Basic Type}) \\
 & & \mathsf{String} & (\text{String type}) \\
 & | & \mathsf{Int} & (\text{Integer type}) \\
 & | & \mathsf{Bool} & (\text{Boolean type}) \\
 & | & t.\mathsf{Id} & (\text{Entity Key type}) \\
 \\
T & ::= & & (\textit{Type}) \\
 & & BT & (\text{Basic type}) \\
 & | & \mathsf{WebPage} & (\text{Web Page type}) \\
 & | & \{\ \overline{m : T}\ \} & (\text{Structure type}) \\
 & | & \mathsf{List} < T > & (\text{List type}) \\
 & | & \{\ x : T \mid C\ \} & (\text{Refinement type}) \\
 \\
C & ::= & \textit{same as } \lambda_{DB} & (\textit{Proposition})
\end{array}
$$

Figure 2.14: LiveWeb type grammar

Screens specify the web page layout by means of a sequence of *web page blocks* which directly correspond to many HTML elements such as line breaks (br), divs, text fields and buttons. As with most server-side languages and frameworks, the content of the web page can be dynamically generated for each request either by parametrizing the screen (like a typical function) or by embedding queries (or any other expression) in the screen definition. In Listing 2.10, we define a simple login page with two text fields and a submit button, which is parametrized by an error message. Figure 2.15 shows a possible rendering of the login page, if we attempted to login previously and got redirected back.

Listing 2.10: Defining a login screen in LiveWeb

```
def screen loginPage (errorMsg: String) {
  div error { label errorMsg };
  div loginForm {
    label "Username: "; textfield uname; br;
    label "Password: "; textfield pwd; br;
    button "Login" to performLogin(uname, pwd)
  }
}
```

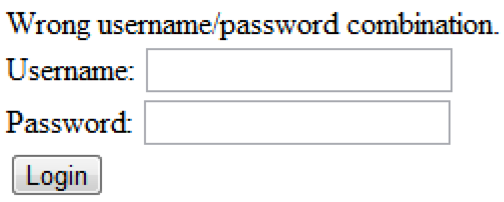

Wrong username/password combination.

Figure 2.15: A login form generated by LiveWeb

Actions are executed in the server-side and encode the application's business logic. Like screens, they can be parametrized, but unlike them no explicit web page blocks occur, only an expression that constitutes the action's body. For instance, a common need among web applications is to perform the login of a user. Besides the layout aspects, already presented, we would need to define an action that, given an user name and a password, tries to perform a login and renders an appropriate web page in response, as exemplified in Listing 2.11. The assume is required in order to provide knowledge that the integer `uid` is authenticated, i.e. has type `{ x: Int | Auth(x) }`.

Listing 2.11: Defining a login action in LiveWeb

```
def action performLogin (uname: String, pwd: String): WebPage {
  let uids =
    from (u in User)
    where u.username == uname and u.password == pwd
    select { id = u.id }
  in
    if isNotEmpty(uids) then
      let uid = getHead(uids).id in (
        assume Auth(uid);
        homePage(uid, "Login was successful.")
      )
    else
      loginPage("Wrong username/password combination.")
}
```

Since its original release, LiveWeb has been the target of some extensions besides the one already mentioned, including an extension with the information flow type system proposed in this thesis. For this reason, the top-level description presented in this section is complemented, in Chapter 6, by a discussion of the framework's implementation and its relation to our work .

3 Related Work

There is a long thread of research on information-flow languages, which includes two major pragmatic approaches that extend mainstream programming languages. Jif [Mye99] is an extension of the Java programming language with information-flow analysis featuring polymorphic labels, first-class labels, runtime principals and declassification mechanisms. Flow Caml [Sim03] is an extension of the Objective Caml programming language with information-flow analysis fully supported by polymorphism and type inference.

While they both provide a powerful support to enforce security policies, in our opinion the focus of Jif is too general to fit as related work, more so as it addresses the problem in an object-oriented setting whose challenges are distinct from ours. Flow Caml, although with a general focus, overlaps with our language in its functional paradigm and is a pragmatic example of advanced features (polymorphism, type inference) that our approach lacks so far. Even so, our work will present an analysis where security levels are first-order logic propositions, allowing for a fine-grained specification of information-flow policies in the context of data-centric systems that is not addressed directly by either of these languages. Thus, in this chapter we discuss both Flow Caml and two works that share the same data-centric setting as ours.

3.1 Ur/Web

Ur/Web is a domain-specific programming language focused on web applications, featuring a non-standard approach to information-flow analysis based in the concept of *SQL queries as policies* [Chl10]. The base idea is that we can use SQL queries as a declarative and familiar language to specify an upper bound on the data that is actually manipulated

Listing 3.1: A sendClient policy in Ur/Web

```
table user : { Name : string, Password : string }

policy sendClient (SELECT user.Name FROM user WHERE true)
```

Listing 3.2: Explicit information-flows in Ur/Web

```
fun displayPassword(name) =
  queryX (SELECT * FROM user WHERE user.Name = {name}) (row =>
    write(row.Password)
  )
```

by database operations. Thus, the analysis must be able to verify that the data manipulated by a given query is a subset of the results of one of the applicable policies.

For instance, in Listing 3.1 we define a database table `user` with two fields `Name` and `Password`, and a `sendClient` policy specifying that only the name of a user can be sent to the web client — that is, field `Name` is public and field `Password` is confidential. Policies of type `sendClient` are used to specify which database information can be sent to the web client (e.g. the browser), corresponding to select statements, and other types of policies exist for insert, delete and update statements.

The language's only loop constructs are *query loops*, used to iterate the results of a query and repeatedly produce or accumulate a result. For instance, in Listing 3.2 we have a `displayPassword` function that selects all the users with a given `name`, specified as an argument, and for each resulting `row` writes the user's password to the web client using the `write` primitive. Given the policy we defined, Ur/Web is able to statically detect that writing the password consists in an information leak, because there is no policy – in our example – that allows for the password values of table `user` to be sent to the web client.

In order to statically detect information leaks, the provenance of the values is tracked throughout the program, as in standard information-flow approaches, but it is achieved by maintaining an abstract state of the program specified in terms of logical propositions. In the context of our previous example, the select query would augment the abstract state with the knowledge that `row` belongs to table `user`, and that the value of `row.Name` equals that of parameter `name`. Additionally, the policy in Listing 3.1 states that if there is some row r that belongs to the `user` table, then we are allowed to send the value of r`.Name` to the web client. Thus, the program's state has no way to prove that we are allowed to write the value of `row.Password`, resulting in a compile-time error as expected.

The expressiveness of the approach is enhanced with the addition of a special predicate `known(x)`, that can be used in queries to assert that the current user of the software system – the client – knows a given information `x`. This allows for fine-grained

Listing 3.3: Undetected information-flow in Ur/Web

```
fun login(name, password) =
  queryX (
    SELECT *
    FROM user
    WHERE user.Name = {name} AND user.Password = {password}
  ) (row =>
    write("Logged in!")
  )
```

table permissions that depend on the current user, based on the notion that all the information coming from the client is known to the user, which is then propagated transitively through equality assertions and container relationships introduced by the various queries.

In spite of this powerful model, their analysis support for implicit flows is lacking, as they do not address the problem at all in the case of where clauses. For instance, the `login` function in Listing 3.3 sends the string `"Logged in!"` to the web client when the user's `name` and `password` match those in the database, and does not send anything otherwise. Thus, it is leaking information about whether the password is right or not, because the query depends on the password's value. Although in this specific setting it is an acceptable consequence, Ur/Web considers that function to be secure without realizing *there is* a consequence, and does not prevent the developer from accidentally leaking information about a table field.

Discussion Standard approaches to compile-time information-flow analysis are based in the use of a *type system* to enforce no information leaks occur, typically requiring explicit security annotations in a program's values and types but guaranteeing absence of leaks caused by programming mistakes and allowing for a local analysis of each part of the system. On the other hand, Ur/Web's approach to information-flow analysis is based in *program verification*, requiring a global analysis of the software system, and does not prevent the developer from causing implicit leaks inadvertently. In spite of this limitations on scalability and soundness, Ur/Web provides a pragmatic model for declarative policy specification, which partially covers common use cases without requiring further program annotations.

3.2 Flow Caml

Flow Caml [Sim03] is an extension of the Objective Caml programming language featuring a type-based information-flow analysis, similar to that of λ_{SEC}^{REF}, which covers a large subset of its source language including functions, records, lists, user-defined data types, pattern matching, references, modules, functors and second-class exceptions. The type system is presented and proven correct in [PS02], including a proof that it guarantees a *non-interference* property, and features the polymorphism and full type inference characteristic of ML dialects, covering security labels as well.

Security labels are written as `!name` and correspond to regular type parameters of every data type, with `int` or `string` being type constructors with one argument, the label, and `list` being a type constructor with two arguments, a security label related to its structure and the type of the list's elements, for example. Every data type with more than one constructor must have at least one security label type argument associated to its structure, in order for the type system to be able to track which information is gained by de-structuring it in one of its constructors. For instance, in the case of the user-defined `list` data type in Listing 3.4, the polymorphic argument `'b` is designated in line 4 as the security level of the information we gain by knowing whether the list is empty or not.

Listing 3.4: A user-defined list data type in Flow Caml

```
1 type ('a, 'b) list =
2     Nil
3     | Cons of 'a * ('a, 'b) list
4     # 'b
5 ;;
```

Function types in Flow Caml are of the form `'a -{'b | 'c | 'd}-> 'e`, where `'a` is the type of the parameter, `'e` is the type of the result, `'b` is the security label of the context in which the function can execute (i.e. its *program counter*), `'d` is the security label associated to the function's structure, and `'c` is a list of the exceptions raised by the function, paired with the security label of the information they reveal. Apart from this last `'c` parameter, related to exceptions, these function types directly correspond to $([\texttt{'b}]\ \texttt{'a} \rightarrow \texttt{'e})_{\texttt{'d}}$ in λ^{REF}_{SEC}. This polymorphic characterization is fully exploited by a precise constraint-based inference algorithm with the inclusion of type (and label) constraints in the type, as exemplified in the type of the function `f` in Listing 3.5. The function receives two integers of different levels, `'a` and `'b`, and produces a pair consisting of the first integer and the sum of both, which has a security label `'c` that is *higher or equal* to the security labels of both operands[1].

Listing 3.5: Type inference and constraints in Flow Caml

```
let f x y = (x, x + y) ;;
val f : 'a int -> 'b int -> 'a int * 'c int
    with 'a < 'c and 'b < 'c
```

Input and output are addressed uniformly in Flow Caml by using a different security label for each different input source or output target, a detail that is not directly addressed by the related calculi we studied. For instance, in the login example of Listing 3.6, we

[1]The type parameters `'b`, `'c` and `'d` are usually omitted when irrelevant and Flow Caml actually simplifies the type further to only using `'a` and `'b`, by taking subtyping into account.

begin by stating that the security label of the standard input (`!stdin`) is lower than both the security label of the standard output (`!stdout`) and the `!secret` security label, informally meaning that information that is *read* from the standard input can be *written* to the standard output and used in private expressions of level `!secret`, respectively.

Listing 3.6: A login example in Flow Caml

```
 1  flow !stdin < !stdout ;;
 2  flow !stdin < !secret ;;
 3
 4  let password: !secret string = "apple" ;;
 5  let login () =
 6     print_string "Type your password: " ;
 7     let p: !stdin string = read_line () in
 8        if p = password then
 9           print_string "Logged in successfully"
10        else
11           print_string "The password is wrong"
12  ;;
```

Next, we declare a `!secret` password and a simplified login function, which reads a string `p` from the standard input (line 7) and writes a different message to the standard output (lines 9 and 11) depending on whether `p` equals the password or not (line 8). Given our policies, the security label inferred for the if-condition is `!secret`, raising the program counter in both branches to `!secret` as well, which reveals the (expected) information leak of the login procedure as we have no policy stating that `!secret` information may be output (i.e. `!secret` $\not<$ `!stdout`).

Discussion Flow Caml is a language which addresses a large amount of significant features – that we cannot possibly fully cover here – bridging the gap between theory and practice by applying information-flow analysis to a mainstream functional language. Our language, on the other hand, only addresses a small part of those features, but does so in a domain-specific setting that is not addressed directly by general purpose languages, and specializes in the use of data-dependent logical propositions as security levels, instead of using only simple labels, resulting in an increased expressiveness of the security policies. However, the importance of polymorphism and type inference cannot be overlooked in a pragmatic information-flow approach, more so when using precise but verbose security levels such as ours.

3.3 Information-Flow in Data-Manipulation Primitives

In a recent work [LC12], the authors perform a characterization of the information-flow in data manipulation primitives, providing intuitions towards the definition of a type-based information flow analysis for a language inspired in the λ_{DB} calculus (presented in Subsection 2.2.1). The characterization is performed by means of examples, illustrating the information-flow in each of the query primitives (select, insert, update and delete) using a lattice with two security levels, $\bot$ and $\top$, with $\bot \sqsubseteq \top$. In order to justify the proposed information-flow, the examples are compared with semantically similar expressions in a traditional imperative setting with memory locations, whose intuitions are the same as the ones described for λ^{REF}_{SEC} in Subsection 2.1.1.

For instance, in Listing 3.7 we declare a relational entity `User` with a public ($\bot$) identifier `id_u`, a public `name` and a private ($\top$) `password`; and proceed to perform a select query to fetch the names of all users whose password is the string `"12345"`.

Listing 3.7: Selecting public values depending on private information

```
entity User(id_u: int⊥, name: string⊥, password: string⊤) in
  from (x in User) where x.password == "12345" select x.name
```

The intuition presented by the authors is that, since the values we select are *dependent* on the where condition, the security level of the result should depend on it as well, similarly to what occurs with conditional statements in traditional imperative approaches. Specifically, the information-flow in the referred example is similar to that found in Listing 3.8, where the value being assigned to the public variable `l` has its security level raised to $\top$ because of the if-condition's security level.

Listing 3.8: Assigning a public value depending on private information

```
if x.password == "12345" then
  l := x.name
```

In turn, insert queries are compared to simple assignments of the form `t := r ::` $\overline{rs}$, where we construct a collection by adding the new record r to the collection of records already inserted, $\overline{rs}$, and assign it to a variable t, representing the entity we insert into. In terms of information flow, assignments require the security level of the value we write to be *at most* as restrictive as the security level of the location we write to. Accordingly, the insert query in Listing 3.9 is not secure because it is assigning a private value, `priv_value`, to the public `name` field.

Listing 3.9: Inserting a private value in a public field

```
let priv_value = head (from (x in User) where true select x.password) in
  insert [ u_id = 1, name = priv_value, password = "12345" ] in User
```

A similar comparison is performed for update and delete queries, which simply combine the writing principles of the insert query and the where condition dependency of the select query. The update query literally writes new values depending on a where condition, like in Listing 3.8, while the delete query does not write new values but changes the collection by removing them, also depending on a where condition.

Discussion Most approaches to type-based information flow analysis focus on the building blocks of general purpose languages and do not address the problem of guaranteeing information security when dealing with domain-specific abstractions such as data manipulation primitives. The work discussed in this section addresses part of the problem, the characterization of the information-flow involved and the definition of an appropriate *non-interference* property, serving as a building step for type-based approaches to information flow in the setting of data-centric software systems.

Given its recency, it seems more fitting to present these developments as related work, even though with a different timing they would be an appropriate contribution to our background, as our own approach overlaps with the authors' in the base characterization of the data manipulation primitives in the context of a λ_{DB}-inspired calculus. Despite the overlap, the work presented in this thesis distinguishes itself by focusing on the exploration of the advantages and limitations of using data-dependent logical propositions as security levels in such a setting.

As for the concrete type system, not presented in the article, the authors' hint at a differentiating factor from most value-oriented approaches – such as ours or λ_{SEC}^{REF} – regarding the meaning of the security level of an expression with type unit: it represents the level of the *least confidential* information that was *written* in the expression, while the remaining types' security levels represent the level of the *most confidential* information that was *read* in the expression, suggesting a similarity to the *write* and *read* levels used in the SMC calculus, a store-oriented approach presented in Subsection 2.1.3.

4

Language and Type System

In type-based information-flow analysis, the majority of the approaches [SM03] involve annotating every value with a security level. However, an equally expressive approach is introduced by Crary et al. [CKP05] in which security annotations are only added to references. They classify the common approaches as following a *value-oriented* view of security, in opposition to the *store-oriented* view of security in which their calculus, the Secure Monadic Calculus (SMC), is based.

During the early drafts of our type system, we tried to pursue this store-oriented view of security since our language is also centered in a store, the database, whose tables' fields have security policies. Although described in detail in Subsection 2.1.3, in general terms the approach uses two security levels r and w when typing an expression, corresponding to the *most restrictive* level of the information we *read* and to the *least restrictive* level of the information we *write*, respectively. The primary obstacle we found in our take to apply it to our setting, however, was the lack of precision when dealing with the records we use to model rows of the database tables. When we insert a record $[\mathrm{a} = x, \mathrm{b} = y]$, we are performing two independent write operations, in practice, but the analysis was unable to distinguish this independence and determined r and w security levels for the whole record, rejecting many programs which leaked no information.

As a consequence, we turned our attention to the value-oriented view of security followed by λ^{REF}_{SEC} and Flow Caml, where we annotate our types with security levels, in our case logical propositions. In this chapter, we begin by describing the core language we used to model our type system (Section 4.1), followed by the type system itself (Section 4.2) and the main extensions required to apply it to our prototype (Section 4.3). Finally, we end up with a general description of possible approaches to prove our type system guarantees a *non-interference* property (Section 4.4).

4.1 Core Language

In order to allow the core concepts of our information-flow type system to be modelled independently of the concrete language used in our prototype, we defined a core calculus largely similar to the λ_{DB} calculus presented in Subsection 2.2.1. The $\lambda_{DB-Flow}$ calculus is essentially a dependently-typed lambda calculus [Bar92] extended with booleans, collections, dependent records and database entities.

More concretely, each of the data types used in the $\lambda_{DB-Flow}$ calculus brings its own set of related constructs, besides the base lambda calculus' features of identifiers (x), lambda abstraction and function application. Although the grammar (Figure 4.1) divides these constructs between three main syntactic categories (terms, conditions and expressions), distinguishing the first two is only relevant for the type system: terms (v) represent irreducible language values that may occur in the security levels, while conditions (c) are simple boolean expressions that directly correspond to a fragment of the type language's formulas, and may appear in conditionals or where clauses.

The base boolean values (true, false) are accompanied by the conditional expression (if-then-else), term equality (=) and boolean operators, represented by conjunction (and) and negation (not). Other common boolean operators, such as disjunction (or) and implication ($\Rightarrow$), are easily defined in terms of those two.

Record values (r) are constituted by empty records, $[\ ]$, and by non-empty records recursively defined as terms of the form $[f = v, r]$, where the meta variable f ranges over field names. Although precise, this notation will be replaced with the equivalent notation $[f_1 = v_1, \ldots, f_n = v_n]$ where deemed clearer. Pairing with the record values' constructor is the field selection ($v.f$) operation, used to inspect the record's value associated with the label f. The primary motivation for the inclusion of records in the language is their use to model the *rows* of a database entity.

Collections are constructed as a list of n expressions, denoted $\{e_1, \ldots, e_n\}$, and are paired with expressions used to destructor collections in their first (head) and remaining values (tail). Additionally, the calculus features a mapping construct (map-in-to) to iterate the collection producing a new one, similar to LiveWeb's foreach-in-do construct. For example, consider:

```
map x in { true, false, true, false } to (not x)
```

When the map is executed, as the elements of the collection are iterated, they are bound to the identifier x and the body is executed, producing a collection with the same size but negated values, `{ false, true, false, true }`. The primary motivation for the inclusion of collections in the language is the necessity to model the fact that database entities may contain multiple rows or none at all.

Entities work as mutable collections of records, as already hinted, which can be created (create-in) with a name, ranged by the meta variable *ent*, and a schema defined by

Terms

v	$::=$		(Terms)
		true $\mid$ false	(Booleans)
	$\mid$	r	(Records)
	$\mid$	x	(Variables)
	$\mid$	$v.\mathrm{f}$	(Field Selection)
r	$::=$	$[\,]$	(Empty Record)
	$\mid$	$[\mathrm{f} = v, r]$	(Non-Empty Record)

Expressions

c	$::=$		(Conditions)
		$v = v$	(Equality)
	$\mid$	c and c	(Conjunction)
	$\mid$	not c	(Negation)
e	$::=$		(Expressions)
		v	(Term Values)
	$\mid$	$(\lambda[pc]\ x : \tau.e)$	(Abstraction)
	$\mid$	$e\ e$	(Application)
	$\mid$	c	(Boolean Conditions)
	$\mid$	if e then e else e	(Conditional)
	$\mid$	$\{\bar{e}\}$	(Collections)
	$\mid$	head e	(Head Projection)
	$\mid$	tail e	(Tail Projection)
	$\mid$	map x in e to e	(Collection Map)
	$\mid$	create $ent : \epsilon$ in e	(Entity Creation)
	$\mid$	from x in ent where c select e	(Row Selection)
	$\mid$	insert e into ent	(Row Insertion)
	$\mid$	update x in ent where c with e	(Row Update)
	$\mid$	delete x from ent where c	(Row Deletion)
	$\mid$	assume ℓ in e	(Knowledge Assumption)

Figure 4.1: $\lambda_{DB-Flow}$ grammar for expressions

a type ϵ. Similarly to λ_{DB} and LiveWeb, there are data manipulation primitives to insert (insert-into), select (from-in-where-select), update (update-in-with-where) and delete (delete-from-where) rows from the entities currently in scope. With the exception of the select query, the result of performing a query is a boolean value indicating its success.

Other basic data types, such as integers or strings, will be used in examples for illustrative purposes, but are not formally included because they present no additional challenges beyond the ones presented by booleans. Additionally, although let-expressions are not formally included, they are commonly defined as syntactic sugar for the application of a lambda abstraction to an expression:

$$\text{let } x = e_1 \text{ in } e_2 \triangleq (\lambda[pc]x : \tau.e_2)\ e_1$$

With pc being the current program counter and τ the type of expression e_1.

4.2 Type System

Our type system follows the standard approach [SM03] to type-based information-flow analysis – the value-oriented approach followed by λ_{SEC} and λ_{SEC}^{REF} – in that it is based in annotating every type with a security level. Like Flow Caml, we do not annotate literal values with security levels as in those calculi, but instead use the current program counter as their security level when determining their type. However, this difference is only apparent, as we can still force the security level of a literal at the type-level and achieve the same expressibility. Besides our domain, a major distinction from other information-flow analyses lies in our security levels, first-order logic propositions that may depend on runtime values, which motivate the need for $\lambda_{DB-Flow}$ to be a dependently-typed calculus in order to correctly model the dependency.

The type language (Figure 4.2) includes a type for each of the data types mentioned in the previous section, annotated with security levels: bool, the type of boolean values; $\Pi[pc]\ x : \alpha.\beta$, the type of dependent functions that take an argument of type α and produce a value of type β in a context (i.e. with program counter) pc; $[f_1 : \tau_1, \ldots, f_n : \tau_n]$, the type of dependent records (R) with n fields, f_1 through f_n, associated with types τ_1 through τ_n, whose considerations about notation are the same as with record values; τ^*, the type of collections whose elements have type τ; entity (R, p), the type of entities (ϵ) with rows of type R and local security policies p; and t_ℓ, the type of values with type t and security level ℓ, ranged over by the meta variables α, β and τ. Security levels include term equality ($=$), predicates with arbitrary arity, conjunction (and), negation (not) and universal quantification ($\forall_x$), ranged over by the meta variables ℓ, pc and p. The other typical first-order logic formulas of disjunction (or), implication ($\Rightarrow$) and existential quantification ($\exists_x$) are easily encoded in the included constructs.

Additionally, although not technically part of the type language, the importance of the assume expression lies only with the type system: the expression is used to increase

Types

$$\begin{array}{llll}
t & ::= & & \text{(Types)} \\
 & & \mathsf{bool} & \text{(Boolean Type)} \\
 & \mid & \prod[pc]\ x : \alpha.\beta & \text{(Dependent Function Types)} \\
 & \mid & R & \text{(Dependent Record Types)} \\
 & \mid & \tau^* & \text{(Collection Types)} \\
 & & & \\
R & ::= & [\,] & \text{(Empty Record Type)} \\
 & \mid & [\mathrm{f} : \tau, R] & \text{(Non-Empty Record Types)} \\
 & & & \\
\alpha, \beta, \tau & ::= & t_\ell & \text{(Labelled Types)} \\
 & & & \\
\epsilon & ::= & \mathsf{entity}\ (R, p) & \text{(Entity Types)}
\end{array}$$

Levels

$$\begin{array}{llll}
\ell, pc, p & ::= & & \text{(Security Levels)} \\
 & & v = v & \text{(Term Equality)} \\
 & \mid & P(v_1, \ldots, v_n) & \text{(n-ary Predicates)} \\
 & \mid & \ell\ \mathsf{and}\ \ell & \text{(Conjunction)} \\
 & \mid & \mathsf{not}\ \ell & \text{(Negation)} \\
 & \mid & \forall_x.\ell & \text{(Universal Quantification)}
\end{array}$$

Figure 4.2: $\lambda_{DB-Flow}$ grammar for types

the knowledge currently available, allowing us to set up information flow relations that are not native to first-order logic. For instance, if I have predicates `User(id)` and `System()`, respectively representing the information a user identified by `id` can read and the information that is confidential to the software system, I might want to say that the security level `System()` is higher than the security level `User(id)`, for any user.

```
assume (forall id: Int. User(id) ⇐ System()) in ...
```

Notice, however, that the same expression can be used like a *declassification* mechanism, "lowering" a security level ℓ to a lower security level ℓ' by assuming $\ell \Leftarrow \ell'$. For instance, to declassify information with level `System` to be readable by the user with id 2, we could perform the following assume locally:

```
assume (System() ⇐ User(2)) in ...
```

Thus, it is not completely clear when the assume expression is being used legitimately or not, that is, without compromising *non-interference*. Even so, we could argue that policies assumed initially for the whole program, i.e. *global policies*, are the exact equivalent of admitting a fixed security lattice in the approaches we studied and, consequently, are not enabling declassification.

In the following, we begin with a definition of our security lattice (Subsection 4.2.1), typing judgements (Subsection 4.2.2) and type operators used (Subsection 4.2.3), and proceed to discuss the typing rules for $\lambda_{DB-Flow}$'s constructs (Subsection 4.2.4).

4.2.1 Lattice

As already mentioned, the security levels used in $\lambda_{DB-Flow}$'s information-flow type system are first-order logic propositions in the style of λ_{DB}. Given a set of axioms $\mathcal{A}$, they form a lattice $(\mathcal{L}, \sqsubseteq)$ where, modulo logical equivalence:

- $\mathcal{L}$ is the set of all logical propositions;
- $\ell_1 \sqsubseteq \ell_2$ if, and only if, $\mathcal{A} \vdash \ell_2 \Rightarrow \ell_1$.

From this characterization, we are able to conclude that the lattice's:

- Minimum element ($\bot$) is the proposition true;
- Maximum element ($\top$) is the proposition false;
- Greatest lower bound operation ($\sqcap$) is the logical disjunction ($\vee$);
- Least upper bound operation ($\sqcup$) is the logical conjunction ($\wedge$).

This corresponds to the intuition that public information carries no specific knowledge, as proposition true trivally holds. Conversely, by constraining the security level, the knowledge gets more and more specific to the point where it does not hold, as is the case for proposition false.

A proof that the described pair $(\mathcal{L}, \sqsubseteq)$ defines a lattice can be found in Appendix A.

4.2.2 Typing Judgements

$$\begin{array}{lcll} \Gamma & ::= & \cdot \mid \Gamma, x : \tau & \text{(Type Environment)} \\ \Delta & ::= & \cdot \mid \Delta, \ell & \text{(Knowledge Environment)} \end{array}$$

Figure 4.3: $\lambda_{DB-Flow}$ grammar for environments

In the typing rules we use three kinds of typing judgements, one to state the type of expressions, one to state subtyping relations and one to state the validity of propositions.

- $\Gamma; \Delta \ [pc] \vdash e : \tau$ states that expression e has type τ in a context with program counter pc under the type environment Γ, mapping names to types, and given the knowledge environment Δ, which is composed by a set of axioms known to be true;
- $\Gamma; \Delta \vdash \alpha <: \beta$ states that the type α is a subtype of the type β, under the type environment Γ and given the knowledge environment Δ;

- $\Delta \vdash \ell$ states that from the knowledge environment Δ, we may conclude that proposition ℓ is valid.

The knowledge environment plays a similar role to the use of refined unit values in the type environment of λ_{DB} to assert additional knowledge, that is, $\lambda_{DB-Flow}$'s knowledge environment $\Delta = \ell_1, \ldots, \ell_n$ roughly corresponds to a type environment consisting only of such assertions, $\_ : \{\_ : \mathsf{unit} \mid \ell_1\}, \ldots, \_ : \{\_ : \mathsf{unit} \mid \ell_n\}$ in λ_{DB}.

The program counter plays the same role as in λ_{SEC} and λ_{SEC}^{REF}, serving as a reference of the security level of the information that the current control-flow of the program depends on. Any expression that occurs in the program is guaranteed by the type system to have *at least* security level equivalent to the respective program counter, resulting both from its use as a default security level for literals, and from the fact that the remaining expressions determine their levels by combining their sub-expressions' security levels in some non-decreasing way.

4.2.3 Type Operators and Notation

The operator $\sqcup$ between two security levels is well defined as the least upper bound operation. However, there is often the need to *increase* the security level of a given type $\tau = t_{\ell_1}$ with a security level ℓ_2, resulting in $t_{\ell_1 \sqcup \ell_2}$, which led to the shortcut notation $\tau \sqcup \ell_2$, already introduced in [Zda02]. We slightly extend the notation to be applicable to unlabelled record types, with the semantics of applying the $(\sqcup\ \ell)$ operation to the type of each field, as defined in Figure 4.4.

$$\begin{aligned} t_{\ell_1} \sqcup \ell_2 &= t_{\ell_1 \sqcup \ell_2} \\ [\,] \sqcup \ell &= [\,] \\ [f : \tau, R] \sqcup \ell &= [f : \tau \sqcup \ell, R \sqcup \ell] \end{aligned}$$

Figure 4.4: The $\sqcup$ operation between types and security levels

Given that our security levels can reference runtime values, when we exit the scope in which those values were given names, there is the need to *close* the free occurrences of those names in some sensible way. We close the free occurrences of a name x in a type by universally quantifying x in all the (structurally reachable) security levels which have free occurrences, an operation designated as $[\![\cdot]\!]^x$ and defined in Figure 4.5. By universally quantifying the security levels, we maintain the *monotonicity* of the information-flow type system because their confidentiality is *increased* – the proposition $P(a) \Leftarrow \forall_x.P(x)$ is valid for any property P and value a.

Notice that when we apply $[\![\cdot]\!]^x$ to dependent function or record types, there is the need to distinguish the case when x is equal to the parameter or field name, since x is no longer a free name in that case: the parameter of dependent functions is bound in both

$$\begin{aligned}
[\![\ell]\!]^x &= \forall_x.\ \ell \\
[\![t_\ell]\!]^x &= t_{[\![\ell]\!]^x} \\
[\![\mathsf{bool}]\!]^x &= \mathsf{bool} \\
[\![\Pi[pc]\ y : \alpha.\beta]\!]^x &= \Pi[pc]\ y : \alpha.\beta && \text{if } x = y \\
[\![\Pi[pc]\ y : \alpha.\beta]\!]^x &= \Pi[[\![pc]\!]^x]\ y : [\![\alpha]\!]^x.[\![\beta]\!]^x && \text{if } x \neq y \\
[\![[\,]]\!]^x &= [\,] \\
[\![[f : \tau, R]]\!]^x &= [f : \tau, R] && \text{if } x = f \\
[\![[f : \tau, R]]\!]^x &= [f : [\![\tau]\!]^x, [\![R]\!]^x] && \text{if } x \neq f \\
[\![\tau^*]\!]^x &= ([\![\tau]\!]^x)^*
\end{aligned}$$

Figure 4.5: The $[\![\cdot]\!]^x$ operation applied to types and formulas

pc, α and β, and a dependent record's field name is bound in its own type τ and the types of all subsequent fields.

The set $subr(R)$, defined recursively in Figure 4.6, is the set of all record types that are considered a *subrecord* of the record type R. Informally, a subrecord of R is a record type whose set of fields is a subset of R's fields, with the same types. We use the notation $(\!|R|\!)$ to denote a record type that is a subrecord of R.

$$\begin{aligned}
subr([\,]) &= \{[\,]\} \\
subr([f : \tau, R]) &= \{[f : \tau, R'] \mid R' \in subr(R)\} \\
&\quad \cup \{R' \mid R' \in subr(R) \wedge f \text{ is not free in } R'\}
\end{aligned}$$

Figure 4.6: Definition of subrecord

Finally, we use the notation $\tau\{w/v\}$ to represent the type that results from replacing all the occurrences of value v in τ with the value w. Similarly, $\ell\{w/v\}$ represents the formula that results from replacing all occurrences of value v in ℓ with the value w. For multiple substitutions, we adopt the notation $\{\overline{w_i/v_i}\}_{1\leq i\leq n}$ to mean the same as $\{w_1/v_1\}\ldots\{w_n/v_n\}$.

4.2.4 Typing Rules

In this subsection we present the typing rules and intuitions behind them, using small examples where necessary. We begin by discussing a set of base typing and subtyping rules with no particular *theme*, and then proceed to group them by data type similarly to the description of the constructs in Section 4.1.

Base typing and subtyping

The typing rule for variables works as expected, inspecting the type environment to find the variable's type, but also increases its security level with that of the program counter (*pc*) to guarantee it is at least as restrictive.

$$\text{Variable} \quad \frac{\Gamma(x) = \tau}{\Gamma; \Delta\ [pc] \vdash x : \tau \sqcup pc}$$

The same occurs with variables in λ^{REF}_{SEC}, but not as directly given the separation of literal values and other expressions in the type system, requiring the combination of the typing rules for variables, values and the use of subsumption.

Assume-expressions have a type τ determined by typing the expression e in an environment increased with knowledge ℓ.

$$\text{Assume} \quad \frac{\Gamma; \Delta, \ell\ [pc] \vdash e : \tau}{\Gamma; \Delta\ [pc] \vdash \mathsf{assume}\ \ell\ \mathsf{in}\ e : \tau}$$

The approach to subsumption is standard, stating that an expression with type α can be said to have type β, as long as α is a subtype of β.

$$\text{Subsumption} \quad \frac{\Gamma; \Delta\ [pc] \vdash e : \alpha \qquad \Gamma; \Delta \vdash \alpha <: \beta}{\Gamma; \Delta\ [pc] \vdash e : \beta}$$

$$\text{Identity-Subtyping} \quad \frac{}{\Gamma; \Delta \vdash \tau <: \tau}$$

Security levels follow the same principle, with the Level-Subtyping rule reflecting the fact that it is safe to increase the confidentiality of a value by raising its security level. Exceptionally, u is used as an unlabelled type distinct from t to keep the rule readable.

$$\text{Level-Subtyping} \quad \frac{\Gamma; \Delta \vdash t <: u \qquad \Delta \vdash \ell_1 \Leftarrow \ell_2}{\Gamma; \Delta \vdash t_{\ell_1} <: u_{\ell_2}}$$

Booleans

Boolean values true and false are trivially typed as booleans whose security level is the current program counter, since their creation depends on the current control-flow of the program.

$$\text{Boolean-True} \quad \frac{}{\Gamma; \Delta\ [pc] \vdash \mathsf{true} : \mathsf{bool}_{pc}}$$

$$\text{Boolean-False} \quad \frac{}{\Gamma; \Delta\ [pc] \vdash \mathsf{false} : \mathsf{bool}_{pc}}$$

Negation (not), conjunction (and) and equality (=) are typed conservatively with a security level obtained by performing the least upper bound ($\sqcup$) of their sub-expressions' security levels. As expected, the sub-expressions of negation and conjunction must be well-typed as booleans, while equality's sub-expressions are well-typed with any labelled type τ, as long as it is common to both.

$$\text{Negation}\quad \frac{\Gamma;\Delta\ [pc] \vdash c : \mathsf{bool}_\ell}{\Gamma;\Delta\ [pc] \vdash \mathsf{not}\ c : \mathsf{bool}_\ell}$$

$$\text{Conjunction}\quad \frac{\Gamma;\Delta\ [pc] \vdash c_1 : \mathsf{bool}_{\ell_1} \qquad \Gamma;\Delta\ [pc] \vdash c_2 : \mathsf{bool}_{\ell_2}}{\Gamma;\Delta\ [pc] \vdash c_1\ \mathsf{and}\ c_2 : \mathsf{bool}_{(\ell_1 \sqcup \ell_2)}}$$

$$\text{Equality}\quad \frac{\Gamma;\Delta\ [pc] \vdash v_1 : t_{\ell_1} \qquad \Gamma;\Delta\ [pc] \vdash v_2 : t_{\ell_2}}{\Gamma;\Delta\ [pc] \vdash v_1 = v_2 : \mathsf{bool}_{(\ell_1 \sqcup \ell_2)}}$$

The conditional (if-then-else) requires two typing rules simply because of syntactic flexibility. The first rule, Conditional-E, is applicable with any expression as its condition and directly corresponds to the typing rule for conditionals in λ_{SEC}^{REF}, with both branches typed in a context with a raised program counter $pc \sqcup \ell$ to model their dependency on the information read in the condition e_1.

$$\text{Conditional-E}\quad \frac{\Gamma;\Delta\ [pc] \vdash e_1 : \mathsf{bool}_\ell \qquad \Gamma;\Delta\ [pc \sqcup \ell] \vdash e_2 : \tau \qquad \Gamma;\Delta\ [pc \sqcup \ell] \vdash e_3 : \tau}{\Gamma;\Delta\ [pc] \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

The second rule, Conditional-C, is only applicable to conditionals whose condition belongs to the syntactical category c, enabling its direct use as an axiom in the knowledge environment Δ. Consequently, we type the first branch with the increased knowledge that c holds, and type the second branch with the increased knowledge that c does not hold, following the example of λ_{DB}.

$$\text{Conditional-C}\quad \frac{\begin{array}{c}\Gamma;\Delta\ [pc] \vdash c : \mathsf{bool}_\ell \\ \Gamma;\Delta, c\ [pc \sqcup \ell] \vdash e_1 : \tau \qquad \Gamma;\Delta, \neg c\ [pc \sqcup \ell] \vdash e_2 : \tau\end{array}}{\Gamma;\Delta\ [pc] \vdash \mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

Example 1 In order to clarify the use of Conditional-C and illustrate the use of the typing rules introduced thus far, consider an example based on the following program fragment:

```
if x = 1 then b else false
```

Suppose that $\Gamma = \{x : \mathsf{int}_{\mathsf{true}},\ b : \mathsf{bool}_{P(x)}\}$, $pc = \mathsf{true}$ and we want to prove that the conditional expression above has type $\mathsf{bool}_{P(1)}$. Intuitively, since the condition only depends on public values, the security level of the branches does not rise.

$$\frac{\dfrac{\Gamma(x) = \mathsf{int}_{\mathsf{true}}}{\Gamma;\cdot\ [\mathsf{true}] \vdash x : \mathsf{int}_{\mathsf{true}}} \qquad \Gamma;\cdot\ [\mathsf{true}] \vdash 1 : \mathsf{int}_{\mathsf{true}}}{\Gamma;\cdot\ [\mathsf{true}] \vdash x = 1 : \mathsf{bool}_{\mathsf{true}}}$$

Consequently, the result of the second branch is also a public value, enabling us to raise its security level to $P(1)$ using subsumption.

$$\frac{\Gamma; x \neq 1\ [\mathsf{true}] \vdash \mathsf{false} : \mathsf{bool}_{\mathsf{true}} \qquad \dfrac{\Gamma; x \neq 1 \vdash \mathsf{bool} <: \mathsf{bool} \qquad \Delta \vdash \mathsf{true} \Leftarrow P(1)}{\Gamma; x \neq 1 \vdash \mathsf{bool}_{\mathsf{true}} <: \mathsf{bool}_{P(1)}}}{\Gamma; x \neq 1\ [\mathsf{true}] \vdash \mathsf{false} : \mathsf{bool}_{P(1)}}$$

Now, by observing that the result of the first branch, b, has security level $P(x)$, we can use the knowledge that $x = 1$ to prove b has security level $P(1)$.

$$\dfrac{\dfrac{\Gamma(b) = \mathsf{bool}_{P(x)}}{\Gamma; x = 1\ [\mathsf{true}] \vdash b : \mathsf{bool}_{P(x)}} \qquad \dfrac{\Gamma; x = 1 \vdash \mathsf{bool} <: \mathsf{bool} \qquad x = 1 \vdash P(x) \Leftarrow P(1)}{\Gamma; x = 1 \vdash \mathsf{bool}_{P(x)} <: \mathsf{bool}_{P(1)}}}{\Gamma; x = 1\ [\mathsf{true}] \vdash b : \mathsf{bool}_{P(1)}}$$

Finally, if we join the intermediate deductions together we are able to apply Conditional-C and prove the program fragment is well-typed as a boolean with security level $P(1)$, as intended.

$$\dfrac{\begin{array}{c}\Gamma; \cdot\ [\mathsf{true}] \vdash x = 1 : \mathsf{bool}_{\mathsf{true}} \\ \Gamma; x = 1\ [\mathsf{true}] \vdash b : \mathsf{bool}_{P(1)} \qquad \Gamma; x \neq 1\ [\mathsf{true}] \vdash \mathsf{false} : \mathsf{bool}_{P(1)}\end{array}}{\Gamma; \cdot \vdash \mathsf{if}\ x = 1\ \mathsf{then}\ b\ \mathsf{else\ false} : \mathsf{bool}_{P(1)}}$$

Functions

Apart from the peculiarities of dependent functions, all the typing rules concerning functions follow the same logic regarding information-flow as the ones in λ^{REF}_{SEC}. In function abstraction in particular, the only difference resides in the fact that we default the security level of standalone values to be the current program counter, pc_1.

$$\text{Abstraction} \quad \dfrac{\Gamma, x : \alpha; \Delta\ [pc_2] \vdash e : \beta}{\Gamma; \Delta\ [pc_1] \vdash (\lambda[pc_2]\ x : \alpha.e) : (\textstyle\prod[pc_2]\ x : \alpha.\beta)_{pc_1}}$$

Similarly to the conditional expression, function application has two distinct rules in order to be precisely typed: one that only allows terms (v) as arguments but has increased precision, and other that allows any expression as argument but is more conservative. The first rule, Application-V, closes the free occurrences of the parameter x by substituting them with the argument v, while the second rule, Application-E, closes the free occurrences of the parameter x using the $[\![\cdot]\!]^x$ operator presented earlier.

$$\text{Application-V} \quad \dfrac{\begin{array}{c}\Gamma; \Delta\ [pc_1] \vdash e_1 : (\prod[pc_2]\ x : \alpha.\beta)_\ell \\ \Gamma; \Delta\ [pc_1] \vdash v : \alpha\{x/v\} \qquad \Delta \vdash pc_1 \Leftarrow pc_2\{v/x\}\end{array}}{\Gamma; \Delta\ [pc_1] \vdash e_1\ v : \beta\{v/x\} \sqcup \ell}$$

$$\text{Application-E} \quad \dfrac{\begin{array}{c}\Gamma; \Delta\ [pc_1] \vdash e_1 : (\prod[pc_2]\ x : \alpha.\beta)_\ell \\ \Gamma; \Delta\ [pc_1] \vdash e_2 : \alpha \qquad \Delta \vdash pc_1 \Leftarrow [\![pc_2]\!]^x\end{array}}{\Gamma; \Delta\ [pc_1] \vdash e_1\ e_2 : [\![\beta]\!]^x \sqcup \ell}$$

Both rules determine if the argument v has the right type α and ensure the call can only be made if the function executes with a higher (or equal) program counter (pc_2) than the current one (pc_1), guaranteeing no information with security level lower than pc_2 is written in the database (as will be shown further ahead).

Example 2 To clarify the differences between both rules, consider the application of a function $f : (\Pi[\mathsf{true}]\ x : \mathsf{int}_{\mathsf{true}}.\mathsf{int}_{Q(x)})_{\mathsf{true}}$ to the value 2, in a public context. If we apply the Application-V rule, the result of the call is proven to have security level $Q(2)$.

$$\frac{\Gamma; \cdot\ [\mathsf{true}] \vdash f : (\Pi[\mathsf{true}]\ x : \mathsf{int}_{\mathsf{true}}.\mathsf{int}_{Q(x)})_{\mathsf{true}} \qquad \Gamma; \cdot\ [\mathsf{true}] \vdash 2 : \mathsf{int}_{\mathsf{true}} \qquad \Delta \vdash \mathsf{true} \Leftarrow \mathsf{true}}{\Gamma; \cdot\ [\mathsf{true}] \vdash f\ 2 : \mathsf{int}_{Q(2)}}$$

On the other hand, if we apply Application-E rule, the result of the call is proven to have the higher security level $\forall_x.Q(x)$.

$$\frac{\Gamma; \cdot\ [\mathsf{true}] \vdash f : (\Pi[\mathsf{true}]\ x : \mathsf{int}_{\mathsf{true}}.\mathsf{int}_{Q(x)})_{\mathsf{true}} \qquad \Gamma; \cdot\ [\mathsf{true}] \vdash 2 : \mathsf{int}_{\mathsf{true}} \qquad \Delta \vdash \mathsf{true} \Leftarrow \mathsf{true}}{\Gamma; \cdot\ [\mathsf{true}] \vdash f\ 2 : \mathsf{int}_{\forall_x.Q(x)}}$$

Example 3 The loss of precision from Application-E can even invalidate its application. Consider the application of a function with type $g : (\Pi[\mathsf{true}]\ x : \mathsf{int}_{P(x)}.\mathsf{int}_{Q(x)})_{\mathsf{true}}$ to the value 3, in a public context. In order to apply the Application-V rule, we must use subsumption to raise our argument's security level to $P(3)$, obtaining a value with security level $Q(3)$ as a result of the function.

$$\frac{\ldots \qquad \dfrac{\Gamma; \cdot\ [\mathsf{true}] \vdash 3 : \mathsf{int}_{\mathsf{true}} \qquad \Gamma; \cdot \vdash \mathsf{int}_{\mathsf{true}} <: \mathsf{int}_{P(3)}}{\Gamma; \cdot\ [\mathsf{true}] \vdash 3 : \mathsf{int}_{P(3)}} \qquad \Delta \vdash \mathsf{true} \Leftarrow \mathsf{true}}{\Gamma; \cdot\ [\mathsf{true}] \vdash g\ 3 : \mathsf{int}_{Q(3)}}$$

Applying the Application-E rule, however, would imply using subsumption to raise our argument's security level to $P(x)$, which is illegal because x is not bound in this scope.

In the rule for dependent function subtyping, the same rules apply as with normal functions besides the need to have the parameter x bound in the current scope. The program counter varies contravariantly, corresponding to the idea that functions with a higher pc may be called in, *at least*, the same contexts as a function with a lower pc.

$$\text{Function-Subtyping} \quad \frac{\begin{array}{c}\Delta \vdash pc_2 \Leftarrow pc_1 \\ \Gamma, x : \alpha_2; \Delta \vdash \alpha_2 <: \alpha_1 \qquad \Gamma, x : \alpha_2; \Delta \vdash \beta_1 <: \beta_2\end{array}}{\Gamma; \Delta \vdash \prod[pc_1]\ x : \alpha_1.\beta_1 <: \prod[pc_2]\ x : \alpha_2.\beta_2}$$

Records

Since records are defined recursively, there are two distinct rules used to type them. The rule for empty records, Record-Empty, is a trivial rule which uses the current program counter as security level, similarly to other values.

$$\text{Record-Empty} \quad \frac{}{\Gamma; \Delta\ [pc] \vdash [\,] : [\,]_{pc}}$$

The rule for non-empty records, Record-Fields, is defined recursively in such a way that the value v associated to each field f is substituted in all subsequent fields' types (and its own) by f, since the field name is bound both in the type τ and the types of the remaining fields.

$$\text{Record-Fields}\quad \frac{\Gamma;\Delta\ [pc] \vdash v : \tau\{v/\mathrm{f}\} \qquad \Gamma;\Delta\ [pc] \vdash r : (R\{v/\mathrm{f}\})_{pc}}{\Gamma;\Delta\ [pc] \vdash [\mathrm{f} = v, r] : [\mathrm{f} : \tau, R]_{pc}}$$

In turn, field selection requires all the field names to be substituted in the selected field's type to guarantee it has no free names. As is common with data type destructors, the resulting type is augmented with the record's security level.

$$\text{Field-Selection}\quad \frac{\Gamma;\Delta\ [pc] \vdash v : [\mathrm{f}_1 : \tau_1, \ldots, \mathrm{f}_n : \tau_n]_\ell \qquad 1 \le i \le n}{\Gamma;\Delta\ [pc] \vdash v.\mathrm{f}_i : \tau_i\{\overline{v.\mathrm{f}_j/\mathrm{f}_j}\}_{i\le j\le n} \sqcup \ell}$$

Record subtyping is defined recursively by two main typing rules in order to exploit the record's structure. The first rule, Record-Subtyping-Fields, states the subtyping relation between two records starting with the same field. The second rule, Record-Subtyping-Swap, allows the order of fields to be swapped as long as their names do not occur in each other's types.

$$\text{Record-Subtyping-Fields}\quad \frac{\Gamma, \mathrm{f} : \alpha; \Delta \vdash \alpha <: \beta \qquad \Gamma, \mathrm{f} : \alpha; \Delta \vdash R_1 <: R_2}{\Gamma;\Delta \vdash [\mathrm{f} : \alpha, R_1] <: [\mathrm{f} : \beta, R_2]}$$

$$\text{Record-Subtyping-Swap}\quad \frac{\mathrm{f}_1 \text{ does not occur in } \tau_2 \qquad \mathrm{f}_2 \text{ does not occur in } \tau_1}{\Gamma;\Delta \vdash [\mathrm{f}_1 : \tau_1, \mathrm{f}_2 : \tau_2, R] <: [\mathrm{f}_2 : \tau_2, \mathrm{f}_1 : \tau_1, R]}$$

Together they define the informal idea of record subtyping where a record R_1 is a subtype of a record R_2 if, and only if, the set of R_1's fields is the equal to the set of R_2's fields and there is a valid ordering (according to the swapping rule) that allows the subtyping relation to be proven pairwise for their types. There is no support for the standard *width subtyping*, which only requires R_1's fields to be a superset of R_2's fields, because it overcomplicates the typing rule for update in a way that obfuscates the important details.

Finally, one can observe that contrary to booleans, functions and collections, destructuring a record does not give any new information on its structure that is not already tracked by the type system. Consequently, the security level associated with a record may be safely distributed by all the fields and lowered to $\bot$.

$$\text{Record-Subtyping-Label-In}\quad \frac{}{\Gamma;\Delta \vdash R_\ell <: (R \sqcup \ell)_\bot}$$

By the same reasons, if we are able to identify a common security level of every field in the record, it is safe to augment the record's security level with it.

$$\text{Record-Subtyping-Label-Out}\quad \frac{}{\Gamma;\Delta \vdash (R \sqcup \ell)_\bot <: R_\ell}$$

The only reason for record types to have a security level associated with their structure is consistency with the rest of the type system.

Collections

A collection value is well-typed as a collection τ^*_{pc}, if τ is a common type for all its elements. Similarly to other values, the program counter is used as the security level of the collection, in order to reflect the informations revealed by the program's control-flow when the evaluation reaches this expression.

$$\text{Collection}\quad \frac{\Gamma;\Delta\ [pc] \vdash e_i : \tau \qquad \text{for all } i \in [1,n]}{\Gamma;\Delta\ [pc] \vdash \{e_1,\ldots,e_n\} : \tau^*_{pc}}$$

Since the head operation works as a typical destructor, with the absence of a run-time error revealing that the collection is not empty, its type is that of the elements of the collection augmented with the security level of the collection itself.

$$\text{Head-Projection}\quad \frac{\Gamma;\Delta\ [pc] \vdash e : \tau^*_\ell}{\Gamma;\Delta\ [pc] \vdash \mathsf{head}\ e : \tau \sqcup \ell}$$

The typing rule for the tail operation states that the tail of a collection e has exactly the same type as the collection e itself.

$$\text{Tail-Projection}\quad \frac{\Gamma;\Delta\ [pc] \vdash e : \tau^*_\ell}{\Gamma;\Delta\ [pc] \vdash \mathsf{tail}\ e : \tau^*_\ell}$$

Iterating a collection while executing an expression with side-effects reveals information on whether the collection is empty or not. To account for that, the typing rule for the map construct types the body expression e_2 in a context where the program counter is augmented with the collection's security level. Additionally, to prevent the cursor x from being a free name of the resulting element type, β, we need to close the free names using the $[\![\cdot]\!]^x$ operation.

$$\text{Collection-Map}\quad \frac{\Gamma;\Delta\ [pc] \vdash e_1 : \alpha^*_\ell \qquad \Gamma, x:\alpha;\Delta\ [pc \sqcup \ell] \vdash e_2 : \beta}{\Gamma;\Delta\ [pc] \vdash \mathsf{map}\ x\ \mathsf{in}\ e_1\ \mathsf{to}\ e_2 : ([\![\beta]\!]^x)^*_\ell}$$

Since we are working with immutable collections, it is safe to state that collections of elements with type α are a subtype of collections of elements with type β.

$$\text{Collection-Subtyping}\quad \frac{\Gamma;\Delta \vdash \alpha <: \beta}{\Gamma;\Delta \vdash \alpha^* <: \beta^*}$$

Database entities

Entities are not first-class values of the language, but declarations visible in a given expression e. Accordingly, the typing rule for the creation of entities states the whole expression is well-typed with type τ, if expression e is well-typed with type τ in a type environment which includes the entity.

$$\text{Create-Entity}\quad \frac{\Gamma, ent:\epsilon;\Delta\ [pc] \vdash e : \tau}{\Gamma;\Delta\ [pc] \vdash \mathsf{create}\ ent:\epsilon\ \mathsf{in}\ e : \tau}$$

The typing rule for insertion states that the type of the record we are inserting, R, must be the same as the entity's schema. In practice, since we can use subsumption, the rule states that the record type obtained by typing expression e must be a subtype of the entity's schema, R, which corresponds to the requirement that the information we are *writing* cannot have a higher security level than the information we are able to *read* later from the database, the same logic used with reference types in λ^{REF}_{SEC}.

$$\text{Insert}\quad \frac{\Gamma(ent) = \mathsf{entity}\ (R, \_) \qquad \Gamma; \Delta\ [pc] \vdash e : R_\bot}{\Gamma; \Delta\ [pc] \vdash \mathsf{insert}\ e\ \mathsf{into}\ ent : \mathsf{bool}_{pc}}$$

Although the rule seems to require that e must be a public expression, recall that we are able to distribute the security level of a record by its fields using the Record-Subtyping-In rule, lowering the security level of the record type itself to $\bot$. Notice also that the entity's local policies are irrelevant because we are inserting a *new* row, with no relation to the entity.

Example 4 Consider a User entity with schema $R_{\mathsf{User}} = [\mathrm{id} : \mathsf{int}_{\mathsf{true}}, \ \mathrm{name} : \mathsf{string}_{\mathsf{false}}]$, and the following expression:

```
insert [ id = 4, name = "Delta" ] into User
```

If we type the expression in a context with $pc = \mathsf{true}$, the type of the record would be $[\mathrm{id} : \mathsf{int}_{\mathsf{true}}, \ \mathrm{name} : \mathsf{string}_{\mathsf{true}}]$. Since both fields' types are subtypes of their homonyms in the User's schema, which means their security levels are *at most* as restrict, we may assert no information is leaked by performing the insert query.

On the other hand, if we type the expression in a context with $pc = \mathsf{false}$, the type of the record would be $[\mathrm{id} : \mathsf{int}_{\mathsf{false}}, \ \mathrm{name} : \mathsf{string}_{\mathsf{false}}]$, which is not a subtype of the User's schema because the record's id field has a higher security level (false) than the one expected by the entity (true). Consequently, the insertion is rejected as ill-typed because it leaks private information to the public field id.

Similarly to λ_{DB}, the where clause of a query plays a key role in its typing rule. In $\lambda_{DB-Flow}$, not only does the expression c contribute to the knowledge that is available, but also its security level affects the current program counter in the same way the condition of a if-expression does.

The typing rule for the select expression embodies this principle by typing the expression e in an environment augmented with the knowledge from the where condition, c, and with a program counter raised by its security level, $pc \sqcup \ell$. The whole select expression is typed as a collection of the selected elements' type, with the free occurrences of the cursor x closed with $[\![\cdot]\!]^x$.

$$\text{Select}\quad \frac{\begin{array}{c}\Gamma(ent) = \mathsf{entity}\ (R, p) \qquad R = [f_1 : \tau_1, \ldots, f_n : \tau_n] \qquad P = p\{\overline{x.f_j/f_j}\}_{1 \le j \le n} \\ \Gamma, x : R_\bot; \Delta, P\ [pc] \vdash c : \mathsf{bool}_\ell \qquad\qquad \Gamma, x : R_\bot; \Delta, P, c\ [pc \sqcup \ell] \vdash e : \beta\end{array}}{\Gamma; \Delta\ [pc] \vdash \mathsf{from}\ x\ \mathsf{in}\ ent\ \mathsf{where}\ c\ \mathsf{select}\ e : [\![\beta^*_\ell]\!]^x}$$

Notice also that, since the cursor x represents an actual row from the entity, the entity's local policies p hold for each specific record that is retrieved. Consequently, both the condition c and the expression e are typed in environments augmented with the knowledge P that the local policies hold for x. The same is evident in the typing rules for the update and delete expressions.

Example 5 Consider the same User entity from the last example and suppose we want to give a type to the following expression with $pc = \textsf{true}$:

```
from u in User where u.name = "Epsilon" select u.id
```

Since we are filtering the results of the query depending on the name field, which is private, the selected expression will be typed in a private context (i.e. $pc = \textsf{false}$). Consequently, although the id field is public, the result of the query is a collection of private integers, $(\textsf{int}_{\textsf{false}})^*_{\textsf{false}}$, reflecting the fact that we know they are the ids associated with a user named "Epsilon", a private information.

The typing rule for the update expression works, fittingly, as a mixture of the typing rules for the insert expression and the select expression. Since we are iterating the entity's rows, the typing rule looks a lot like the one for select, but since our goal is to *write* updated records, the type of the record we use to express the update must be a subrecord of the entity's schema.

$$\text{Update}\quad \frac{\begin{array}{c}\Gamma(ent) = \textsf{entity}\ (R, p) \qquad R = [f_1 : \tau_1, \ldots, f_n : \tau_n] \qquad P = p\{\overline{x.f_j / f_j}\}_{1 \leq j \leq n} \\ \Gamma, x : R_\perp; \Delta, P\ [pc] \vdash c : \textsf{bool}_\ell \qquad\quad \Gamma, x : R_\perp; \Delta, P, c\ [pc \sqcup \ell] \vdash e : (\!|R|\!)_\perp\end{array}}{\Gamma; \Delta\ [pc] \vdash \textsf{update}\ x\ \textsf{in}\ ent\ \textsf{where}\ c\ \textsf{with}\ e : [\![\textsf{bool}_\ell]\!]^x}$$

Example 6 Consider the same User entity from the previous examples and suppose we want to determine if the following expression is well-typed with $pc = \textsf{true}$:

```
update u in User where u.name = "Zeta" with [ id = 6 ]
```

Since we are filtering the entity's rows by the name field, which is private, we should only be able to update private fields in order to avoid leaking information about which rows are named "Zeta". Accordingly, the type of the updating record is $[\text{id} : \textsf{int}_{\textsf{false}}]$ which is not a subtype of the entity schema's subrecord $[\text{id} : \textsf{int}_{\textsf{true}}]$.

Typing the delete expression directly corresponds to performing an update query in which all the entity's fields are updated with themselves. Another way to look at the typing rule is to consider that a public observer will only notice the deleted rows are absent if they contain public information. Consequently, any private information that the query depends on will be leaked to the public observer, just as if we performed an update under the same conditions.

$$\text{Delete}\quad \frac{\begin{array}{c}\Gamma(ent) = \textsf{entity}\ (R, p) \qquad R = [f_1 : \tau_1, \dots, f_n : \tau_n] \qquad P = p\{\overline{x.f_j / f_j}\}_{1 \le j \le n} \\ \Gamma, x : R_\perp; \Delta, P\ [pc] \vdash c : \textsf{bool}_\ell \qquad \Gamma, x : R_\perp; \Delta, P, c\ [pc \sqcup \ell] \vdash x : R_\perp\end{array}}{\Gamma; \Delta\ [pc] \vdash \textsf{delete}\ x\ \textsf{from}\ ent\ \textsf{where}\ c : [\![\textsf{bool}_\ell]\!]^x}$$

Example 7 Consider the same User entity from the previous examples and suppose we want to determine if the following expression is well-typed with $pc = \textsf{false}$:

```
delete u from User where u.id = 7
```

As the lowest security level of a User's fields is public, performing a delete operation which depends on the entity's public id incurs in no information leak. However, since $pc = \textsf{false}$, the records that we want to delete are typed as $[\text{id} : \textsf{int}_{\textsf{false}}, \text{name} : \textsf{string}_{\textsf{false}}]_\perp$, whose record type is not a subtype of R_{User}, reflecting the fact that the *act* of performing that delete operation depends on private information.

4.3 Extended LiveWeb

The type system we developed for $\lambda_{DB-Flow}$ was validated experimentally by means of an extension to the LiveWeb framework, whose implementation is discussed in Chapter 6. Since both expression languages are very similar, the type rules for LiveWeb's expressions closely resemble the ones we defined for our core calculus. Consequently, the greatest novelty when defining the typing rules for LiveWeb's constructs lies in its web block language, which is the focus of this section.

Blocks are the way LiveWeb interacts with an external user, allowing the output and input of information, a concern that is not addressed directly in any of the calculi we studied. Ur/Web, discussed in Section 3.1, addresses the problem by using the mechanism of *queries-as-policies* to define an upper bound on the information that is sent to the client, and uses a special predicate to mark the information that comes from the user. FlowCaml, discussed in Section 3.2, addresses the problem in the context of a value-oriented information-flow type system by defining two special security levels, `!stdin` and `!stdout`, which are used in the type of the input and output primitives, respectively.

Inspired by FlowCaml's approach, we defined a special security level, `Input()`, which is the level of all the information that is input through the web block primitives. For the output, however, the approach of requiring every screen to have the same result security level, `Output()`, seemed limiting as it forces a binary distinction between values that can always be sent to the client and values that can never be sent to the client. Consequently, the output level ℓ of each screen is specified individually and the blocks in its body cannot contain information with security level higher than ℓ, an approach that directly coincides with the original idea inspired in FlowCaml if every screen's output level is `Output()`.

4.3.1 Typing Judgements

The typing judgements we use in LiveWeb's type system are the ones used in $\lambda_{DB-Flow}$, plus a new typing judgement for blocks. The necessity of a new judgement stems from the fact that blocks are composed sequentially in such a way that a name declared in a block is bound in all the blocks that follow, i.e. it is not local.

The judgement $\Gamma_1; \Delta\ [pc] \vdash b : \mathsf{block}_\ell \dashv \Gamma_2$ states that the block b is well typed with security level ℓ in a context with program counter pc, under a type environment Γ_1 and a knowledge environment Δ, and produces the new typing environment Γ_2. Its typical use is apparent in the typing rule for block sequencing below, reflecting the fact that the second block in a sequence, b_2, should be typed in the type environment Γ_2 produced by typing the first block, b_1.

$$\text{Block-Sequence}\quad \frac{\Gamma_1; \Delta\ [pc] \vdash b_1 : \mathsf{block}_{\ell_1} \dashv \Gamma_2 \qquad \Gamma_2; \Delta\ [pc] \vdash b_2 : \mathsf{block}_{\ell_2} \dashv \Gamma_3}{\Gamma_1; \Delta\ [pc] \vdash b_1; b_2 : \mathsf{block}_{\ell_1 \sqcup \ell_2} \dashv \Gamma_3}$$

4.3.2 Typing Rules

The typing rule for the line break block, br, is similar to the typing rules for values in that, in the absence of any sub-expressions that might influence its security level, we default the block's security level to that of the program counter pc.

$$\text{Block-Br}\quad \frac{}{\Gamma; \Delta\ [pc] \vdash \mathsf{br} : \mathsf{block}_{pc} \dashv \Gamma}$$

Alert messages, labels and images are typed conservatively, using the security level of their sub-expression as their own. All three are simple output blocks, meaning the typing environment Γ is not altered.

$$\text{Block-Alert}\quad \frac{\Gamma; \Delta\ [pc] \vdash e : \mathsf{string}_\ell}{\Gamma; \Delta\ [pc] \vdash \mathsf{alert}\ e : \mathsf{block}_\ell \dashv \Gamma}$$

$$\text{Block-Label}\quad \frac{\Gamma; \Delta\ [pc] \vdash e : t_\ell}{\Gamma; \Delta\ [pc] \vdash \mathsf{label}\ e : \mathsf{block}_\ell \dashv \Gamma}$$

$$\text{Block-Image}\quad \frac{\Gamma; \Delta\ [pc] \vdash e : \mathsf{string}_\ell}{\Gamma; \Delta\ [pc] \vdash \mathsf{image}\ e : \mathsf{block}_\ell \dashv \Gamma}$$

The same applies to LiveWeb's two types of buttons, with the only difference being that they have more than one sub-expression and thus need to use the least upper bound ($\sqcup$) of all their security levels.

$$\text{Block-Button-Simple}\quad \frac{\Gamma; \Delta\ [pc] \vdash e_1 : \mathsf{string}_{\ell_1} \qquad \Gamma; \Delta\ [pc] \vdash e_2 : \mathsf{block}_{\ell_2}}{\Gamma; \Delta\ [pc] \vdash \mathsf{button}\ e_1\ \mathsf{to}\ e_2 : \mathsf{block}_{\ell_1 \sqcup \ell_2} \dashv \Gamma}$$

$$\text{Block-Button-AJAX}\quad \frac{\begin{array}{c}\Gamma; \Delta\ [pc] \vdash e_1 : \mathsf{string}_{\ell_1} \\ \Gamma; \Delta\ [pc] \vdash e_2 : \mathsf{string}_{\ell_2} \qquad \Gamma; \Delta\ [pc] \vdash e_3 : \mathsf{block}_{\ell_3}\end{array}}{\Gamma; \Delta\ [pc] \vdash \mathsf{button}\ e_1\ \mathsf{update}\ e_2\ \mathsf{with}\ e_3 : \mathsf{block}_{\ell_1 \sqcup \ell_2 \sqcup \ell_3} \dashv \Gamma}$$

Div blocks are simple containers of other blocks b, which feature a CSS class identifier cl and an expression e that determines their HTML id. The class identifier is known statically and does not constitute a source of information, as the expression e and the block b do. Since both are independent of each other, the typing rule for the div construct types them separately and uses their security levels' least upper bound as its security level. The typing rule for link blocks follows the same reasoning.

$$\text{Block-Div}\quad \frac{\Gamma_1;\Delta\ [pc] \vdash e : \mathsf{string}_{\ell_1} \qquad \Gamma_1;\Delta\ [pc] \vdash b : \mathsf{block}_{\ell_2} \dashv \Gamma_2}{\Gamma_1;\Delta\ [pc] \vdash \mathsf{div}\ cl\ \mathsf{as}\ e\ \{\ b\ \} : \mathsf{block}_{\ell_1 \sqcup \ell_2} \dashv \Gamma_2}$$

$$\text{Block-Link}\quad \frac{\Gamma_1;\Delta\ [pc] \vdash b : \mathsf{block}_{\ell_1} \dashv \Gamma_2 \qquad \Gamma_1;\Delta\ [pc] \vdash e : \mathsf{block}_{\ell_2}}{\Gamma_1;\Delta\ [pc] \vdash \mathsf{link}\ \{\ b\ \}\ \mathsf{to}\ e : \mathsf{block}_{\ell_1 \sqcup \ell_2} \dashv \Gamma_2}$$

Text fields and text areas have exactly the same typing rules. The security level ℓ of the content e is the highest information they output, reason by which ℓ is also their security level. Since both are primarily input constructs, the resulting type environment is augmented with a string x whose security level combines the block's security level and the primitive `Input()` security level discussed earlier.

$$\text{Block-TextField}\quad \frac{\Gamma;\Delta\ [pc] \vdash e : \mathsf{string}_{\ell}}{\Gamma;\Delta\ [pc] \vdash \mathsf{textfield}\ x\ \mathsf{with}\ e : \mathsf{block}_{\ell} \dashv \Gamma, x : \mathsf{string}_{\ell \sqcup \mathrm{Input}()}}$$

$$\text{Block-TextArea}\quad \frac{\Gamma;\Delta\ [pc] \vdash e : \mathsf{string}_{\ell}}{\Gamma;\Delta\ [pc] \vdash \mathsf{textarea}\ x\ \mathsf{with}\ e : \mathsf{block}_{\ell} \dashv \Gamma, x : \mathsf{string}_{\ell \sqcup \mathrm{Input}()}}$$

The option block corresponds to the HTML select element whose options are given by expression e_3, a collection of records. The field names f_1 and f_2 correspond to the fields of the record that contain the information on the *id* and the *display value* of each option, respectively. Expression e_4 corresponds to the default id of the select and, for that reason, its type t_{ℓ_1} must be the same as the type of field f_1.

$$\text{Block-Option}\quad \frac{\begin{array}{c}\Gamma;\Delta\ [pc] \vdash e_3 : \tau^*_{\ell_3} \qquad \Gamma;\Delta \vdash \tau <: [f_1 : t_{\ell_1}, f_2 : \mathsf{string}_{\ell_2}]_\bot \\ \Gamma;\Delta\ [pc] \vdash e_4 : t_{\ell_1} \qquad \ell_b = \ell_1 \sqcup \ell_2 \sqcup \ell_3 \qquad \ell_x = \ell_1 \sqcup \ell_3 \sqcup \mathrm{Input}()\end{array}}{\Gamma;\Delta\ [pc] \vdash \mathsf{option}\ x\ \mathsf{fill}\ f_1 \Rightarrow f_2\ \mathsf{with}\ e_3\ \mathsf{default}\ e_4 : \mathsf{block}_{\ell_b} \dashv \Gamma, x : t_{\ell_x}}$$

The block's security level, ℓ_b, is obtained by performing the least upper bound of its sub-expressions' security levels, while the resulting type environment contains a new identifier x – corresponding to the selected id – whose security level combines the security levels of the collection ℓ_3 and field ℓ_1, from which the value comes, with the primitive `Input()` level to reflect the choice of the user.

The typing rule for the iterator block is very similar to the typing rule for $\lambda_{DB-Flow}$'s map expression: when we type the body block b, the program counter is raised with the collection's security level ℓ_1.

$$\text{Block-Iterator}\quad \frac{\Gamma_1;\Delta\ [pc] \vdash e : \tau^*_{\ell_1} \qquad \Gamma_1, x : \tau;\Delta\ [pc \sqcup \ell_1] \vdash b : \mathsf{block}_{\ell_2} \dashv \Gamma_2}{\Gamma_1;\Delta\ [pc] \vdash \mathsf{iterator}\ (x\ \mathsf{in}\ e)\ \{\ b\ \} : \mathsf{block}_{\ell_1 \sqcup [\![\ell_2]\!]^x} \dashv \Gamma_1}$$

In the end, the iterator's security level combines both the security level of the collection ℓ_1 and the security level of its body ℓ_2, although there is the need to close the free occurrences of the identifier x using $[\![\cdot]\!]^x$, and the typing environment is left unchanged because all the new names are local to b.

4.4 Approaching Non-interference

Although out of the scope of this thesis, in order to prove well-typed programs in our language do not leak confidential information, we would need to define a *non-interference* property and prove it holds for well-typed programs. Essentially, such a property says that confidential information does not *interfere* with public information, meaning that we cannot infer any confidential information just by looking at public information.

The base approach used in the calculi we studied [Zda02, CKP05], which is also the usual approach according to [SM03], is centred in the definition of an equivalence relation $\approx_\zeta$ between computation states, based on the language's labelled operational semantics. Intuitively, two computation states are equivalent according to $\approx_\zeta$ if they are indistinguishable to an observer with security level ζ or lower. Given the equivalence relation, we are able to formalize the non-interference property.

If we abstract computation states as a tuple (S, e), where S represents state information (e.g. the memory) and e is the program being executed, the non-interference property holds if, and only if:

- for all computation states (S, e) such that $\Gamma, x : \tau; \Delta\ [pc] \vdash e : t_\ell$, and;
- for all values v_1 and v_2 of type τ such that $v_1 \approx_\ell v_2$

we can prove that

$$(S, e\{v_1/x\}) \approx_\ell (S, e\{v_2/x\})$$

Finally, both approaches we studied prove that if two computation states are equivalent with respect to $\approx_\zeta$, then both evaluate in a finite number of steps to computation states that are also equivalent, meaning the equivalence relation is preserved through evaluation. If the semantics are deterministic and closed under evaluation contexts, this implies non-interference holds.

Alternatively, we could attempt to encode our language in another for which non-interference was already proven, an approach taken in [Zda02] to prove non-interference for λ_{SEC}^{REF}, and in [ABHR99] by encoding several calculi into DCC.

5

Example

In order to validate our approach to information-flow analysis, detailed in Chapter 4, we developed a prototype implementation as an extension of the LiveWeb framework and used the prototype to validate a series of small examples. In this chapter, we discuss the modelling of a medium-sized web application in our language, in order to illustrate its expressivity and the major insights and problems faced during the development process. We begin with a broad description of our scenario in Section 5.1, enunciating potential confidentiality concerns, and proceed to discuss the concrete application and its implementation difficulties in Section 5.2.

5.1 Scenario

The most common way of disseminating and reporting progresses in scientific research is the publication of *papers* in specialized scientific conferences and journals pertaining to a specific field of study. The conferences' organizing committees assign each submitted paper to a number of other researchers knowledgeable in the same sub-field, so that the paper's validity is assessed and a decision can be made to accept or reject the submission, based on their reviews.

Nowadays, the management of submissions and peer reviews is often performed by specialized software systems, providing a common medium that coordinates the interaction between committee members, authors and reviewers during the whole process. Each of these groups of users interacts with the system in a distinct way: from the authors' point of view, the interaction is based in submitting papers, reading the reviews they receive and consulting the papers' acceptance status; from the reviewers' point of view, the interaction is based in voting the papers they would like to review, consulting the list of assigned papers and submitting reviews; from the committee members' point of view, the interaction is based in managing the start and end dates for the submission and reviewing processes, assigning reviewers to papers and deciding whether a paper should be accepted or not, based on the reviewers' opinions.

In such systems, there are a number of security policies and guarantees that the committee may wish to ensure:

- The reviewers of a specific paper must be anonymous;
- The authors of a specific paper should remain anonymous until the paper is accepted;
- The title and short summary (usually known as *abstract*) of a specific paper should only be readable by its authors, potential reviewers and committee members, until the paper is accepted;
- The contents of a paper should only be readable by the authors and assigned reviewers, until the paper is accepted;
- If a paper is rejected, the information should only be available to its authors, its reviewers and committee members;
- A review should only be readable by its author, the authors of the paper and committee members.

The following section presents a possible modelling of such a system in our prototype, taking advantage of information flow analysis to ensure that these confidentiality concerns are addressed.

5.2 Application

Consider a web application whose registered users are modelled by the `User` entity in Listing 5.1, featuring a public `id`, a public `username`, a private `password` that should only be readable by the system, and a public `role` field containing a pre-defined number used to distinguish between the three types of users in the system: authors of papers (0), reviewers (1) and committee members (2). At the policy level, we use the predicate `User(id)` to label information that a specific user can read, and use the predicates `AuthorRole(id)`, `ReviewerRole(id)` and `CommitteeRole(id)` to refer to information that should only be readable by a specific user with the corresponding role. Accordingly, the entity has three local policies encoding that definition, for each specific row. For instance, information readable by an author with identifier `id`, should be readable (i.e. is implied) by an user with the same id and role number 0.

Listing 5.1: The `User` entity

```
def entity User {
  id : Id at true(),
  username : String at true(),
  password : String at System(),
  role : Int at true()
}
flow AuthorRole(id) to (User(id) and (role = 0))
flow ReviewerRole(id) to (User(id) and (role = 1))
flow CommitteeRole(id) to (User(id) and (role = 2))
```

In turn, the predicates `AuthorOf(id)` and `ReviewerOf(id)` represent the information that is readable, respectively, by an author or an assigned reviewer of a specific paper with id `id`. Papers, modelled by entity `Paper` in Listing 5.2, feature an identifier `id`, readable by any registered user; a `title` and an `abstract`, readable by the authors of the paper, reviewers (in general) and committee members; a `content`, only readable by the authors and assigned reviewers of the paper; and a `status` field indicating whether the paper is pending evaluation (0), rejected (1) or accepted (2), which is only readable by the authors, assigned reviewers and committee members, according to the confidentiality concerns laid out in our scenario.

Listing 5.2: The `Paper` entity

```
def entity Paper {
  id : Id at (exists uid: Int. User(uid)),
  title : String at (
    AuthorOf(id)
    or (exists uid: Int. ReviewerRole(uid))
    or (exists uid: Int. CommitteeRole(uid))
  ),
  abstract : String at (
    AuthorOf(id)
    or (exists uid: Int. ReviewerRole(uid))
    or (exists uid: Int. CommitteeRole(uid))
  ),
  content : String at (AuthorOf(id) or ReviewerOf(id)),
  status : Int at (
    AuthorOf(id)
    or ReviewerOf(id)
    or (exists uid: Int. CommitteeRole(uid))
  )
}
```

In order to illustrate the interplay between the two entities defined above and their policies, suppose we want to write a query that allows the committee member with id uid to fetch the titles of all submitted papers. Disregarding security policies, a first attempt may be similar to the getTitles action in Listing 5.3. The base security level will be that of the title field, allowing us to prove the information has the precise security level CommitteeRole(uid). However, if we specialize our functions in each kind of role, we end up replicating the whole system many times in order to propagate the appropriate security levels throughout all the actions and screens.

Listing 5.3: The getTitles action

```
def true() action getTitles(uid: Int at true()):
  List[String at CommitteeRole(uid)] at true()
{
  from (p in Paper)
  where true
  select p.title
}
```

Consequently, as a means of *modularization*, we use the security level User(id) in the interfaces of actions and screens and prove *locally* that our information can be read by the current user. In order to prove that information with level CommitteeRole(uid) can be read by the current user, we need to prove the current user is in fact a committee member, achieved by performing a join with the entity User, as shown in the action of Listing 5.4 which receives the current user id as a parameter and executes with program counter User(uid).

Listing 5.4: The getTitlesForCommittee action

```
def User(uid) action getTitlesForCommittee (uid: Int at User(uid)):
  List[String at User(uid)] at User(uid)
{
  from (p in Paper, u in User)
  where u.id == uid and u.role == 2
  select p.title
}
```

The authorship relation between users and papers is represented by entity Author in Listing 5.5, with the expected foreign key fields paper_id and user_id. Both fields may only be read by authors of the paper, so that information on authorship remains confidential as required. A local policy states that each row of the Author's table proves the corresponding user (user_id) is able to read information available to the authors of the corresponding paper (paper_id). For simplicity, we did not consider the "until the paper is accepted" condition mentioned in the scenario, but the state change could be modelled by using an Accepted(paper_id) predicate related to the paper's status, as we did with user roles.

Listing 5.5: The `Author` entity

```
def entity Author {
  id : Id at (exists uid: Int. User(uid)),
  paper_id : Int at AuthorOf(paper_id),
  user_id : Int at AuthorOf(paper_id)
}
flow AuthorOf(paper_id) to User(user_id)
```

Getting back to our paper titles example, if an user with role author wants to fetch all the papers' titles, the type system would detect an information leak according to our policies, even if we performed a join with the `Users` table, because an author is only able to read the titles of his/her own papers. Considering this (intended) restriction, suppose we decide to define a general action that fetches the title of a paper with id `pid` if the current user satisfies any of the security requirements, as presented in Listing 5.6.

Listing 5.6: The `getTitle` action

```
def User(uid) action getTitle (
  uid: Int at User(uid),
  pid: Int at User(uid)
): String at User(uid) {
  getHead(
    from (p in Paper, u in User, a in Author)
    where p.id == pid and (
      (a.paper_id == pid and a.user_id == uid) // for AuthorOf(pid)
      or (u.id == uid and u.role == 1) // for ReviewerRole(uid)
      or (u.id == uid and u.role == 2) // for CommitteeRole(uid)
    )
    select p.id
  )
}
```

Although the where condition contemplates each of the required cases, the action is not well-typed because we would be reading fields from the `Author` table, which are authors-only, even if the user is a reviewer or a committee member, leading the type system to conclude there is a possible information leak. On the other hand, a `getTitle` action that worked only for reviewers and committee members would type check, because both roles can read all the fields involved.

Based on the title and abstract of submitted papers, reviewers can manifest their preferences regarding which papers they would like to review, to be considered in the assignment process. The `ReviewIntention` entity (Listing 5.7) expresses the intention of a user with id `user_id` to review the paper with id `paper_id`, with a boolean field `positive` indicating whether the user would or not like to review the specified paper, and a `priority` field specifying the order of the preference. While the specific reviewer that manifested the intention should be anonymous, other fields could be readable by a committee member, thus enabling the assignment of reviews without the need for the system to reveal the reviewer's identity.

Listing 5.7: The `ReviewIntention` entity

```
def entity ReviewIntention {
  id : Id at (exists uid: Int. User(uid)),
  user_id : Int at ReviewerRole(user_id),
  paper_id : Int at (
    ReviewerRole(user_id)
    or (exists uid: Int. CommitteeRole(uid))
  ),
  positive : Bool at (
    ReviewerRole(user_id)
    or (exists uid: Int. CommitteeRole(uid))
  ),
  priority : Int at (
    ReviewerRole(user_id)
    or (exists uid: Int. CommitteeRole(uid))
  )
}
```

The `ReviewAssignment` entity in Listing 5.8 models the assignment of reviews and contains a subset of the fields of the `ReviewIntention` entity, maintaining the same security levels: the `id` field can be read by any registered user, as it only reveals information about the number of rows in the entity; the `user_id` field should only be readable by the reviewer with the same id to preserve the anonymity; and the `paper_id` only needs to be readable to the reviewer, so that he/she can know which papers to review, and for the committee members, so that they know how many reviews to expect until the reviewing process is finished. Similarly to the `Author` entity, a local policy states that, for each row in the entity, the user that is assigned to review a paper can read any information a reviewer of that paper can.

Listing 5.8: The `ReviewAssignment` entity

```
def entity ReviewAssignment {
  id : Id at (exists uid: Int. User(uid)),
  user_id : Int at ReviewerRole(user_id),
  paper_id : Int at (
    ReviewerRole(user_id)
    or (exists uid: Int. CommitteeRole(uid))
  )
}
flow ReviewerOf(paper_id) to User(user_id)
```

After the submission process has ended and the reviewers are assigned papers, they can submit reviews to those papers. The `Review` entity (Listing 5.9) is defined by the id of the reviewer (`user_id`), once again only readable by the reviewer himself/herself, the `paper_id` of the paper being reviewed and the review's `content`. The last two fields can be read by the reviewer, an author of the paper or a committee member, allowing the authors to receive feedback and the committee members to base their decision regarding the validity of the paper, in line with the policies discussed earlier. Similarly to the `getTitle` action, we are not able to fetch all the reviews for a paper using a single query, if we want to be able to prove the result can be read by the current user (i.e. `User(uid)`): proving the current user is an author requires reading fields from the `Author`'s table and proving the current user is the author of the review requires reading the `user_id` field of the `Review` entity itself, both unreadable by other roles. Consequently, we are not even able to combine the queries for reviewers and committee members as with the `getTitle` example, but would instead need to use three separate queries, one for each role.

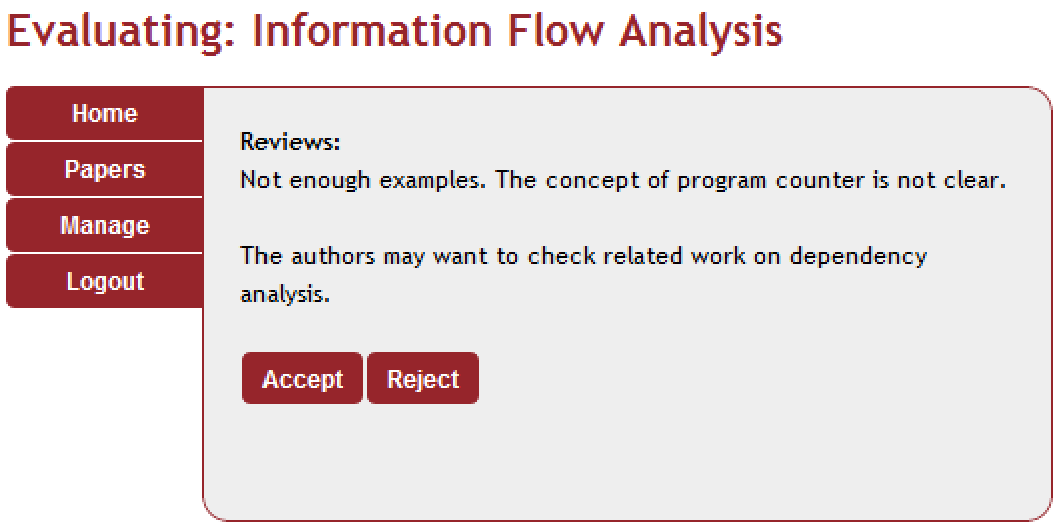
Listing 5.9: The `Review` entity

```
def entity Review {
  id : Id at (exists uid: Int. User(uid)),
  user_id : Int at ReviewerRole(user_id),
  paper_id : Int at (
    ReviewerRole(user_id)
    or AuthorOf(paper_id)
    or (exists uid: Int. CommitteeRole(uid))
  ),
  content : String at (
    ReviewerRole(user_id)
    or AuthorOf(paper_id)
    or (exists uid: Int. CommitteeRole(uid))
  )
}
```

As the reviewing process progresses and all the assigned reviewers for a given paper submit their reviews, the committee members are ready to evaluate if the paper should be accepted or not. A possible screen definition for such interaction is shown in Listing 5.10, parametrized in the current user and the paper being judged. The first occurrence of the `User(uid)` predicate in the screen's signature is the program counter, similarly to the actions on the previous examples, and the last occurrence specifies the screen's *output security level*, an upper bound on the information that may be included in the screen's blocks and, thus, sent to the client. The screen is composed by a div displaying the paper's title, the navigation menu available to committee members and another div holding the reviews of the paper, generated by a call to the `reviews` screen, and buttons targeting the `submitEvaluation` action, with the last parameter defining whether the evaluation is positive or negative. A possible rendering of the screen is displayed in Figure 5.1.

Listing 5.10: The `evaluatePaper` screen

```
def User(uid) screen evaluatePaper (
  uid: Int at User(uid),
  pid: Int at User(uid)
) at User(uid) {
  div title {
        label ("Evaluating: " + getTitle(uid, pid))
  };
  label committeeMenu(uid);
  div content {
    label reviews(uid, pid);
    button "Accept" to submitEvaluation(uid, pid, true);
    button "Reject" to submitEvaluation(uid, pid, false)
  }
}
```


Evaluating: Information Flow Analysis

Reviews:
Not enough examples. The concept of program counter is not clear.

The authors may want to check related work on dependency analysis.

Figure 5.1: Possible rendering of the paper evaluation page

Notice that every screen and action call features the current user id as a parameter. Most of the time the id is required by the task at hand but, regardless, it is always necessary to be able to propagate the `User(uid)` security levels in the definitions' interface. If the `uid` parameter is not included in the parameter list, the program counter would need to be raised to forall `uid: Int. User(uid)` in order to avoid having `uid` as a free name, which consequently would force the security level of the definition's result to be raised as well, preventing the current user from reading the resulting value.

The `submitEvaluation` action, mentioned earlier and included in Listing 5.11, exemplifies another concern raised by the approach of using the `User(uid)` security level to achieve modularity. Since the action has a program counter of `User(uid)` and parameters with security level `User(uid)`, anticipating calls in contexts with that program counter, we need to perform declassification using an assume expression in order to update the status of the paper, otherwise the query is rejected by the type system as potentially leaking information.

Listing 5.11: The `submitEvaluation` action

```
def User(uid) action submitEvaluation (
  uid: Int at User(uid),
  pid: Int at User(uid),
  accept: Bool at User(uid)
): WebPage at User(uid) {
  if (isCommitteeMember(uid)) then
    assume User(uid) in (
      update p in Paper with {
        status = if (accept) then 2 else 1
      } where (p.id == pid)
    )
  else
    false;
  papers(uid)
}
```

The main reason for the ubiquitous `User(uid)` program counter in this application is the use of a common menu in every screen, made of simple buttons. Since each time we choose an item of the menu we are taken to a screen which renders a new menu, the type system detects a dependency between the menu we draw and the source page, in this case irrelevant. Thus, once we have a web page featuring the menu whose program counter is `User(uid)`, we are forced to have the menu with program counter `User(uid)`, which in turn forces all the screens linked by the menu to at least have that same program counter. A clever way of dealing with the situation is to eliminate the apparent dependency by using *ajax buttons* in the menu, consequently redesigning the target screens in such a way that only their specific information is included, allowing the menu and other screens to lower their program counter to true. Even so, complex applications will always require the use of declassification in some places, although adhering to the proposed design principle removes the necessity for some assumes.

6 Implementation

One of the contributions of this thesis is a prototype implementation of our core language on top of the LiveWeb framework (Subsection 2.2.1), in order to illustrate the approach and understand its limits. The implementation was split into two core modules: a direct extension of the LiveWeb framework with our type system, and a interface module with a *satisfiability modulo theories* (SMT) solver, used by the type checking algorithm to prove the validity of formulas.

In this chapter, we present an overview of the project's architecture and try to give a notion of the implementation effort when compared to what was already implemented by the LiveWeb framework. In Section 6.1, we begin by describing the starting state of the framework's implementation and the extensions that were made since its release. Based on that knowledge, in Section 6.2 we discuss how the LiveWeb framework was extended in order to implement our type system, emphasising the main challenges of the typing algorithm, and in Section 6.3 we discuss how the SMT solver was abstracted, taking into account its input language and the encoding required for some of our data types. Finally, in Section 6.4 we illustrate the type checking process with a small example, focusing on the interaction between the solver and the type checker.

6.1 LiveWeb

Framework description

LiveWeb is a domain-specific language for web applications whose implementation constitutes the base framework on top of which we developed our type system's prototype.

Composed by a runtime system and a web-based development environment, both supported by SQLite databases, its code comprises near 1400 lines of JavaScript and around 36000 lines of Java, 16000 of which are generated by a compiler construction tool, BNF Converter, used to generate the lexer, parser and base abstract syntax representation from a *labelled BNF grammar* [FR05].

The remaining 20000 lines are scattered across 177 Java files consisting of: an *abstract syntax tree* (AST) representation of the language; a conversion visitor from the generated base abstract syntax representation to the AST representation; an HTTP server; typing and execution environments; a compile-time representation for types; a run-time representation for values; a database manager used by the runtime environment to query the entities of an application; a database manager used by both sub-systems to fetch, save, publish and delete definitions; and a set of 18 AST visitors with various purposes: execution, type checking, pretty printing, collecting free names, converting boolean expressions to logical formulas and database management.

Prior extensions

Since LiveWeb's original release [Dom10], several improvements and minor extensions were made, which can be grouped around three main goals: increase the coverage of database and user interface primitives; leverage the multi-layer integration provided by the language in the verification of security-related properties; and empower the development of increasingly customizable applications.

The first group includes the addition of a delete query primitive, important in typical applications but originally absent, and frequently used web page elements such as text areas, drop-down selectors and JavaScript alert messages. Support for Asynchronous JavaScript and XML (AJAX) requests was implemented through the addition of HTML *ids* to the `div` primitive and the creation of specialized buttons that update a `div`, referenced by its id, using an AJAX request. Finally, observing the recurrent need to create a specialized screen to use as target for an AJAX button, the language was further extended with *anonymous screens* to enable the inline definition of screens inside expressions.

Regarding the verification of security-related properties, LiveWeb was extended with the access-control type system presented in [CPS$^+$11], taking advantage of the similarity between the expressions of both languages. Besides the changes made to the type checker and typing environment, the implementation required an extension of the type language with refinement types, and the addition of read and write policies to the definition of entities.

As in the original type system, LiveWeb's access-control policies are defined by means of logical formulas. Each time the type checker needs to verify the validity of a logical formula, an external *satisfiability modulo theories* (SMT) solver, CVC3, is run with a specific benchmark containing the proposition we want to prove and the information we can assume in that context. The benchmark in Listing 6.1 is an example where the solver is asked to prove if a user with id `uid` is authenticated, knowing that an user with id `user_id` is authenticated and both ids are equal. A definition of SMT solvers is given in Section 6.3, along with further discussion on their use and input languages in the context of our extension.

Listing 6.1: LiveWeb – SMT solver benchmark example

```
(benchmark example
  :logic AUFLIRA
  :extrasorts (Term)
  :extrapreds ((Auth Term))
  :extrafuns ((uid Term) (user_id Term))
  :assumption (Auth user_id)
  :assumption (= uid user_id)
  :formula (not (Auth uid))
)
```

Lastly, in order to provide support for the development of dynamically customizable applications, LiveWeb was extended with functional values, allowing actions and screens – hereafter referred as definitions – to be stored in the database and be passed as parameters to other definitions. Since a definition's body may contain calls to other definitions and queries to specific entities, the definition's interface was extended to include an import declaration of the free names in its body and their expected types, allowing the type system to enforce that a definition may only be inserted in the database if its body's free names were all imported.

On top of the support for functional values, the language was enriched with a construct to abstract the import of new code in runtime, similar to a type casing expression. Given a string containing the code, which is parsed and type checked in runtime, the inferred type is matched against the types declared in one of several *branches*, using subtyping and checking its imports, and executes the code in the first branch that does. As with a typical case construct, the type system is able to provide compile-time guarantees adapted to the branch that is executed, that is, the type of the code that is imported.

6.2 LiveWeb Extension Core Module

Extension description

One of the two core modules of our prototype consists in a direct extension of the LiveWeb framework with our information-flow type system. The implementation was supported by the latest version of the framework, described in the previous section, with the exception of the access-control type system extension, which was discarded because it required us to tackle a larger problem lying outside the scope of this thesis, the integration of access-control and information-flow analyses.

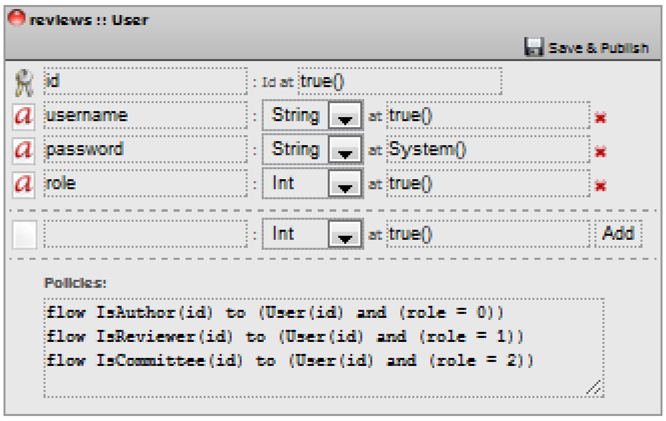
Concretely, we estimate about 4000 lines of Java code were discarded, starting with the original type checking visitor using refinement types. Furthermore, although our extension also uses an SMT solver (discussed in Section 6.3), its integration with the LiveWeb framework was too tightly coupled with the use of refinement types, both in the implementation of predicates and the typing environment, motivating our decision to rewrite them from scratch modularly.

The new core LiveWeb module by itself comprises near 42000 lines of Java code, 23500 of which are non-generated and scattered across 222 files, resulting in an absolute increase of 3500 non-generated lines versus the original implementation. On the other hand, the total number of lines written to develop our prototype can be estimated to be near the 10000 lines, considering the amount of discarded code and the size of the SMT solver module (2700 lines).

In practice, the extension effort consisted in: changes to the grammar and correspondent additions to the *abstract syntax tree* (AST), in order to accommodate the security annotations and related constructs (flow declarations, formulas and terms); AST and compile-time representations of dependent function types; an abstraction of a lattice by means of an interface parametric in the elements' type; a concrete lattice of formulas, using the SMT solver module to check if one element is lower or equal to another element; a typing environment which ensures its declarations and scope are properly mirrored by the solver; and a total of 12 new AST visitors with various purposes: type checking, renaming, term substitution, subtyping, performing conversions (namely to solver formulas and terms), injection of premises in formulas, closing free names and erasing security labels.

The implementation of the type checker and typing environment was greatly influenced by the option to keep the solver constantly running throughout the type checking process, in order to avoid the overhead of restarting the solver dozens of times. The solver state must be kept synchronized with the type environment's state, a task that is ensured by the latter, and the identifiers of the AST must be uniquely renamed prior to the start of type checking, because the solver language does not allow name overloading and its scoping instructions only affect assertions. Consequently, apart from the synchronization carried by the environment, the interaction with the solver is made through the addition (assume, if conditions, where clauses) or verification (assert) of assertions, and by querying the lattice on the relation between two security levels. This interaction is explored further in Section 6.4 through a small example.

As expected, despite all the server-side changes required to implement our type system, the development environment only required small changes to fully accommodate our extension: the addition of editable security levels to the entity editor (Figure 6.1) and the creation of a text-based editor for global policies, similar to the one used to edit actions and screens.



```
id        : Id at true()
username  : String at true()
password  : String at System()
role      : Int at true()
          : Int at true()
```

Policies:

```
flow IsAuthor(id) to (User(id) and (role = 0))
flow IsReviewer(id) to (User(id) and (role = 1))
flow IsCommittee(id) to (User(id) and (role = 2))
```

Figure 6.1: Entity editor with security levels

A concrete example

The type checking visitor class alone spans near 2000 lines of code to cover the typing rules of all the language's constructs, as could probably be expected given the size of the language and the focus of this thesis. In order to illustrate the mapping process between typing rules and visitor methods, we will present a concrete example of how the typing rules for the let-expression were implemented as a `visit` method in the type checker.

$$\text{Let-Value} \qquad \frac{\Gamma;\Delta\ [pc] \vdash v : \tau_1 \qquad \Gamma, x : \tau_1; \Delta\ [pc] \vdash e : \tau_2}{\Gamma;\Delta\ [pc] \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau_2\{v/x\}}$$

$$\text{Let-Expression} \qquad \frac{\Gamma;\Delta\ [pc] \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1; \Delta\ [pc] \vdash e_2 : \tau_2}{\Gamma;\Delta\ [pc] \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : [\![\tau_2]\!]^x}$$

The first rule is used to accurately type let-expressions whose first expression is a term, taking advantage of term substitution, and the second rule is used to cover the remaining cases, using universal quantification to close the free occurrences of the identifier in the resulting type. In the implementation, presented in Listing 6.2, there is only one type of AST node to represent a let-expression, so we opted for a general solution that covers both cases uniformly.

First, in line 4, we determine the type of the first expression – v or e_1 in the typing rules, labelled as *id expression* in the code – by using the `typecheckLabelled` method, which simply wraps a `visit` call on the argument node to guarantee the resulting type is a `LabelledType`. Next, in lines 5–6, we use a specialized visitor – wrapped in the `transformInTerm` method – to transform the expression in a term by replacing all non-term sub-expressions with a fresh variable, which is then bound to the type of the expression it replaces. For instance, if the expression is the record `{ a = [1,2,3] }`, the list sub-expression would be replaced by a fresh variable y, bound to a labelled list type, and the resulting term would be `{ a = y }`.

Listing 6.2: Visit method for let-expressions

```
1  public IType visit(ASTExpressionLet node, AASTFormula pc)
2    throws VisitorException
3  {
4    LabelledType idLT = typecheckLabelled(node.getIdExpression(), pc);
5    Entry<AASTTerm, Map<String, IType>> idTerm =
6      transformInTerm(node.getIdExpression(), pc);
7
8    env.beginScope();
9    env.assoc(node.getId(), idLT);
10   LabelledType inLT = typecheckLabelled(node.getInExpression(), pc);
11   env.endScope();
12
13   return substituteIdByTerm(inLT, node.getId(), idTerm.getKey())
14     .accept(new CloseFreeNamesVisitor(idTerm.getValue()), null);
15 }
```

Typing the second expression – e_2 in the typing rules, labelled as *in expression* in the code – proceeds as expected (lines 8–11), inside a new scope with the identifier bound to the first expression's type. Similarly to the first typing rule, the resulting type is obtained by taking the second expression's type and replacing all its occurrences of the identifier with the term obtained earlier, using the `substituteIdByTerm` method. Building on our earlier example, if the type of the second expression is `Int at P(x)`, after substitution we obtain `Int at P({ a = y })`.

Finally, since transforming the first expression in a term may have generated fresh identifiers, we use another visitor to universally quantify their occurrences in the resulting type. For instance, if we close the free occurrence of y in the type `Int at P({ a = y })`, we obtain the type `Int at forall y: List[Int]. P({ a = y })`.

6.3 SMT Solver Module

Given that our information-flow type system uses logic formulas as security levels, we need to be able to verify whether a given formula is valid or not. For that purpose, our prototype includes a core module that abstracts the use of an external *satisfiability modulo theories* solver, commonly known as SMT solver, during the type checking process. A SMT solver is an algorithm that decides the satisfiability of a logical formula $\mathcal{F}$ with respect to (modulo) a given set of axioms (a theory) $\mathcal{T}$, that is, it decides the satisfiability of the formula $\mathcal{T} \wedge \mathcal{F}$. Although the algorithm focuses on satisfiability, we can also check the validity of a logical formula by observing that, if $\mathcal{T} \wedge \neg\mathcal{F}$ is *not* a satisfiable proposition, then $\mathcal{T} \Rightarrow \mathcal{F}$ is a valid proposition.

As mentioned in Section 6.1, LiveWeb already made use of a SMT solver, CVC3, in its extension with the access-control type system proposed in [CPS+11]. To interact with the solver, LiveWeb uses the SMT-LIB format [RT06], a standard language used in SMT solver benchmarks and competitions, allowing the replacement of CVC3 by some other compliant solver with minimal effort. Even so, the implementation of the interaction with the solver was too tightly coupled with LiveWeb's typing environment to be easily applicable to our extension, motivating the decision to write it from scratch as a separate module. Additionally, the version of the SMT-LIB format used in LiveWeb is not amenable for interactive use, requiring to restart the solver for each new query, a problem that is overcome in the latest version [BST10].

The implementation comprises nearly 2700 lines of code, spanning 53 Java files, and consists of: an *abstract syntax tree* (AST) representation of the necessary subset of the SMT-LIB format; an unparser from the AST representation to the textual script representation expected by a SMT-LIB compliant solver; and a set of classes and interfaces that abstract the concrete solver being used and the encoding of certain data types, such as records and strings, that are not defined in any of SMT-LIB's built-in theories. More precisely, we implement an `ISolver` interface that abstracts the concrete solver being used, exposing methods to query, add assertions, declare new values and declare new predicates; and two interfaces, `IScriptBuilder` and `IScriptGenerator`, that define the process of building the script and the generation of AST terms, respectively, in the level of abstraction used by the type checker.

The remaining of this section (Subsection 6.3.1), details the encoding that was used to bridge the gap between the data types used in our source language and the ones built-in to the solver language. The interaction with the type checker is further explored in Section 6.4.

6.3.1 Encoding

A SMT-LIB v2 compliant solver allows interactive use by means of *commands* with a syntax based in S-expressions [McC60], used in the Lisp family of programming languages. Each session begins by specifying which *sub-logic* will be used, among the many that are built-in to the solver or user-defined. A sub-logic is essentially a predefined set of specialized *theories*, such as quantifier-free boolean logic, linear integer arithmetic or a theory of arrays, and defines the initial set of definitions in the solver's environment. Our encoding is supported by a built-in sub-logic called `AUFLIA` [Cok11], which allows reasoning over quantified logical propositions that may include both arbitrary uninterpreted functions and linear integer arithmetic.

Listing 6.3: Example of a solver interaction

```
> (set-logic AUFLIA)
> (declare-sort Unit 0)
> (declare-fun unit () Unit)
> (assert (forall ((x Unit)) (= x unit)))
> (declare-fun a () Unit)
```

```
 6  > (declare-fun b () Unit)
 7  > (assert (not (= a b)))
 8  > (check-sat)
 9  unsat
10  ...
```

After specifying the sub-logic, the interaction is mainly a sequence of sort (type) and function declarations (declare-sort, declare-fun), assertions (assert) and satisfiability queries (check-sat) which yield one of three results: satisfiable (sat), unsatisfiable (unsat) or unknown (unknown) when the solver is unable to ensure its answer is consistent. For instance, in Listing 6.3 we have a simple example where we check if two specific values of the `Unit` sort are equal, following the sort's relevant declarations and axioms. Note that we want to test validity, so the assertion we want to verify must be negated and the result interpreted symmetrically: a negative result (unsat) in the satisfiability test means that the equality is valid. Given the assertion that any value of the `Unit` sort is equal to the `unit` value (line 4), the solver is able to conclude that both values must be equal by applying transitivity.

AUFLIA has built-in sorts for logical propositions (`Bool`) and for integer terms (`Int`), but lacks support for the majority of terms used in our logic, as it has no notion of records, strings or boolean values. In the remainder of this section, we discuss the encodings used to support the data types and predicates of our logic, evidencing the expressibility restrictions imposed by the SMT-LIB standard as it is. A complete listing of the preamble included at the beginning of each type checking session can be found in Appendix B.

Boolean The sort of *boolean values*, here named `MyBool` to distinguish it from the built-in `Bool` sort of logical propositions, has the two base values `myTrue` and `myFalse`, known to be distinct, and the three base boolean operations of negation (`myNot`), conjunction (`myAnd`) and disjunction (`myAnd`), whose semantic is defined symbolically by means of assertions.

Listing 6.4: Preamble for the boolean type

```
(declare-sort MyBool 0)
(declare-fun myTrue () MyBool)
(declare-fun myFalse () MyBool)
(assert (distinct myTrue myFalse))

(declare-fun myNot (MyBool) MyBool)
(assert (= (myNot myTrue) myFalse))
(assert (= (myNot myFalse) myTrue))

(declare-fun myAnd (MyBool MyBool) MyBool)
(assert (= (myAnd myTrue myTrue) myTrue))
(assert (= (myAnd myFalse myTrue) myFalse))
```

```
(assert (= (myAnd myFalse myFalse) myFalse))
(assert (= (myAnd myTrue myFalse) myFalse))

(declare-fun myOr (MyBool MyBool) MyBool)
(assert (= (myOr myTrue myTrue) myTrue))
(assert (= (myOr myFalse myTrue) myTrue))
(assert (= (myOr myFalse myFalse) myFalse))
(assert (= (myOr myTrue myFalse) myTrue))
```

Strings A possible encoding for strings is as lists of characters. Since characters are not built-in, the simplest way to encode strings, albeit not the most readable, is as lists of integer character codes. Thus, the `String` sort has one base value, the empty string (`nil`), and the usual binary constructor (`cons`) known not to produce empty strings. For instance, using the ASCII code of each character the string literal `"SMT"` would be encoded as `(cons 83 (cons 77 (cons 84 nil)))`.

Listing 6.5: Preamble for the string type (1)

```
(declare-sort String 0)
(declare-fun nil () String)
(declare-fun cons (Int String) String)
(assert (forall ((ch Int) (s String)) (distinct nil (cons ch s))))
```

We get equality for free, because structural equality is already defined for uninterpreted functions, and define the concatenation function between two strings (`concat`) symbolically by means of assertions.

Listing 6.6: Preamble for the string type (2)

```
(declare-fun concat (String String) String)
(assert (forall ((s String)) (= (concat nil s) s)))
(assert (forall ((s String)) (= (concat s nil) s)))
(assert (forall ((x Int) (xs String) (ys String))
  (= (concat (cons x xs) ys) (cons x (concat xs ys)))
))
```

Predicates In the SMT-LIB format, predicates are simply functions with result sort `Bool` that must be declared before they are used, like any other function. Unlike our language, however, the SMT-LIB standard does not support name overloading for predicates, requiring us to use different names for predicates with different argument types. To work around this mismatch without undermining the readability of the script – that is, adhere to a naming convention that includes the argument types – we define a generic `Term` sort and (purely symbolic) conversion functions to simulate untyped behaviour.

Listing 6.7: A generic `Term` sort

```
(declare-sort Term 0)

// For every declared sort named <SORT> add:
(declare-fun <SORT>ToTerm (<SORT>) Term)
(declare-fun termTo<SORT> (Term) <SORT>)
(assert (forall ((t Term) (x <SORT>)) (and
  (=> (= t (<SORT>ToTerm x)) (= (termTo<SORT> t) x))
  (=> (= (termTo<SORT> t) x) (= t (<SORT>ToTerm x)))
)))
```

For the same reason, predicates with the same name but different arities must also be differentiated, but we deemed it as acceptable to simply append the predicate's name with its arity, usually a single digit. With the `Term` sort in place and naming convention defined, declaring a predicate P with arity n is the same as declaring a function named say, $P\#n$, with n parameters of sort `Term` and a result of sort `Bool`. Additionally, in order to use a predicate in a logical proposition, the argument values must be wrapped in the appropriate `<SORT>ToTerm` function, as we can see in Listing 6.8: the integer arguments of the `Author` predicate must be wrapped using the function `intToTerm`. Given that structural equality between uninterpreted function terms is already built-in, the solver has no problem proving the validity of the implication by taking into account the equality in line 3.

Listing 6.8: Declaring and using the `Author` predicate

```
1 > (declare-fun Author#1 (Term) Bool)
2 > (declare-fun id () Int)
3 > (assert (= id 3))
4 > (assert (not
5   (=> (Author#1 (intToTerm id)) (Author#1 (intToTerm 3)))
6 ))
7 > (check-sat)
8 unsat
```

Records When encoding record types, we are restricted not only by the lack of name overloading as with predicates, but also because we are unable to quantify over sorts, consequently being unable to abstract over field types when defining selectors or equality between records. Given that an encoding of typed records would require a different sort for each record type and correspondent accessor functions, our encoding admits that the upper layer (i.e. the type checker) is typing records correctly. Every record has the same `Record` sort and for every distinct *field name* we generate a function with a `Record` parameter and a `Term` result. For instance, to declare the record $[\,\text{user\_id} = 1, \text{paper\_id} = 3\,]$ with the name `author` in Listing 6.9, we need to ensure that both field selectors were declared and that the equality assertions in lines 5–6 properly wrap the integer terms.

Listing 6.9: Declaring a record (1)

```
1 (declare-sort Record 0)
2 (declare-fun RF-user_id (Record) Term)
3 (declare-fun RF-paper_id (Record) Term)
4 (declare-fun author () Record)
5 (assert (= (RF-user_id author) (intToTerm 1)))
6 (assert (= (RF-paper_id author) (intToTerm 3)))
```

There is also the need to tackle the problem of record equality, taking into account the previously stated restrictions. For every record declaration, we generate an equality assertion based on the record's type, taking into account the value of the record's fields and requiring the other record to have the same exact number of fields. In order to have the information about the number of fields, we add to our preamble a `fieldsNumber` function and assert its value for each specific record upon declaration. Listing 6.10 illustrates the changes that would be needed to complete the previous example.

Listing 6.10: Declaring a record (2)

```
(declare-sort Record 0)
(declare-fun fieldsNumber (Record) Int)
...
(declare-fun author () Record)
(assert (= (fieldsNumber author) 2))
(assert (forall ((r Record)) (and
  (=> (= r author) (and
    (= (fieldsNumber r) (fieldsNumber author))
    (= (R-user_id r) (R-user_id author))
    (= (R-paper_id r) (R-paper_id author))
  ))
  (=> (and
    (= (fieldsNumber r) (fieldsNumber author))
    (= (R-user_id r) (R-user_id author))
    (= (R-paper_id r) (R-paper_id author))
  ) (= r author))
)))
...
```

Other types Although our logic has no term representation for functions, collections and web pages, we can declare variables of those types and compare them for name-based equality. Given that no extra information is required, the encoding just needs to include a new sort for each of them (`Function`, `Collection` and `Block`, respectively) and the corresponding conversion functions from and to the `Term` sort.

6.4 Type Checking Process: an example

In this section, we illustrate the interaction between the typing algorithm and the solver by means of a small example. Consider a simplified fragment of the paper reviewing system presented earlier (Chapter 5). A `Paper` is described by a public `id` and a confidential `content` which may only be read by an author or a reviewer of the paper.

```
def entity Paper {
  id : Id at true(),
  content : String at (Author(id) or Reviewer(id))
}
```

The `Author` entity describes the authorship relation by pairing users and papers using their ids, with a *local policy* stating that information reserved to authors of a paper may also be read by an user that is an author, provided there is a row in the table proving the authorship.

```
def entity Author {
  id : Id at true(),
  paper_id : Int at Author(paper_id),
  user_id : Int at Author(paper_id)
}
flow Author(paper_id) to User(user_id)
```

An auxiliary action `getPapersBy` fetches the ids of the papers authored by the given user with id `uid`, performing a select query in the `Author`'s table with the appropriate condition, and requires the resulting information to be readable by the user by labelling the return type with security level `User(uid)`.

```
def User(uid) action getPapersBy (uid: Int at User(uid)):
  List[Int at User(uid)] at User(uid)
{
  from (a in Author) where a.user_id == uid select a.paper_id
}
```

Finally, we have a `deletePapersBy` action that attempts to delete all the papers authored by the given user, obtaining the ids through a call to the `getPapersBy` action described above and performing a delete query in the `Paper`'s table with the expected condition for each of them.

```
def true() action deletePapersBy (uid: Int at true()): {} at true() {
  (foreach pid in getPapersBy(uid) do
    delete p from Paper where p.id == pid
  );
  {}
}
```

Recall that, since the paper ids depend on the author's id, the information is no longer public. Consequently, the delete query incurs in an information leak which is detected by the type system, caused independently by the context it is performed in (`User(uid)`), potentially revealing information on the *number of papers* authored by the user, and by the information provided in its where clause (also of level `User(uid)`), potentially revealing information on *which papers* were authored by the user. We will now explore the type checking process of this action from the point of view of the solver.

When we begin to type check a module, the solver is started with the full script for the preamble (included in Appendix B). Next, the type checker needs to process all the entities and signatures of actions/screens, in order to pre-declare their type. When processing the `Paper` entity to determine its type, the type checker begins a new scope and declares its two fields in order for them to be bound when checking if the local policies are well-typed. Although unnecessary, the solver mirrors these actions uniformly, using the `(push 1)` and `(pop 1)` commands to begin and end the *assertion scope*[1].

```
1  > ...
2  > (push 1)
3  > (declare-fun id () Int)
4  > (declare-fun content () String)
5  > (pop 1)
```

Afterwards, when the entity's record type is declared by the type checker, the solver mirrors it and declares a `Paper` constant with sort `Record`, resulting from the conversion of its original type to the solver encoding. During the conversion, both field accessors are declared in anticipation for their use. The remaining definitions undergo a similar process, producing the following script, with the unnecessary commands omitted:

```
6  > (declare-fun R-id (Record) Term)
7  > (declare-fun R-content (Record) Term)
8  > (declare-fun Paper () Record)
9  > (declare-fun R-paper_id (Record) Term)
10 > (declare-fun R-user_id (Record) Term)
11 > (declare-fun Author () Record)
12 > (declare-fun getPapersBy () Function)
13 > (declare-fun deletePapersBy () Function)
```

Moving forward to the type verification of `deletePapersBy`, the type of the `uid` parameter is determined and declared inside a new scope (lines 14–15) in order to be bound during the verification of the body, a sequence beginning with a `foreach` expression.

```
foreach pid in getPapersBy(uid) do ...
```

The typing rule for the `foreach` expression requires the first expression to be a collection, leading us to type the call to `getPapersBy`:

```
getPapersBy(uid)
```

[1]Recall from Section 6.2 that the solver only allows the control of the scope of assertions, not names.

Besides confirming the identifier is bound, we need to check if the type of the argument (i.e. `uid`) is a subtype of the action's first parameter, that is, check if Int at true() is a subtype of Int at `User(uid)`. From the perspective of the solver, this amounts to checking if the security level true() is implied by `User(uid)`. As the `User` predicate was not declared previously, this also triggers its declaration (line 16).

```
14  > (push 1)
15  > (declare-fun uid () Int)
16  > (declare-fun User#1 (Term) Bool)
17  > (push 1)
18  > (assert (not (=> (User#1 (intToTerm uid)) true)))
19  > (check-sat)
20  unsat
21  > (pop 1)
```

As expected, the solver answers that adding the negated assertion leads to an unsatisfiable theory, proving our claim is valid and, consequently, the subtyping relation holds. Notice that the whole query is performed inside a new assertion scope, so that the negative assertion we perform to check validity does not affect future queries. A similar query is performed to guarantee the call, with program counter `User(uid)`, can be made in the current context, true().

```
22  > (push 1)
23  > (assert (not (=> (User#1 (intToTerm uid)) true)))
24  > (check-sat)
25  unsat
26  > (pop 1)
```

Now that we are sure to be iterating a collection, the cursor `pid` is declared inside a new scope (line 28) and we proceed to type the foreach's body, the delete query, with a higher program counter, `User(uid)`.

```
delete p in Paper where p.id == pid
```

After ensuring `Paper` is indeed an entity, we declare the query's cursor `p` inside a new scope (line 30) and check whether the where clause is well-typed as a boolean.

```
p.id == pid
```

In turn, to ensure the corresponding equality expression is well-typed, both sides of the equality must share a common supertype, leading us to verify if any of the two sides' types is a supertype of the other. Given the current program counter, both sides end up with security level `User(uid)`, triggering a trivial query to the solver.

```
27  > (push 1)
28  > (declare-fun pid () Int)
```

```
29  > (push 1)
30  > (declare-fun p () Record)
31  > (push 1)
32  > (assert (not
33    (=> (User#1 (intToTerm uid)) (User#1 (intToTerm uid)))
34  ))
35  > (check-sat)
36  unsat
37  > (pop 1)
```

Finally, the typing rule for the delete expression requires the security level of each field to be at least as high as the program counter, given that we augment the current knowledge with the where condition (line 36).

```
def entity Paper { id : Int at true(), ... }
```

The algorithm begins by verifying if the `Paper`'s `id` field fulfils the criteria – which it does not – and the type checking process reaches a halt, since the missing ids would allow a public observer of the system to know *which papers* were deleted and conclude they all had the *same author*.

```
38  > (assert (= (R-id p) (intToTerm pid)))
39  > (push 1)
40  > (assert (not (=> true (User#1 (intToTerm uid)))))
41  > (check-sat)
42  sat
```

Concretely, suppose I know from the list of papers that their ids range from 1 to 10 and that I am one of two co-authors of the paper with id 3. Right after the action is executed, if I look at the papers list and see that papers with ids 3 and 5 are missing, I am able to conclude that my co-author was an author of the paper with id 5, although that information should not be available unless I was an author of that paper.

7

Conclusions and Future Work

In this thesis, we developed a type-based information-flow analysis in the setting of database-backed software systems, using first-order logic propositions as security levels which can depend on actual values of the program, provided a prototype implementation on top of the LiveWeb framework and developed a medium-sized example in our language.

While information-flow analysis techniques are not new – and, in fact, we follow a mostly standard approach – the use of logical propositions that depend on data allows for an expressive, and precise, specification of the confidentiality policies that govern the interaction with a database. Moreover, simple labels can easily be encoded in our lattice by representing them as nullary predicates, whereas there is no direct correspondence for the inverse translation since our levels depend on data. The use of data-dependent security levels also implies additional concerns in the definition of the type system, related to the scope of the identifiers that appear in the security levels, as the free occurrences of identifiers must be closed in a sensible way. In our particular case, we universally quantify the identifiers in order to close their occurrences conservatively by increasing the security level.

Without the ability to experiment and validate our ideas, there would be little meaning for our exploratory approach, as we still lack formal proofs that our type system guarantees a *non-interference* property. Our prototype provides a medium for direct experimentation with the development of web applications benefiting from confidentiality guarantees given by the type system, useful in understanding the pragmatic trade-offs involved in the secure implementation of such applications and the modelling of their security policies.

As possible future work supported by the developments of this thesis, we envision:

Proof of Non-interference Proving a non-interference property for our type system would be a crucial development in ensuring it fulfils its purpose. This should be facilitated by the fact that our runtime semantics is that of λ_{DB}, as we extended the language conservatively, and that we could try encoding our language in Flow Caml, since it covers all our data types and has non-inference formally proven. The major challenges then, would be to correctly encode the data manipulation primitives and our data-dependent security levels.

Label Polymorphism Pragmatic approaches to information-flow analysis like Jif and Flow Caml feature label polymorphism in some way to avoid heavily duplicating functions because of distinct security levels. For instance, it would be useful to write actions with generic program counters, such that the parameters' *highest security level* and the result's *lowest security level* can be automatically deduced.

Label Inference A feature that is prominent in Flow Caml, given its ML heritage, is full inference of security labels. In our setting, it appears to be even more relevant from a pragmatic point of view, as logical propositions easily get verbose to write, hindering the understandability of function signatures.

Bibliography

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, January 1999.

[AF12] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. of the 39th ACM Symposium on Principles of Programming Languages*, January 2012.

[Bar92] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[BST10] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. of the 8th International Workshop on Satisfiability Modulo Theories*, December 2010.

[Chl10] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010.

[CKP05] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(2):249–291, March 2005.

[Cok11] D. R. Cok. The SMT-LIBv2 Language and Tools: A Tutorial. Technical report, GrammaTech, Inc., 2011.

[CPS^{+}11] L. Caires, J. A. Pérez, J. C. Seco, H. T. Vieira, and L. Ferrão. Type-based access control in data-centric systems. In *Proc. of the 20th European Symposion on Programming*, pages 136–155, 2011.

[Dom10] M. Domingues. Core language for web applications. Master's thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2010.

[FP91] T. Freeman and F. Pfenning. Refinement types for ML. In *Proc. of the SIGPLAN '91 Symposium on Programming Language Design and Implementation*, pages 268–277. ACM Press, 1991.

[FR05] M. Forsberg and A. Ranta. The labelled BNF grammar formalism. Technical report, Department of Computer Science, Chalmers University of Technology, February 2005.

[GF09] A. D. Gordon and C. Fournet. Principles and applications of refinement types. Technical Report MSR-TR-2009-147, Microsoft Research, 2009.

[JW93] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. of the 20th ACM Symposium on Principles of Programming Languages*, pages 71–84, January 1993.

[LC12] L. Lourenço and L. Caires. Segurança de dados em aplicações centradas em dados por análise de fluxo de informação. In *Proc. of INForum 2012 - Simpósio de Informática*, September 2012.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[Mog89] E. Moggi. Computational lambda-calculus and monads. *4th IEEE Symposium on Logic in Computer Science*, pages 14–23, June 1989.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, January 1999.

[PS02] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. of the 29th ACM Symposium on Principles of Programming Languages*, pages 319–330, January 2002.

[Ree79] T. Reenskaug. Models-views-controllers. Technical report, Xerox PARC, December 1979.

[RT06] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, University of Iowa, August 2006.

[Sim03] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[YYSL12] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proc. of the 39th ACM Symposium on Principles of Programming Languages*, January 2012.

[Zda02] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

[Zda04] S. Zdancewic. Challenges for information-flow security. In *Proc. of the 1st International Workshop on Programming Language Interference and Dependence*, August 2004.

A

A Lattice of First-Order Logic propositions

Notation 1. *We denote the set of all* ***first-order logic formulas*** *as $\mathcal{F}$.*

Notation 2. *We denote by $\leftarrow_{\mathcal{A}}$ the* ***is-implied-by*** *relation ($\leftarrow$) over logical formulas in $\mathcal{F}$, given a set of axioms $\mathcal{A}$. That is, for any two elements a and b of $\mathcal{F}$:*

$$\vdash a \leftarrow_{\mathcal{A}} b \;:\Leftrightarrow\; \mathcal{A} \vdash a \leftarrow b$$

Definition 1. *A* ***partial order*** *$\leq$ is a binary relation on the elements of a set L which is:*

1. *reflexive,*
2. *transitive,*
3. *and anti-symmetric.*

Definition 2. *A* ***partially ordered set*** *(poset) $(L, \leq)$ is defined by a set L and a partial order $\leq$ on the elements of a set L.*

Definition 3. *A* ***lattice*** *$(L, \leq)$ is a poset that satisfies the following axioms. For any two elements a and b of L:*

1. *There exists an $x \in L$ such that:*
    - (a) $a \leq x$
    - (b) $b \leq x$
    - (c) $\forall_{y \in L}. a \leq y \wedge b \leq y \Rightarrow x \leq y$

2. *There exists an $x \in L$ such that:*
   (a) $x \leq a$
   (b) $x \leq b$
   (c) $\forall_{y \in L}. y \leq a \wedge y \leq b \Rightarrow y \leq x$

Lemma 4. *The $\leftarrow_{\mathcal{A}}$ relation over logical formulas in $\mathcal{F}$ is a* ***partial order*** *if we consider equality ($=$) to be defined as logical equivalence ($\leftrightarrow$).*

Proof. By definition 1:

1. **Reflexivity** follows by the definition of logical implication;
2. **Transitivity** follows by the definition of logical implication;
3. **Anti-symmetry** follows by the definition of logical equivalence.

□

Corollary 5. *The set of logical formulas $\mathcal{F}$ with the $\leftarrow_{\mathcal{A}}$ relation is a* ***poset*** *if we consider equality ($=$) to be defined as logical equivalence ($\leftrightarrow$).*

Proof. By definition 2 and lemma 4. □

Lemma 6. *The poset $(\mathcal{F}, \leftarrow_{\mathcal{A}})$ is a* ***lattice*** *if we consider ($=$) to be defined as logical equivalence ($\leftrightarrow$).*

Proof. By corollary 5, we know $(\mathcal{F}, \leftarrow_{\mathcal{A}})$ is a poset. Thus, by definition 3 we must prove that for any two formulas $P, Q \in \mathcal{F}$:

1. There exists a formula $R \in \mathcal{F}$ such that:
   (a) $P \leftarrow_{\mathcal{A}} R$
   (b) $Q \leftarrow_{\mathcal{A}} R$
   (c) $\forall_{S \in \mathcal{F}}. P \leftarrow_{\mathcal{A}} S \wedge Q \leftarrow_{\mathcal{A}} S \Rightarrow R \leftarrow_{\mathcal{A}} S$
2. There exists a formula $R \in \mathcal{F}$ such that:
   (a) $R \leftarrow_{\mathcal{A}} P$
   (b) $R \leftarrow_{\mathcal{A}} Q$
   (c) $\forall_{S \in \mathcal{F}}. S \leftarrow_{\mathcal{A}} P \wedge S \leftarrow_{\mathcal{A}} Q \Rightarrow S \leftarrow_{\mathcal{A}} R$

We can prove 1 by considering the formula $P \wedge Q$:

1. $P \leftarrow_{\mathcal{A}} P \wedge Q$ (conjunction elimination)
2. $Q \leftarrow_{\mathcal{A}} P \wedge Q$ (conjunction elimination)

3. $\forall_{S \in \mathcal{F}}.P \leftarrow_{\mathcal{A}} S \wedge Q \leftarrow_{\mathcal{A}} S$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash P \leftarrow S) \wedge (\mathcal{A} \vdash Q \leftarrow S)$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash P \leftarrow S \wedge Q \leftarrow S)$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash (P \vee \neg S) \wedge (Q \vee \neg S))$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash (P \wedge Q) \vee \neg S)$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash (P \wedge Q) \leftarrow S)$
$\Rightarrow \forall_{S \in \mathcal{F}}.(P \wedge Q) \leftarrow_{\mathcal{A}} S$

Similarly, we can prove 2 by considering the formula $P \vee Q$:

1. $P \vee Q \leftarrow_{\mathcal{A}} P$ (disjunction introduction)

2. $P \vee Q \leftarrow_{\mathcal{A}} Q$ (disjunction introduction)

3. $\forall_{S \in \mathcal{F}}.S \leftarrow_{\mathcal{A}} P \wedge S \leftarrow_{\mathcal{A}} Q$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash S \leftarrow P) \wedge (\mathcal{A} \vdash S \leftarrow Q)$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash S \leftarrow P \wedge S \leftarrow Q)$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash (S \vee \neg P) \wedge (S \vee \neg Q))$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash S \vee (\neg P \wedge \neg Q))$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash S \vee \neg(P \vee Q))$
$\Rightarrow \forall_{S \in \mathcal{F}}.(\mathcal{A} \vdash S \leftarrow (P \vee Q))$
$\Rightarrow \forall_{S \in \mathcal{F}}.S \leftarrow_{\mathcal{A}} (P \vee Q)$

□

B

Solver Preamble

Listing B.1: Solver encoding preamble

```
(set-logic AUFLIA)
(declare-sort MyBool 0)
(declare-fun myTrue () MyBool)
(declare-fun myFalse () MyBool)
(assert (distinct myTrue myFalse))
(declare-fun myNot (MyBool) MyBool)
(assert (= (myNot myTrue) myFalse))
(assert (= (myNot myFalse) myTrue))
(declare-fun myAnd (MyBool MyBool) MyBool)
(assert (= (myAnd myTrue myTrue) myTrue))
(assert (= (myAnd myTrue myFalse) myFalse))
(assert (= (myAnd myFalse myTrue) myFalse))
(assert (= (myAnd myFalse myFalse) myFalse))
(declare-fun myOr (MyBool MyBool) MyBool)
(assert (= (myOr myTrue myTrue) myTrue))
(assert (= (myOr myTrue myFalse) myTrue))
(assert (= (myOr myFalse myTrue) myTrue))
(assert (= (myOr myFalse myFalse) myFalse))
(declare-sort String 0)
(declare-fun nil () String)
(declare-fun cons (Int String) String)
(assert (forall ((s String) (ch Int)) (distinct nil (cons ch s))))
(declare-fun concat (String String) String)
(assert (forall ((s String)) (= (concat nil s) s)))
(assert (forall ((s String)) (= (concat s nil) s)))
```

```
(assert (forall ((ys String) (xs String) (x Int))
  (= (concat (cons x xs) ys) (cons x (concat xs ys)))
))
(declare-sort Record 0)
(declare-fun numFields (Record) Int)
(declare-sort Block 0)
(declare-sort Function 0)
(declare-sort Collection 0)
(declare-sort Term 0)
(declare-fun termToInt (Int) Term)
(declare-fun intToToTerm (Term) Int)
(assert (forall ((t Term) (x Int)) (and
  (=> (= t (termToInt x)) (= (intToTerm t) x))
  (=> (= (intToTerm t) x) (= t (termToInt x)))
)))
(declare-fun termToMyBool (MyBool) Term)
(declare-fun myBoolToTerm (Term) MyBool)
(assert (forall ((t Term) (x MyBool)) (and
  (=> (= t (termToMyBool x)) (= (myBoolToTerm t) x))
  (=> (= (myBoolToTerm t) x) (= t (termToMyBool x)))
)))
(declare-fun termToString (String) Term)
(declare-fun stringToTerm (Term) String)
(assert (forall ((t Term) (x String)) (and
  (=> (= t (termToString x)) (= (stringToTerm t) x))
  (=> (= (stringToTerm t) x) (= t (termToString x)))
)))
(declare-fun termToRecord (Record) Term)
(declare-fun recordToTerm (Term) Record)
(assert (forall ((t Term) (x Record)) (and
  (=> (= t (termToRecord x)) (= (recordToTerm t) x))
  (=> (= (recordToTerm t) x) (= t (termToRecord x)))
)))
(declare-fun termToBlock (Block) Term)
(declare-fun blockToTerm (Term) Block)
(assert (forall ((t Term) (x Block)) (and
  (=> (= t (termToBlock x)) (= (blockToTerm t) x))
  (=> (= (blockToTerm t) x) (= t (termToBlock x)))
)))
(declare-fun termToFunction (Function) Term)
(declare-fun functionToTerm (Term) Function)
(assert (forall ((t Term) (x Function)) (and
  (=> (= t (termToFunction x)) (= (functionToTerm t) x))
  (=> (= (functionToTerm t) x) (= t (termToFunction x)))
)))
(declare-fun termToCollection (Collection) Term)
```

```
(declare-fun collectionToTerm (Term) Collection)
(assert (forall ((t Term) (x Collection)) (and
  (=> (= t (termToCollection x)) (= (collectionToTerm t) x))
  (=> (= (collectionToTerm t) x) (= t (termToCollection x)))
)))
```