Maria Luísa Sobreira Gouveia Lourenço

B.Sc., M.Sc., Mestre em Engenharia Informática

A Type System for Value-dependent Information Flow Analysis

Dissertação para obtenção do Grau de
Doutor em Engenharia Informática

Orientador: Luís Caires, Prof. Catedrático,
Universidade Nova de Lisboa

Júri:	
Presidente:	[Nome do presidente do júri]
Arguentes:	[Nome do primeiro arguente]
	[Nome do segundo arguente]
Vogais:	[Nome do primeiro vogal]
	[Nome do segundo vogal]

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Julho, 2015

A Type System for Value-dependent Information Flow Analysis

To my loving family

Acknowledgements

First and foremost I would like to thank my advisor Luís Caires not only for all his support and expertise but also for setting the bar high. If we do not strive for excellence then we will not be able to achieve the best possible results nor will we better ourselves. I am sure the outcome of this thesis would have been very different if the bar was set lower. I deeply appreciate all that I gained from working with Luís, from the rigorous work methodology to the simple but effective presentation and communication skills that he taught me in the past years. I am also thankful for all Luís's classes, from the compilers classes to the foundations of programming languages' classes, which were very inspirational and without them I am sure I would not have enrolled in a Ph.D. and much less in this area of expertise. I would like to also thank my thesis advisory committee for their comments and feedback, namely Carla Ferreira, Marzia Buscemi, and Vasco Vasconcelos.

I would like to thank my office colleagues for all the discussions and comments I received over the years as well as the PLASTIC research group. I thank Bernardo Toninho for the discussions regarding dependent type theory but also other subjects related to this work. I thank Tiago Santos for the discussions and ideas he contributed for this work, and also for programming such a useful Scala SMT library which he kindly let me use for this thesis's prototype. I thank Miguel Domingues for all the help he so patiently gave in the past years, namely for helping me setting up a web service for this thesis prototype so I could submit a request for rise4fun's tool page. I thank Mário Pires and Paulo Ferreira for all the discussions related to language-based security we had while they stayed in the office. I thank Jorge Pérez for all his feedback and helpful ideas. I also thank João Seco and Carla Ferreira for their comments during this work.

I would like to thank all the great teachers that formed me academically from undergraduate years through graduate courses. In particular I would like to thank João Lourenço, José Cardoso e Cunha, Margarida Mamede and Luís Monteiro for their invaluable contributions to my academic formation.

I thank all my buddies from doctoral program whether for the lunch conversations or just coffee: Sofia Gomes, Jorge Costa, Filipa Peleja, Ricardo Silva, Sinan Elgimez, João Martins, Mário Pires, Miguel Lourenço.

This work would not be possible if I did not have such great friends, whom I do not need to enumerate, in my life. To them thank you for your patience and for helping me distract from my Ph.D. when I needed the most. I must also mention how grateful I am for coming back to my long awaited passion after nearly 8 years of inactivity: Kendo.

It was great to re-establish old friendships and make new ones. Kendo has been very inspirational to me and helped me through rough times throughout my life. Similar to what I have also learned during these past years during my Ph.D., I have learned to strive for the best by giving the best of me, to better myself. Better yet, to overcome myself and my barriers as well as any obstacle life throws at me. Without Kendo I am sure my path would have been harder, if not impossible, to bear.

To conclude, I would like to thank my family for their love and support during these past years, namely my mother and brother but also my sister-in-law, aunts, uncles and cousins who always encouraged me to pursue my dreams and keep on fighting for what I believe is best for me. I would also like to thank my old buddy, "canine brother", Gastão from whom I am lucky to still count with his loyalty and friendship after nearly 16 years. A special note to my late grandparents and father, whom I am sure would be very proud of my achievements so far.

I am sure it was not easy to put up with me, thank you all for your love and patience.

This thesis work was supported by CITI PEst-OE/EEI/UI0527/2014, FCT/MEC grant SFRH/BD/68801/2010, and FLEX-Agile grant by OutSystems SA.

Abstract

Information systems are widespread and used by anyone with computing devices as well as corporations and governments. It is often the case that security leaks are introduced during the development of an application. Reasons for these security bugs are multiple but among them one can easily identify that it is very hard to define and enforce relevant security policies in modern software. This is because modern applications often rely on container sharing and multi-tenancy where, for instance, data can be stored in the same physical space but is logically mapped into different security compartments or data structures. In turn, these security compartments, to which data is classified into in security policies, can also be dynamic and depend on runtime data.

In this thesis we introduce and develop the novel notion of dependent information flow types, and focus on the problem of ensuring data confidentiality in data-centric software. Dependent information flow types fit within the standard framework of dependent type theory, but, unlike usual dependent types, crucially allow the security level of a type, rather than just the structural data type itself, to depend on runtime values.

Our dependent function and dependent sum information flow types provide a direct, natural and elegant way to express and enforce fine grained security policies on programs. Namely programs that manipulate structured data types in which the security level of a structure field may depend on values dynamically stored in other fields

The main contribution of this work is an efficient analysis that allows programmers to verify, during the development phase, whether programs have information leaks, that is, it verifies whether programs protect the confidentiality of the information they manipulate. As such, we also implemented a prototype typechecker that can be found at `http://ctp.di.fct.unl.pt/DIFTprototype/`.

Keywords: Information Flow, Type Systems, Dependent Types, Language-based Security

RESUMO

Os sistemas de informação estão generalizados e são usados por qualquer individuo com dispositivos de computação, bem como empresas e entidades governamentais. Em muitos casos, as fugas de segurança são introduzidas durante o desenvolvimento de uma aplicação. As razões para tal são múltiplas, mas entre elas pode-se facilmente identificar que é muito difícil definir e aplicar políticas de segurança relevantes no software moderno. Isto se deve ao facto das aplicações modernas dependerem muitas vezes de partilha de armazenamento e *multi-tenancy*, onde, por exemplo, os dados podem ser armazenados no mesmo espaço físico mas são logicamente mapeados em compartimentos diferentes de segurança ou estruturas de dados. Por sua vez, esses compartimentos de segurança, para os quais os dados são classificadas nas políticas de segurança, também podem ser dinâmicos e depender de dados de tempo de execução.

Nesta tese introduzimos e desenvolvemos o novo conceito de tipos de fluxo de informação dependentes, e focamos no problema de assegurar a confidencialidade dos dados em software centrado em dados. Os tipos de fluxo de informação dependentes enquadram-se no *standard* da teoria de tipos dependentes mas, ao contrário dos tipos dependentes habituais, crucialmente permitem que o nível de segurança de um tipo, em vez de apenas o próprio tipo de dados, dependa de valores de tempo de execução.

Os nossos tipos de fluxo de informação dependentes funcionais e soma fornecem uma maneira directa, natural e elegante de expressar e aplicar políticas de segurança refinadas sobre os programas. Nomeadamente em programas que manipulam tipos de dados estruturados em que o nível de segurança de um campo na estrutura pode depender de valores armazenados de forma dinâmica em outros campos.

A principal contribuição deste trabalho consiste numa análise eficiente que permite aos programadores verificar, durante a fase de desenvolvimento, se os programas contêm fugas de informação, isto é, verifica se os programas protegem a confidencialidade da informação que manipulam. Como tal, também implementámos um prótotipo do typechecker que pode ser encontrado em `http://ctp.di.fct.unl.pt/DIFTprototype/`.

Palavras-chave: Fluxo de Informação, Sistema de Tipos, Tipos Dependentes, Segurança via Linguagens de Programação

CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

Information systems are widespread and used by anyone with computing devices as well as corporations and governments. While information systems can be found in a wide variety of architectures and configurations (going from centralised systems, e.g. in a business corporation, to distributed systems, e.g. web applications for online stores or social networks), there is one point in common among these systems: they deal with huge amounts of data.

It then comes as no surprise that data security is a critical issue in information systems, deserving much attention and focus on both academia and business corporations in the past decades. Moreover, it is often the case that security leaks are introduced during the development of an application. Reasons for these security bugs are multiple but among them one can easily identify that it is very hard to define and enforce relevant security policies in modern software.

Defining relevant security policies is challenging since modern applications often rely on container sharing and multi-tenancy where, for instance, data can be stored in the same physical space but is logically mapped into different security compartments or data structures. In turn, these security compartments, to which data is classified into in security policies, can also be dynamic and depend on runtime data (including configuration parameters).

For instance, suppose the photos of a user are classified at security compartment `usr`. Then the photos of user `joe` are classified at security compartment `usr`("joe").

These sort of security policies are known as "row-level" policies in the databases community. We can define "row-level" security as follows:

"*Row-Level Security enables customers to control access to rows in a database table based on the characteristics of the user executing a query (e.g., group membership or execution context).*"[1].

[1] https://msdn.microsoft.com/en-us/library/dn765131.aspx

"Row-level" policies are usually used in multi-tenant applications since it allows to create a policy to enforce a logical separation of each tenant's data rows from every other tenant's rows. Enabling the application to use a single table to store data for many tenants.

In this thesis we introduce the novel notion of *dependent information flow types*, and focus on the problem of ensuring data confidentiality in data-centric software. Namely, this thesis aims to defend the following statement:

> **Thesis statement**: *Dependent information flow types are suitable to reason about data confidentiality in data-centric software, providing an elegant and lightweight theoretical system to express and reason about "row-level" security properties that naturally occur in the setting of data-centred systems.*

In the following section we discuss how one can specify security policies in system.

1.1 Modelling and Reasoning about Security Policies

Security policies establish rules and procedures that must be met in order to gain access to protected information. Since not all information holds the same value within a system, data must be classified into security compartments (which define degrees of protection) and must be treated differently.

Key concepts in information security are: confidentiality, integrity, and availability. Confidentiality consists in preventing the disclosure of sensitive data to anyone who does not have permission to access the data. Integrity, however, is important to maintain data uncorrupted and coherent, meaning it is crucial that we ensure that no unauthorised operation is executed over the data. Finally, there is no use in ensuring data confidentiality and integrity if such data cannot be made available when necessary – this is known as availability.

Two complementary approaches are used to enforce security policies in information systems: access control mechanisms and information flow analysis. The former consists in defining access control policies over resources, while the latter concerns in preventing insecure information flows throughout the execution flow of the system.

Access Control. Access control allows us to specify policies over data such that the policy describes which permissions are required in order for data to be accessible by a user. Permissions must then be granted, or revoked, to system's users by the administrator. Thus, access control defines who can access data.

Access control models have been, and still are, widely studied. We mention some of the most relevant: Mandatory Access Control (MAC), Discretionary Access Control (DAC), and Role-based Access Control (RBAC).

Mandatory Access Control [18] key idea consists in enforcing a system wide policy that states who has access to what resources. This policy can only be set by the system

administrator. On the other hand, in Discretionary Access Control [32] the system users are allowed to define the access policies over the objects they own.

In Role-based Access Control [23, 50], permissions are not assigned to users. Instead we have a set of roles to which we can assign permissions. The members of a role will then inherit those permissions. This adds more flexibility to access control policies (in contrast to MAC and DAC), since we can manage access control policies more easily and intuitively. This is because, intuitively, roles represent a user's responsibility/job inside an organisation, so for example we can define an hierarchy of roles that correspond to the organisation's hierarchy and assign users to their respective roles. Also we can change a role's permission set without having to re-assign permissions to each member.

As we already stated, confidentiality of data is essential for information systems. While access control policies are enough to ensure sensitive data is only obtainable for those who have the correct permissions, it cannot give any guaranties concerning how the data will be used afterwards. These guaranties are given by information flow analyses.

Information Flow. Information Flow [17, 50, 52] ensures data confidentiality by classifying information with levels of security (the highest the level the more sensitive the data is), forming a (security) lattice, and then ensuring that the flow of information goes only from lower to higher levels of security (meaning there is no information flow that leaks private data).

The enforcement of *confining* a system such that it will not leak private data is known as the confinement problem [31] in the literature. One of the greatest challenges in information flow consists in dealing with channels that exploit mechanisms that are not intended for information transmission and allow an attacker to infer some confidential data by observing the behaviour of the program, known as covert channels [31]. A couple of examples of such channels are implicit flows and termination channels.

Typically, in the literature [17, 64], insecure flows can either be explicit or implicit. Moreover, data confidentiality is ensured by enforcing programs to preserve a noninterference property. In practice, this means that data should only flow from a lower level to an higher level of security.

So, an explicit flow corresponds to a direct mapping of classified information to a lower classified container (data-flow based), while an implicit flow corresponds to public information that depends on classified one (control-flow based).

Classic examples of such flows are the assignment of a low level variable with a high level value, `l:=h`, for explicit flow; and a high guarded conditional whose branches are classified as low level, **`if`** `h > 0` **`then`** `l:= 1` **`else`** `l:= 0`, for implicit flows. A termination channel, however, occurs when the termination behaviour of a program depends on sensitive data. For e.g., **`if`** `h` **`then`** `(`**`while true do`** `skip)`**`else`** `skip`.

A property of non-interference [26] is usually employed to enforce information flow security of an application. This property states that changing sensitive data of a program does not change the perception that an external observer has on the output of a program,

which implies that no public data depends on protected data. So, in other words, noninterference ensures data confidentiality by certifying that a compliant program does not have insecure flows.

This, however, can be very restrictive if we take into consideration that applications sometimes need to release sensitive data. For example, any application that requires authentication will have to disclose to the user if the typed password was correct or incorrect, thus leaking some information regarding the protected information. This is known as information declassification [54].

In this thesis we do not deal with declassification nor termination channels, thus we ensure our analysis enforces a termination-insensitive noninterference property.

Next, we discuss some of the relevant techniques employed in the programming languages community to enforce security policies in systems.

1.2 Language-based Security Techniques

In the past decade, the usage of language-based security techniques – such as compilers [48], proof-carrying code [47] , inline reference monitors [20, 55], and type systems [6, 10, 24] – to enforce security policies in computer systems has shown promise in real-world scenarios.

These techniques can be applied during software's development time (static analysis) or during their execution (dynamic analysis). In static language-based techniques, the main idea consists in analysing the source code before being deployed for execution, preventing its deployment if a security policy is violated. Dynamic language-based techniques take a different approach, they rely on observation of a program's behaviour during its execution to detect violations of the security policy, stopping the program before the insecure operation is executed.

Dynamic Analysis. Dynamic approaches for security consist in techniques such as inline reference monitors (IRM) [20, 55] to guarantee an application's security policies.

An IRM shares the address space of the application it monitors, this requires that the IRM be merged into the application's code, at compile time. This merge is achieved through program rewriting (code instrumentation) to insert security checks in the code. IRM enforces security policies while the application is running: should the application attempt to violate the security policy, the IRM halts it. Therefore, an IRM mediates between the client and the application.

We mention some of the relevant work achieved in the area of IRM. In [22], Erlingsson and Schneider present SASI, an IRM that generalises Software Fault Isolation [65] to any security policy that can be described with a security automaton [55].

However, in [20, 21] Erlingsson and Schneider introduce an IRM, denoted as PoET/PSLang, that targets Java application by adding checks in the Java Virtual Machine (JVM) code.

Recent works have begun to target web applications, for instance in [49] Phung et al. mediate access to sensitive DOM objects and properties, and in [33, 58] a mediation for flash applications is introduced. IRM implementations, however, must take into consideration possible actions to circumvent the added checks in a program (for example, jump over those checks). In order to prevent such actions, IRM implementations usually impose restrictions on control flow [37].

A complementary approach, named Control-Flow Integrity (CFI) [2], consists in defining security policies with a Control-Flow Graph (CFG) and then ensuring that an application's execution proceeds along paths in the CFG. This enforcement, much like IRM's, is achieved through program rewriting to insert dynamic checks in the application's code.

Both these approaches can enforce access control policies so we can only specify security policies that talk about operations over data. This is not enough to ensure security over data itself, we need to be able to state and enforce security properties over data.

Static Analysis. On the realm of static language-based security approaches, we point out some of the relevant works. For instance, code certification is a well studied static technique to ensure applications are safe with respect to a security policy. It consists on having the program developer produce a certificate, i.e. a proof, that his code complies with the security policy. This certificate is then checked by the client before executing the application, thus preventing malicious code (those that do not pass the certificate checker). Moreover, the certificate is produced by a certifying compiler [48] and then checked by a theorem prover.

One form of code certification is proof-carrying code [47] (PPC). The programmer annotates his code with properties that must hold during execution, this is required for non-programs or non-trivial properties since these annotations are program specific and therefore cannot be inferred from the security policy. Thus the certifying compiler must have a module to generate proof obligations (the Verification Conditions Generator, VCG) when compiling the annotated code.

Furthermore, the compiler will carry over the annotations to the object-code level. In order for the programmer to verify if his program complies with a security policy, he needs to run the VCG over the annotated object-code, along with the security policy, to generate proof obligations. These proof obligations are then proved to hold for the program by a automatic theorem prover, otherwise it will mean that the program does not comply with the security policy.

Another kind of code certification is type assembly language (TAL) [25, 40] where type annotations are perceived as a proof of type safety and the type checker corresponds to the proof checker. TAL transforms, at compile time, type information from source language to a platform independent typed intermediate language (TIL) [62], and then to a typed object-code. This allows the typed object code to be verified by any type checker. TAL can enforce any security property that can be expressed with a type system.

Works such as [13], [30], and [39, 40] focus on using certifying compilers to prove

standard type safety properties.

In [24], Fournet et al. present a type system that statically checks if a program respects an authorisation policy for access control over sensitive resources. Their work extends a typed version of Spi calculus, a process calculus with cryptographic operations, with an authorisation logic and code annotations to state the authorisation policy (unguarded statements). Their goal is to verify facts about data arising at run-time (input guarded statements), and to statically check pre-condition over sensitive resources (expectations). For instance, they can encode a simple RBAC policy by defining roles and permissions via logic rules and members with facts.

In [10], Caires et al. present a functional language with SQL-based constructions to represent and manipulate information. Their goal consists in statically enforcing, via a type-based approach, access control policies in a data-centric setting. They use refinement types that can specify policies that depend on the current state of a database. Since they have database entities as a value of the language, they express access control concerns by annotating each entity with read and write permissions using predicates. These permissions are then statically checked each time an operation is executed on the entity by checking if the current knowledge (predicates that we know to be true so far, for e.g. by applying dependently typed functions with post-conditions) is enough to derive the required set of permissions. Their work however does not deal with information flow analysis.

We favour static language-based security since, in some cases, an error during execution time can be, by itself, a security breach. Moreover, with static based techniques we are able to detect more insecure programs since these techniques reason about all possible execution paths.

In the following section, we discuss some works that explore type systems within the context of an information flow analysis.

1.3 Type-based Information Flow Analysis

Type-based information flow analysis has gained great focus in the research community in the past years. Several proposals for dynamic information flow analysis on web languages have been put forward.

In [4], Austin and Flanagan propose a dynamic information flow mechanism for a Javascript-like language based on a notion of faceted values. Faceted values offer different views of a value given the execution context's principal. Other recent work by Hedin and Sabelfeld [27] proposes a dynamic information flow analysis for a subset of the ECMA standard for Javascript.

In [19], Enck et al. introduce a taint analysis for mobile applications, where implicit flows are not taken into account to minimise performance overhead, and in [16] Davis and Chen develop a dynamic analysis to prevent insecure cross-application information flows.

Other works based on dynamic analysis for operating systems include [11, 56, 69]. While their focus is not language-based security, they use concepts first introduced by language-based approaches, for e.g. the decentralised label model (DLM)[43] is widely used in these systems.

Our work, however, is based on static analysis, as we seek to obtain compile time security guarantees, and avoid possible information leaks due to exceptional behaviour (dynamic security errors).

Static approaches for type-based information flow analysis has attracted substantial research effort for a long time (see e.g., [52]). In early works the focus was on imperative languages [53, 64], λ-calculus [1, 28, 51], object-oriented languages [42], and concurrent languages [29, 68].

More recently, there has been a growing interest in studying secure information flows in data-centric applications, to cite a few of the relevant work: [7, 12, 15].

In [15] Corcoran et al. present a static analysis to enforce label-based security policies in the web programming language SELinks. Their analysis is able to enforce relevant information flow policies in web applications although the authors do not discuss the noninterference property.

The approach taken by Chlipala [12] consists in adding program specifications expressed by SQL-queries which are then typechecked, while in [7], Bierman et al. use refinement types and semantic subtyping to enforce properties that may be relevant for security.

Unlike our approach, these works do not provide a value-dependent information flow analysis leading to non-interference results, as this thesis work does. Moreover, our core language can easily encode common data manipulation language (DML) operations (as we will show in Chapter 5) and thus our analysis is general enough to ensure noninterference on data-centric applications, which usually involves expressive security policies, depending on runtime values, often required in realistic applications.

Two interesting ideas put forward recently are the specification of security policies that rely on runtime first-class representations of principals, by Tse and Zdancewic [63], and security labels that can dynamically change, by Zheng and Myers in [70].

The former is based on the (seminal) DLM introduced by Myers and Liskov in [43], and presents a typed λ-calculus where principals are values and thus can be mentioned during a program, for e.g. for conditional testing, increasing the expressiveness of the security policy model. The authors also prove a noninterference result for an information flow type system using this notion of runtime principals.

Although it is conceivable that some dynamically enforced form of value dependent security label could be encoded in some version of the DLM (e.g., using label passing [3]) in this work we deliberately focus on a direct and lightweight static approach.

The second work, introduces a static type-based information flow analysis where security labels can change during execution time and are case-analysed via a label-test

primitive. This construction is used to add label constraints that are statically checked by the type checker.

In this thesis work we do not consider runtime principals nor dynamically changing labels but, instead, use runtime values to index security labels to ensure data dependent security policies.

We proceed in the next section with the discussion of recent work on dependent types applied to language-based security.

1.4 Dependent Types and Security Types

Several recent works explore applications of dependent types [46, 60, 61] to language-based security in the context of stateful static information flow.

In [60] Swamy *et al.* present FINE, a general-purpose and very expressive dependently typed language based on Fable [59], and suggest several encodings in the language of high-level security concerns such as information flow and access control policies.

To express an information flow analysis in such setting, the programmer is required to hardcode the security labels as well as the lattice and all its operations/axioms (meet, join, partial order relation, etc) into inductive types and logic formulae within a module that internalizes the intended information flow policy inside the framework.

Moreover, a value abstraction result is presented, stating that code within a module does not interfere with another module's protected code, which is different from the (standard) notion of noninterference used in our work, and does not primitively and explicitly address the fundamental notion of value dependent classification through dependent typing, which is the core contribution of our work (which, in addition, covers a language with general imperative features).

Also, the use of dependent types to express security properties in such line of work relies on refinement types and relative logical encodings of meta properties, which is very different from what we do here, that does not involve refinement types, and adopts a simple and primitive notion of value dependent classification directly at the level of the type structure, leading to an absolute non-interference theorem.

In [46], Nanevski *et al.*, use a very expressive relational Hoare type theory (RHTT) to reason about access control and information flow in stateful programs.

Besides standard dependent types, this work introduces a special dependent type, `STsec`, to specify security policies via pre and post-conditions, using higher-order logic formulae capable of expressing heap union disjointness. The `STsec` type is used to type potentially side-effectful operations, but the relevant part *w.r.t.* to information flow analysis is the post-condition that specifies the behaviour of two different runs of the program, relating the outputs, input heaps and output heaps of any two terminating executions of the program.

Another interesting work, based on [61] and [46], is RF* [5], where Barthe et al. introduce the notion of relational refinement types. The key idea of relational refinement types

consists in extending classic refinement types to relational formulae, which in turn enables to relate the left and right value of every program variable in scope through projections L and R. With this setting, the author's type system is able to relate expressions at a relational refined type that can describe the results of both expressions.

A distinguishing feature of these latter approaches is that data is not classified with security labels (as expected from traditional information flow analyses). Instead, and similarly to the approach of Swamy et al. [60], the noninterference property is expressed directly in the post-condition via detailed assertions that relate the initial heap with the final heap as well as the output values for any two runs of the program.

While it might be conceivable, in principle, to express value-dependent information flow policies in such a framework, and in fact, in any sufficiently expressive logical framework for imperative programs supporting general functional properties, the goal of our work follows a much lightweight and tractable type-based approach, and aims to single out and address in a direct and explicit way the core notion of value dependent information classification.

We proceed in the next section with the motivation of our approach.

1.5 Data Confidentiality in Data-Centric Software

In this section, we motivate dependent (function and sum) information flow types by means of several examples.

As in any information flow analysis, we are concerned about insecure flows that might arise during the execution of a program but not with how data is accessed (that concerns access control analyses). Our analysis associates security levels s to types τ to classify expressions e, so typing an expression at security level s, denoted $\Delta \vdash e : \tau^{\mathsf{s}}$, means that data used or computed by expression e will only be affected by data classified at security level up to s.

We proceed by illustrating our programming language, a simple λ-calculus with references, using as toy example, a typical data centric web application: a conference manager.

In this scenario, a user of the system can be either a registered user, an author user, or a programme committee (PC) member user. The system stores data concerning its users' information, their submissions, and the reviews of submissions in "database tables" which we will represent in our core programming language as lists of (references to) records (e.g., mutable lists):

$\tau_a \stackrel{\text{def}}{=} [\texttt{uid} : \texttt{int} \times \texttt{name} : \texttt{str} \times \texttt{univ} : \texttt{str} \times \texttt{email} : \texttt{str}]$

$\sigma_a \stackrel{\text{def}}{=} [\texttt{uid} : \texttt{int} \times \texttt{sid} : \texttt{int} \times \texttt{title} : \texttt{str} \times \texttt{abs} : \texttt{str}^* \times \texttt{paper} : \texttt{int}^*]$

$\delta_a \stackrel{\text{def}}{=} [\texttt{uid} : \texttt{int} \times \texttt{sid} : \texttt{int} \times \texttt{PC\_only} : \texttt{str}^* \times \texttt{review} : \texttt{str}^* \times \texttt{grade} : \texttt{int}]$

$$\textbf{let}\ \texttt{Users} = \textbf{ref}_{\texttt{ref}(\tau_a)^{*\perp}}\ (\textbf{ref}_{\tau_a}\ [\,]\ )::\{\}\ \textbf{in}$$

```
let Submissions=ref_{ref(σ_a)*⊥} (ref_{σ_a} [] )::{} in
let Reviews = ref_{ref(δ_a)*⊥} (ref_{δ_a} [] )::{}
```

So `Users` stores information for each registered user; `Submissions` keeps track of each submission in the system by storing its id, the author's id, and the contents of the submission; and `Reviews` stores information regarding the evaluation of each submission, namely the id of the PC member reviewing the submission, the id of the submission, the comments for the other PC members, and the comments and grade to be delivered to the author.

The system offers operations to add new data as well as some listing operations, we exemplify some of them.

Example 1 Operation `assignReviewer` assigns a PC member to review a given submission, initialising the remaining fields.

```
let assignReviewer = λ (u, s).
   let new_rec = ref_{δ_a} [ uid = u, sid = s, PC_only = "",
                          review = "", grade = "" ]
   in Reviews := new_rec :: !Reviews
```

Example 2 Operation `viewAuthorPapers` iterates the `Submissions` collection to build a list of all records with a given author id

```
let viewAuthorPapers = λ (u).
  foreach(x in !Submissions) with y = {} do
    let tuple = !x in
      if tuple.uid = u then tuple::y else y
```

Example 3 Operation `viewAssignedPapers` simulates a join operation between collections `Reviews` and `Submissions` to obtain the list of submissions assigned to the PC member with the given id.

```
let viewAssignedPapers = λ (uid_r).
   foreach (x in !Reviews) with res_x = {} do
      let tuple_rev = !x in
       if tuple_rev.uid = uid_r then
         (foreach(y in !Submissions) with res_y = {} do
            let tuple_sub = !y in
              if tuple_sub.sid = tuple_rev.sid then
                 tuple_sub::res_y
              else res_y )::res_x
       else res_x
```

The **foreach** iterator is a familiar functional collection fold combinator [7] where x is the current item/cursor and res_x denotes the value accumulated from previous iteration, with initial value {}).

For instance,

```
foreach(x in viewAuthorPapers(03)) with count = 0 do count + 1
```

returns the number of submissions of author with id 03.

Our goal is to statically ensure by typing the confidentiality of the data stored in the conference manager system.

As in classical approaches (e.g., [1, 28]), both a type τ and a security label s are assigned to expressions by our typing judgment $\Delta \vdash e{:}\tau^{\mathsf{s}}$, expressing the fact that the value of e will only be affected by computations interfering at security levels $\leq \mathsf{s}$.

Let us now discuss the expressiveness of the security policies using standard type-based information flow approaches.

1.5.1 Expressiveness of Security Policies

As is usual in information flow analysis, a partial order (the so-called security lattice) relating security levels is defined, and information is only allowed to flow upwards (in the order). For the purpose of static code analysis, the given security lattice could be declared as a preamble to the code to be checked.

To specify security policies for our system, we thus classify the data manipulated by our conference manager with security levels from a suitable security lattice (omitting data types when not necessary, for simplicity).

We assume security lattices are bounded by a top, $\top$, and bottom, $\bot$ element denoting the most restrictive (no one can observe) and most permissive (public data, anyone can observe) security levels, respectively.

For the conference manager we can then specify, say, that information is classified in three additional security levels:

- `U(uid)`, for the data that can be disclosed to any registered user;
- `A(uid,sid)`, for data observable to authors; and
- `PC(uid,sid)`, for data that only programme committee members can see.

In such simple case, we may let $\bot{<}\mathtt{U}{<}\mathtt{A}{<}\mathtt{PC}{<}\top$ and specify the according security policy for each conference manager entity:

$\sigma_b \stackrel{\text{def}}{=} [\mathtt{uid} : \bot \times \mathtt{sid} : \bot \times \mathtt{title} : \mathtt{A} \times \mathtt{abs} : \mathtt{A} \times \mathtt{paper} : \mathtt{A}]$

$\delta_b \stackrel{\text{def}}{=} [\mathtt{uid} : \bot \times \mathtt{sid} : \bot \times \mathtt{PC\_only} : \mathtt{PC} \times \mathtt{review} : \mathtt{A} \times \mathtt{grade} : \mathtt{A}]$

$\mathbf{let}\ \mathtt{Users} = \mathbf{ref}_{\mathtt{ref}(\tau_b)^{*\bot}}\ (\mathbf{ref}_{\tau_b}\ [\,]\ )::\{\}\ \mathbf{in}$

let Submissions=**ref**$_{\texttt{ref}(\sigma_b)^{*\perp}}$ (**ref**$_{\sigma_b}$ [])::{} **in**
let Reviews = **ref**$_{\texttt{ref}(\delta_b)^{*\perp}}$ (**ref**$_{\delta_b}$ [])::{}

The security lattice together with these types specify the following policy:

Policy 1 (Bad Policy)
A registered user's information is observable from security level `U`, meaning any registered user (including authors and PC members) can see it. The content of a paper can be seen by authors. And, finally, regarding a submission's review we have that comments to the PC are observable only to its members, while reviews and grade of the submission can be seen by authors.

This policy, however, is not precise enough to protect the confidentiality of the data. An author, who has at least the security level `A`, is able to execute the operation `viewAuthorPapers` (Example 2) using a different id than his own, which clearly violates confidentiality.

Thus, the security policy that we want is the following:

Policy 2 (Good Policy)
A registered user's information is *only* observable *by himself*. The content of a paper can be seen by *its author as well as its reviewers*. And, regarding a submission's review, we have comments to the PC can *only* be observable to other members that are *also reviewers of the submission*, while comments and grade of the submission can be seen by *its authors only*.

To express these kind of data-dependent policies, and make sure that operations that depend on them are secure according to the given policies (such as the operation illustrated above), we introduce a general notion of dependent information flow type, which builds on the notion of *indexed* security label [35].

In the following section, we will give an overview of this thesis work.

1.6 Dependent Information Flow Types

In this section we introduce the main concepts of our dependent information flow types and provide an informal overview of the approach developed in this thesis with a toy example.

Dependent information flow types provide a direct, natural and elegant way to express and statically enforce fine grained security policies on programs. Namely, programs that manipulate structured data types in which the security level of a structure field may depend on values dynamically stored in other fields, still considered a challenge to security enforcement in software systems such as data-centric web-based applications.

In standard information flow type systems [1, 17, 26, 28, 64], a type has the form $\tau^{\mathtt{s}}$, where the structural type τ is tagged with a security label $\mathtt{s}$, an element of a security lattice

modelling an hierarchy of security compartments or levels. For example, one defines $(\texttt{int}^{\top} \to \texttt{int}^{\top})^{\bot}$ as the type of a low security ($\bot$) function that maps a high security ($\top$) integer to a high security integer.

However, as already suggested, it is often the case that the security level of data values depends on the manipulated data itself; such dependencies are obviously not expressible by such basic security labelling approaches.

The key idea behind dependent information flow types is fairly simple. We propose to extend dependent types in such a way that not only the (structural) type assigned to a computation may depend on values but also its security level, expressed by associating to a data type a value dependent security label (cf. [35]), instead of a plain security label, as described above.

In order to achieve this goal, we introduce a theory of dependent information flow types within the framework of dependent type theory, introducing sum and function dependent types, capturing the essence of value dependent security classification.

Next we will present value dependent security labels and dependent sum and product types, crucial to develop our dependent information flow types.

1.6.1 Value-Indexed Security Labels

Value-indexed security labels may partition standard security levels by indexing labels ℓ with values v, so that each partition $\ell(v)$ classify data at a specific level, depending on the value v.

For example, we can partition the security level `U` into n security compartments, each representing a single registered user of the system, so security level `U(01)` represents the security compartment of the registered user with id `01`. Of course, one may also consider indexed labels of arbitrary arity, for instance for security level `A` (author) we can index with both the author's id and submission's id so `A(42,70)` would stand for the security compartment of data relating to author (with id `42`) and his submission (id `70`).

1.6.2 Dependent Sum and Product Types

A simple example of a dependent (function) information flow type is

$$\Pi x{:}\texttt{string}^{\bot}.\texttt{string}^{\mathsf{usr}(x)}$$

One could assign such a type to the function `get_passwd` that given a user name (a string) returns its password (a string). Although the security level of user "pat" is public ($\bot$), pat's password itself belongs to the security level `usr`("pat"), where `usr(`x`)` is a value dependent security label.

For another simple example, consider the dependent (labelled product) information flow type:

$$\Sigma[\texttt{uid:string}^{\bot} \times \texttt{passwd:string}^{\mathsf{usr}(\mathsf{uid})}]$$

This would type records in which the security level of `passwd` field depends on the actual value assigned to the `uid` field.

Value dependent security labels, such as `usr(`x`)`, denote concrete security levels in the given security lattice, along standard lines, but allow security levels to be indexed by program values, useful to express security constraints to depend on dynamically determined data values.

In such a setting, we would expect the security levels `usr`("joe") and `usr`("pat") to be incomparable, thus avoiding insecure information flows between the associated security compartments, representing the private knowledge of users joe and pat respectively. In particular the security level of the password returned by the call `get_passwd`("joe") is `usr`("joe") rather than, say, just `usr`, which in our setting could be denoted by the label `usr(`$\top$`)`, representing the security level of the information available from any user.

Thus dependent types together with value indexed security labels allows secure computations to be expressed with extra precision.

Other key feature of our type system is the way it allows us to capture general data-dependent security constraints within data structures containing elements classified at different security levels, as necessary to represent, e.g., realistic rich security policies on structured documents or databases.

Typically, it is required to flexibly inspect, select, and compose such structure elements during computations, while enforcing all the intended information flow policies. For example, consider a (global) password file `users` modelled by a collection (e.g. a list) of records of dependent product type, the type assigned to such a collection would be:

$$\texttt{users:}\Sigma\texttt{[uid:string}^{\perp}\times\texttt{passwd:string}^{\mathsf{usr(uid)}}\texttt{]}^{*\perp}$$

(notice that s^* is the type of collections (lists) of values of type s).
Then, consider the following function

```
let getPasswords = λ(u).
   foreach (x in users) with acum = {} do
     if x.uid = u then x.pwd :: acum else acum
```

The function `getPasswords` extracts from the global data structure `users` the collection of passwords associated to a user id. Notice that although the collection `users` contains passwords classified in different security levels, the security level of the collection returned by the function is always `usr`(u), with u the user id string passed as argument. Then, the following typing holds `getPasswords:` $\Pi u : \texttt{string}^{\perp}.\texttt{string}^{*\mathsf{usr}(u)}$.

We base our development on a minimal λ-calculus with records, (general) imperative references, and collections. Although extremely parsimonious, we show that our programming language and its dependent information flow type system is already quite expressive, allowing practically relevant scenarios to be modelled and analysed against natural value dependent information flow policies.

1.6.3 Toy Example: A Conference Manager System

Let us now overview our approach by going back to our conference manager example.

We assume the following security levels:

- `U(uid)`, representing registered users with id `uid`;
- `A(uid,sid)`, stating author of submission with id `sid` and whose user id is `uid`; and
- `PC(uid,sid)`, that stands for PC members assigned to review submission with id `sid` and whose user id is `uid`.

In general, the security lattice is required to enforce $\ell(\overline{v}, u, \overline{w}) \leq \ell(v, \top, w)$ and $\ell(\overline{v}, \bot, \overline{w}) \leq \ell(\overline{v}, u, \overline{w})$. So, for instance, for all `uid` we have $\texttt{U(}\bot\texttt{)} \leq \texttt{U(uid)} \leq \texttt{U(}\top\texttt{)}$.

We can see $\texttt{U(}\top\texttt{)}$ as the approximation (by above) of any `U(uid)`, e.g, standing for the standard label `U`. Additionally, we give some of the interpretations we give to security labels indexed by $\top$ or $\bot$:

- $\texttt{A(}\bot\texttt{,}\bot\texttt{)}$, stands for the security compartment accessible to any author;
- $\texttt{A(}\bot\texttt{,}\bot\texttt{)}$, denotes the security compartment accessible to any PC member;
- $\texttt{A(}\top\texttt{,}\top\texttt{)}$, represent the compartment containing the information of all authors;
- $\texttt{PC(}\top\texttt{,}\top\texttt{)}$, stands for the compartment with the information of all PC members;
- $\texttt{A(uid,}\top\texttt{)}$, stands for registered users with id `uid` that are authors;
- $\texttt{A(}\top\texttt{,sid)}$, is the security compartment of authors of submission with id `sid`.
- $\texttt{A(uid,}\bot\texttt{)}$, means a registered author with no authority over submitted papers;

For our running example and in order to define Policy 2, we declare the following collections with the according types:

$\tau_c \stackrel{\text{def}}{=} \Sigma[\texttt{uid} : \bot \times \texttt{name} : \texttt{U(uid)} \times \texttt{univ} : \texttt{U(uid)} \times \texttt{email} : \texttt{U(uid)}]$
$\sigma_c \stackrel{\text{def}}{=} \Sigma[\texttt{uid:}\bot \times \texttt{sid:}\bot \times \texttt{title:A(uid,sid)} \times \texttt{abs:A(uid,sid)} \times \texttt{paper:A(uid,sid)}]$
$\delta_c \stackrel{\text{def}}{=} \Sigma[\texttt{uid:}\bot \times \texttt{sid:}\bot \times \texttt{PC\_only:PC(uid,sid)} \times \texttt{review:A(}\top\texttt{,sid)} \times \texttt{grade:A(}\top\texttt{,sid)}]$

$\textbf{let}\ \texttt{Users} = \textbf{ref}_{\texttt{ref}(\tau_c)^{*\bot}}\ (\textbf{ref}_{\tau_c^{\bot}}\ [\,]\ )::\{\}\ \textbf{in}$
$\textbf{let}\ \texttt{Submissions} = \textbf{ref}_{\texttt{ref}(\sigma_c)^{*\bot}}\ (\textbf{ref}_{\sigma_c^{\bot}}\ [\,]\ )::\{\}\ \textbf{in}$
$\textbf{let}\ \texttt{Reviews} = \textbf{ref}_{\texttt{ref}(\delta_c)^{*\bot}}\ (\textbf{ref}_{\delta_c^{\bot}}\ [\,]\ )::\{\}$

and the partial order defining the sample security lattice by the following axioms (quantifiers ranging over natural numbers):

$$\forall \texttt{uid,sid.}\ \texttt{U(uid)} \leq \texttt{A(uid,sid)} \qquad \text{(Axiom 1)}$$

$$\forall \texttt{uid1,uid2,sid.}\ \texttt{A(uid1,sid)} \leq \texttt{PC(uid2,sid)} \qquad \text{(Axiom 2)}$$

Axiom 2 states that information observable by an author of a given submission is also observable to a PC member of said submission, while Axiom 1 denote that data visible to

a registered user is also observable by an author, if the ids match (the id represents the same user).
We will illustrate below with some examples on how these work to disallow insecure programs.

For brevity, as in the example below, when writing new record values based on existing ones, we just mention the fields being assigned a new value, and a sample field indicating the record value from which the other values are to be copied.

For example, in `[uid = t_sub.uid, title = t + "!", ...]` we mean that fields `sid`, `abs`, and `paper` are just copied from `t_sub` like `uid = t_sub.uid`, `sid = t_sub.sid`, etc. Consider then the following code

Example 4

```
let t = first( ( foreach(x in !Submissions) with y={} do
                  let t_sub = !x in
                     if t_sub.uid = 42 and t_sub.sid = 70 then
                        t_sub.title::y else y ) )
in ( foreach(x in !Submissions) with y = {} do
       let t_sub = !x in
        if t_sub.sid = 70 and t_sub.uid = 42 then
          let new_rec = [uid = t_sub.uid, title = t+"!", ...]
          in x:= new_rec )
```

In this example `Submissions` is a (mutable) collection of references of type σ_c. The type σ_c is a dependent sum type where the security level of some fields depends on the actual values bound to other fields (as previously explained).

For example, notice the security level of field `title` is declared as `A(uid,sid)` where `uid` and `sid` are other fields of the (thus dependent) record type. Also, `t` gets security level `A(42,70)` since we are retrieving a record with `uid` value `42` and `sid` value `70`.

To type record `new_rec`, we need to obtain type

$$\Sigma[\texttt{uid:}\bot\times\texttt{sid:}\bot\times\texttt{title:A(uid,sid)}\times\ldots]$$

which in turn needs to check the type of expression `t+"!"` for field `title`.
But since we know `t` has security level `A(42,70)` and that `t_sub.sid=70` and `t_sub.uid=42` (so fields `uid` and `sid` have value `42` and `70`, respectively, in `new_rec`), we can deem secure the assignment `x:= new_rec`.

On the other hand, if we change the last conditional to be **`if`** `t_sub.sid = 24`, then we would be attempting to associate data of security level `A(42,70)`, value `t`, within the security compartment `A(24,`$\top$`)` for author with `uid 24`. So, in other words, data from author `42`'s submission is being associated to submissions of author `24`, inducing an ilegal flow of information.

Let us now discuss the code fragment below

```
foreach (x in !Submissions) with y = {} do
 let t_sub = !x in
   if (t_sub.uid = 42) then
     [uid = t_sub.uid, sid = t_sub.sid, title = t_sub.title]::y
  else y
```

This program evaluates to a collection of records of sum type (resulting from projecting part of submission records of type σ_c). The expected type, given the definition of σ_c, is

$$\Sigma[\texttt{uid}{:}\bot \times \texttt{sid}{:}\bot \times \texttt{title: A(uid, sid)}]$$

However, our type system can track value dependencies and constraints imposed by programs, so a more precise type is assigned in this case, namely

$$\Sigma[\texttt{uid}{:}\bot \times \texttt{sid}{:}\bot \times \texttt{title: A(42,sid)}]$$

Such ability to track dependencies is crucial to rigorously analyse fine grained security policies. For instance, in order to check if PC member with id `10` could observe the result of the above operation, we need to establish that $\texttt{A(42,sid)} \leq \texttt{PC(10,sid)}$, which would not be possible had we approximated the field `sid` with $\top$.

Let us consider the following code for a function `viewUserProfile`

```
let viewUserProfile = λ (u).
  foreach(x in !Users) with y = {} do
    let usr = !x in
      if usr.uid = u then usr::y else y
```

Function `viewUserProfile` returns a collection of records of dependent sum type whose security labels on fields `title`, `abs`, and `paper` depend on the value of the parameter `uid_a`. A precise typing for `viewUserProfile` is

$$\Pi(\texttt{u}{:}\bot).\Sigma[\texttt{uid}{:}\bot \times \texttt{name:U(u)} \times \texttt{univ:U(u)} \times \texttt{email:U(u)}]^{*\bot}$$

Notice that the return type depends on the value of the function argument, so the type of `viewUserProfile` is a functional dependent type. Namely, the type of `viewUserProfile` states the function retrieves the profile of user with id `u`, so, for instance, expression `first(viewUserProfile(42)).email` has type `U(42)`.

Example 5 The `addCommentSubmission` operation is used by the PC members to add comments to other PC members during the evaluation of a given submission.

```
let addCommentSubmission = λ(uid_r, sid_r).
  foreach (p in viewAssignedPapers(uid_r)) with _ do
   if p.sid = sid_r then
    foreach(y in !Reviews) with _ do
     let t_rev = !y in
      if t_rev.sid = p.sid then
       let up_rec = [uid=t_rev.uid,PC_only=comment(p.uid,p.sid,p),...]
       in y := up_rec
```

Function `viewAssignedPapers` has type $(\Pi(\texttt{uid\_r}:\bot).\texttt{C}$, where type `C` is

$$\Sigma[\texttt{uid}:\bot\times\texttt{sid}:\bot\times\texttt{title:A(uid,sid)}\times\texttt{abs:A(uid,sid)}\times\texttt{paper:A(uid,sid)}]^{*\bot}$$

Since there is no dependency (`uid_r` not free in `C`), we may abbreviate the functional type by $\texttt{int}^{\bot}\rightarrow\texttt{C}$, thus identifier `p` has type `C`.

Function *comment* returns a given paper's PC comments, and has type

$$\Pi\texttt{u}:\bot.\Pi\texttt{s}:\bot.\Pi\texttt{r:C.A(u,s)}$$

Notice that its return type in the call *comment*`(p.uid,p.sid,p)` has security label `A(p.uid,p.sid)`. Additionally, we know `t_rev` has type δ_c:

$$\Sigma[\texttt{uid}:\bot\times\texttt{sid}:\bot\times\texttt{PC\_only:PC(uid,sid)}\times\texttt{review:A}(\top,\texttt{sid})\times\texttt{grade:A}(\top,\texttt{sid})]^{\bot}$$

So, in order to type check the assignment expression, `y := up_rec`, we need to check that `up_rec` has the same type as the prescribed type for the collection's elements, type δ_c. Namely, we have to check if *comment*`(p.uid, p.sid,p)` has type `PC(t_rev.uid,p.sid)`.

As we said, the type for *comment*`(p.uid, p.sid,p)` has security label `A(p.uid, p.sid)` but since it has field dependencies, we need to infer values for them. In this case, we cannot infer a value for field `uid` so we approximate to $\top$ obtaining $\texttt{A}(\top,\texttt{p.sid})$.

However, because we know by the conditional that `t_rev.sid = p.sid`, we can index the security level by field `sid` instead, which allows us to type the assignment operation since field `sid` is bounded by the dependent sum type of the record being used for the assignment.

Then we can type *comment*`(p.uid, p.sid,p)` with type $\texttt{A}(\top,\texttt{p.sid})$ and thus, due to $\texttt{A}(\top,\texttt{p.sid}) \leq \texttt{PC}(\bot,\texttt{p.sid})$ (Axiom 2), we can up-classify *comment*`(p.uid, p.sid,p)` with `PC(t_rev.uid, p.sid)`.

Meaning we can, finally, type the record `up_rec` with the dependent sum type

$$\Sigma[\texttt{uid}:\bot\times\texttt{sid}:\bot\times\texttt{PC\_only:PC(uid,sid)}\times\texttt{review:A}(\top,\texttt{sid})\times\texttt{grade:A}(\top,\texttt{sid})]$$

We refer back to this example in Chapter 3 - Example 22, where we detail the relevant steps taken by the system to typecheck the program.

Without dependent types, we would lose precision in the typing of *comment*`(p.uid, p.sid,p)` (obtaining $\texttt{A}(\top,\top)$ instead) and not be able to raise the security level to the required level, thus `addCommentSubmission` would not type check despite being secure.

Although approaching a substantial level of simplicity, our type system tackles relevant technical challenges, necessary to handle heterogeneously classified data structures and security level dependency.

As in classical approaches (e.g., [1, 28]), both a type τ and a security label s are assigned to expressions by our typing judgment $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, reflecting the fact that the value of e does not depend on data classified with security levels above s or incomparable with s, where s is in general a value dependent label.

The analysis of implicit flows is also particularly interesting in our setting, even if we adopt standard techniques to track the computational context security level (the "program counter") r. The additional component $\mathcal{S}$ represents a set of the equational constraints, used to refine label indices, and establish type equality.

We follow up with the contributions of this thesis in the next section.

1.7 Contributions

The contribution of this thesis work is the detailed study of the notion of value-dependency on security labels within the framework of a general type theory, particularly suited to express so-called "row-level" security policies.

A concept of indexed security label was introduced in [35], as an useful yet isolated feature to express security policies in a domain specific language with high-level monolithic data manipulation operations, much less general and expressive than what we achieve in this thesis work.

The developments in this thesis put forward, in a principled way and for the first time, the notion of data/state dependent information flow in terms of a fairly canonical dependent type theory with first-order sum and arrow types, defined by a set of simple type rules, and for a parsimonious λ-calculus with references and collections.

We now outline the contributions of this thesis in more detail:

- **Notion of Value-indexed Security Labels**. We introduce the notion of value-*indexed* security label [35]. Value-indexed security labels may partition standard security levels by indexing labels ℓ with values v, so that each partition $\ell(v)$ classify data at a specific level, depending on the value v.

  For example, we can partition the security level `U` into n security compartments, each representing a single system's registered user, so security level `U(01)` stands for the security compartment of registered user with id `01`.

  We show value-indexed security labels, together with a dependent type theory, are useful to define "row-level" security policies, like the one shown in Policy 2 in Section 1.5, where logically different security compartments, that may be statically

mapped into different physical compartments, are dynamic and dependent on runtime data.

- **Dependent Information Flow Types**. In this thesis, we develop the notion of *dependent information flow types* [36].We believe dependent information flow types provide a direct, natural and elegant way to express and statically enforce fine grained security policies on programs (such as Policy 2 in Section 1.5), including programs that manipulate structured data types in which the security level of a structure field may depend on values dynamically stored in other fields.

  With our analysis, we are able to reason about data confidentiality of data-centric applications. We build our analysis on top of an expressive λ-calculus with records, collections, and references.

  We also show the core language is expressive enough to encode Data Manipulation Language (DML) primitives. This means our dependent information flow type system can also reason about data confidentiality in typical DML applications.

- **Noninterference for Dependent Information Flow Types**. We prove that well-typed programs in our dependent information flow types comply with a termination-insensitive noninterference theorem, thus ensuring data confidentiality. To do so, we present notions of store and expression equivalence up to a given security level.

- **A Typechecker Prototype Implementation**. Finally, we address the formalisation of a typecheck algorithm, and implemented a *proof-of-concept* prototype typechecker using our dependent information flow types.

The main contributions of this thesis are published in the following:

- [35] L. Lourenço and L. Caires. Information Flow Analysis for Valued-Indexed Data Security Compartments. In Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers. Ed. by M. Abadi and A. Lluch-Lafuente. Springer, 2013, pages 180–198.

- [36] L. Lourenço and L. Caires. Dependent Information Flow Types. In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '15. Mumbai, India ACM, 2015, pages 317–328.

- DIFT Typechecker Prototype website, `http://ctp.di.fct.unl.pt/DIFTprototype/`, and its live version in Microsoft's rise4fun `http://rise4fun.com/DIFT/`.

The structure of this thesis for the remaining chapters are as follows:

- Chapter 2 introduces the core language used in this work. We achieve this by first introducing an untyped version of the core language in order to present its syntax and semantics. Then, we proceed to its typed version and present a type system for

information flow analysis. We conclude with a toy example to further illustrate the core language as well as the limitations of standard type-based information flow analysis.

- In Chapter 3 we formalise our dependent information flow types. We begin introducing the notion of value-dependent security labels and then extend the type system presented in Chapter 2 with dependent function and sum types. In this chapter, we discuss the challenges value-dependency on labels imposes in our analysis and how we tackle them. Then we present our type system and illustrate our analysis and type system via examples and typing derivations, respectively. Next, we show our type safety results: well-typed programs preserve their typing under their evaluation and never get "'stuck". Finally, we conclude with an overview of the relevant related work.

- The formulation of the soundness result is presented in Chapter 4. We also introduce notions of store equivalence and expression equivalence up to a security level to achieve the formulation of noninterference. We outline the proof of noninterference and conclude with a example to illustrate how one can interpret noninterference result.

- In Chapter 5 we discuss some of the applications of this thesis work. Namely, we show how we can encode a typical data-centric application, via a toy example, and reason about the confidentiality of its data. We show our core language is capable of encoding Data Manipulation Language (DML) primitives and, thus, our analysis can be applied to DML applications. To achieve the latter claim, we present typing derivations of the encodings that matches the expected typing rules for DML primitives (in accordance to those introduced in [35]).

- The focus in Chapter 6 is on a typechecking algorithm. We discuss the algorithm and discuss its implementation into a *proof-of-concept* prototype. Namely, we give the intuition behind our constraint solving procedure (which relies on Z3 SMT solver) and discuss the algorithm's efficiency. To illustrate the syntax and the output of the typechecker prototype, we give some examples (including those presented in Chapter 1). We conclude with the discussion of some of the open problems of the current implementation of our prototype typechecker, and make a comparison to implementations of other type-based information flow analysis works.

- Finally, in Chapter 7 we give some concluding remarks, and outline possible future directions for this work.

With the aim to gradually introduce the concepts to the reader, in this chapter we outlined the relevant work in the area of language-based security, and more concretely type-based information flow analysis. We opt to discuss the related work throughout this document.

CHAPTER 2

REASONING WITH A TYPE-BASED INFORMATION FLOW ANALYSIS

In this chapter we introduce the core language used to develop this thesis work, which we already illustrated in Chapter 1 via examples. Namely, we define an imperative λ-calculus with collections, records and variants. We begin by introducing the language's syntax, along with auxiliary abbreviations that help the presentation, then we define the operational semantics via a small-step operational semantics. We then proceed with a typed version of the core language to illustrate a "typical" type-based information flow analysis. Finally, we will illustrate our language using our conference manager system, introduced in Chapter 1.

2.1 λ_{RCV}: An Imperative λ-calculus with Records and Collections

In this section we present an imperative λ-calculus with records, collections and variants, which serves as the underlying core language for our analysis in subsequent chapters. We start by introducing some syntactic conventions and abbreviations to be used to simplify the presentation and examples.

Basic Notation

Let $\mathcal{X}$ be a infinite set of variables such that $x, y, z, \ldots \in \mathcal{X}$, $\mathcal{M}$ be a infinite set of identifiers such that $m, n, \ldots \in \mathcal{M}$, and $\texttt{Loc}$ be a infinite set of memory locations such that $l, l', \ldots \in \texttt{Loc}$.

Syntactic Conventions:

$e ::=$		(*expression*)
	$\mid \lambda x.e$	(abstraction)
	$\mid e_1(e_2)$	(application)
	$\mid x$	(variable)
	$\mid [\overline{m = e}]$	(record)
	$\mid e.m$	(field access)
	$\mid \{\bar{e}\}$	(collection)
	$\mid e_1 :: e_2$	(cons)
	$\mid$ **foreach**$(e_1,\ e_2,\ x.y.e_3)$	(iteration)
	$\mid \#m(e)$	(variant)
	$\mid$ **case** $e\ (\overline{m \cdot x \Rightarrow e})$	(case)
	$\mid$ **let** $x = e_1$ **in** e_2	(let)
	$\mid$ **if** c **then** e_1 **else** e_2	(conditional)
	$\mid$ **ref** e	(reference)
	$\mid e_1 := e_2$	(assign)
	$\mid\ !e$	(deref)
	$\mid v$	(value)

(a) Expressions

$c ::=$		(*conditions*)
	$\mid \neg c$	(negation)
	$\mid c_1 \vee c_2$	(disjunction)
	$\mid V = V$	(equality)
	$\mid V$	(term)

(b) Logical Expressions

Figure 2.1: Abstract Syntax (Part 1)

foreach$(e_1, e_2, x.y.e_3)$ is to be read as **foreach** $(x$ **in** $e_1)$ **with** y **=** e_2 **do** e_3
$\lambda(x, \ldots, z).e$ is to be read as $\lambda x.(\ldots).\lambda z.e$

Abbreviations:
$[\overline{m = e}]$ stands for $[m_1 = e_1, \ldots, m_n : e_n]$
$\{\bar{e}\}$ stands for $\{e_1, \ldots, e_n\}$
case $e(\overline{m \cdot x \Rightarrow e})$ stands for **case** $e(m_1 \cdot x_1 \Rightarrow e_1, \ldots, m_n \cdot x_n \Rightarrow e_n)$

Syntax

The syntax of our core language is given by the grammar in Figure 3.2a. Being an extension of the $\lambda-$calculus, we naturally have the abstraction $\lambda x.e$, where x is bound in expression e; application $e_1(e_2)$; and variables x as expressions.

Additionally, we have record expressions $[\overline{m = e}]$, that associates field identifiers m to expressions e, and field selection $e.m$ to project the value associated to the field identifier.

Our core language also includes collections $\{\bar{e}\}$ and some operations over collections, such as: cons operator $e_1 :: e_2$, to add an element to the beginning of a collection, and a collection iterator **foreach**$(e_1,\ e_2,\ x.y.e_3)$, to iterate and compute over the elements of a collection.

More concretely, the **foreach** iterator is a familiar functional collection fold combinator [7], where x is the current item of collection denoted by e_1, y denotes the value accumulated from previous iteration (with initial value e_2) and e_3 is the expression to be evaluated at each iteration. Notice that x, y are bound in e_3.

Let us see an example to illustrate this primitive.

Example 6 Suppose we have a collection of integer and we want the sum of all its elements. We can code this operation as follows

```
foreach (x in {1,2,3,4,5})
  with sum = 0 do
     x + sum
```

The result of this operation would be value `15`.

We also have variant expressions, $\#m(e)$, and a case primitive, **case** e $(\overline{m \cdot x \Rightarrow e})$, to case-analyse variant expressions. So e denotes the variant value to be analysed, m_i the possible identifiers of the variant value, e_i the corresponding expression in case of a match and x_i the variable denoting the value in e for the matched identifier. Each x_i is bounded in their corresponding e_i. We now illustrate these primitives with a common use of variants.

Example 7 Suppose we declare a function to compute the division between two integers. We must then test whether the denominator is zero and give the appropriate output accordingly.

Since our language does not have exceptions, one way to handle division by zero is to return a variant value representing that no result was computed, `#None`. Otherwise we compute the result of the division, say v, and wrap it in a variant value, `Some(`v`)`.

```
let division =
  λ (x,y).
     if y == 0 then
       #None(skip)
     else #Some(x/y)
in let result = division(12,2)
   in case result( None · x ⇒ ''Err: Division by Zero'',
                   Some · y ⇒ y)
```

Then, for this snippet, we would obtain the variant value `#Some(6)` upon the application `division(12,2)`. So when evaluating the case primitive, we can match `result` with identifier `Some` and replace variable y with the value `6`, thus obtaining the integer `6` for this example.

As also illustrated above, we also have let-expressions, **let** x = e_1 **in** e_2, and conditionals, **if** c **then** e_1 **else** e_2. As expected, variable x is bounded in e_2 by the value denoted by e_1 in a let-expression. We restrict condition, c, in a conditional expression to be pure-expressions given by the grammar in Figure 2.1b. Logical expressions c use terms, syntactic category V defined in Figure 2.2a, to check equality of values. Essentially terms

$V ::=$		(*terms*)
	$\lambda x.e$	(abstraction)
	$[\overline{m = V}]$	(record)
	$\overline{V}$	(collection)
	$\#m(V)$	(variant)
	`true`	(true)
	`false`	(false)
	$()$	(unit)
	x	(identifier)
	$V.m$	(field access)

(a) Terms

$v ::=$		(*values*)
	$\lambda x.e$	(abstraction)
	$[\overline{m = v}]$	(record)
	$\overline{v}$	(collection)
	$\#m(v)$	(variant)
	`true`	(true)
	`false`	(false)
	$()$	(unit)
	l	(locations)

(b) Values

Figure 2.2: Abstract Syntax (Part 2)

are the values of our language and, additionally, variables and field projection (useful to compare field values of a record value in conditionals).

So pure expressions are those side-effect free, in concrete, all expressions except assignment, reference expressions and deference. However, to simplify our analysis, we also require expressions in fragment c to be logical expressions (disjunction, negation or equality) between terms. This is a necessary restriction since in our analysis conditionals are used to gather constraints to be solved by an equational theory, as we will discuss in Chapter 3.

Finally, imperative expressions of our language include creation of a new reference (variable) with initial content denoted by e, **`ref`** e; assignment of a new value to a given reference, $e_1 := e_2$; and dereference operation, `!`e, to obtain the contents of a reference.

In segment v, defined in Figure 3.2b, we represent the possible output of evaluating an expression.

So values of our language include abstractions, $\lambda x.e$; records, $[m_1 = v_1, \ldots, m_n = v_n]$; collections (list of values), $\{v_1, \ldots, v_n\}$; variants, $\#m(v)$; booleans; unit value; and locations l. We consider two values v, u equal, $v = u$, when they are the same up to reordering of record identifiers, and assume intensional equality on lambda abstractions.

We assume other basic data types (such as integers, strings) and corresponding operators, such as: **`first`**$(-)$ to retrieve the first element of a collection, and **`rest`**$(-)$ to retrieve a collection without its first element. As usual, we consider expressions/terms up-to renaming of bound variables (α-equivalence).

We next define the semantics of our language.

Semantics

The semantics of our language is defined with respect to a store representing the current state of the program. Programs in our language are closed expressions (i.e., no free variables). We define in the expected way, but omit technical details, the free variables of an expression, $fv(e)$, and capture avoiding substitution on expressions, $e\{^v/_x\}$. We denote by $[l \mapsto v]$ the substitution that maps location l to value v.

$$\frac{(S; e_1) \longrightarrow (S'; e_1')}{(S; e_1(e_2)) \longrightarrow (S'; e_1'(e_2))}\ (\text{APP-LEFT}) \qquad \frac{(S; e_2) \longrightarrow (S'; e_2')}{(S; (\lambda x.e)(e_2)) \longrightarrow (S', (\lambda x.e)(e_2'))}\ (\text{APP-RIGHT})$$

$$(S; (\lambda x.e)(v)) \longrightarrow (S; e\{^v/_x\})\ (\text{APP})$$

$$\frac{\mathcal{C}[\![c]\!]}{(S; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \longrightarrow (S; e_1)}\ (\text{IF-TRUE}) \qquad \frac{\neg\mathcal{C}[\![c]\!]}{(S; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \longrightarrow (S; e_2)}\ (\text{IF-FALSE})$$

$$\frac{(S; e_1) \longrightarrow (S'; e_1')}{(S; \textbf{let } x = e_1 \textbf{ in } e_2) \longrightarrow (S'; \textbf{let } x = e_1' \textbf{ in } e_2)}\ (\text{LET-LEFT}) \qquad (S; \textbf{let } x = v \textbf{ in } e_2) \longrightarrow (S; e_2\{^v/_x\})\ (\text{LET-RIGHT})$$

Figure 2.3: Operational Semantics for Expressions (Part 1)

We now give notions of store and store operations.

Definition 1 (Store) A Store S is a mapping from *locations* to values. The store that assigns v_i to l_i for $i \in 1, \ldots, n$ is written $\{l_1 = v_1, \ldots, l_n = v_n\}$, and the empty store is written as $\emptyset$.

As for notations for store operations, we write $S(l)$ to denote the value associated with location l in S, $S[l \mapsto v]$ to denote a store S updated with a new value for location l, and $dom(S)$ to denote the domain set of S.

Before presenting our operational semantics, we provide an auxiliary definition for the evaluation of (store independent) logical expressions.

Definition 2 (Logical Expressions Semantics) The value of a closed logical expression c is given by the interpretation map $\mathcal{C} : c \to \{\textbf{true}, \textbf{false}\}$, as well as the auxiliary interpretation function for closed terms $\mathcal{T} : V \to v$ as follows:

$\mathcal{C}[\![\neg c]\!] = \neg\mathcal{C}[\![c]\!]$
$\mathcal{C}[\![c_1 \vee c_2]\!] = \mathcal{C}[\![c_1]\!] \vee \mathcal{C}[\![c_2]\!]$
$\mathcal{C}[\![V_1 = V_2]\!] = (\mathcal{T}[\![V_1]\!] = \mathcal{T}[\![V_2]\!])$
$\mathcal{T}[\![\textbf{true}]\!] = \textbf{true}$
$\mathcal{T}[\![\textbf{false}]\!] = \textbf{false}$
$\mathcal{T}[\![()]\!] = ()$
$\mathcal{T}[\![\{V_1, \ldots, V_n\}]\!] = \{\ \mathcal{T}[\![V_1]\!], \ldots, \mathcal{T}[\![V_n]\!]\ \}$
$\mathcal{T}[\![\lambda(x : \tau_1^{s_1}).e]\!] = \lambda(x : \tau_1^{s_1}).e$
$\mathcal{T}[\![[m_1{:}V_1, \ldots, m_n{:}V_n]]\!] = [m_1{:}\mathcal{T}[\![V_1]\!], \ldots, m_n{:}\mathcal{T}[\![V_n]\!]]$
$\mathcal{T}[\![V.m]\!] = field(\mathcal{T}[\![V]\!], m)$ with $field([\ldots, m : v, \ldots], m) = v$
$\mathcal{T}[\![\#m(V)]\!] = \#m(\mathcal{T}[\![V]\!])$

We now define our language's semantics by means of a small-step operational semantics, using a call-by-value evaluation strategy. The operational semantics is based on a reduction step, defined between configurations of the form $(S; e)$, where S is a store and e an expression, and is denoted as follows

$$(S; e) \longrightarrow (S'; e')$$

A reduction step states that expression e under store S evolves in one computational step to expression e' under store S'.
We now define our reduction relation on configurations.

Definition 3 (Reduction) Reduction, denoted as $(S; e) \longrightarrow (S'; e')$, is inductively defined by the rules in Figure 2.3, Figure 2.4, Figure 2.5, and Figure 2.6.

In Figure 2.3 we have the set of rules expected for a call-by-value λ-calculus with let-declarations and conditionals. Let us see in more detail

Rule (APP-LEFT) evaluates the left expression on an application until it reduces to a (abstraction) value

$$\frac{(S;\ e_1) \longrightarrow (S';\ e_1')}{(S;\ e_1(e_2)) \longrightarrow (S';\ e_1'(e_2))}$$

Then rule (APP-RIGHT) takes reduction steps on the right expression of an application

$$\frac{(S;\ e_2) \longrightarrow (S';\ e_2')}{(S;\ (\lambda x.e)(e_2)) \longrightarrow (S',\ (\lambda x.e)(e_2'))}$$

And, lastly, rule (APP) makes a β-reduction.

$$(S;\ (\lambda x.e)(v)) \longrightarrow (S;\ e\{^{v}/_{x}\})$$

Regarding the evaluation of let-declarations, rule (LET-LEFT) reduces the first expression of a let-declaration until it is a value

$$\frac{(S;\ e_1) \longrightarrow (S';\ e_1')}{(S;\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2) \longrightarrow (S';\ \textbf{let}\ x = e_1'\ \textbf{in}\ e_2)}$$

while (LET-RIGHT) applies a β-reduction on the second expression with the obtained value.

$$(S;\ \textbf{let}\ x = v\ \textbf{in}\ e_2) \longrightarrow (S;\ e_2\{^{v}/_{x}\})$$

The semantics of a conditional rely on the the logical expressions semantics, Definition 2, in order to reduce the logical expressions.

$$\text{(COLLECTION)}\quad \frac{(S;e) \longrightarrow (S';e')}{(S;\{\ldots e \ldots\}) \longrightarrow (S';\{\ldots e' \ldots\})}$$

$$\text{(CONS-LEFT)}\quad \frac{(S;e_1) \longrightarrow (S';e_1')}{(S;e_1::e_2) \longrightarrow (S';e_1'::e_2)} \qquad \text{(CONS-RIGHT)}\quad \frac{(S;e_2) \longrightarrow (S';e_2')}{(S;v::e_2) \longrightarrow (S';v::e_2')}$$

$$\text{(CONS)}\quad (S;v::\{v_1,\ldots,v_n\}) \longrightarrow (S;\{v,v_1,\ldots,v_n\})$$

$$\text{(FOREACH-LEFT)}\quad \frac{(S;\ e_1) \longrightarrow (S';\ e_1')}{(S;\ \textbf{foreach}(e_1,\ e_2,\ x.y.e_3) \longrightarrow (S';\ \textbf{foreach}(e_1',\ e_2,\ x.y.e_3))}$$

$$\text{(FOREACH-RIGHT)}\quad \frac{(S;\ e_2) \longrightarrow (S';\ e_2')}{(S;\ \textbf{foreach}(v,\ e_2,\ x.y.e_3) \longrightarrow (S';\ \textbf{foreach}(v,\ e_2',\ x.y.e_3))}$$

$$\text{(FOREACH)}\quad \frac{v_l = h::hs}{(S;\ \textbf{foreach}(v_l,\ v,\ x.y.e_3) \longrightarrow (S;\ \textbf{foreach}(hs,\ e_3\{^h/_x\}\{^v/_y\},\ x.y.e_3))}$$

$$\text{(FOREACH-BASE)}\quad (S;\textbf{foreach}(\{\},\ v,\ x.y.e_3) \longrightarrow (S;v)$$

Figure 2.4: Operational Semantics for Expressions (Part 2)

So rule (IF-TRUE) evaluates a conditional expression to the second (then branch) expression if the first (logical) expression reduces to **true**, $\mathcal{C}[\![c]\!]$

$$\frac{\mathcal{C}[\![c]\!]}{(S;\ \textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \longrightarrow (S;\ e_1)}$$

Dually, rule (IF-FALSE) evaluates a conditional expression to the third (else branch) expression if the first (logical) expression reduces to **false**, $\neg\mathcal{C}[\![c]\!]$

$$\frac{\neg\mathcal{C}[\![c]\!]}{(S;\ \textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \longrightarrow (S;\ e_2)}$$

We now discuss the evaluation of collections and its operations, defined in Figure 2.4. As expected, rule (COLLECTION) evaluates a collection expression to a collection value by reducing each element's expression to a value.

$$\frac{(S;\ e) \longrightarrow (S';\ e')}{(S;\ \{\ldots e \ldots\}) \longrightarrow (S';\ \{\ldots e' \ldots\})}$$

The evaluation of the cons primitive also follows as one would expect. Rule (CONS-LEFT) reduces the left-side expression of a cons expression until a value is obtained

$$\frac{(S;\ e_1) \longrightarrow (S';\ e_1')}{(S;\ e_1{::}e_2) \longrightarrow (S';\ e_1'{::}e_2)}$$

Once the left-side of a cons expression is a value, then rule (CONS-RIGHT) can be applied to reduce the right-side expression until a collection value is obtained.

$$\frac{(S;\ e_2) \longrightarrow (S';\ e_2')}{(S;\ v{::}e_2) \longrightarrow (S';\ v{::}e_2')}$$

Finally, rule (CONS) evaluates the cons expression to a collection value that includes the left-side value as the head of the final collection value.

$$(S;\ v{::}\{v_1, \ldots, v_n\}) \longrightarrow (S;\ \{v, v_1, \ldots, v_n\})$$

Less standard rules are those regarding the evaluation of the **`foreach`** primitive. Rule (FOREACH-LEFT) evaluates the first expression until we get a collection value

$$\frac{(S;\ e_1) \longrightarrow (S';\ e_1')}{(S;\ \textbf{foreach}(e_1,\ e_2,\ x.y.e_3) \longrightarrow (S';\ \textbf{foreach}(e_1',\ e_2,\ x.y.e_3))}$$

Afterwards, rule (FOREACH-RIGHT) can be applied to reduce the second expression until a value is obtained. This value corresponds to the initial value of the accumulated value of the iteration on each step, denoted by variable y.

$$\frac{(S;\ e_2) \longrightarrow (S';\ e_2')}{(S;\ \textbf{foreach}(v,\ e_2,\ x.y.e_3) \longrightarrow (S';\ \textbf{foreach}(v,\ e_2',\ x.y.e_3))}$$

Then, rule (FOREACH) is applied when we have a value in the first and second expression of the iterator operator. Namely, the first value is a non-empty collection value v_l and the second value is the accumulated value v. This rule represents an iteration step and reduces to another iteration expression.

$$\frac{v_l = h{::}hs}{(S;\ \textbf{foreach}(v_l,\ v,\ x.y.e_3) \longrightarrow (S;\ \textbf{foreach}(hs,\ e_3\{^h/_x\}\{^v/_y\},\ x.y.e_3))}$$

Notice that the third expression remains unchanged since it represents the computation to be done in each step. This new iteration expression will have as first expression the tail of the collection value v_l. As second expression it will have the resulting expression of the substitution of all free occurrences of x and y, in the third expression, by the head of the collection value and the accumulated value, respectively.

Finally, rule (FOREACH-BASE) is the base case of the iteration operator, stating that if we iterate an empty collection value then it reduces to the accumulated value (second expression).

$$(S; \textbf{foreach}(\{\},\ v,\ x.y.e_3) \longrightarrow (S; v)$$

We revisit example Example 6 to illustrate the semantics of the **`foreach`** primitive.

Example 8 Let us then recall the code snippet:

```
foreach (x in {1,2,3,4,5})
  with sum = 0 do
     x + sum
```

To evaluate this code, we have the following reduction steps (using the abstract syntax):

- `(`S`; foreach({1,2,3,4,5}, 0,` x`.sum.(`x `+ sum)))`
- `(`S`; foreach({2,3,4,5}, 1,` x`.sum.(`x `+ sum)))` by (FOREACH)
  as result of `(`x `+ sum)`$\{^1/_x\}\{^{sum}/_0\}$
- `(`S`; foreach({3,4,5}, 3,` x`.sum.(`x `+ sum)))` by (FOREACH)
  as result of `(`x `+ sum)`$\{^2/_x\}\{^{sum}/_1\}$
- `(`S`; foreach({4,5}, 6,` x`.sum.(`x `+ sum)))` by (FOREACH)
  as result of `(`x `+ sum)`$\{^3/_x\}\{^{sum}/_3\}$
- `(`S`; foreach({5}, 10,` x`.sum.(`x `+ sum)))` by (FOREACH)
  as result of `(`x `+ sum)`$\{^4/_x\}\{^{sum}/_6\}$
- `(`S`; foreach({ }, 15,` x`.sum.(`x `+ sum)))` by (FOREACH)
  as result of `(`x `+ sum)`$\{^5/_x\}\{^{sum}/_{10}\}$
- `(`S`; 15)` by (FOREACH-BASE)

Let us see how records, variants and its operations are evaluated, their rules can be found in Figure 2.5. Rule (RECORD) reduces a record expression to a record value by taking evaluation steps for each field expression.

$$\frac{(S;\ e) \longrightarrow (S';\ e')}{(S;\ [\ldots m{:}e\ldots]) \longrightarrow (S';\ [\ldots m{:}e'\ldots])}$$

Rule (FIELD-LEFT) evaluates the left-side of a field projection up to a record value

$$\frac{(S;\ e) \longrightarrow (S';\ e')}{(S;\ e.m) \longrightarrow (S';\ e'.m)}$$

$$\frac{(S;e) \longrightarrow (S';e')}{(S;[\ldots m{:}e\ldots]) \longrightarrow (S';[\ldots m{:}e'\ldots])}\ (\textsc{record}) \qquad \frac{(S;e) \longrightarrow (S';e')}{(S;\#m(e)) \longrightarrow (S';\#m(e'))}\ (\textsc{variant})$$

$$\frac{(S;e) \longrightarrow (S';e')}{(S;e.m) \longrightarrow (S';e'.m)}\ (\textsc{field-left}) \qquad (S[\ldots m{:}v\ldots].m) \longrightarrow (S;v)\ (\textsc{field-right})$$

$$\frac{(S;e) \longrightarrow (S';e')}{(S;\textbf{case}\ e(\ldots,m_i\cdot x_i \Rightarrow e_i,\ldots)) \longrightarrow (S';\textbf{case}\ e'(\ldots,m_i\cdot x_i \Rightarrow e_i,\ldots))}\ (\textsc{case-left})$$

$$(S;\textbf{case}\ \#m_i(v)(\ldots,m_i\cdot x_i \Rightarrow e_i,\ldots)) \longrightarrow (S;e_i\{^v/_{x_i}\})\ (\textsc{case-right})$$

Figure 2.5: Operational Semantics for Expressions (Part 3)

While rule (FIELD-RIGHT) retrieves the projected field's value.

$$(S;\ [\ldots m{:}v\ldots].m) \longrightarrow (S;\ v)$$

Regarding variants, rule (VARIANT) evaluates a variant expression to a variant value.

$$\frac{(S;\ e) \longrightarrow (S';\ e')}{(S;\ \#m(e)) \longrightarrow (S';\ \#m(e'))}$$

Evaluating a case expression follows as expected. Rule (CASE-LEFT) reduces the case expression until we obtain a variant value

$$\frac{(S;\ e) \longrightarrow (S';\ e')}{(S;\ \textbf{case}\ e(\ldots,m_i\cdot x_i \Rightarrow e_i,\ldots)) \longrightarrow (S';\ \textbf{case}\ e'(\ldots,m_i\cdot x_i \Rightarrow e_i,\ldots))}$$

And rule (CASE-RIGHT) makes a β-reduction on the corresponding expression e_i whose identifier m_i matches that of the variant value being case-analysed.

$$(S;\ \textbf{case}\ \#m_i(v)(\ldots,m_i\cdot x_i \Rightarrow e_i,\ldots)) \longrightarrow (S;\ e_i\{^v/_{x_i}\})$$

Let us revisit Example 7 to illustrate the evaluation of a **`case`** primitive.

Example 9 Recall the code snippet that declares a function `division` and how we use a **`case`** operator to analyse the result of an application of `division` function.

```
let division =
  λ (x,y).
     if y == 0 then
```

```
        #None(skip)
      else #Some(x/y)
in let result = division(12,2)
   in case result( None · x ⇒ ''Err: Division by Zero'',
                   Some · y ⇒ y)
```

The evaluation of this code, given a store S, is as follows:

- (S; `let division = λ (x,y).`
  `if y == 0 then`
  `#None(skip)`
  `else #Some(x/y)`
  `in let result = division(12,2)`
  `in case result( None · x ⇒ ''Err: Division by Zero'',`
  `Some · y ⇒ y) )`

- (S; `let result = (λ (x,y).`
  `if y == 0 then`
  `#None(skip)`
  `else #Some(x/y) )(12,2)`
  `in case result( None · x ⇒ ''Err: Division by Zero'',`
  `Some · y ⇒ y) )` by (LET-RIGHT)

  as result of replacing `division` with the λ value.

- (S; `let result = if 2 == 0 then`
  `#None(skip)`
  `else #Some(12/2)`
  `in case result( None · x ⇒ ''Err: Division by Zero'',`
  `Some · y ⇒ y) )` by (APP)

  as result of replacing `x` with `12` and `y` with `2` in the body of the λ value

- (S; `let result = #Some(12/2)`
  `in case result( None · x ⇒ ''Err: Division by Zero'',`
  `Some · y ⇒ y) )` by (IF-FALSE)

  as result of $\mathcal{C}[\![2 == 0]\!] =$ `false`.

- (S; `case #Some(12/2)( None · x ⇒ ''Err: Division by Zero'',`
  `Some · y ⇒ y) )` by (LET-RIGHT)

  as result of replacing `result` with the variant expression `#Some(12/2)`

- (S; `case #Some(6)( None · x ⇒ ''Err: Division by Zero'',`
  `Some · y ⇒ y) )` by (CASE-LEFT)

  as result of evaluating expression `12/2` to `6` inside the variant expression.

$$\frac{(S;e) \longrightarrow (S';e')}{(S;!e) \longrightarrow (S';!e')}\ (\textsc{deref-left}) \qquad \frac{S(l) = v}{(S;!l) \longrightarrow (S;v)}\ (\textsc{deref})$$

$$\frac{(S;e) \longrightarrow (S';e')}{(S;\mathbf{ref}_{\tau^s} e) \longrightarrow (S';\mathbf{ref}\ _{\tau^s} e')}\ (\textsc{ref-left}) \qquad \frac{l \notin dom(S)}{(S;\mathbf{ref}\ _{\tau^s} v) \longrightarrow (S \cup \{l \mapsto v\};\ l)}\ (\textsc{ref-right})$$

$$\frac{(S;e_1) \longrightarrow (S';e_1')}{(S;e_1 := e_2) \longrightarrow (S';e_1' := e_2)}\ (\textsc{assign-left})$$

$$\frac{(S;e_2) \longrightarrow (S';e_2')}{(S;l := e_2) \longrightarrow (S';l := e_2')}\ (\textsc{assign-right}) \qquad \frac{l \in dom(S)}{(S;l := v) \longrightarrow (S[l \mapsto v];())}\ (\textsc{assign})$$

Figure 2.6: Operational Semantics for Imperative Primitives

- `(S; 6)` by (CASE-RIGHT)
  as result of $\mathtt{y}\{^{6}/_{y}\}$ for the matched identifier `Some`.

Finally, we present the semantics of the imperative core of our language (Figure 2.6). So rule (REF-LEFT) evaluates the expression in a reference expression up to a value

$$\frac{(S;e) \longrightarrow (S';e')}{(S;\mathbf{ref}\ e) \longrightarrow (S';\mathbf{ref}\ \ e')}$$

And then rule (REF-RIGHT) augments the store with a fresh location (mapped to the evaluated value). That location is the result of evaluating a reference expression.

$$\frac{l \notin dom(S)}{(S;\mathbf{ref}\ \ v) \longrightarrow (S \cup \{l \mapsto v\};\ l)}$$

Rule (DEREF-LEFT) reduces a dereference expression up to a location value

$$\frac{(S;e) \longrightarrow (S';e')}{(S;!e) \longrightarrow (S';!e')}$$

While rule (DEREF) retrieves from the store the associated value given the location.

$$\frac{S(l) = v}{(S;!l) \longrightarrow (S;v)}$$

Lastly, assignment expressions are evaluated from left-to-right until we obtain a location value on the left-side expression, rule (ASSIGN-LEFT),

$$\frac{(S;e_1) \longrightarrow (S';e_1')}{(S;e_1 := e_2) \longrightarrow (S';e_1' := e_2)}$$

and a value on the right-side expression, rule (ASSIGN-RIGHT).

$$\frac{(S; e_2) \longrightarrow (S'; e_2')}{(S; l := e_2) \longrightarrow (S'; l := e_2')}$$

Then, rule (ASSIGN) updates the store, given that the location is valid, and evaluates the assignment expression to unit value.

$$\frac{l \in dom(S)}{(S; l := v) \longrightarrow (S[l \mapsto v]; ())}$$

Next we will show a type version of our core language to illustrate a type-based information flow analysis.

2.2 Type-based Information Flow Analysis on λ_{RCV}

In this section we introduce a typed version of our core language, λ_{RCV}, and a type system to ensure type safety of the language as well as noninterference. The former means that "well-typed programs do not go wrong", while the latter ensures data confidentiality of well-typed programs with respect to the prescribed security policy.

The goal of this section is to explain some basic techniques related to type-based information flow analysis as well as establish some limitations of these analyses using standard security labels.

We begin by introducing abbreviations to be used for ease of presentation, and some basic notions related to information flow analysis.

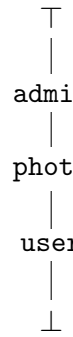
Abbreviations:
$\{\overline{m : \tau^{\ell}}\}^{\ell}$ stands for $\{m_1 : {\tau_1}^{\ell_1}, \ldots, m_n : {\tau_n}^{\ell_n}\}^{\ell}$
$[\overline{m : \tau^{\ell}}]^{\ell}$ stands for $[m_1 : {\tau_1}^{\ell_1}, \ldots, m_n : {\tau_n}^{\ell_n}]^{\ell}$

Basic Notions of Information Flow
We assume a multilevel security approach that classifies information into security compartments, according to some given security lattice, and mediates users access to data according to the security clearance they possess.

A *security label* ℓ represents a security compartment in the system and is used to classify its data. Security labels form a pre-order that establishes a *security lattice*, denoted as $\mathcal{L}$.

A security lattice represents the allowed flows of information throughout the execution of a program. For data confidentiality, the allowed flows of information are represented by "upward paths" in the lattice. That is, if $\ell_1 \leq \ell_2$ then information classified by security label ℓ_1 is allowed to flow to containers (variables or output channels) of a higher security label ℓ_2.

For example, suppose we have the following pre-order relation $\bot \leq \texttt{user} \leq \texttt{photo} \leq \texttt{admin} \leq \top$ giving us the following security lattice:



Then data classified with security label `users` can be stored in a container classified with label `admin` but not the other way around. That is, data with security label `admin` cannot be stored in a container classified with label `user`.

Insecure information flows can be classified into explicit flows or implicit flows. An *explicit flow* corresponds to a direct mapping of classified information to a lower classified container (data-flow based), while an *implicit flow* corresponds to public information that depends on classified one (control-flow based).

Classic examples of such flows are the assignment of a low level variable with a high level value, `l := h`, for explicit flow; and a high guarded conditional whose branches are classified as low level, **`if`** `h > 0` **`then`** `l := 1` **`else`** `l := 0`, for implicit flows.

The noninterference property [26] is usually employed to enforce information flow security. Noninterference states that changing sensitive data of a program does not change the perception that an external observer has on the output of a program, which implies that no public data depends on protected data.

For illustration purposes, let us formalise noninterference for a system with only two security levels, $\bot$ and $\top$, such that $\bot \leq \top$.

We begin by defining a store equivalence relation between stores, denoted as $S_1 =_\bot S_2$.

Definition 4 (Store Equivalence) Let S_1 and S_2 be stores that map locations to values of either security level $\bot$ or $\top$. Then we say we say S_1 is equivalent to S_2 if they only differ in stored values of security level $\top$. That is,

$$S_1 =_\bot S_2 \stackrel{\text{def}}{=} S_1 \text{ differs from } S_2 \text{ in stored values classified at } \top$$

We now define the noninterference property.

Definition 5 (Noninterference) Let S_1 and S_2 be two equivalent stores that map locations to values of either security level $\bot$ or $\top$. Then we say we say e is compliant to the noninterference property if given two equivalent stores, it evaluates to the same value and store equivalence is preserved for the resulting stores. That is,

$$S_1 =_\bot S_2 \ \wedge\ (S_1;\ e) \xrightarrow{*} (S_1';\ v_1) \ \wedge\ (S_2;\ e) \xrightarrow{*} (S_2';\ v_2) \implies v_1 = v_2 \ \wedge\ S_1' =_\bot S_2'$$

$$
\begin{array}{lll}
\tau^{\ell}, \sigma^{\ell} ::= & & \textit{(security types)} \\
& \mid \texttt{bool}^{\ell} & \text{(bool type)} \\
& \mid \texttt{cmd}^{\ell} & \text{(command type)} \\
& \mid \texttt{ref}(\tau^{\ell'})^{\ell} & \text{(reference type)} \\
& \mid \tau^{*\ell} & \text{(collection type)} \\
& \mid \{\overline{m : \tau^{\ell}}\}^{\ell} & \text{(variant type)} \\
& \mid (\tau^{\ell_1} \xrightarrow{\mathsf{r}} \sigma^{\ell_2})^{\ell} & \text{(function type)} \\
& \mid [\overline{m : \tau^{\ell}}]^{\ell} & \text{(record type)}
\end{array}
$$

Figure 2.7: Abstract Syntax of Types

So noninterference states that executing a program e under two equivalent stores, $S_1 =_{\perp} S_2$, will output the same result, $v_1 = v_2$, and have the same side-effects on the resulting stores, $S_1' =_{\perp} S_2'$. In other words, noninterference ensures data confidentiality by certifying that a compliant program does not have insecure flows.

Next we discuss some assumptions we make in our type-based information flow analysis.

Assumptions

As in typical type-based information flow analyses, types τ are annotated with a security label ℓ. So, if an expression e is assigned type τ^{ℓ} then the system must ensure that only users with enough permissions to read information at security level ℓ have access to the value computed by e. Otherwise, the result of e is assumed to be opaque and thus cannot be observed by such a user.

As for the attacker model, we assume an attacker can observe information, including stored data, that has security level $\perp$ (public), and may be a user of the system. So interaction with the system is possible using the core language we show here.

This view is extended to any given security level, so that attackers with access to data classified at security level ℓ can only observe information classified up to ℓ.

Types

Let us now introduce our language of types, following standard definition for typed λ-calculus for information flow [51].

Definition 6 (Types) The types $\mathcal{T}_{RCV}$ of λ_{RCV} are defined by the abstract syntax in Figure 2.7.

Types, τ, in λ_{RCV} are annotated with a security label ℓ, also denoted as security types τ^{ℓ}. So (security) types, τ^{ℓ}, can be boolean $\texttt{bool}^{\ell}$, unit $\texttt{cmd}^{\ell}$ (here denoted as command), reference $\texttt{ref}(\tau^{\ell'})^{\ell}$, variant $\{\overline{m : \tau^{\ell}}\}^{\ell}$, record $[\overline{m : \tau^{\ell}}]^{\ell}$, function $(\tau^{\ell_1} \xrightarrow{\mathsf{r}} \sigma^{\ell_2})^{\ell}$, and collection types $\tau^{*^{\ell}}$. In collection type $\tau^{*\ell}$ each collection element has type τ^{ℓ}.

We require the security label of a record type, $[\overline{m : \tau^{\ell}}]^{\ell}$, to always be the greatest lower bound (glb) of its field's security labels in order to prevent implicit flows on writes. For

$e ::=$	(*expression*)
$\mid \lambda(x : \tau^{\ell}).e$	(abstraction)
$\mid e_1(e_2)$	(application)
$\mid x$	(variable)
$\mid [\overline{m = e}]$	(record)
$\mid e.m$	(field access)
$\mid \{\overline{e}\}$	(collection)
$\mid e_1 :: e_2$	(cons)
$\mid$ **foreach**$(e_1,\ e_2,\ x.y.e_3)$	(iteration)
$\mid \#m(e)$ **as** τ^{ℓ}	(variant)
$\mid$ **case** e $(\overline{m \cdot x \Rightarrow e})$	(case)
$\mid$ **let** $x = e_1$ **in** e_2	(let)
$\mid$ **if** c **then** e_1 **else** e_2	(conditional)
$\mid \mathbf{ref}_{\tau^{\ell}}\ e$	(reference)
$\mid e_1 := e_2$	(assign)
$\mid\ !e$	(deref)
$\mid v$	(value)

(a) Expressions

$v ::=$	(*values*)
$\mid \lambda(x : \tau^{\ell}).e$	(abstraction)
$\mid [\overline{m = v}]$	(record)
$\mid \overline{v}$	(collection)
$\mid \#m(v)$ **as** τ^{ℓ}	(variant)
$\mid$ **true**	(true)
$\mid$ **false**	(false)
$\mid ()$	(unit)
$\mid l$	(locations)

(b) Values

Figure 2.8: Abstract Syntax of Typed λ_{RCV}

the same reason, we have the same invariant condition for variant types, $\{\overline{m : \tau^{\ell}}\}^{\ell}$. We will illustrate later on this section the need for a security label on record and variant types.

Also, notice we annotate the function type, $(\tau^{\ell_1} \xrightarrow{\mathsf{r}} \sigma^{\ell_2})^{\ell}$, with security label r, which is a lower bound on the function effects (writes). If omitted, security label r is assumed to be $\bot$. We will give more examples later on this section to illustrate why it is necessary to record the computational context on the function type.

While not formalised, for simplicity, we assume other basic types, such as integers and strings with their associated operations, which are used in examples.

We now use the type language to define an explicitly typed version of the λ_{RCV}, denoted as $\lambda_{\tau RCV}$.

Syntax

The syntax of the typed version of λ_{RCV} is given by Figure 2.8. Notice that the only difference with respect to untyped λ_{RCV} are the type annotations in abstractions, variant expressions and references. Type annotations are needed here to simplify the implementation of the type system defined next.

Operational Semantics

The semantics of $\lambda_{\tau RCV}$ is actually the same as the one presented for λ_{RCV}, the only difference is the abstract syntax. For that reason we omit the rules of the operational semantics and move on to the introduction of the type system.

$$
\begin{array}{lll}
\Delta ::= & & (\textit{typing environment}) \\
& \mid \phi & (\text{empty environment}) \\
& \mid \Delta, x : \tau^{\ell} & (\text{type assignment to a variable}) \\
& \mid \Delta, l : \tau^{\ell} & (\text{type assignment to a location})
\end{array}
$$

Figure 2.9: Abstract Syntax of Typing Environments

$$
\text{(ENV-EMPTY)}\ \frac{}{\phi \vdash \diamond} \qquad \text{(ENV-VAR)}\ \frac{\Delta \vdash \diamond \quad x \notin dom(\Delta) \quad \Delta \vdash \tau^{\ell}}{\Delta, x : \tau^{\mathsf{s}} \vdash \diamond} \qquad \text{(ENV-LOC)}\ \frac{\Delta \vdash \diamond \quad l \notin dom(\Delta) \quad \Delta \vdash \tau^{\ell}}{\Delta, l : \tau^{\ell} \vdash \diamond}
$$

Figure 2.10: Valid Typing Environments

Type System

To type $\lambda_{\tau RCV}$ expressions, we define a typing judgment of the form

$$\Delta \vdash^{\mathsf{r}} e : \tau^{\ell}$$

It asserts expression e has type τ^{ℓ} under typing environment Δ. The label ℓ states that the value of expression e does not depend on data classified with security labels above ℓ or incomparable with ℓ. Label r expresses the security level of the computational context (cf. the "program counter" [42, 52]), which is necessary to prevent implicit flows.

So the goal of the type system is to ensure information only flows upwards the security lattice, e.g., only from a level l to a level h such that $\mathsf{l} \leq \mathsf{h}$.

We now give some definitions before presenting our type system.

Definition 7 (Typing Environment) For $x \in \mathcal{X}$, $l \in \mathtt{Loc}$, and $\tau^{\ell} \in \mathcal{T}_{RCV}$ the set Δ of all typing environments is defined by the abstract syntax in Figure 2.9.

Typing declarations assign types to identifiers $x{:}\tau^{\mathsf{s}}$, and types to locations, $l{:}\tau^{\mathsf{s}}$. A typing environment Δ is a list of typing declarations. We write $dom(\Delta)$ to denote the declared variables and locations in Δ, and define the notion of valid typing environment which, in turn, relies on the notion of well-formed types. We then define how to form valid types as follows:

Definition 8 (Well-formed Types) Well-formed types are denoted by judgment $\Delta \vdash \tau^{\ell}$, stating that type τ^{ℓ} is well-formed under typing context Δ, and is given by the set of rules shown in Figure 2.11.

Note the invariants on record's and variant's types security label are enforced by the definition of well-formed types. So valid typing environments are defined as follows:

Definition 9 (Valid Typing Environment) A typing environment Δ is valid if the judgement $\Delta \vdash \diamond$ is derivable by the rules in Figure 2.10.

Essentially, validity ensures all types are correctly build on basic types, or use valid type expressions under the typing environment.

$$\frac{\Delta \vdash \diamond}{\Delta \vdash \texttt{cmd}^{\ell}}\ (\text{W-CMD}) \qquad \frac{\Delta \vdash \tau^{\ell}}{\Delta \vdash \tau^{*\ell}}\ (\text{W-COLLECTION}) \qquad \frac{\Delta \vdash \diamond \quad \Delta \vdash \tau^{\ell}}{\Delta \vdash \texttt{ref}(\tau^{\ell'})^{\ell}}\ (\text{W-REF}) \qquad \frac{\Delta \vdash \diamond}{\Delta \vdash \texttt{Bool}^{\ell}}\ (\text{W-BOOL})$$

$$\frac{\Delta \vdash \diamond \quad \forall_i\, \Delta \vdash \tau_i^{\ell_i} \quad \ell = \sqcap \ell_i}{\Delta \vdash \{\overline{m : \tau^{\ell}}\}^{\ell}}\ (\text{W-VARIANT}) \qquad \frac{\forall_i\, \Delta \vdash \tau_i^{\ell_i} \quad \ell = \sqcap \ell_i}{\Delta \vdash [\overline{m : \tau^{\ell}}]^{\ell}}\ (\text{W-RECORD}) \qquad \frac{\Delta \vdash \diamond \quad \Delta \vdash \tau^{\ell_1} \quad \Delta, x{:}\tau^{\ell_1} \vdash \sigma^{\ell_2}}{\Delta \vdash (\tau^{\ell_1} \xrightarrow{\mathsf{r}} \sigma^{\ell_2})^{\ell}}\ (\text{W-ARROW})$$

Figure 2.11: Well-formed types

$$\frac{}{\tau^{\ell} <: \tau^{\ell}}\ (\text{S-REFLEX}) \qquad \frac{\tau^{\ell} <: \tau^{\ell''} \quad \tau^{\ell''} <: \tau^{\ell'}}{\tau^{\ell} <: \tau^{\ell'}}\ (\text{S-TRANS}) \qquad \frac{\ell \leq \ell'}{\tau^{\ell} <: \tau^{\ell'}}\ (\text{S-EXPR})$$

$$\frac{\forall_i\, \tau_i^{\ell_i} <: \tau_i^{\ell'_i} \quad \ell' = \sqcap \ell'_i}{\{\overline{m : \tau^{\ell}}\}^{\ell} <: \{\overline{m : \tau'^{\ell'}}\}^{\ell'}}\ (\text{S-VARIANT}) \qquad \frac{\tau^{\ell'_1} <: \tau^{\ell_1} \quad \sigma^{\ell_2} <: \sigma^{\ell'_2} \quad \mathsf{r}' \leq \mathsf{r}}{(\tau^{\ell_1} \xrightarrow{\mathsf{r}} \sigma^{\ell_2})^{\ell} <: (\tau^{\ell'_1} \xrightarrow{\mathsf{r}''} \sigma^{\ell'_2})^{\ell}}\ (\text{S-ARROW}) \qquad \frac{\forall_i\, \tau_i^{\ell_i} <: \tau_i^{\ell'_i} \quad \ell' = \sqcap \ell'_i}{[\overline{m : \tau^{\ell}}]^{\ell} <: [\overline{m : \tau^{\ell'}}]^{\ell'}}\ (\text{S-RECORD})$$

Figure 2.12: Subtyping rules

We now define our type system by means of a typing relation.

Definition 10 (Type System) Typing is expressed by the judgment $\Delta \vdash^{\mathsf{r}} e : \tau^{\ell}$, stating that expression e is well-typed by τ^{ℓ} in environment Δ, given computational context security label r.

The type system asserts, through a set of typing rules, that an expression is well-typed. If there is a valid typing for an expression then it is well-typed.

Definition 11 (Valid Typing) The judgement $\Delta \vdash^{\mathsf{r}} e : \tau^{\ell}$ is valid if it is derivable by the typing rules.

Our analysis also relies on a simple subtyping relation, denoted $<:$, which allows up-classification of security labels.

Note that this subtyping relation is defined only on security labels instead of the type structure itself. This is standard in type-based information flow analysis since our goal is to ensure secure information flows, thus we only need to inspect security labels. However, it could easily be extended to include such standard subtyping rules.

Definition 12 (Subtyping Relation) Our subtyping relation is expressed as $\tau^{\ell} <: \tau^{\ell'}$ and is defined by the rules given in Figure 2.12.

The subtyping relation is as expected on a type-based information flow analysis with function, record and variant types. In rule (S-ARROW) the security label is contravariant on the argument's type and on the recorded computational context security label r, and covariant on the return type. Notice rules (S-RECORD) and (S-VARIANT) preserve the record's and variant's label invariant, respectively.

We will now discuss our typing rules. Since base types play no role in our analysis, we will omit them in our examples for presentation purposes. Also, with few exceptions that we will point out, values in our language are typed with security types at security label $\bot$. The institution is that basic values are initially "public" ($\bot$) unless declared otherwise or if, given the context where they are used, they are classified to a higher security level.

We begin with our subsumption rule, (T-SUB), which is used to raise the security level of expressions or lower the computational context label, whenever necessary.

$$\frac{\Delta \vdash^{\mathsf{r}} e : \tau^{\ell} \quad \tau^{\ell} <: \tau^{\ell'} \quad \mathsf{r}' \leq \mathsf{r}}{\Delta \vdash^{\mathsf{r}'} e : \tau^{\ell'}} \text{ (T-SUB)}$$

Next we show how an abstraction is typed. So the typing rule for λ expressions, rule (T-LAMBDA), is as one would expect in a typed λ-calculus where we type the abstraction with a function type relating the parameter and result type

$$\frac{\Delta, x : \tau^{\ell_1} \vdash^{\mathsf{r}} e : \sigma^{\ell_2}}{\Delta \vdash^{\mathsf{r}} \lambda(x : \tau^{\ell_1}).e : (\tau^{\ell_1} \xrightarrow{\mathsf{r}} \sigma^{\ell_2})^{\bot}} \text{ (T-LAMBDA)}$$

However, we also record on the function type the computational context r under which the function was typed. As stated earlier, this registers the effects the function has on the store since the computational context serves as a lower bound on the write operations over the store. We will get back to this in the discussion of the typing rule for assignment primitive to further illustrate the need of recording the computational context in a function type.

Then, in rule (T-APP), we type an application by checking if the argument type matches the function parameter type, typing the result accordingly.

$$\frac{\Delta \vdash^{\mathsf{r}} e_1 : (\tau^{\ell_1} \xrightarrow{\mathsf{r}'} \sigma^{\ell_2})^{\ell} \quad \Delta \vdash^{\mathsf{r}} e_2 : \tau^{\ell_1} \quad \mathsf{r} \leq \mathsf{r}'}{\Delta \vdash^{\mathsf{r}} e_1(e_2) : \sigma^{\ell_2 \sqcup \ell}} \text{ (T-APP)}$$

Notice that, since the security label of the function type ℓ classifies the identity of the function, then the effects of calling the function, security label of the result ℓ_2, should not reveal anything about the function being called. For that reason, the resulting security label of an application, $\ell_2 \sqcup \ell$, is the greatest lower bound (glb) between the security labels of the result, ℓ_2, and the function type itself, ℓ.

Suppose for instance the following code snippet:

Example 10 Suppose we have a function `f` with type $(\bot \xrightarrow{\bot} \bot)^{\top}$

```
let f = λ(x:⊥). x + 1
   in f(2)
```

Then when calling `f` for integer `2`, if we do not restrict the security label of the application, we would obtain the result type's label which is $\bot$. This means that while the identity of function `f` was protected by label $\top$ (that is, function `f` could not be observed at lower than $\top$ security labels) its call is not. In fact, one can now observe the result since it has security label $\bot$.

So we have to preserve the security label of the function's identity. However, since it can be the case where the label of the function type is actually lower than the result type's, we compute the glb between these two labels as the resulting label of an application.

Another detail to note is that we enforce the computational context upon application, r, to be bounded by the computational context under which the λ value was created, r'. This is better understood once we have discussed the typing rules of the imperative primitives, so we delay the comments on this detail for later in this section.

Regarding let-declarations, rule (T-LET) is as expected: we type the let-declaration with the type of the second expression, $\tau'^{\ell'}$, under the typing environment augmented with the type of the first expression, τ^{ℓ}, associated to identifier x.

$$\frac{\Delta \vdash^{\mathsf{r}} e_1 : \tau^{\ell} \qquad \Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}} e_2 : \tau'^{\ell'}}{\Delta \vdash^{\mathsf{r}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau'^{\ell'}}\ (\text{T-LET})$$

As for the conditional, rule (T-IF), we need to be careful with potential implicit flows introduced by this primitive.

$$\frac{\Delta \vdash^{\mathsf{r}} c : \mathtt{Bool}^{\ell} \qquad \Delta \vdash^{\mathsf{r}\sqcup\ell} e_1 : \tau^{\ell} \qquad \Delta \vdash^{\mathsf{r}\sqcup\ell} e_2 : \tau^{\ell}}{\Delta \vdash^{\mathsf{r}} \mathbf{if}\ c\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau^{\ell}}\ (\text{T-IF})$$

So, in order to prevent implicit flows from occurring on write operations, we raise the security level of the computational context to the least upper bound (lub) of its current computational context, r, with the logical expression's security label, ℓ. Moreover, we enforce the security level of both branches as well as the logical expression to be the same (by means of up-classification via subtyping, if necessary) so as to prevent implicit flows on the result of the conditional.

Let us look at an example for illustration.

Example 11 Assume identifier `high` is classified at security label $\top$.

```
if high then
  true
else false
```

Then, if we do not enforce the labels on the logical expression to be the same as those on the branches, we would be able to infer the value of identifier `high` by observing the result of the conditional, which clearly violates data confidentiality.

So in this case we need to raise the security label of both branches from $\bot$ top $\top$ to prevent information computed in the branches to be known at security level $\bot$. This can be achieved using our subtyping relation.

We now introduce the typing rules for collections and their operations. We type a collection, (T-COLLECTION), with collection type $\tau^{*\ell}$ after checking that all its elements share the same type τ^{ℓ}.

$$\frac{\forall_i \quad \Delta \vdash^{\mathsf{r}} e_i : \tau^{e}ll}{\Delta \vdash^{\mathsf{r}} \{e_1, \ldots, e_n\} : \tau^{*\ell}} \text{(T-COLLECTION)}$$

One important thing to note here is that collections are homogeneous not only in the base types, but also on the security levels of its elements. So we cannot have a collection of integers where some of its elements are classified at $\bot$ while the remaining are classified at $\top$.

Regarding the cons operator, rule (T-CONS), it is typed as one would expect: with a collection type after checking compatibility with the type of the collection's elements.

$$\frac{\Delta \vdash^{\mathsf{r}} e_1 : \tau^{\ell} \quad \Delta \vdash^{\mathsf{r}} e_2 : \tau^{*\ell}}{\Delta \vdash^{\mathsf{r}} e_1 :: e_2 : \tau^{*\ell}} \text{(T-CONS)}$$

Notice that we can add an element to a collection of higher security level by raising the security level of the new element via subtyping, but not the other way around.

In order to type a **`foreach`** primitive, rule (T-FOREACH), we require the security level of all sub expressions to be the same. Also, to type the iterator's body e_3, we augment the typing environment, Δ, with the type of the collection being iterated, $\tau^{*\ell}$, associated to x and the type of the initial value of the accumulator,τ'^{ℓ}, mapped to y.

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}} e_1 : \tau^{*\mathsf{s}} \\ \Delta \vdash^{\mathsf{r}} e_2 : \tau'^{\mathsf{s}} \\ \Delta, x : \tau^{\mathsf{s}}, y : \tau'^{\mathsf{s}} \vdash^{\mathsf{r}} e_3 : \tau'^{\mathsf{s}}\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{foreach}\ (e_1,\ e_2,\ x.y.e_3) : \tau'^{\mathsf{s}}} \text{(T-FOREACH)}$$

The security labels must be the same to disallow insecure programs such as, e.g., in which one could count the elements of a collection classified with a high security level, and assign the result to a low level. For instance:

Example 12 Suppose we have collection `top_secrets` with elements classified at security level $\top$ and consider the code snippet.

```
foreach (x in top_secrets) with count = 0 do count + 1
```

This code can only be typed as $\texttt{int}^{\top}$. If we allowed the body of the foreach loop to be typed at a level lower that $\top$, we could type the result of the above program at security level $\bot$ since the computation only involves values at that level. That, however, would represent an implicit flow since one could then observe some information about collection `top_secrets` at level $\bot$, namely its number of elements, breaking noninterference.

While in other approaches ([51, 52]) a record type only has security labels in its field's types, in our system we require a record type to have a security label. This has to do with our decision of treating all types uniformly - all types have a security label - and such is reflected in our typing rules.

With that in mind, our rule (T-RECORD), which introduces record types,

$$\frac{\forall_i\ \Delta \vdash^{\mathsf{r}} e_i : \tau_i^{\ell_i}}{\Delta \vdash^{\mathsf{r}} [\ldots, m_i = e_i, \ldots] : [\ldots \times m_i{:}\tau_i^{\ell_i} \times \ldots]^{\sqcap \ell_i}}\ \text{(T-RECORD)}$$

requires the security label of record values to be, at most, the greatest lower bound (glb) of all the security labels occurring in their fields, otherwise implicit flows could occur on assignments of record values.

An example of such implicit flows is the assignment of a reference containing a record value. If such operation is executed under a computational context of a higher security level than some of the record's fields security levels, then an implicit flow occurs. We will get back to this example in the discussion of rule (T-ASSIGN) to further illustrate the need of a security label on a record and its invariant.

Notice that this allows (but does not force) records storing both private and public data to be classified as public. Such a scenario is in fact, secure, as will only leak, at most, information that a record is present, but not the field contents (except those classified as public). Let us illustrate with an example:

Example 13 Assume `boxed` to be a collection of records typed as

$$\texttt{boxed: ([public:}\bot \times \texttt{secret:}\top\texttt{])}^{*\bot}$$

Some fields contents of the collection's records are classified as high ($\top$), but the records themselves and the collection itself is classified as low ($\bot$). In this case, we can type

```
foreach (x in boxed) with count = 0 do count + 1
```

with type $\texttt{int}^{\bot}$. This means that the collection and its records (borders) are visible entities at level $\bot$, while the actual record field contents are concealed from the same level. With this specification, it would be allowed to a low observer to observe the collection size, but not the contents of the `secret` fields, preserving non-interference.

We type field projection, rule (T-FIELD), with the type associated to the field being projected, m_i, in the record type of expression e .

$$\frac{\Delta \vdash^{\mathsf{r}} e : [\ldots \times m_i{:}\tau_i^{\ell_i} \times \ldots]^{\ell'}}{\Delta \vdash^{\mathsf{r}} e.m_i : \tau_i^{\ell_i}} \text{(T-FIELD)}$$

Typing rules of logical expressions and the remaining values of the language – (T-UNIT), (T-TRUE), (T-FALSE) and (T-LOC) – are as expected so we will omit them in this discussion.

In order to type a variant, we need to check the compatibility of the expression in the variant, e, with the declared type for identifier m_i.

$$\frac{\Delta \vdash^{\mathsf{r}} e : \tau_i^{\ell_i}}{\Delta \vdash^{\mathsf{r}} \#m_i(e)\ \textbf{as}\ \{\ldots, m_i{:}\tau_i^{\ell_i}, \ldots\}^{\sqcap \ell_i} : \{\ldots, m_i{:}\tau_i^{\ell_i}, \ldots\}^{\sqcap \ell_i}} \text{(T-VARIANT)}$$

Typing a variant expression is similar to how we type records with respect to the security label given to the variant type. So, for the same reasons as in the record types, we require the security label of variants to be, at most, the greatest lower bound (glb) of all the security labels of their possible identifiers, otherwise implicit flows could occur on assignments of variant values. Again, we will illustrate and defer the discuss for later in this section.

We type a case primitive, rule (T-CASE), by enforcing each case branch has the same type under a typing environment that maps the associated case branch's variable, x_i, with the corresponding type, $\tau_i^{\ell_i}$, in the variant type of the variant vale being case-analysed.

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}} e : \{\ldots, m_i : \tau_i^{\ell_i}, \ldots\}^{\ell'} \\ \forall_i\ \ \Delta, x_i{:}\tau_i^{\ell_i} \vdash^{\mathsf{r}} e_i : \tau^{\ell}\end{array}}{\Delta \vdash^{\mathsf{r}} \textbf{case}\ e(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) : \tau^{\ell}} \text{(T-CASE)}$$

Lastly, we conclude our discussion with the typing rules for the imperative core of the language: (T-REF), (T-DEREF), and (T-ASSIGN).

Starting with the reference allocation primitive, rule (T-REF), we check if the expression being used to initialise the reference is compatible with the declared type. Then, we type a reference allocation with a reference type, $\texttt{ref}(\tau^\ell)$, of the type of the expression used to initialise the reference, τ^ℓ, and classify at security level $\bot$.

$$\frac{\Delta \vdash^{\mathsf{r}} e : \tau^\ell \quad \mathsf{r} \leq \ell}{\Delta \vdash^{\mathsf{r}} \mathbf{ref}_{\tau^\ell}\ e : \texttt{ref}(\tau^\ell)^\bot} \text{ (T-REF)}$$

We also impose a lower bound on the security level of the expression initialising the reference allocation to the computational context security level. Otherwise, illegal implicit flows could occur. For example:

Example 14 If we allowed the conditional on the snippet below,

```
let high = ref⊤ true in
  let x = ( if high then ref⊥ true else ref⊥ false)
    in !x
```

Then we would be able to observe that a new reference was allocated and obtain some information regarding the protected identifier `high` (in this case, we would be able to determine its value).

So this program does not comply with noninterference and is deemed insecure.

We type a deference operation, rule (T-DEREF), with the type of the reference's content, as one would expect.

$$\frac{\Delta \vdash^{\mathsf{r}} e : \texttt{ref}(\tau^\ell)^{\ell'} \quad \ell' \leq \ell}{\Delta \vdash^{\mathsf{r}} !e : \tau^\ell} \text{ (T-DEREF)}$$

However, we require the reference's security level, ℓ', to be the lower bound for the deference's security level, ℓ, in order to prevent implicit flows. This is because references are initially typed at security level $\bot$ but may raise to a different security level, given the computational context. For instance,

Example 15 Let us see the following snippet

```
let high = ref⊤ true
  let x = ref⊥ true in
    let y = ref⊥ false in
      let z = (if high then x else y) in !z
```

The conditional typechecks and is deemed secure, since we can raise the security level of both references `x` and `y` to $\top$. However, upon the dereference operation, the computational context is $\bot$ (it was only raised to type the conditional's branches), so we would leak the value of `high` (classified as $\top$), via an implicit flow, at a lower security level ($\bot$).

So we must deem this program as insecure because of the dereference operation.

Finally, the assignment primitive can introduce explicit flow. So when typing an assignment, (T-ASSIGN), we check the compatibility between the content's type in the reference denoted by e_1, and the type of the expression being used for the new content e_2.

$$\frac{\begin{array}{l}\Delta \vdash^{\mathsf{r}} e_1 : \mathtt{ref}(\tau^{\ell})^{\ell'} \\ \Delta \vdash^{\mathsf{r}} e_2 : \tau^{\ell} \\ \mathsf{r} \sqcup \ell' \leq \ell\end{array}}{\Delta \vdash^{\mathsf{r}} e_1 := e_2 : \mathtt{cmd}^{\bot}} \text{ (T-ASSIGN)}$$

Notice that this allows us to store values with lower security labels with respect to the reference's declared content's type via up-classification, but not the other way around.

We also require that the least upper bound (lub) of the computational context security label, r, and the reference's security level, ℓ', to be the lower bound of the content's security level ℓ. So the computational context r plays a key role in preventing explicit flows by ensuring only values classified at security levels above, or equal, to the computational context are altered in the store. This way one can safely type basic values and commands at the $\bot$ level, even at higher computational contexts.

This leads us back to our earlier discussion of our records/variants needing a security label which is the greatest lower bound (glb) of the security levels of the record/variant's fields/identifiers.

Example 16 Let us see

```
let cond = ref_⊤ true in
 let r = ref_[a:⊥×b:⊤]⊥ [a = 0, b = 1] in
   if !cond then
     r := [a = 2, b= 2]
```

In this snippet, we are updating a reference whose content is a record value containing a field of security level $\bot$, given that the logical condition `cond` holds.

This logical condition, however, is classified at security level $\top$, so this code snippet must be deemed insecure in our system. Otherwise, an implicit flow would occur and field `b` of the record store in reference `r` would depend on `cond`, which has a higher security level, thus violating noninterference.

In order to do so, and since our treatment of types in the typing rules does not distinguishes if the content of a reference is a record type, we must inspect the record's security label and be able to determine if the assignment operation is secure.

So in this case, the security label of the record is $\bot$ – corresponding to security level ℓ in rule (T-ASSIGN) – and, in the assignment operation, the computational context security label is $\top$ – security level r in rule (T-ASSIGN) – because we raised the security level of the branch's computational context to match the level of conditional's logical expression `cond`.

Therefore, the condition of the assignment typing rule $\mathsf{r} \sqcup \ell' \leq \ell$ does not hold in this example, and this program is deemed insecure by our analysis.

Example 17 Now let us look at the same issue but for variant values.

In this snippet, we might be updating a reference of a variant value that can either be an integer with identifier `low` and security level $\bot$, or a boolean with identifier `high` and security level $\top$.

```
let cond = ref_⊤ true in
 let r = ref_{low:int^⊥,high:bool^⊤}^⊥ #high(true) in
   if !cond then
     r := #low(0)
```

Since the logical expression`cond` is classified at security level $\top$, this code snippet must be deemed insecure in our system otherwise an implicit flow can occur in the assignment operation.

Indeed, when condition `cond` holds we can observe a change in the state of the program: now variant value obtained through reference `r` is an integer where once was a boolean. This clearly violates noninterference.

In order to disallow this program, and again because we treat our types uniformly in the typing rules, we must inspect the variant's security label and be able to determine if the assignment operation is secure.

So in this case, the security label of the variant is $\bot$ – corresponding to security level ℓ in rule (T-ASSIGN) – and, in the assignment operation, the computational context security label is $\top$ – security level r in rule (T-ASSIGN) – because we raised the security level of the branch's computational context to match the level of conditional's logical expression `cond`.

Therefore, the condition of the assignment rule $\mathsf{r} \sqcup \ell' \leq \ell$ does not hold in this example, and this program is deemed insecure by our type system.

Thus, as illustrated above, the security label of a record/variant is an upper bound on the computational context under which the record/variant value can be altered.

To finish our discussion, we go back to our function type and application typing rule.

Recall that our function types, $(\tau^{\ell_1} \xrightarrow{\mathsf{r}} \sigma^{\ell_2})^{\ell}$, keep track of the computational context under which the function was typed.

As we stated earlier, this is necessary to prevent implicit flows via write operations since the computational context serves as a lower bound on the writes to a program's state. Take as example the following snippets:

Example 18 We begin with an explicit flow by updating a reference, whose content is classified at security level $\bot$, under a computational context of higher security level, $\top$.

```
let cond = ref⊤ true in
  let low = ref⊥ 0 in
      in if !cond then
             low := 1
```

As we have seen, this is detected by our typing rules and deemed insecure. More concretely, rule (T-ASSIGN) disallows this assignment since the side-condition $\mathsf{r} \sqcup \ell' \leq \ell$ does not hold. However, one can attempt to circumvent this check by wrapping the assignment in a λ value:

```
let cond = ref⊤ true in
  let low = ref⊥ 0 in
    let f = λ (cell: ref(int⊥)⊥, value: int⊥). cell := value
     in if !cond then
             f(low,1)
```

In this case, the side condition of rule (T-ASSIGN) holds since function `f` would be typed with $(\bot \xrightarrow{\bot} \bot)^{\bot}$. So this is an implicit flow introduced by the λ value upon its application.

To solve this, we keep track of the computational context under which the λ was typed, using it as an upper bound of the computational context under which the function can be called. This corresponds to the side-condition $\mathsf{r} \leq \mathsf{r}'$ in rule (T-APP).

Thus, for this example, we are able to deem this program insecure using our typing rules because the computational context at the moment of the application is $\top$ but the computational context in the function type is $\bot$, so $\top \leq \bot$ does not hold.

We conclude this chapter by showing our type system is safe, that is, well-typed programs always preserve their typing and never get stuck.

2.2.1 Type Safety

We now show that our core language is type safe, that is, that a program in our language evaluates to a value of the expected type and never gets stuck.

We start by introducing some preliminary definitions. Namely, we introduce notions of store consistency and well-typed configurations.
We say that a store S is well-typed with relation to a typing environment Δ if the values referred by its locations have the expected type. We define the typing of stores as follows:

Definition 13 (Store Consistency) Let Δ be a typing environment and S a store, we say store S is consistent with respect to typing environment Δ, denoted as $\Delta \vdash S$, if $dom(S) \subseteq dom(\Delta)$ and $\forall l \in dom(S)$ then $\Delta \vdash^{\mathsf{r}} S(l) : \Delta(l)$.

From the store consistency definition, we define what it means for a configuration to be well-typed.

Definition 14 (Well-typed Configuration)
A configuration $(S;\ e)$ is well-typed in typing environment Δ if $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}} e{:}\tau^{\ell}$.

So a configuration $(S;\ e)$ is well-typed if there is a typing environment Δ that types both the store and the expression, for an arbitrary computational context.

To prove type preservation, we introduce the substitution lemma on which it relies. This lemma states that the type of an expression is preserved under substitution, allowing us to prove the cases in type preservation where a β-reduction occurs.

Lemma 1 (Substitution Lemma)
If $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}} e : \tau^{\ell}$ and $\Delta \vdash^{\mathsf{r}'} v : \tau'^{\ell'}$ then $\Delta \vdash^{\mathsf{r}} e\{v/x\} : (\tau^{\ell})\{v/x\}$.
Proof: Induction on the derivation of $\Delta, x : \tau'^{\ell'} \vdash^{\mathsf{r}} e : \tau^{\ell}$.

Theorem 3 says that well-typed configurations remain well-typed after a reduction step, and possibly the final configuration is extended with new locations in the state.

Theorem 1 (Type Preservation)
Let $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}} e : \tau^{\ell}$.
If $(S; e) \longrightarrow (S'; e')$ then there is Δ' such that $\Delta' \vdash^{\mathsf{r}} e' : \tau^{\ell}$, $\Delta' \vdash S'$ and $\Delta \subseteq \Delta'$.
Proof: Induction on the derivation of $\Delta \vdash^{\mathsf{r}} e : \tau^{\ell}$,

We now present progress theorem, Theorem 4, which states that well-typed programs never get stuck.

Theorem 2 (Progress)
Let $\Delta \vdash^{\mathsf{r}} e : \tau^{\ell}$ and $\Delta \vdash S$. If e is not a value then $(S;\ e) \longrightarrow (S';\ e')$.
Proof: Induction on the derivation of $\Delta \vdash^{\mathsf{r}} e : \tau^{\ell}$.

These theorems ensure that our semantics preserves typability and well-typed programs never get stuck, thus making our type system safe. However, the soundness result, with respect to our information flow analysis, is noninterference (Definition 5).

Thus noninterference together with Theorem 3 and Theorem 4, establishes that our system ensures well-typed programs do not leak confidential information under the security policy prescribed by the assumed security lattice. In other words, data does not flow from a security compartment to another if they are unrelated or if it is a down-flow in the security lattice.

Next, as conclusion to this chapter, we illustrate the limitations of the type system above via a revisit of the toy example introduced in Chapter 1.

2.3 Toy Example: A Conference Manager System

In this section, we revisit the conference manager system used to introduce our approach in Chapter 1. Our focus will be on the limitations of using the type system presented on the previous section with standard security labels.

We start by defining some useful abbreviations to be used later in examples.

Abbreviations:
$[m_1 = r.m_1,\ m_i = e,\ \ldots]$ stands for $[m_1 = r.m_1, m_{i-1} = r.m_{i-1}, m_i = e, \ldots, m_n = r.m_n]$

The above abbreviation is useful when writing new record values based on existing ones: we just mention the fields being assigned a new value, and a sample field indicating the record value from which the other values are to be copied.

Recall that a user of this system can be either a registered user, an author user, or a PC member user. Moreover, the system stores data concerning its users' information, their submissions, and the reviews of submissions in "database tables" which we represent in our core programming language as collections of (references to) records (e.g., mutable collections):

$\textbf{let}\ \texttt{Users} = \textbf{ref}_{\texttt{ref}(\tau)^{*\perp}}\ (\textbf{ref}_{\tau}\ [\,]\ )::\{\}\ \textbf{in}$
$\textbf{let}\ \texttt{Submissions} = \textbf{ref}_{\texttt{ref}(\sigma)^{*\perp}}\ (\textbf{ref}_{\sigma}\ [\,]\ )::\{\}\ \textbf{in}$
$\textbf{let}\ \texttt{Reviews} = \textbf{ref}_{\texttt{ref}(\delta)^{*\perp}}\ (\textbf{ref}_{\delta}\ [\,]\ )::\{\}$

Before explaining the types declared for each collection, we introduce the security labels used in this system to classify data. Thus, we assume the following security levels for our conference manager system:

- $\perp$, for data observable by anyone;
- $\texttt{U}$, for data observable by registered users;
- $\texttt{A}$, for data observable by authors;
- $\texttt{PC}$, for data observable by PC members;
- $\top$, for data observable by the admin user.

These security levels follow the pre-order $\perp \leq \texttt{U} \leq \texttt{A} \perp \leq \texttt{PC} \leq \top$, establishing the security lattice for our analysis in this scenario.
We can now discuss the types given for the above collections. So we have the following types for the contents of `Users`, `Submissions`, and `Reviews`, respectively:

$\tau \stackrel{\text{def}}{=} [\texttt{uid} : \perp \times \texttt{name} : \texttt{U} \times \texttt{univ} : \texttt{U} \times \texttt{email} : \texttt{U}]$
$\sigma \stackrel{\text{def}}{=} [\texttt{uid} : \perp \times \texttt{sid} : \perp \times \texttt{title} : \texttt{A} \times \texttt{abs} : \texttt{A} \times \texttt{paper} : \texttt{A}]$
$\delta \stackrel{\text{def}}{=} [\texttt{uid} : \perp \times \texttt{sid} : \perp \times \texttt{PC\_only} : \texttt{PC} \times \texttt{review} : \texttt{A} \times \texttt{grade} : \texttt{A}]$

These types, together with the security lattice, establish the following security policy:

- A registered user's information is observable from security level `U`, meaning any registered user (including authors and PC members) can see it;
- The content of a paper can be seen by authors (as well as PC members);
- Comments to the PC, on a submission's review, are observable only to PC members, while reviews and grades of the submission can be seen by authors.

We now proceed with some examples on how our type system works to disallow insecure programs, while also highlighting its downfalls. Consider then the following code

Example 19 The code below retrieves the submission of author with id 42, associating to identifier `sub42`, and then attempts to store some of its protected data in a public container `leak`.

```
let leak = ref⊥ "" in
let sub42 = foreach (x in !Submissions) with y = {} do
         let t_sub = !x in
           if (t_sub.uid = 42) then
             [uid = t_sub.uid, sid = t_sub.sid, title = t_sub.title]::y
           else y
in leak := sub42.title
```

The result of evaluating the collection iterator is a collection of records of record type. More concretely, the expected type for `sub42` would be $\texttt{[uid:}\bot \times \texttt{sid:}\bot \times \texttt{title: A]}^{\bot}$.

So, on the assign operation, our type system detects an insecure information flow since `sub42.title` has a higher security label than `leak`.

Now suppose we have a function *contains* that given two strings, returns whether the first argument string contains the second argument string. So *contains* has type $\texttt{A} \xrightarrow{\bot} \bot \xrightarrow{\bot} \texttt{A}$, notice the result type is classified with security label `A`. Let us consider the snippet below

```
if contains(sub42.title, "DIFT") then
  ref⊥ true
```

Our analysis detects an implicit information flow due to the reference creation under a higher computational context than its contents.

As expected, our type system is able to detect explicit flow as well as implicits flows of information with respect to the specified security policy.

However, security policies relying on standard security labels are inadequate to express "row-level" security concerns. Take, for instance, the following example

Example 20 Function `viewUserProfile` obtains a given user's profile

```
let viewUserProfile = λ (u).
  foreach(x in !Users) with y = {} do
    let usr = !x in
      if usr.uid = u then usr::y else y
```

Thus, our analysis types `viewUserProfile` with type

$$\bot \xrightarrow{\bot} \texttt{[uid:}\bot \times \texttt{name: U} \times \texttt{univ: U} \times \texttt{email: U]}^{*\bot}$$

So we can retrieve the email of user with id 42 by calling the function and then projecting the corresponding field, **first**(viewUserProfile(42)).email, which would be typed as U. However, any registered user can observe this piece of information since it is classified at security level U. Thus, the following snippet is deemed secured

```
foreach(x in !Users) with y = {} do
    let usr = !x in
      if usr.uid = 70 then
        x := [uid = 70, email = first(viewUserProfile(42)).email, ...]
```

which leaks a user's email to another registered user.

So, as we have seen, standard security labels are unable to express fine-grained security concerns such as "row-level" policies.

2.4 Remarks

In this chapter, we have presented our core language, λ_{RCV}, used to support our analysis by introducing its syntax and semantics. We then proceeded with a typed version of our core language, $\lambda_{\tau RCV}$, as a means to introduce basic concepts of type-based information flow analysis.

For that purpose, we presented a type system for $\lambda_{\tau RCV}$ based on standard security labels, discussing its typing rules and how they disallow insecure information flows. We then showed type safety of the type system presented, that is, that well-typed programs can always progress with the correct typing.

We finally concluded with a brief discussion, via a toy example, of the limitations of the type system using standard security labels with respect to the expressiveness of the security policies.

In the following chapter, we introduce our dependent information flow types and show how they can express "row-level" security concerns.

CHAPTER 3

DEPENDENT INFORMATION FLOW TYPES

In this chapter we formally present our dependent information flow types. As already discussed in previous chapters, our type system for information flow builds on fairly traditional concepts from information flow type systems [1, 28], but crucially explores a notion of type dependency on security labels, in a way that cleanly fits within a standard framework of dependent type theory with canonical dependent functional and sum types.

We illustrate our analysis with some of the previously given examples and discuss some of our key typing rules. We follow with the presentation of our type safety results that ensures our semantics preserve typability and that a well-typed program never gets stuck. We finish this chapter with a discussion of the relevant related work.

3.1 λ_{DIFT}: A Dependent Information Flow Typed λ-calculus

In this section we introduce another typed version of our core language, λ_{DIFT}, using dependent information flow types. We then proceed with the type system using dependent information flow types, discussing the challenges posed by our approach and how we tackled them.

Let us begin by introducing abbreviations to be used for ease of presentation.

Syntactic Conventions:
$\lambda(x, \ldots, z).e$ is to be read as $\lambda x.(\ldots).\lambda z.e$

Abbreviations:
$\ell(\overline{\top})$ stands for $\ell(\top, \ldots, \top)$
$\ell(\overline{\bot})$ stands for $\ell(\bot, \ldots, \bot)$
$\{\overline{m : \tau^{s}}\}^{s}$ stands for $\{m_1 : {\tau_1}^{s_1}, \ldots, m_n : {\tau_n}^{s_n}\}^{\ell}$
$\Sigma[\overline{m : \tau^{s}}]^{s}$ stands for $\Sigma[m_1 : {\tau_1}^{s_1}, \ldots, m_n : {\tau_n}^{s_n}]^{\ell}$

$[m_1 = r.m_1,\ \ m_i = e,\ \ \ldots]$ stands for $[m_1 = r.m_1, m_{i-1} = r.m_{i-1}, m_i = e, \ldots, m_n = r.m_n]$

We make the same assumptions we did in Chapter 2 for the type-based information flow analysis. Before introducing λ_{DIFT}, we introduce our value-dependent security labels and our new syntax of types.

3.1.1 Value-dependent Security Labels

Value-dependent security labels may partition standard security levels by indexing labels ℓ with values v, so that each partition $\ell(v)$ classify data at a specific level, depending on the value v.

For example, we can partition the security level `U` into n security compartments, each representing a single registered user of the system, so security level `U(01)` represents the security compartment of the registered user with id `01`.

Of course, one may also consider indexed labels of arbitrary arity, for instance for security level `A` (author) we can index with both the author's id and submission's id so `A(42,70)` would stand for the security compartment of data relating to author (with id `42`) and his submission (id `70`).

Syntax of Security Labels.

So our security labels, which we consider in general to be value dependent, have the form $\ell(\overline{\mathtt{v}})$, where $\overline{\mathtt{v}}$ is a list of security label indexes. Label indexes are given by:

$\mathtt{v} ::=$		(*label indexes*)		
	$\mid \top$	(top)	$\mid \bot$	(bot)
	$\mid$ **`true`**	(true)	$\mid$ **`false`**	(false)
	$\mid \overline{v}$	(collection)	$\mid [\overline{m : v}]$	(record)
	$\mid x.m$	(field selection)	$\mid x$	(variable)
	$\mid m$	(field identifier)		

We define in the expected way, but omit technical details, the free variables of a security label, $fv(\mathtt{s})$, the free names of a security label, $fn(\mathtt{s})$, and substitution on security labels, $\mathtt{s}\{^{v}/_{x}\}$.

Let us now define *concrete* security labels as those whose indexes are either $\bot$,$\top$, or a value v, not a (record) field identifier m (or field selection $x.m$), or a variable x.

Definition 15 (Concrete Security Labels)

Let `s` be a security label. We say `s` is concrete if, and only if, $fv(\mathtt{s}) \cap fn(\mathtt{s}) = \emptyset$.

So, for instance, `S(42,70)`, `S(`$\top$`,70)`, and `S(`$\top$`,`$\top$`)` are *concrete* security labels but `S(uid,70)` and `S(uid,sid)` are not.

As we will see below, labels indexed by a simple field identifier, e.g., $\ell(m)$, only make sense in the scope of a field m in a dependent sum type.

$$
\begin{array}{rll}
\mathsf{s},\mathsf{r},\mathsf{t},\mathsf{q} ::= & \ell(\overline{\mathsf{v}}) & \text{(\textit{security labels})} \\
\tau^{\mathsf{s}},\sigma^{\mathsf{s}} ::= & & \text{(\textit{security types})} \\
& \mid \mathtt{Bool}^{\mathsf{s}} & \text{(bool type)} \\
& \mid \mathtt{cmd}^{\mathsf{s}} & \text{(command type)} \\
& \mid \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}} & \text{(reference type)} \\
& \mid \tau^{*\mathsf{s}} & \text{(collection type)} \\
& \mid \{\overline{m:\tau^{\mathsf{s}}}\}^{\mathsf{t}} & \text{variant type} \\
& \mid (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r};\sigma^{\mathsf{q}})^{\mathsf{t}} & \text{dependent function type)} \\
& \mid \Sigma[\overline{m:\tau^{\mathsf{s}}}]^{\mathsf{r}} & \text{dependent sum type)}
\end{array}
$$

Figure 3.1: Abstract Syntax of Types

Similarly, labels indexed by field selection, e.g., $\ell(x.m)$, only make sense in the scope of a field n, in a dependent sum type, that is related to field m of record value denoted by x.

We will give some examples, in later sections, where these indexes are used.

Next, we discuss our requirements regarding the assumed security lattice for our analysis.

3.1.2 Security Lattice

We consider a general notion of security lattice.

We require the lattice $\mathcal{L}$ elements to be concrete security labels, with $\top$ the top element (the most restrictive security level), and $\bot$ the bottom element (the most permissive security level), and $\sqcup$, $\sqcap$, denote the join and meet operations respectively.

The lattice partial order is noted $\leq$ and $<$ its strict part; we write $\mathsf{s}\#\mathsf{s}'$ to assert that neither $\mathsf{s} \leq \mathsf{s}'$ nor $\mathsf{s}' \leq \mathsf{s}$.

As mentioned earlier, dependent security labels $\ell(\overline{\bot})$ and $\ell(\overline{\top})$ are interpreted as approximations to the "standard" non-value dependent label ℓ. We thus require that for any values $v_1, \ldots, v_n$ that make the label $\ell(v_1, \ldots, v_n)$ concrete, $\ell(\overline{\bot}) \leq \ell(v_1, \ldots, v_n) \leq \ell(\overline{\top})$ holds in the given security lattice $\mathcal{L}$, and that the ordering between labels is well defined and satisfies the lattice property (i.e., well defined meets and joins, etc).

We also assume the intended security lattice, required for each particular security analysis, is specified by a set of schematic assertions of the form $\forall\overline{x}.\ell_1(\overline{u}) \leq \ell_2(\overline{v})$, where the (optional) $\overline{x}$ may occur in $\overline{u}, \overline{v}$. Lattice assertions are useful to relate indexes between two distinct security labels, as shown earlier in Chapter 1.

We can now proceed with the introduction of our syntax of types for λ_{DIFT}.

3.1.3 Types

Let us now introduce our language of types for λ_{DIFT}.

Definition 16 (Dependent Information Flow Types) The types $\mathcal{T}_{DIFT}$ of λ_{DIFT} are defined by the abstract syntax in Figure 3.1.

Our (security) types now have the form τ^{s}, so all types are annotated with a security label s that may be indexed.

Notice that the difference between this syntax of types, $\mathcal{T}_{DIFT}$, and the syntax presented in the previous chapter, $\mathcal{T}_{RCV}$, consists in the security labels (that now can be indexed) and the introduction of dependent function and sum types.

So our (security) types, τ^{s}, can be boolean $\mathtt{bool}^{\mathsf{s}}$, unit $\mathtt{cmd}^{\mathsf{s}}$ (denoted as command), reference $\mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{s}}$, variant $\{\overline{m : \tau^{\mathsf{s}}}\}^{\mathsf{s}}$, dependent sum type $\Sigma[\overline{m : \tau^{\mathsf{s}}}]^{\mathsf{s}}$, dependent function type$(\Pi(x : \tau^{\mathsf{s}}).\mathsf{r}; \sigma^{\mathsf{q}})^{\mathsf{t}}$, and collection types $\tau^{*^{\mathsf{s}}}$. As we have seen, in collection type $\tau^{*\mathsf{s}}$ each collection element has type τ^{s}.

As stated previously in Chapter 1, our dependent sum types and dependent function types take a key role in our type system by allowing us to express (runtime) value dependency on security labels, as already illustrated.

A dependent sum type has the general form

$$\Sigma[m_1 : \tau_1^{\mathsf{s}_1} \times \ldots \times m_n : \tau_n^{\mathsf{s}_n}]^{\mathsf{r}}$$

where any security label s_i with $i > 1$ may be dependent on previous fields (via the field identifier). For example, the type

$$\Sigma[\mathsf{uid} : \mathtt{int}^{\perp} \times \mathsf{photos} : \mathtt{bytes}^{*\mathsf{user}(\mathsf{uid})}]^{\perp}$$

is a dependent sum type where field photos has the value dependent security level, $\mathsf{user}(\mathsf{uid})$, which is indexed by the (runtime) value in field uid.

Like before for record types, we require the security level of a dependent sum type to always be the greatest lower bound (glb) of its field's security labels in order to prevent implicit flows on writes. This invariant is preserved by the subtyping relation.

Note that in order to compute the glb, we take intro consideration that a field's security label s may have dependencies, and, therefore, we approximate the values they denote with $\perp$, via operation $|\mathsf{s}|^{\downarrow}$.

A dependent function type has the form

$$(\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\mathsf{t}}$$

where the security level of the return type σ^{q} may depend on the value of the argument (denoted by the bound variable x).

Like for standard function types, we annotate the dependent function type with security level r , which is a lower bound on the function effects (writes). If omitted, security level r is assumed to be $\perp$. When x does not occur free in σ^{q} we write $(\tau^{\mathsf{s}} \xrightarrow{\mathsf{r}} \sigma^{\mathsf{q}})^{\mathsf{t}}$ for the type above, or simply $(\tau^{\mathsf{s}} \to \sigma^{\mathsf{q}})^{\mathsf{t}}$ if r is $\perp$.

For instance, the type

$$(\Pi x{:}\mathtt{int}^{\perp}; \mathtt{bytes}^{*\mathsf{user}(x)})^{\perp}$$

could be given to the function that retrieves a given user's photos, and whose effects are *may be* observable up to security level $\perp$.

$e ::=$		(*expression*)
	$\lambda(x : \tau^{\ell}).e$	(abstraction)
	$e_1(e_2)$	(application)
	x	(variable)
	$[\overline{m = e}]$	(record)
	$e.m$	(field access)
	$\{\overline{e}\}$	(collection)
	$e_1 :: e_2$	(cons)
	foreach$(e_1,\ e_2,\ x.y.e_3)$	(iteration)
	$\#m(e)$ **as** τ^{ℓ}	(variant)
	case e $(\overline{m \cdot x \Rightarrow e})$	(case)
	let $x = e_1$ **in** e_2	(let)
	if c **then** e_1 **else** e_2	(conditional)
	$\mathbf{ref}_{\tau^{\ell}}\ e$	(reference)
	$e_1 := e_2$	(assign)
	$!e$	(deref)
	v	(value)

(a) Expressions

$v ::=$		(*values*)
	$\lambda(x : \tau^{\ell}).e$	(abstraction)
	$[\overline{m = v}]$	(record)
	$\overline{v}$	(collection)
	$\#m(v)$ **as** τ^{ℓ}	(variant)
	true	(true)
	false	(false)
	$()$	(unit)
	l	(locations)

(b) Values

Figure 3.2: Abstract Syntax of Typed λ_{RCV}

We now use the type language $\mathcal{T}_{DIFT}$ to define another typed version of the λ_{RCV}, denoted as λ_{DIFT}.

Syntax

The syntax of λ_{DIFT} is given by Figure 3.2, using the abstract syntax of types defined in Figure 3.1.

Notice that the only difference with respect to $\lambda_{\tau RCV}$ (Figure 2.8 in Chapter 2) is the abstract syntax of types, that now includes value-dependent security labels and dependent sum and function types.

Notice that in source types and programs non-concrete security labels may only occur in the context of dependent sum types and dependent functional types.

Also, even if types for collections are homogeneous, due to the presence of dependent sum types, the system accommodates collections of elements containing data in different, possibly incomparable, security compartments.

Operational Semantics

The semantics of λ_{DIFT} is the same as the one presented for λ_{RCV} in Chapter 2, the only difference is the abstract syntax. For that reason we omit the rules of the operational semantics and move on to the introduction of the type system.

Before presenting our type system, let us first discuss the issue of dependencies in value-indexed security labels.

3.1.4 Dependencies in Indexed Security Labels

As mentioned earlier, the security lattice only relates concrete labels. As such, at some points, our type system is required to approximate runtime values to eliminate dependencies occurring in security labels.

For instance, should we project field `name` of a record typed with $\Sigma[\texttt{uid} : \texttt{int}^{\bot} \times \texttt{name} : \texttt{string}^{\texttt{user(uid)}}]^{\bot}$, then we would need to eliminate the field dependency in the resulting type's security label, `user(uid)`, into either `user(s)` if the actual name `s` can be deduced from the computational context as is often the case, or, at least, by `user(`$\top$`)`.

Dually, it may also be necessary to capture value dependencies in security labels, e.g., if we declare a reference of type $\Sigma[\texttt{uid} : \texttt{int}^{\bot} \times \texttt{name} : \texttt{string}^{\texttt{user(uid)}}]^{\bot}$ and then initialise with a record with type $\Sigma[\texttt{uid} : \texttt{int}^{\bot} \times \texttt{name} : \texttt{string}^{\texttt{user(0)}}]^{\bot}$, then we would need to introduce the field dependency in `user(0)`. We give more examples later in this chapter.

We achieve such introduction and elimination of dependencies in security labels by:

- Tracking knowledge regarding dependencies in a constraint set $\mathcal{S}$ carried along in typing judgements;
- Using a equational theory to entail runtime values or dependencies, depending whether we are eliminating or capturing dependencies in security labels.

We restrict constraints to talk about pure expressions, without side-effects.

Definition 17 (Constraint Set)
A constraint set $\mathcal{S}$ is a finite set of constraints of the form $e \doteq e'$ where e, e' are pure (without side-effects) expressions.

We assume a decidable sound equational theory, talking about basic data such as booleans, integers, records, etc, and write $S \models e \doteq e'$ for the entailment of $e \doteq e'$ given the constraints in $\mathcal{S}$. We also require $\doteq$ to be compatible with reduction in the sense that for any e, e' pure if $(S; e) \longrightarrow (S; e')$ then $\models e \doteq e'$. For instance, if $(S; 1 + 1) \longrightarrow (S; 2)$ then $\models 1 + 1 \doteq 2$.

We denote by $\mathcal{S}\{x \doteq e\}$ the set $\mathcal{S} \cup \{x \doteq e\}$ if e is a pure expression, and $\mathcal{S}$ otherwise. For example $\mathcal{S}\{x \doteq \texttt{true}\ \textbf{and}\ y.m = 42\}$ would be $\mathcal{S} \cup \{x \doteq \texttt{true}\ \textbf{and}\ y.m = 42\}$, but $\mathcal{S}\{x \doteq x := 1\}$ would remain just $\mathcal{S}$.

We give some examples of expected equational axioms:

$$(c \wedge c') \doteq \texttt{true} \Rightarrow c \doteq \texttt{true}$$
$$[\ldots, m_i{:}v_i, \ldots].m_i \doteq v_i$$
$$(x \doteq v) \wedge e \doteq e' \Rightarrow e\{^v/_x\} \doteq e'\{^v/_x\}$$
$$v \doteq v$$

So, for example, $\{x.\texttt{uid} \doteq \texttt{uid\_r}, \texttt{uid\_r} \doteq 42\} \models x.\texttt{uid} \doteq 42$.

As for any equational theory, we assume that $\mathcal{S} \models E$ and $\mathcal{S} \cup \{E\} \models E'$ implies $\mathcal{S} \models E'$ (deduction closure).

For the purpose of this work we consider constraint solving issues inside a black-box, subject to the mentioned general requirements. We do not specify any particular equational

$$
\begin{array}{lll}
\Delta ::= & & \textit{(typing environment)} \\
& \mid \phi & \text{(empty environment)} \\
& \mid \Delta, x : \tau^{\mathsf{s}} & \text{(type assignment to a variable)} \\
& \mid \Delta, l : \tau^{\mathsf{s}} & \text{(type assignment to a location)}
\end{array}
$$

Figure 3.3: Abstract Syntax of Typing Environments

theory since its precise formulation is orthogonal to our analysis, as long as it is decidable and sound (the more complete the theory the better).

We now proceed with the discussion of our type system.

3.1.5 Type System

To type λ_{DIFT} expressions, we define a typing judgment of the form

$$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$$

It asserts that expression e has type τ^{s} under typing environment Δ, given constraints $\mathcal{S}$.

The label s states that the value of expression e does not depend on data classified with security levels above s or incomparable with s. As expected from type-based approaches to information flow analysis, our type system ensures that information only flows upwards the security lattice, e.g., only from a level l to a level h such that $\mathsf{l} \leq \mathsf{h}$.

Label r is concrete and expresses the security level of the computational context (cf. the "program counter" [42, 52]), a familiar technique to prevent implicit flows as shown in Chapter 2.

Before formalizing our type system, we give some definitions.

Definition 18 (Typing Environment) For $x \in \mathcal{X}$, $l \in \mathtt{Loc}$, and $\tau^{\mathsf{s}} \in \mathcal{T}_{DIFT}$ the set Δ of all typing environments is defined by the abstract syntax in Figure 3.3.

Typing declarations assign types to identifiers $x{:}\tau^{\mathsf{s}}$, and types to locations, $l{:}\tau^{\mathsf{s}}$. A typing environment Δ is a list of typing declarations.

For simplicity, and without loss of generality, we consider in our presentation that security labels are indexed by a single label index, assuming the obvious extension of type and subtyping rules to deal with labels with multiple indexes, when necessary, e.g., in examples.

We now redefine well-formedness of types to take into account the new syntax of types.

Definition 19 (Well-formed Types) Well-formed types are denoted by judgment $\Delta \vdash^{\mathcal{N}} \tau^{\mathsf{s}}$, stating that type τ^{s} is well-formed under typing context Δ, given names set $\mathcal{N}$, and is given by the set of rules shown in Figure 3.5.

The difference essentially consists in checking for dependencies in the security labels to check if they are within the domain of the typing environment (if variables) or within the set of names $\mathcal{N}$ (for field identifiers).

$$\text{(S-INDEXLEFT)}\ \frac{\ell(\top) \leq \mathsf{s}}{\tau^{\ell(\mathsf{v})} <: \tau^{\mathsf{s}}} \qquad \text{(S-INDEXRIGHT)}\ \frac{\mathsf{s} \leq \ell(\bot)}{\tau^{\mathsf{s}} <: \tau^{\ell(\mathsf{v})}} \qquad \text{(S-EXPR)}\ \frac{\mathsf{s} \leq \mathsf{s}'}{\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}}$$

$$\text{(S-VARIANT)}\ \frac{\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau_i^{\mathsf{s}'_i} \quad \mathsf{t}' = \sqcap \mathsf{s}'_i}{\{\overline{m : \tau^{\mathsf{s}}}\}^{\mathsf{t}} <: \{\overline{m : \tau'^{\mathsf{s}'}}\}^{\mathsf{t}'}} \qquad \text{(S-ARROW)}\ \frac{\tau^{\mathsf{s}'} <: \tau^{\mathsf{s}} \quad \sigma^{\mathsf{q}} <: \sigma^{\mathsf{q}'} \quad \mathsf{r}' \leq \mathsf{r}}{(\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\mathsf{t}} <: (\Pi x{:}\tau^{\mathsf{s}'}.\mathsf{r}'; \sigma^{\mathsf{q}'})^{\mathsf{t}}}$$

$$\text{(S-RECORD)}\ \frac{\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau_i^{\mathsf{s}'_i} \quad \mathsf{t}' = \sqcap |\mathsf{s}'_i|^{\downarrow}}{\Sigma[\overline{m : \tau^{\mathsf{s}}}]^{\mathsf{t}} <: \Sigma[\overline{m : \tau^{\mathsf{s}'}}]^{\mathsf{t}'}}$$

Figure 3.4: Subtyping rules

$$\text{(W-COLLECTION)}\ \frac{\Delta \vdash^{\mathcal{N}} \tau^{\mathsf{s}}}{\Delta \vdash^{\mathcal{N}} \tau^{*\mathsf{s}}} \qquad \text{(W-REF)}\ \frac{\Delta \vdash \diamond \quad \Delta \vdash^{\mathcal{N}} \tau^{\mathsf{s}} \quad \mathit{fv}(\mathsf{t}) \subseteq \mathit{dom}(\Delta) \quad \mathit{fn}(\mathsf{t}) \in \mathcal{N}}{\Delta \vdash^{\mathcal{N}} \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}}} \qquad \text{(W-BOOL)}\ \frac{\Delta \vdash \diamond \quad \mathit{fn}(\mathsf{s}) \in \mathcal{N} \quad \mathit{fv}(\mathsf{s}) \subseteq \mathit{dom}(\Delta)}{\Delta \vdash^{\mathcal{N}} \mathtt{Bool}^{\mathsf{s}}}$$

$$\text{(W-CMD)}\ \frac{\Delta \vdash \diamond \quad \mathit{fv}(\mathsf{s}) \subseteq \mathit{dom}(\Delta) \quad \mathit{fn}(\mathsf{s}) \in \mathcal{N}}{\Delta \vdash^{\mathcal{N}} \mathtt{cmd}^{\mathsf{s}}} \qquad \text{(W-SUBTYPE)}\ \frac{\Delta \vdash^{\mathcal{N}} \tau^{\mathsf{s}} \quad \Delta \vdash^{\mathcal{N}} \tau'^{\mathsf{s}'}}{\Delta \vdash^{\mathcal{N}} \tau^{\mathsf{s}} <: \tau'^{\mathsf{s}'}}$$

$$\text{(W-VARIANT)}\ \frac{\Delta \vdash \diamond \quad \forall_i\ \Delta \vdash^{\mathcal{N}} \tau_i^{\mathsf{s}_i} \quad \mathit{fv}(\mathsf{t}) \subseteq \mathit{dom}(\Delta) \quad \mathsf{t} = \sqcap \mathsf{s}_i}{\Delta \vdash^{\mathcal{N}} \{\overline{m : \tau^{\mathsf{s}}}\}^{\mathsf{t}}}$$

$$\text{(W-RECORD)}\ \frac{\forall_i\ \Delta \vdash^{\mathcal{N} \cup \{m_1, \ldots, m_{i-1}\}} \tau_i^{\mathsf{s}_i} \quad \mathsf{s} = \sqcap |\mathsf{s}_i|^{\downarrow} \quad \mathit{fv}(\mathsf{s}) \subseteq \mathit{dom}(\Delta) \quad \mathit{fn}(\mathsf{s}) \in \mathcal{N}}{\Delta \vdash^{\mathcal{N}} \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}}$$

$$\text{(W-ARROW)}\ \frac{\Delta \vdash \diamond \quad \Delta \vdash^{\mathcal{N}} \tau^{\mathsf{s}} \quad \Delta, x{:}\tau^{\mathsf{s}} \vdash^{\mathcal{N}} \sigma^{\mathsf{q}} \quad \mathit{fv}(\mathsf{t}) \subseteq \mathit{dom}(\Delta) \quad \mathit{fn}(\mathsf{t}) \in \mathcal{N}}{\Delta \vdash^{\mathcal{N}} (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\mathsf{t}}}$$

Figure 3.5: Well-formed types

We now define our type system by means of a typing relation.

Definition 20 (Type System) Typing is expressed by the judgment $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, stating that expression e is well-typed by τ^{s} in environment Δ, given constraints in $\mathcal{S}$, and concrete context security level r.

The type system asserts, through a set of typing rules, that an expression is well-typed. We omit the ones already discussed for λ_{RCV} in Chapter 2, and present the typing rules that are new or modified in Figure 3.6. Like in the type system presented for λ_{RCV}, our analysis also relies on a simple subtyping relation, denoted $<:$, which allows up-classification of security labels.

Definition 21 (Subtyping Relation) Our subtyping relation is expressed as $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ and is defined by the rules given in Figure 3.4.

Notice that we introduce a couple of subtyping rules, rules (S-INDEXLEFT) and (S-INDEXRIGHT), with respect to the ones previously presented.

Rules (S-INDEXLEFT), (S-INDEXRIGHT), and (S-EXPR) rely on the lattice order, where we consider $\mathsf{s} \leq \mathsf{s}'$ to be an instance of a lattice assertion, in rule (S-EXPR) we assume τ not to be dependent type nor a variant type.

$$\frac{\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \sigma^{\mathsf{q}}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x : \tau^{\mathsf{s}}).e : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\perp}} \text{(T-LAMBDA)}$$

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'\sigma^{\mathsf{q}})^{\mathsf{t}} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}} \quad (\mathsf{v} = \top) \vee (\mathcal{S}\{x \doteq e_2\} \models x \doteq \mathsf{v}) \quad \mathsf{r} \leq \mathsf{r}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1(e_2) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}} \text{(T-APP)}$$

$$\frac{\forall_i \, \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i{=}e_i, \ldots] : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\sqcap |\mathsf{s}_i|^{\downarrow}}} \text{(T-RECORD)}$$

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}} \quad \mathcal{S}\{x \doteq e\} \models x.m_j \doteq \mathsf{v}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}}} \text{(T-REFINERECORD)}$$

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}} \quad \mathcal{S}\{x \doteq e\} \models x.m_j \doteq \mathsf{v}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}}} \text{(T-UNREFINERECORD)}$$

Figure 3.6: Typing Rules

The idea behind rule (S-INDEXLEFT) is since we know $\ell(\perp) \leq \ell(v) \leq \ell(\top)$ holds in the security lattice for any value v (as stated earlier in Section 3.1.2), and if we have $\ell(\top) \leq \mathsf{s}$ then it also holds that any instance of the value-dependent security label $\ell(\mathsf{v})$ is $\leq$ than security label s. Therefore, we can raise value-dependent security label $\ell(\mathsf{v})$ to security label s. Likewise for rule (S-INDEXRIGHT). The remaining rules are the expected and were already discussed in Chapter 2.

We will now discuss the relevant typing rules for the type system of λ_{DIFT}. Like before, since base types play no role in our analysis, we will omit them in our examples for presentation purposes.

Dependent function types are introduced via rule (T-LAMBDA)

$$\frac{\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \sigma^{\mathsf{q}}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x : \tau^{\mathsf{s}}).e : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\perp}} \text{(T-LAMBDA)}$$

where, as stated previously, x may occur in σ^{q}.

Now rule (T-APP)

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}} \\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}} \\ (\mathsf{v} = \top) \vee (\mathcal{S}\{x \doteq e_2\} \models x \doteq \mathsf{v}) \quad \mathsf{r} \leq \mathsf{r}'\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1(e_2) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}} \text{ (T-APP)}$$

is the expected rule for any value dependent function application where free occurrences of x in the result type are replaced with a value v.

In our system we approximate the argument value v of e_2 via constraint entailment given the additional knowledge $x \doteq e_2$, otherwise we set $\mathsf{v} = \top$.

Although rules (T-LET) and (T-IF) are as expected, they play a key role in collecting constraints in our system – which may be used to approximate runtime values.
So rule (T-LET)

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}} \\ \Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq e_1\}} e_2 : \tau'^{\mathsf{s}'}\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau'^{\mathsf{s}'}} \text{ (T-LET)}$$

collects the binding of variable x to expression e_1 in a constraint $\{x \doteq e_1\}$ that is added to the constraint set $\mathcal{S}$ only if expression e_1 is pure. Otherwise the operation $\mathcal{S}\{x \doteq e_1\}$ returns the constraint set $\mathcal{S}$ unmodified.

In rule (T-IF)

$$\frac{\begin{array}{c}\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} c : \texttt{Bool}^{\mathsf{s}} \\ \Delta \vdash^{\mathsf{r} \sqcup \mathsf{s}}_{\mathcal{S} \cup \{c \doteq \texttt{true}\}} e_1 : \tau^{\mathsf{s}} \\ \Delta \vdash^{\mathsf{r} \sqcup \mathsf{s}}_{\mathcal{S} \cup \{c \doteq \texttt{false}\}} e_2 : \tau^{\mathsf{s}}\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\mathsf{s}}} \text{ (T-IF)}$$

in order to prevent implicit flows from occurring, we raise the security level of the computational context to the least upper bound of its current level with the logical condition's security level. Moreover, we enforce the security level of both branches and the logical condition to be the same, and track knowledge to the constraint set $\mathcal{S}$ about the condition's value in each branch.

Recall that in λ_{RCV}'s type system we require a record type to have a security label. This has to do with our decision of treating all types uniformly - all types have a security label - and such is reflected in our typing rules. The same goes for dependent sum types.

So our rule (T-RECORD) introduces dependent sum types,

$$\frac{\forall_i \; \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i{=}e_i, \ldots] : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\sqcap |\mathsf{s}_i|^{\downarrow}}} \text{ (T-RECORD)}$$

requiring the security label of record values to be, at most, the greatest lower bound of all the security labels occurring in their fields, otherwise implicit flows could occur on assignments of record values. We have seen examples of such in Chapter 2.

Since a field's security label s may have dependencies, we approximate the values they denote with $\bot$, via operation $|\mathsf{s}|^{\downarrow}$. Notice that this allows (but does not force) records storing both private and public data to be classified as public. Such a scenario is in fact, secure, as will only leak, at most, information that a record is present, but not the field contents (except those classified as public). We refer back to the discussion in Chapter 2.

Rules (T-REFINERECORD) and (T-UNREFINERECORD), adequate to our dependent labeled sum types, correspond to introduction and elimination rule for (value-dependent) existential types.

Rule (T-REFINERECORD)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}} \qquad \mathcal{S}\{x \doteq e\} \models x.m_j \doteq \mathsf{v}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}}} \text{ (T-REFINERECORD)}$$

potentially introduces a dependent sum type by indexing a label with field m_j, given that a concrete witness value v can be identified from m_j via constraint entailment.

The converse is achieved with rule (T-UNREFINERECORD),

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}} \qquad \mathcal{S}\{x \doteq e\} \models x.m_j \doteq \mathsf{v}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}}} \text{ (T-UNREFINERECORD)}$$

that is, one may eliminate a field dependency (and potentially a dependent sum type) by replacing such a field with a concrete value witness, derivable as discussed for the (T-REFINERECORD) rule.

We illustrate with some examples.

Example 21 Recall the `viewAuthorPapers` from Chapter 1

```
let viewAuthorPapers = λ (u).
  foreach(x in !Submissions) with y = {} do
    let tuple = !x in
      if tuple.uid = u then tuple::y else y
```

When typing expression `tuple::y`, while typing the **then** branch, we obtain type

$$\Sigma[\texttt{uid:}\bot \times \texttt{sid:}\bot \times \texttt{title: A(uid,sid)} \times \texttt{abs: A(uid,sid)} \times \texttt{paper: A(uid,sid)}]^{*\bot}$$

However, at this point, we know that `tuple.uid = u`, which was added to the constraint set $\mathcal{S}$ according to rule (T-IF).

So, to type `tuple`, we apply rule (T-UNREFINERECORD), adding a new constraint $\{x \doteq \text{tuple}\}$ for a fresh identifier x, and entail

$$\mathcal{S} \cup \{\texttt{tuple.uid=u} \doteq \texttt{true}, x \doteq \texttt{tuple}\} \models x\texttt{.uid} \doteq \texttt{u}$$

to eliminate the field dependency `uid` in the security label, obtaining the type

$$\Sigma[\texttt{uid}:\bot\times\texttt{sid}:\bot\times\texttt{title:A(u,sid)}\times\texttt{abs:A(u,sid)}\times\texttt{paper:A(u,sid)}]^{\bot}$$

Finally, in both branches, `y` is typed as the collection type with element type the dependent sum type above (since we are adding `tuple` to `y` and the conditional branches must have the same type). So function `viewAuthorPapers` is assigned type

$$\Pi(\texttt{u}:\bot).\Sigma[\texttt{uid}:\bot\times\ \texttt{sid}:\bot\times\ \texttt{title:A(u,sid)}\times\ \ldots]^{*\bot}$$

Example 22 We now refer back to Example 5. For clarity, we abbreviate dependent sum types and mention only the record fields relevant for the discussion.

```
let addCommentSubmission = λ(uid_r: ⊥, sid_r: ⊥).
  foreach (p in viewAssignedPapers(uid_r)) with _ do
    if p.sid = sid_r then
      foreach(y in !Reviews) with _ do
        let t_rev = !y in
          if t_rev.sid = p.sid then
           let up_rec =
              [uid=t_rev.uid,
               PC_only=comment(p.uid,p.sid,p),...]
           in y := up_rec
```

To typecheck `let up_rec= [uid=t_rev.uid, PC_only=`*comment*`(p.uid,p.sid,p),...] in y:= up_rec`, we begin with typechecking the record value for identifier `up_rec` and then, in the scope of the let-declaration, we need to type `up_rec` with the declared type for the elements of collection `Reviews` (denoted as δ), which we have seen in Chapter 1 to be

$$\Sigma[\texttt{uid}:\bot\times\texttt{sid}:\bot\times\texttt{PC\_only:PC(uid,sid)}\times\texttt{review:A}(\top,\texttt{sid})\times\texttt{grade:A}(\top,\texttt{sid})]^{\bot}$$

We also know identifier `p` has type $\Sigma[\texttt{uid}:\bot\times\texttt{sid}:\bot\times\ldots\times\texttt{title:A(uid,sid)}]^{\bot}$.

The type for the application *comment*`(p.uid,p.sid,p)` has level `A(p.uid,t_rev.sid)` but, in order for the dependent sum type of the record value used in the let-declaration to be well-formed, we must either introduce dependency `uid` in field `PC_only` or approximate, by subtyping, to $\top$.

However, we cannot refine the dependent sum type with the given constraint set, so we approximate the dependency `p.uid` to $\top$ and the application *comment*`(p.uid,p.sid,p)` has type `A(`$\top$`,t_rev.sid)`.

Afterwards, while typing expression `up_rec`, we must obtain the expected type for contents of reference `y`, which is δ.

To do so we first apply subtyping rule (S-INDEXRIGHT), since we have

$$\texttt{A}(\top,\texttt{t\_rev.sid})\leq\texttt{PC(uid,t\_rev.sid))}$$

and get type `PC(uid,t_rev.sid)` for record field `PC_only`.

The last step consists in refining the type for record field `PC_only` in order to match the expected type in type δ. So we refine the type of the `PC_only` field to `PC(uid,sid)`, by applying (T-REFINERECORD), since we know

$$\{\texttt{up\_rec} \doteq \texttt{[uid=t\_rev.uid, PC\_only=}\mathit{comment}\texttt{(p.uid,p.sid,p),...]}\}$$

so, by adding constraint $\{x \doteq \texttt{up\_rec}\}$ (x is fresh), we can entail $x.\texttt{sid} \doteq \texttt{t\_rev.sid}$.

Thus we obtain type δ and can typecheck the assignment operation.

By allowing the (T-REFINERECORD) and (T-UNREFINERECORD) rules to approximate the security label to a field identifier of another record, as we just did in Example 22, we retrieve essential precision in our analysis, required to obtain the correct typing for `PC_only`, `PC(uid,sid)`, and to typecheck function `addCommentSubmission`.

Rule (T-FIELD) is the expected for field projection. Notice that, since the security lattice is formed by *concrete* security labels, if we type a projection of a field whose security label has a dependency then it will be eliminated either via (T-UNREFINERECORD) or via (T-SUB) before applying rule (T-FIELD).

Next, we illustrate some of our key typing rules with some typing derivations.

3.1.5.1 Examples of Typing Derivations

We show some typing derivations to illustrate some of our rules. For the sake of presentation, we omit basic types `bool` and `cmd`, and only mention those that may play a key role in the application of a typing rule (e.g. dependent sum and dependent function types) as well as type's security labels.

We begin with rules (T-APP) and (T-LET) to show how we collect information in our system and later apply it to approximate runtime values, namely in the case of a dependent function application. We avoid showing all the premises of a rule, for brevity sake, and only show those that are illustrative for the given example.

Example 23 Let us go back to example Example 21,

```
let viewAuthorPapers = λ (uid_a).
  foreach(x in !Submissions) with y = {} do
    let tuple = !x in
      if tuple.uid = uid_a then tuple::y else y
in let id = 42 in viewAuthorPapers(id)
```

Function `viewAuthorPapers`, as we have previously seen, has type

$$(\Pi(\texttt{uid\_a:}\bot).\Sigma[\texttt{uid:}\bot \times \texttt{sid:}\bot \times ... \times \texttt{paper:A(uid\_a,sid)}]^{*\bot})^{\bot}$$

For presentation purposes, we define $\mathcal{S}' = \mathcal{S} \cup \{\texttt{id} \doteq 42\}$.
So the derivation that types the last let-declaration is the following

1. Δ,id:$\bot \vdash^{\mathsf{r}}_{\mathcal{S}'}$viewAuthorPapers:
$(\Pi(\texttt{uid\_a}{:}\bot).\Sigma[\texttt{uid}{:}\bot \times \texttt{sid}{:}\bot \times \ldots \times \texttt{paper}{:}\texttt{A(uid\_a,sid)}]^{*\bot})^{\bot}$
by (T-ID)

2. Δ,id:$\bot \vdash^{\mathsf{r}}_{\mathcal{S}'}$id: $\bot$
by (T-ID)

3. $(\mathtt{v} = \top) \vee \mathcal{S}'\{\texttt{uid\_a} \doteq \texttt{id}\} \models \texttt{uid\_a} \doteq \mathtt{v}$
such that $\mathcal{S}'\{\texttt{uid\_a} \doteq \texttt{id}\} \models \texttt{uid\_a} \doteq 42$ thus $\mathtt{v} = 42$

---

4. Δ,id:$\bot \vdash^{\mathsf{r}}_{\mathcal{S}'}$viewAuthorPapers(id):
$(\Sigma[\texttt{uid}{:}\bot \times \texttt{sid}{:}\bot \times \ldots \times \texttt{paper}{:}\texttt{A(uid\_a,sid)}])\{^{\mathtt{v}}/_{\texttt{uid\_a}}\}^{*\bot\{^{\mathtt{v}}/_{\texttt{uid\_a}}\}\sqcup\bot}$
by (T-APP), 1, 2, 3

5. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} 42$: $\bot$
by (T-NUM)

---

6. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ **let** id = 42 **in** viewAuthorPapers(id) :
$(\Sigma[\texttt{uid}{:}\bot \times \texttt{sid}{:}\bot \times \ldots \times \texttt{paper}{:}\texttt{A(42,sid)}])^{*\bot}$
by (T-LET), 4, 5

Notice that had we not gathered constraint $\{\texttt{id} \doteq 42\}$, then we could not entail $\texttt{uid\_a} \doteq 42$. So, in that case, $\mathtt{v} = \top$ and we would have obtained the less concrete type

$$\Sigma[\texttt{uid}{:}\bot \times \texttt{sid}{:}\bot \times \ldots \times \texttt{paper}{:}\texttt{A}(\top,\texttt{sid})]^{*\bot}$$

Next, we will show how our system disallows insecure assignments via rule (T-ASSIGN).

Example 24 Recall Example 16

```
let cond = ref_{bool^⊤} true in
 let r = ref_{Σ[a:⊥×b:⊤]^⊥} [a = 0, b = 1] in
   if !cond then
     r := [a = 2, b= 2]
```

As we explained earlier, this code snippet must be deemed insecure because the assignment operation depends on data classified at a security level higher than the security level of some of the data stored in reference r.
Let us see the derivation of this snippet, take into account the following
$\Delta' = \Delta, \texttt{cond} : \texttt{ref}(\texttt{bool}^{\top})^{\bot}$,
$\mathcal{S}' = \mathcal{S}$ since expression **ref**$_{\texttt{bool}^{\top}}$ **true** is not pure,
$\Delta'' = \Delta', \mathtt{r} : \texttt{ref}(\Sigma[\mathtt{a} : \bot \times \mathtt{b} : \top]^{\bot})^{\bot}$, and
$\mathcal{S}'' = \mathcal{S}'$ since a reference expression is not pure.

1. (...)
   we omit derivation that raises **`true`** security label

---

2. $\Delta \vdash^{\perp}_{\mathcal{S}}$ **`true`** : $\texttt{bool}^{\top}$
   by (T-SUB), 1

---

3. $\Delta \vdash^{\perp}_{\mathcal{S}}$ $\textbf{ref}_{\texttt{bool}^{\top}}$ **`true`** : $\texttt{ref}(\texttt{bool}^{\top})^{\perp}$
   by (T-REF), 2

4. (...)
   we omit derivation that raises `[a = 0, b = 1]` security labels

---

5. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ `[a = 0, b = 1]`: $\Sigma[\texttt{a} : \perp \times \texttt{b} : \top]^{\perp}$
   by (T-SUB), 4

6. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ $\textbf{ref}_{\Sigma[\texttt{a}:\perp\times\texttt{b}:\top]^{\perp}}$ `[a = 0, b = 1]`: $\texttt{ref}(\Sigma[\texttt{a} : \perp \times \texttt{b} : \top]^{\perp})^{\perp}$
   by (T-REF), 5

7. (...)
   we omit derivation for logical expression `!cond`

8. $\Delta'' \vdash^{\top}_{\mathcal{S}''}$ `r`: $\texttt{ref}(\Sigma[\texttt{a} : \perp \times \texttt{b} : \top]^{\perp})^{\perp}$
   by (T-ID)

9. (...)
   we omit derivation raising `[a = 2, b= 2]` security labels

10. $\Delta'' \vdash^{\top}_{\mathcal{S}''}$ `[a = 2, b= 2]`: $\Sigma[\texttt{a} : \perp \times \texttt{b} : \top]^{\perp}$
    by (T-SUB), 9

11. $\top \sqcup \perp \leq \perp$
    side-condition $\texttt{r} \sqcup \texttt{s}' \leq \texttt{s}$ FAILS!

12. $\Delta'' \vdash^{\perp \sqcup \top}_{\mathcal{S}''}$ `r := [a = 2, b= 2]`: $\texttt{cmd}^{\perp}$
    by (T-ASSIGN),8,10,11 but TYPECHECK FAILED in 11

13. $\Delta'' \vdash^{\perp}_{\mathcal{S}''}$ **`if`** `!cond` **`then`** `r := [a = 2, b= 2]`: $\texttt{cmd}^{\perp}$
    by (T-IF),7,12 but TYPECHECK FAILED in 12

---

14. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ **`let`** `r =` $\textbf{ref}_{\Sigma[\texttt{a}:\perp\times\texttt{b}:\top]^{\perp}}$ `[a = 0, b = 1]` **`in`**
    `if` `!cond` **`then`** `r := [a = 2, b= 2]`: $\texttt{cmd}^{\perp}$
    by (T-LET),6,13 but TYPECHECK FAILED in 13

---

15. $\Delta \vdash^{\perp}_{\mathcal{S}}$ **let** cond = **ref**$_{\texttt{bool}^{\top}}$ **true in**
let r = **ref**$_{\Sigma[\texttt{a}:\perp \times \texttt{b}:\top]^{\perp}}$ [a = 0, b = 1] **in**
if !cond **then** r := [a = 2, b= 2]: $\texttt{cmd}^{\perp}$
by (T-LET), 3, 14 but TYPECHECK FAILED in 14

Because the logical expression in the conditional has security level $\top$, we raised the computational security level to $\top$ (as result of $\top \sqcup \perp$) when typing the branch of the conditional. Then, when we attempt to apply rule (T-ASSIGN), we check if condition $\top \sqcup \perp \leq \perp$ – corresponding to premise $\texttt{r} \sqcup \texttt{s}' \leq \texttt{s}$ in the rule – holds. Since it does not hold, our analysis deems the program insecure.

We end this section with a typing derivation that relies on rule (T-REFINERECORD), exemplifying how we introduce dependencies in dependent sum types.

Example 25 Let us refer back to Example 22.

```
let addCommentSubmission = λ(uid_r: ⊥, sid_r: ⊥).
  foreach (p in viewAssignedPapers(uid_r)) with _ do
    if p.sid = sid_r then
      foreach(y in !Reviews) with _ do
        let t_rev = !y in
          if t_rev.sid = p.sid then
            let up_rec =
              [uid=t_rev.uid,
              PC_only=comment(p.uid,p.sid,p), ...]
            in y := up_rec
```

Recall the type for the elements of collection `Reviews` (denoted δ from now on)

$$\Sigma[\texttt{uid}:\perp \times \texttt{sid}:\perp \times \texttt{PC\_only}:\texttt{PC(uid,sid)} \times \texttt{review}:\texttt{A}(\top,\texttt{sid}) \times \texttt{grade}:\texttt{A}(\top,\texttt{sid})]^{\perp}$$

We also know identifier `p` has type $\Sigma[\texttt{uid}:\perp \times \texttt{sid}:\perp \times \ldots \times \texttt{title}:\texttt{A(uid,sid)}]^{\perp}$ denoted as v, and *comment* is a dependent function of type $\Pi \texttt{u}:\perp.\Pi \texttt{s}:\perp.\Pi \texttt{r}:v.\texttt{A(u,s)}$.

For the sake of presentation, we assume the extension of rules (T-LAMBDA) and (T-APP) for multiple parameters/arguments.

Let us discuss the derivation of the last let-declaration in the above snippet, where:
$\{\texttt{p}:v,\ \texttt{y}:\texttt{ref}(\delta)^{\perp},\ \texttt{t\_rev}:\delta\} \subseteq \Delta$,
$\{\texttt{p.sid=sid\_r} \doteq \texttt{true},\ \texttt{t\_rev.sid=p.sid} \doteq \texttt{true}\} \subseteq \mathcal{S}$,
$\mathcal{S}' = \mathcal{S} \cup \{\texttt{up\_rec} \doteq \texttt{[uid=t\_rev.uid, PC\_only=}\textit{comment}\texttt{(p.uid,p.sid,p),...]}\}$,
$\Delta' = \Delta$,$\texttt{up\_rec}: \Sigma[\texttt{uid}:\perp \times \texttt{sid}:\perp \times \texttt{PC\_only: A}(\top,\texttt{t\_rev.sid}) \times$
$\texttt{review: A}(\top,\texttt{sid}) \times \texttt{grade: A}(\top,\texttt{sid})]^{\perp}$.

1. $\Delta \vdash^{\perp}_{\mathcal{S}}$ *comment* : Πu:$\perp$.Πs:$\perp$.Πr:v.A(u,s)
   by (T-ID)
2. $\Delta \vdash^{\perp}_{\mathcal{S}}$ p.uid : $\perp$
   by (T-FIELD)
3. $\Delta \vdash^{\perp}_{\mathcal{S}}$ p.sid : $\perp$
   by (T-FIELD)
4. $\Delta \vdash^{\perp}_{\mathcal{S}}$ p : v
   by (T-ID)
5. $\mathcal{S} \cup \{\mathtt{u} \doteq \mathtt{p.uid}\} \models \mathtt{u} \doteq \mathtt{p.uid}$
6. $\mathcal{S} \cup \{\mathtt{s} \doteq \mathtt{p.sid}\} \models \mathtt{s} \doteq \mathtt{t\_rev.sid}$
7. $\mathcal{S} \cup \{\mathtt{r} \doteq \mathtt{p}\} \models \mathtt{r} \doteq \mathtt{p}$

---

8. $\Delta \vdash^{\perp}_{\mathcal{S}}$ *comment*(p.uid,p.sid,p): A(u,s)$\{^{\mathtt{p.uid}}/_{\mathtt{u}}\}\{^{\mathtt{t\_rev.sid}}/_{\mathtt{s}}\}$
   by (T-APP),1,2,3,4,5,6,7
9. A(p.uid,t_rev.sid)$\leq$ A($\top$,t_rev.sid)
10. $\Delta \vdash^{\perp}_{\mathcal{S}}$ *comment*(p.uid,p.sid,p): A($\top$,t_rev.sid)
    by (T-SUB), 8, 9
11. (...)
    we omit derivation for fields uid, sid, review, and grade

---

12. $\Delta \vdash^{\perp}_{\mathcal{S}}$ [uid=t_rev.uid, PC_only=*comment*(p.uid,p.sid,p), ...]:
    Σ[uid:$\perp$ $\times$ sid:$\perp$ $\times$ PC_only: A($\top$,t_rev.sid) $\times$
    review: A($\top$,sid) $\times$ grade: A($\top$,sid)]$^{\perp}$
    by (T-RECORD), 10, 11
13. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ y: ref$(\delta)^{\perp}$
    by (T-ID)
14. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ up_rec:
    Σ[uid:$\perp$ $\times$ sid:$\perp$ $\times$ PC_only: A($\top$,t_rev.sid) $\times$
    review: A($\top$,sid) $\times$ grade: A($\top$,sid)]$^{\perp}$
    by (T-ID)
15. A($\top$,t_rev.sid) $\leq$ PC(uid,t_rev.sid)
    by lattice assertion $\forall_{\mathtt{uid_1,uid_2,sid}}$ A(uid$_1$,sid) $\leq$ PC(uid$_2$,sid)

---

16. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ `up_rec`:
Σ[`uid:`$\perp$ $\times$ `sid:`$\perp$ $\times$ `PC_only: PC(uid,t_rev.sid)` $\times$
`review: A(`$\top$`,sid)` $\times$ `grade: A(`$\top$`,sid)`]$^{\perp}$
by (T-SUB),14,15

17. $\mathcal{S}' \cup \{x \doteq$`up_rec`$\} \models x.$`sid`$\doteq$`t_rev.sid`

---

18. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ `up_rec`:
Σ[`uid:`$\perp$ $\times$ `sid:`$\perp$ $\times$ `PC_only: PC(uid,sid)` $\times$
`review: A(`$\top$`,sid)` $\times$ `grade: A(`$\top$`,sid)`]$^{\perp}$
by (T-REFINERECORD),16, 17

19. $\perp \sqcup \perp \leq \perp$

---

20. $\Delta' \vdash^{\perp}_{\mathcal{S}'}$ `y := up_rec`: `cmd`$^{\perp}$
by (T-ASSIGN),13, 18, 19

---

21. $\Delta \vdash^{\perp}_{\mathcal{S}}$ **`let`** `up_rec= [uid=t_rev.uid, PC_only=`*comment*`(p.uid,p.sid,p), ...]`
`in` `y := up_rec` : `cmd`$^{\perp}$
by (T-LET),12, 19

Since `A(p.uid,t_rev.sid)` is not well-formed because dependency `p.uid` is not related to any field of the dependent record where the field dependency occurs, we have to apply subtyping to raise `A(p.uid,t_rev.sid)` to `A(`$\top$`,t_rev.sid)` (step 10), while typing the record value to be associated to `up_rec`.
Also, as we have seen before, the security lattice has assertion

$$\forall_{\texttt{uid}_1,\texttt{uid}_2,\texttt{sid}} \texttt{A(uid}_1\texttt{,sid)} \leq \texttt{PC(uid}_2\texttt{,sid)}$$

so, while typing expression `up_rec` we must obtain the expected type for contents of reference `y`, which is δ.

To do so we first apply subtyping rule (S-INDEXRIGHT) – via typing rule (T-SUB)–, since we have `A(`$\top$`,t_rev.sid)` $\leq$ `PC(uid,t_rev.sid))`, and get type `PC(uid,t_rev.sid)` for record field `PC_only`.

Then, the only type that does not match the expected type δ is the type for field `PC_only` because its security label is indexed by `t_rev.sid` instead of field `sid`.

So we refine the type of the `PC_only` field to `PC(uid,sid)`, by applying (T-REFINERECORD), since we know $\{$`up_rec` $\doteq$ `[uid=t_rev.uid, PC_only=`*comment*`(p.uid,p.sid,p), ...]`$\}$, then we can entail the projection of their fields.Namely $\{$`up_rec.sid`$\doteq$ `t_rev.sid`$\}$, and, finally, by adding constraint $\{x \doteq$ `up_rec`$\}$ (x is fresh) we can entail $x.$`sid` $\doteq$ `t_rev.sid`.

Thus we obtain type δ and can typecheck the assignment operation

We proceed by showing our dependent information flow type system is safe in the following section.

3.1.6 Type Safety

We now present our main technical results: Theorem 3 (Type Preservation) - types are preserved by the reduction relation; Theorem 4 (Progress) - well-typed expressions are either a value or have a reduction step; and Theorem 5 (Non-interference) - well-typed expressions preserve non-interference. We will defer the presentation of the soundness result, non-interference, to the next chapter and focus on the standard safety results on this section.

We start by introducing some preliminary definitions. These are essentially the same presented for $\lambda_{\tau RCV}$ but extended to include the constraint set $\mathcal{S}$ in the judgments.

Namely, we introduce notions of store consistency and well-typed configurations. We say that a store S is well-typed with relation to a typing environment Δ if the values referred by its locations have the expected type. We define the typing of stores as follows:

Definition 22 (Store Consistency) Let Δ be a typing environment and S a store, we say store S is consistent with respect to typing environment Δ, denoted as $\Delta \vdash S$, if $dom(S) \subseteq dom(\Delta)$ and $\forall l \in dom(S)$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} S(l) : \Delta(l)$.

From the store consistency definition, we define what it means for a configuration to be well-typed.

Definition 23 (Well-typed Configuration)
A configuration $(S; e)$ is well-typed in typing environment Δ if $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$.

So a configuration $(S;\ e)$ is well-typed if there is a typing environment Δ that types both the store and the expression, for an arbitrary computational context.

To prove type preservation, we introduce the substitution lemma on which it relies. This lemma states that the type of an expression is preserved under substitution.

Lemma 2 (Substitution Lemma)
If $\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$ and $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$.
Proof: Induction on the derivation of $\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$ (see Appendix B.1, Lemma 12).

Theorem 3 says that well-typed configurations remain well-typed after a reduction step, and possibly the final configuration is extended with new locations in the state.

Theorem 3 (Type Preservation)
Let $fv(e) \cup fv(\tau^{\mathsf{s}}) = \emptyset$, $vars(\Delta) = \emptyset$, $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$.
If $(S; e) \longrightarrow (S'; e')$ then there is Δ' such that $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau^{\mathsf{s}}$, $\Delta' \vdash S'$ and $\Delta \subseteq \Delta'$.
Proof: Induction on the derivation of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$, (see Appendix B.1, Theorem 6).

Theorem 4, states that well-typed programs never get stuck.

Theorem 4 (Progress)
Let $\Delta \vdash^{r}_{\mathcal{S}} e{:}\tau^{s}$ and $\Delta \vdash S$. If e is not a value then $(S; e) \longrightarrow (S'; e')$.
Proof: Induction on the derivation of $\Delta \vdash^{r}_{\mathcal{S}} e : \tau^{s}$, (see Appendix B.1, Theorem 7).

These theorems ensure that our semantics preserves typability and well-typed programs never get stuck, thus making our type system safe. However, the soundness result, with respect to our information flow analysis, is noninterference (Definition 5).

Thus noninterference together with Theorem 3 and Theorem 4, establishes that our system ensures well-typed programs do not leak confidential information under the security policy prescribed by the assumed security lattice. In other words, data does not flow from a security compartment to another if they are unrelated or if it is a down-flow in the security lattice.

3.2 Remarks

In this chapter, we have introduced our *novel* dependent information flow types. We began by presenting an extension of the core language presented in Chapter 2, λ_{DIFT}, that accommodates in its abstract syntax of types our dependent function and sum types as well as our value-dependent security labels.

We discussed some challenges that value-dependency in security labels brings to our analysis and presented our type system. To illustrate some of our system's key rules, we presented typing derivations of some of the relevant examples already discussed in Chapter 1. We proceeded with the presentation of our type safety results, full proofs can be found in Appendix B.

We now point out some of the relevant related work.

3.2.1 Related Work

The works that most relate to ours are those that have stateful security policies in the context of static information flow analysis. Some we have already presented in Chapter 1: [46, 60, 61].

As we have stated previously, in Fine [60], in order to express an information flow analysis, the programmer is required to one has to hardcode the security labels as well as the lattice and all its operations/axioms (meet, join, partial order relation, etc) into inductive types and logic formulae within a module that internalizes the intended information flow policy inside the framework.

In that sense, our approach is more lightweight – adopting a simple and primitive notion of value dependent classification directly at the level of the type structure, leading to an absolute non-interference theorem – and very expressive for a stateful static information flow analysis.

Moreover, they show a value abstraction result which is different from the the (standard) notion of noninterference used in our work. Their result states that code within a module does not interfere with another module's protected code, not primitively and explicitly addressing the fundamental notion of value dependent classification through dependent typing, which is the core contribution of our work.

In [46], Nanevski *et al.*, use a very expressive relational Hoare type theory (RHTT) to reason about access control and information flow in stateful programs. Besides standard dependent types, this work introduces a special dependent type, `STsec`, to specify security policies via pre and post-conditions, using higher-order logic formulae capable of expressing heap union disjointness.

The `STsec` type is used to type potentially side-effectful operations, but the relevant part *w.r.t.* to information flow analysis is the post-condition that specifies the behaviour of two different runs of the program, relating the outputs, input heaps and output heaps of any two terminating executions of the program.

Another interesting work, based on [61] and [46], is RF* [5], where the authors introduce the notion of relational refinement types. The key idea of relational refinement types consists in extending classic refinement types to relational formulae, which in turn enables to relate the left and right value of every program variable in scope through projections L and R. With this setting, the author's type system is able to relate expressions at a relational refined type that can describe the results of both expressions.

A distinguishing feature of these latter approaches is that data is not classified with security labels (as expected from traditional information flow analyses). Instead, and similarly to the approach of [60], the noninterference property is expressed directly in the post-condition via detailed assertions that relate the initial heap with the final heap as well as the output values for any two runs of the program.

While it might be conceivable, in principle, to express value-dependent information flow policies in such a framework, and in fact, in any sufficiently expressive logical framework for imperative programs supporting general functional properties, the goal of our work follows a much lightweight and tractable type-based approach, and aims to isolate and address in a direct and explicit way the core notion of value dependent information classification.

A concept of indexed security label was introduced in [35], as an useful feature to express security policies in a DSL with high-level monolithic data manipulation operations, much less expressive than what we achieve in this paper.

The developments in this thesis put forward, in a principled way, the notion of data/state dependent information flow in terms of a fairly canonical dependent type theory with first-order sum and arrow types, defined by a set of simple type rules, and for a very parsimonious λ-calculus with references and collections. We believe that our framework, or possible extensions of it, may be used as a suitable foundation for studying information flow data dependency in a broad sense.

To the best of our knowledge, dependent information flow types in the sense introduced here, leading to a general non-interference theorem, are novel; we have no perspective on how to conveniently and precisely express valued dependent security classification in existing dependent type systems.

We proceed in the next chapter with the formulation of our noninterference result and a proof outline of its proof.

CHAPTER 4

FORMULATION OF NONINTERFERENCE

In this chapter we present our main soundness result, the noninterference theorem, and introduce some auxiliary definitions, namely equivalence relations over stores and expressions. We also illustrate how noninterference is interpreted and how it ensures well-typed programs do not violate data confidentiality prescribed by the given security lattice.

4.1 Store and Expression Equivalence Relations

To develop our noninterference theorem, we now introduce some relevant concepts.

We begin with the relation of store equivalence up to a security level s that relies on a couple of auxiliary definitions.

The first definition $\mathit{filterValue}(\tau^{\mathsf{s}'}, v, \mathsf{s})$ redacts (replacing with a dummy value $\star$) all values in value v whose security level is above or incomparable to a given security level s.

Definition 24 (Filter Value) Let v be a value of security type $\tau^{\mathsf{s}'}$. We define a filter function, denoted as $\mathit{filterValue}(\tau^{\mathsf{s}'}, v, \mathsf{s})$, that redacts all the values occurring in value v whose security level is above a given security level s.

$\mathit{filterValue}(\tau^{\mathsf{s}'}, v, \mathsf{s}) \triangleq$
match τ **with**
$\Sigma[m_1{:}\tau_1^{\mathsf{s}_1} \times \ldots \times m_n{:}\tau_n^{\mathsf{s}_n}] \Rightarrow [m_1 = \mathit{filterValue}(\tau_1^{\mathsf{s}_1}, v.m_1, \mathsf{s}), \ldots, m_n = \mathit{filterValue}(\tau_n^{\mathsf{s}_n}, v.m_n, \mathsf{s})]$
$\tau^* \Rightarrow \{\ v_i' \mid v_i \in v \text{ and } v_i' = \mathit{filterValue}(\tau^{\mathsf{s}'}, v_i, \mathsf{s})\ \}$
$\mathtt{ref}(\tau'^{\mathsf{t}}) \Rightarrow \{\ \mathbf{ref}_{\tau''} u \mid v = \mathbf{ref}_{\tau''} v' \text{ and } u = \mathit{filterValue}(\tau'^{\mathsf{t}}, v', \mathsf{s})\ \}$
$\{\ldots, m : \tau'^{\mathsf{t}}, \ldots\} \Rightarrow \{\ \#m(u) \mid v = \#m(v') \text{ and } u = \mathit{filterValue}(\tau'^{\mathsf{t}}, v', \mathsf{s})\ \}$
$\_ \Rightarrow$ **if** $\mathsf{s}' \leq \mathsf{s}$ **then** v **else** $\star$

So, for e.g., if value $v = \{$`[public = 42, secret = 70], [public = 666, secret = 50]`$\}$ has type $\tau^{\mathsf{s}'} = \Sigma$`[public:int`$^{\bot}$ $\times$ `secret:int`$^{\top}$`]`$^{*\bot}$, then $\mathit{filterValue}(\tau^{\mathsf{s}'}, v, \bot)$ results in value $\{$`[public = 42, secret = `$\star$`], [public = 666, secret = `$\star$`]`$\}$.

Auxiliary function $redact(\Delta, S, \mathsf{s})$ returns the store obtained by "redacting" all stored values in S with security level higher than s, or incomparable with s.

Definition 25 (Filter Store) Let S be a well-typed store under typing environment Δ, that is $\Delta \vdash S$. We define a redact function, denoted as $redact(\Delta, S, \mathsf{s})$, that, for all stored values in the locations kept in S, redacts the values whose security level is above (or incomparable to) a given security level s.

$$redact(\Delta, S, \mathsf{s}) \triangleq \{l \mapsto v \mid v = filterValue(\Delta(l), S(l), \mathsf{s})\}$$

We can now present our store equivalence definition.

Intuitively, two well-typed stores S_1, S_2 are said to be equivalent up to level s, written $S_1 =_{\mathsf{s}} S_2$, if $redact(\Delta, S_1, \mathsf{s}) = redact(\Delta, S_2, \mathsf{s})$.

Definition 26 (Store Equivalence) Let S_1 and S_2 be two well-typed stores under typing environment Δ. We define that S_1 is equivalent to S_2 up to level s, denoted as $S_1 =_{\mathsf{s}} S_2$, if all values up to level s in S_1 are the same as the values up to level s in S_2.

$$S_1 =_{\mathsf{s}} S_2 \quad \text{iff} \quad redact(\Delta, S_1, \mathsf{s}) = redact(\Delta, S_2, \mathsf{s})$$

Let us see an example of equivalent stores followed up by one of non-equivalent stores.

Example 26 Assume `user(42)#user(666)`, and let S_1 and S_2 be stores well-typed under typing environment Δ, such that

$$\Delta = \{\texttt{private\_file: } \Sigma\texttt{[uid:}\bot\times\texttt{content:user(uid)]}^{*\bot}\}$$

```
S1(private_file) = {  [uid = 42, content = "walking debt"],
                      [uid = 666, content = "varoufakis"]  }
S2(private_file) = {  [uid = 42, content = "greek minister of awesome"],
                      [uid = 666, content = "varoufakis"]  }
```

We have $S_1 =_{\texttt{user(666)}} S_2$ since values `"walking debt"` and `"greek minister of awesome"`, classified as `user(42)`, are not visible at level `user(666)`, because

```
redact(Δ,S1,user(666)) = redact(Δ,S2,user(666)) =
      { [uid = 42, content = ⋆], [uid = 666, content = "varoufakis"] }
```

However, S_1 and S_2 are not equivalent at security level `user(42)` since values `"walking debt"` and `"greek minister of awesome"` are visible at level `user(42)`. In fact,

```
redact(Δ,S1,user(42)) =
 {[uid = 42, content = "walking debt"],
    [uid = 666, content = ⋆]}
redact(Δ,S2,user(42)) =
 {[uid = 42, content = "greek minister of awesome"],
    [uid = 666, content = ⋆]}
```

Thus, since *redact*(Δ,S_1,user(42)) $\neq$ *redact*(Δ,S_2,user(42)), we have $S_1 \neq_{\texttt{user(42)}} S_2$.

Useful to formulations of non-interference results is the introduction of a relation of expression equivalence, relating expressions at the same type and security level.

Technically, program expressions e_1 and e_2 are equivalent up to level s if they only differ in subexpressions classified at higher (or incomparable) security levels (being indistinguishable to attackers constrained to see only up to level s).

We say two expressions, e_1, e_2, are equivalent up to a security level s, asserted by $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2{:}\tau^{\mathsf{s}'}$, if they compute the same result under all stores equivalent up to s.

The constraint sets $\mathcal{S}_1$ and $\mathcal{S}_2$ are used to approximate runtime values for expression e_1 and e_2, respectively. Therefore, in our expression equivalence relation, to ensure we approximate to equivalent values, we always augment these constraint sets at the same time with equivalent expressions.

Apart from a few exceptions, mentioned below, our expression equivalence rules are similar to our typing rules for λ_{DIFT}. For that reason, we only discuss some of the more relevant rules of our expression equivalence relation.

So, for instance, in rule (E-APP)

$$\frac{\begin{array}{c}\text{(E-APP)}\\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}';\sigma^{\mathsf{q}})^{\mathsf{t}} \\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}} \quad \mathsf{r} \leq \mathsf{r}' \\ (\mathsf{v} = \top) \vee \\ (\mathcal{S}_1 \cup \{x \doteq e_2\} \models x \doteq \mathsf{v} \ \wedge \ \mathcal{S}_2 \cup \{x \doteq e_2'\} \models x \doteq \mathsf{v})\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1(e_2) \ \cong_{\mathsf{s}} e_1'(e_2') : (\sigma^{\mathsf{q}})\{^{v}/_{x}\}}$$

In order to relate two applications, we must have that the function being applied, e_1, e_1', is also equivalent up to security level s as well as the arguments, e_2, e_2'. We also must entail the same value v from the augmented constraint sets $\mathcal{S}_1$ and S_2.

Rules that may approximate runtime values via constraint entailment, like (E-APP), (E-REFINERECORD) and (E-UNREFINERECORD), must ensure that the values entailed also belong in the equivalence relation.

We proceed with the discussion of key rules for expression equivalence with no counterpart in our typing rules.

Rule (E-EXPROPAQUE) relates expressions e_1 and e_2 at security level s, given both the computational context r and expressions security levels s' are above the observational security level s.

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} e_1 : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} e_2 : \tau^{\mathsf{s}'} \quad \mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}} \text{(E-EXPROPAQUE)}$$

Intuitively, the rule states if one can only observe up to security level s, then any expressions classified at a higher or incomparable security level are indistinguishable. Since expressions can change the state via assignment operations, one must ensure the computational context (which is a lower bound on the security level of values altered) is above the observational security level.

In some cases, however, the condition imposed on the computational context can be too restrictive, so we have rule (E-VALOPAQUE) which only imposes that the security level of, potentially different, values s' be above the observational security level s.

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v_1 : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} v_2 : \tau^{\mathsf{s}'} \qquad \mathsf{s} < \mathsf{s}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau^{\mathsf{s}'}} \text{ (E-VALOPAQUE)}$$

Let us see an example to illustrate the difference between these two rules.

Example 27 Let e_1, e_2 be two well-typed expressions under typing environment Δ

$e_1 \stackrel{\text{def}}{=}$
```
let r = ref_int^⊤ 0 in
  if true then
    r := 10
  else r:= 5
```
$e_2 \stackrel{\text{def}}{=}$
```
let r = ref_int^⊤ 0 in
  if true then
    r := 6
  else r:= 2
```

such that $\Delta \vdash^{\bot}_{\mathcal{S}_1} e_1 : \mathtt{cmd}^{\bot}$ and $\Delta \vdash^{\bot}_{\mathcal{S}_2} e_2 : \mathtt{cmd}^{\bot}$.

In both code snippets we create a new reference for values classified at security level $\top$, initialising with value 0. Then, given a condition – which is the same for both codes – we update the reference with distinct values.

However, since these values must be classified at security level $\top$, they are not observable at security level $\bot$. Thus expressions e_1 and e_2 are equivalent up to security level $\bot$, that is $\Delta \vdash^{\bot}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\bot} e_2 : \mathtt{cmd}^{\bot}$.

Notice, though, that in both codes the computational context security level is $\bot$ so condition $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ does not hold and rule (E-EXPROPAQUE) cannot be applied.

So the only way to relate these expressions, with our expression equivalence relation, is to apply rules (E-LET), (E-REF), (E-IF), (E-ASSIGN), and (E-VALOPAQUE) – to relate the distinct integer values used in the assignment operations – instead of applying rule (E-EXPROPAQUE) for the assignment expressions.

Finally, rule (E-VAL) is applied whenever the security level of the values s' is below or equal to the observational security level s' so the values must be the same.

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} v : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} v : \tau^{\mathsf{s}'}} \text{ (E-VAL)}$$

Notice that two expressions may be equivalent up to level s even if they are typed at a different level s'.

We may now present our non-interference theorem.

4.2 Noninterference Theorem

In this section we show our noninterference result and the outline of its proof.

We start with auxiliary lemmas, used to prove noninterference.

Lemma 3 (Non-interference Step)
Let $vars(\Delta) = \emptyset$, $\Delta \vdash S_1$, $\Delta \vdash S_2$, $S_1 =_{\mathsf{s}} S_2$, $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$.
If $(S_1, e_1) \to (S_1', e_1')$ and $(S_2, e_2) \to (S_2', e_2')$, then exists Δ' such that $\Delta \subseteq \Delta'$, $\Delta' \vdash S_1'$, $\Delta' \vdash S_2'$, $S_1' =_{\mathsf{s}} S_2'$, $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}'}$.
Proof: By induction on the derivation of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$.

Lemma 3 states that if two equivalent programs can both perform a step under stores that differ only on information with higher (or incomparable) security level than s, then the resulting stores remain indistinguishable up to security level s, and the resulting program residuals remain equivalent at the same level.

We now discuss the proof sketch of this lemma.

Proof (Outline) We show the proof of cases (E-EXPROPAQUE) and (E-IF), the remaining cases can be found in Appendix B.2.

CASE (E-EXPROPAQUE):

We have as hypothesis
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$,
$\Delta \vdash S_1$, $\Delta \vdash S_2$, $S_1 =_{\mathsf{s}} S_2$,
$(S_1; e_1) \longrightarrow (S_1'; e_1')$, and $(S_2; e_2) \longrightarrow (S_2'; e_2')$.
Then by inversion of the expression equivalence relation on the hypothesis we obtain
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} e_1 : \tau^{\mathsf{s}'}$, $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} e_2 : \tau^{\mathsf{s}'}$, and $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$.
By subject reduction, Theorem 3, we also have $\Delta \subseteq \Delta'$, $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1} e_1' : \tau^{\mathsf{s}'}$, $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_2} e_2' : \tau^{\mathsf{s}'}$, $\Delta' \vdash S_1'$, and $\Delta' \vdash S_2'$.

We then apply rule (E-EXPROPAQUE) to get $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}'}$.
So, finally, we are left to prove the resulting stores S_1' and S_2' preserve the equivalence relation. In order to so we rely on another auxiliary lemma:

Lemma 4
Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}'}$, $\Delta \vdash S$, and $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$.
If $(S, e) \longrightarrow (S', e')$, then there is Δ' such that $\Delta \subseteq \Delta'$, $\Delta' \vdash S'$ and $S =_{\mathsf{s}} S'$.

stating whenever a well-typed expression e reduces, the resulting store S' is equivalent to the store under which the expression reduced, S, given that the security level of the

expression s', as well as the computational security level r, are above the observational security level s.

This auxiliary lemma is proved by induction on $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}'}$, since the computational context security level r is a lower bound on the effects an expression has on the store, it is straightforward to see that if by hypothesis we have $\mathsf{s} < \mathsf{r}$ then any effects expression e might produced in the store are not observable up to security level s.

Thus, to prove $S_1' =_{\mathsf{s}} S_2'$ we apply Lemma 4.

CASE (E-IF):

Here we have four sub-cases, two for when both expressions take the same branch of the conditional, and the remaining two when they diverge in their reduction step to different branches.

The former sub-cases are straightforward application of the induction hypothesis.

The proof for the latter sub-cases, where expressions diverge, are symmetric to each other so we will just outline the sub-case where expression **if** c **then** e_1 **else** e_2 reduces to the then-branch and expression **if** c' **then** e_3 **else** e_4 reduces to the else-branch.

So as hypothesis we have
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2}$ **if** `true` **then** e_1 **else** $e_2 \cong_{\mathsf{s}}$ **if** `false` **then** e_3 **else** $e_4{:}\tau^{\mathsf{s}'}$,
$\Delta \vdash S_1$, $\Delta \vdash S_2$, and $S_1 =_{\mathsf{s}} S_2$,
$(S_1; \textbf{if } \texttt{true } \textbf{then } e_1 \textbf{ else } e_2) \longrightarrow (S_1; e_1)$, and
$(S_2; \textbf{if } \texttt{false } \textbf{then } e_3 \textbf{ else } e_4) \longrightarrow (S_2; e_4)$.
Then by inversion of the expression equivalence relation on the hypothesis we obtain
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \texttt{true} \cong_{\mathsf{s}} \texttt{false} : \texttt{Bool}^{\mathsf{s}'}$,
$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{\texttt{true}\doteq\texttt{true}\},\mathcal{S}_2\cup\{\texttt{false}\doteq\texttt{true}\}} e_1 \cong_{\mathsf{s}} e_3 : \tau^{\mathsf{s}'}$, and
$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{\texttt{true}\doteq\texttt{false}\},\mathcal{S}_2\cup\{\texttt{false}\doteq\texttt{false}\}} e_2 \cong_{\mathsf{s}} e_4{:}\tau^{\mathsf{s}'}$.

Also by definition of expression equivalence (Definition 27) we know
$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{\texttt{true}\doteq\texttt{true}\}} e_1 : \tau^{\mathsf{s}'}$, and $\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_2\cup\{\texttt{false}\doteq\texttt{false}\}} e_4 : \tau^{\mathsf{s}'}$.

Since we know $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \texttt{true} \cong_{\mathsf{s}} \texttt{false} : \texttt{Bool}^{\mathsf{s}'}$, then we can obtain, by inversion of rule (E-EXPROPAQUE), $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$.

Also it is true that $\mathsf{s} < \mathsf{s}' \sqcap (\mathsf{r} \sqcup \mathsf{s}')$ holds, so we can apply rule (E-EXPROPAQUE) and get $\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{\texttt{true}\doteq\texttt{true}\},\mathcal{S}_2\cup\{\texttt{false}\doteq\texttt{false}\}} e_1 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$.

Then, by Constraint Cut Lemma we have $\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$.

Finally, by subtyping we can lower the computational context security level obtaining $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$. □

We proceed with another auxiliary lemma, which is similar to Lemma 3 but for the case where one of the programs already terminated.

Lemma 5 (Value-Step-Equivalence)
Let $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s e{:}\tau^{s'}$, $vars(\Delta) = \emptyset$, $\Delta \vdash S_1$, $\Delta \vdash S_2$ and $S_1 =_s S_2$.
If $(S_2, e) \longrightarrow (S_2', e')$ then there is Δ' such that $\Delta \subseteq \Delta'$, $\Delta' \vdash S_1$, $\Delta' \vdash S_2'$, $S_1 =_s S_2'$ and $\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s e'{:}\tau^{s'}$ (and symmetrically).
Proof: Induction on the derivation of $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s e{:}\tau^{s'}$. Proof can be found in Appendix B.2.

We can then prove our non-interference theorem:

Theorem 5 (Non-interference)
Let $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_s e_2{:}\tau^{s'}$, with $vars(\Delta) = \emptyset$, $\Delta \vdash S_1$, $\Delta \vdash S_2$ and $S_1 =_s S_2$.
If $(S_1, e_1) \xrightarrow{m} (S_1', v_1)$, and $(S_2, e_2) \xrightarrow{n} (S_2', v_2)$ then there is Δ' such that $\Delta \subseteq \Delta'$, $\Delta' \vdash S_1'$, $\Delta' \vdash S_2'$, $S_1' =_s S_2'$ and $\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_s v_2{:}\tau^{s'}$.

Proof By induction on $m + n$, using Lemma 3 in the case $m > 0$ and $n > 0$. For $m = 0$ and $n > 0$ we rely on Lemma 5.

Case $m = 0$ **and** $n = 0$:
Then $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_s v_2{:}\tau^{s'}$ and $S_1 =_s S_2$ by hypothesis.

Case $m > 0$ **and** $n > 0$**, subcase** $m = m' + 1$ **and** $n = n' + 1$:

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_s e_2{:}\tau^{s'}$	(1) - hyp
$\Delta \vdash S_1$ and $\Delta \vdash S_2$	(2) - hyp
$S_1 =_s S_2$	(3) - hyp
$(S_1, e_1) \xrightarrow{1} (S_1'', e_1'')$	(4) - hyp
$(S_2, e_2) \xrightarrow{1} (S_2'', e_2'')$	(5) - hyp
$(S_1'', e_1'') \xrightarrow{m'} (S_1', v_1)$	(6) - hyp
$(S_2'', e_2'') \xrightarrow{n'} (S_2', v_2)$	(7) - hyp
$\Delta \subseteq \Delta''$	(8) - by Lemma 3 with (1,2,3,4,5)
$\Delta'' \vdash S_1''$ and $\Delta'' \vdash S_2''$	(9) - by Lemma 3 with (1,2,3,4,5)
$S_1'' =_s S_2''$	(10) - by Lemma 3 with (1,2,3,4,5)
$\Delta'' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1'' \cong_s e_2''{:}\tau^{s'}$	(11) - by Lemma 3 with (1,2,3,4,5)
$\Delta'' \subseteq \Delta'$	by I.H. with (6,7,9,10,11)
$\Delta' \vdash S_1'$ and $\Delta' \vdash S_2'$	by I.H. with (6,7,9,10,11)
$S_1' =_s S_2'$	by I.H. with (6,7,9,10,11)
$\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_s v_2{:}\tau^{s'}$	by I.H. with (6,7,9,10,11)

Case $m = 0$ **and** $n > 0$**, subcase** $m = 0$ **and** $n = n' + 1$:

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_s e_2{:}\tau^{s'}$	(1) - hyp
$\Delta \vdash S_1$ and $\Delta \vdash S_2$	(2) - hyp
$S_1 =_s S_2$	(3) - hyp
$(S_1, v_1) \xrightarrow{0} (S_1, v_1)$	(4) - hyp

$(S_2, e_2) \xrightarrow{1} (S_2'', e_2')$	(5) - hyp
$(S_2'', e_2') \xrightarrow{n'} (S_2', v_2)$	(6) - hyp
$\Delta \subseteq \Delta''$	(7) by Lemma 5 with (1,2,3,5)
$\Delta'' \vdash S_1$ and $\Delta'' \vdash S_2''$	(8) by Lemma 5 with (1,2,3,5)
$S_1 =_{\mathsf{s}} S_2''$	(9) - by Lemma 5with (1,2,3,5)
$\Delta'' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}'}$	by Lemma 5 with (1,2,3,5)
$\Delta'' \subseteq \Delta'$	by I.H. with (4,6,7,8,9)
$\Delta' \vdash S_1$ and $\Delta' \vdash S_2'$	by I.H. with (4,6,7,8,9)
$S_1 =_{\mathsf{s}} S_2'$	by I.H. with (4,6,7,8,9)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau^{\mathsf{s}'}$	by I.H.with (4,6,7,8,9)

Case $m+1$ **and** $n = 0$: Symmetric to previous case. □

The noninterference theorem states two instances of the same program (differing only in values not observable at security level s), under equivalent stores S_1 and S_2, compute equivalent values and no changes are observable in the resulting stores. In particular, if the result is classified at security level s or below, then both instances return the same value.

In the following section, we give some examples on how to interpret noninterference theorem to check for data confidentiality in programs.

4.3 Interpreting Noninterference

Suppose we apply theorem Theorem 5 to a program $e = e_1 = e_2$ (so $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e : \tau^{\mathsf{s}'}$ holds by reflexivity).

Then, if $\mathsf{s} \not< \mathsf{s}'$, we must have $v_1 = v_2$ (since neither (E-EXPROPAQUE) or (E-VALOPAQUE) are applicable to derive $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau^{\mathsf{s}'}$).

One can thus conclude that an attacker "located" at security level s never distinguishes the result of a program executed under stores that only differ in data that should be considered confidential for level s (data classified at any level l such that $\mathsf{l} \not\leq \mathsf{s}$).

We now illustrate our noninterference results.

Example 28 Recall our conference manager from Chapter 1, and consider the following program that retrieves the profile of author with `uid 42` and then inserts a new profile in collection `Users` using some of the information previously retrieved.

$\tau_c \stackrel{\text{def}}{=} \Sigma[\texttt{uid} : \bot \times \texttt{name} : \texttt{U(uid)} \times \texttt{univ} : \texttt{U(uid)} \times \texttt{email} : \texttt{U(uid)}]$

```
let Users = ref_{ref(τc)*⊥} (ref_{τc} [] )::{} in
let p = first(viewUserProfile(42)) in
Users := ref_{τc} [ uid = 42, name = p.name,
                univ = p.univ, email = p.email ] :: !Users
```

Since the new record value associates information of security level `U(42)` (value `p`) with user id `42`, this program should be deemed secure and the noninterference property checked.

Let us apply the theorem to check. The evaluation of the assignment operation is the relevant part of this program since the program does not compute a value but changes the state at location `Users`.

Thus, to illustrate the compliance of the noninterference theorem, we will just analyse this part of the program's evaluation, referring back to the assignment operation as expression e:

```
Users:= ref_τc[uid=42, name=p.name, univ=p.univ, email=p.email]:: !Users
```

Assume `U(42)#U(666)`, and let S_1 and S_2 be stores such that

$$
\begin{array}{rl}
S_1(\texttt{Users}) = \{ & \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } A_1\texttt{, univ = } A_2\texttt{, email = } A_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}\ \} \\
S_2(\texttt{Users}) = \{ & \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } C_1\texttt{, univ = } C_2\texttt{, email = } C_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}\ \}
\end{array}
$$

We have $S_1 =_{\texttt{U(666)}} S_2$ since the values A_i and C_i, classified as `U(42)`, are not visible at level `U(666)`, by definition of store equivalence and `U(42)#U(666)`.

Also, we have $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\texttt{U(666)}} e : \texttt{cmd}^{\perp}$.

Let us, then, consider the reductions $(S_1; e) \longrightarrow (S_1'; ())$ and $(S_2; e) \longrightarrow (S_2'; ())$.

Then the resulting stores are the following

$$
\begin{array}{rl}
S_1'(\texttt{Users}) = \{ & \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } A_1\texttt{, univ = } A_2\texttt{, email = } A_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } A_1\texttt{, univ = } A_2\texttt{, email = } A_3\texttt{]}\ \} \\
S_2'(\texttt{Users}) = \{ & \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } C_1\texttt{, univ = } C_2\texttt{, email = } C_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } C_1\texttt{, univ = } C_2\texttt{, email = } C_3\texttt{]}\ \}
\end{array}
$$

so noninterference is satisfied, since $\mathit{redact}(\Delta, S_1', \texttt{U(666)}) = \mathit{redact}(\Delta, S_2', \texttt{U(666)})$

$$
\begin{array}{rl}
\{ & \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } \star\texttt{, univ = } \star\texttt{, email = } \star\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } \star\texttt{, univ = } \star\texttt{, email = } \star\texttt{]}\ \}
\end{array}
$$

That is, $S_1' =_{\texttt{U(666)}} S_2'$.

Thus, the effects of expression e are not visible at security level `U(666)`, as expected.

Now let us consider a slight modification to the code above in the assignment operation.

Example 29 This program will now associate the contents of the profile of author with id 42 to a profile of author with id 666, via the assignment expressions (expression e' from this point onwards).

$\tau_c \stackrel{\text{def}}{=} \Sigma[\texttt{uid} : \bot \times \texttt{name} : \texttt{U(uid)} \times \texttt{univ} : \texttt{U(uid)} \times \texttt{email} : \texttt{U(uid)}]$

```
let Users = ref_{ref(τc)*⊥} (ref_τc [] )::{} in
let p = first(viewUserProfile(42)) in
Users := ref_τc [ uid = 666, name = p.name,
                  univ = p.univ, email = p.email ] :: !Users
```

This clearly violates confidentiality, among other things, and is disallowed by the security lattice since `U(42)#U(666)`, so the program should be considered insecure.

Let us look this in detail. Again, we have $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e' \cong_{\texttt{U(666)}} e' : \texttt{cmd}^{\bot}$.
After the reduction steps $(S_1; e') \longrightarrow (S_1'; ())$ and $(S_2; e') \longrightarrow (S_2'; ())$, we have

$$
\begin{aligned}
S_1'(\texttt{Users}) = \{\ & \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } A_1\texttt{, univ = } A_2\texttt{, email = } A_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } A_1\texttt{, univ = } A_2\texttt{, email = } A_3\texttt{]}\ \} \\
S_2'(\texttt{Users}) = \{\ & \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } C_1\texttt{, univ = } C_2\texttt{, email = } C_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}, \\
& \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } C_1\texttt{, univ = } C_2\texttt{, email = } C_3\texttt{]}\ \}
\end{aligned}
$$

But now, $S_1' \neq_{\texttt{U(666)}} S_2'$ since after executing e' the values A_i and C_i of the new record are observable at level `U(666)`. This is captured by the notion of store equivalence because now we have

$$
\begin{aligned}
& \mathit{redact}(\Delta, S_1', \texttt{U(666)}) = \\
& \quad \{\ \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } \star\texttt{, univ = } \star\texttt{, email = } \star\texttt{]}, \\
& \quad\ \ \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}, \\
& \quad\ \ \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } A_1\texttt{, univ = } A_2\texttt{, email = } A_3\texttt{]}\ \}
\end{aligned}
$$

and

$$
\begin{aligned}
& \mathit{redact}(\Delta, S_2', \texttt{U(666)}) = \\
& \quad \{\ \textbf{ref}_{\tau_c}\ \texttt{[uid = 42, name = } \star\texttt{, univ = } \star\texttt{, email = } \star\texttt{]}, \\
& \quad\ \ \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } B_1\texttt{, univ = } B_2\texttt{, email = } B_3\texttt{]}, \\
& \quad\ \ \textbf{ref}_{\tau_c}\ \texttt{[uid = 666, name = } C_1\texttt{, univ = } C_2\texttt{, email = } C_3\texttt{]}\ \}
\end{aligned}
$$

As expected, the thesis of non-interference theorem is not satisfied.

Of course, insecure programs like Example 29 are rejected by our type system. In this particular case, it would not be possible to give the perhaps expected dependent type τ_c, to record `[uid = 666, sid= p.sid, name = p.name, univ = p.univ, email = p.email]` using rule (T-REFINERECORD) because the security level of `p.name`, `p.univ`, and

`p.email` is `U(42)` but field `uid` has value `666`. Thus, as shown in Example 28, well-typed programs do not leak confidential data.

4.4 Remarks

In this chapter, we presented our main soundness result: noninterference. Together with type safety, noninterference ensures that well-typed programs are compliant with the prescribed security policy (according to the defined security lattice).

Before introducing our result, however, we presented a equivalence relation between stores and expressions, which ee illustrated with examples. We proceed then to the formulation of the noninterference theorem and outlined its proof. We concluded with a couple of examples to show how to interpret our noninterference result.

In the following chapter, we will discuss some applications of this work. Namely, how one can reason about data confidentiality in data-centric systems and how one can apply our analysis to Data Manipulation Languages.

CHAPTER 5

REASONING WITH DEPENDENT INFORMATION FLOW TYPES

In this chapter we focus on the applications of this thesis work, which are twofold: a) reasoning about data-centric applications; and b) how our approach can be applied in applications programmed via Data Manipulation Languages (DML).

We show the former through an example of typical data-centric systems programmed in our core language in Section 5.1, and the latter by showing our encodings of typical DML primitives and how its typing rules can be derived from our type system (Section 5.2).

5.1 Data-centric Applications

We have shown in Chapter 1 how to reason about data confidentiality in a conference manager system, in this section we further illustrate with another toy example, an academic information manager system.

5.1.1 An Academic Information Manager System

In this scenario, a user of the system can be either a student or faculty member. The system stores data concerning its users' information, including a student's curriculum and tuition balance, a faculty member's department and salary, the evaluations of students in courses, and a student's final grades in "database tables" which we will represent in our core programming language as lists of (references to) records (e.g., mutable lists).

For our academic information manager example, we declare "database tables" as:

$$\texttt{let Studts = } \mathbf{ref}_{\mathtt{ref}(\tau)^{*\perp}}\ (\mathbf{ref}_{\tau}\ [])::\{\}\ \texttt{in}$$
$$\texttt{let Faculty = } \mathbf{ref}_{\mathtt{ref}(\sigma)^{*\perp}}\ (\mathbf{ref}_{\sigma}\ [])::\{\}\ \texttt{in}$$
$$\texttt{let Evals = } \mathbf{ref}_{\mathtt{ref}(\delta)^{*\perp}}\ (\mathbf{ref}_{\delta}\ [])::\{\}\ \texttt{in}$$

```
let Grades = ref_{ref(v)*⊥}(ref_v [])::{}
```

Where `Studts` stores information for each student; `Faculty` keeps track of faculty information such as their department and salary; `Evals` stores information regarding students' evaluation tests, namely: the id of the professor (evaluator), the id of the course whom the test concerns about, the criteria defined for the test (including its solution), and the scores obtained in a course's evaluation test; and `Grades` registers a student's final grade in its enrolled courses.

The system offers operations to add new data as well as some listing operations, we exemplify some of them.

Example 30 Operation `enrollStudent2Course` enrols a given student to a given course, initialising the final grade as 0.

```
let enrollStudent2Course = λ(s, c).
  let new_rec = ref_v [suid = s, cuid = c, grade = 0]
  in Grades := new_rec :: !Grades
```

Example 31 Operation `viewAverageScore` computes a given student's average of all evaluations of a given course

```
let viewAverageScore = λ (s, c).
 let counter = ref 0 in
  ( foreach (x in !Evaluations) with avg = 0 do
     let tuple = !x in
       if( tuple.cuid = c) then
        foreach (y in !(tuple.scores)) with sum = 0 do
         ( if (y.suid = s) then
             ( counter := !counter + 1;
               y.score ) + sum
           else sum
         ) + avg
       else avg
   )/!counter
```

Example 32 Operation `computeFinalGrade4Course` is a join operation between tables `Evaluations` (via operation `viewAverageScore`) and `Grades` to compute the final grade of all enrolled students in a given course.

```
let computeFinalGrade4Course = λ(c).
  foreach(x in !Grades) with y = skip do
     let tuple = !x in
      let s = tuple.suid in
```

```
let up_rec = [suid = s, cuid = c, grade =
    viewAverageScore(s ,c) ]
in x := up_rec
```

Before explaining the types declared for each collection, we introduce the security labels used in this system to classify data. Thus, we assume the following security levels for our academic information manager system:

- $\bot$, for data observable by anyone;
- `U(uid)`, for data observable by registered user with id `uid`;
- `S(suid,cuid)`, for data observable by student with id `suid` enrolled in course of id `cuid`;
- `P(puid,cuid)`, for data observable by professor with id `puid` that teaches course of id `cuid`;
- $\top$, for data observable by the admin user.

As seen in Chapter 3, the security lattice is required to enforce $\ell(\overline{v}, u, \overline{w}) \leq \ell(\overline{v}, \top, \overline{w})$ and $\ell(\overline{v}, \bot, \overline{w}) \leq \ell(\overline{v}, u, \overline{w})$. So, for example, for all `uid` we have `U(`$\bot$`)` $\leq$ `U(uid)` $\leq$ `U(`$\top$`)`. Moreover, we can see `U(`$\top$`)` as the approximation (by above) of any `U(uid)`, e.g, standing for the standard label `U`.

We interpret security labels indexed by $\top$ or $\bot$ as follows:

- `S(`$\bot$`,`$\bot$`)`, denotes the security compartment accessible to *any* student;
- `P(`$\bot$`,`$\bot$`)`, stands for the security compartment accessible to *any* professor;
- `S(`$\top$`,`$\top$`)`, represents the security compartment containing the information of *all* students;
- `P(`$\top$`,`$\top$`)` denotes the security compartment containing the information of *all* professors;
- `S(suid,`$\bot$`)`, stands for a student that has no authority over enrolled courses;
- `P(puid,`$\bot$`)`, denotes a professor that has no authority over allocated courses;
- `S(suid,`$\top$`)`, stands for registered users with uid `suid` that are students;
- `P(puid,`$\top$`)`, represents registered users with uid `puid` that are professors,
- `S(`$\top$`, cuid)`, stands for the security compartment of all students enrolled in course with id `cuid`;
- `P(`$\top$`, cuid)` represents the security compartment of all professors of the course with id `cuid`.

We can now discuss the types given for the above collections. So we have the following types for the contents of `Studts`, `Faculty`, `Evals`, and `Grades`, respectively:

$\tau \stackrel{\text{def}}{=} \Sigma[\ \mathtt{suid} : \bot \times \mathtt{CV} : \mathsf{U}(\mathsf{suid}) \times \mathtt{tuitions} : \mathsf{U}(\mathsf{suid})\ ]^{\bot}$

$\sigma \stackrel{\text{def}}{=} \Sigma[\ \mathtt{puid} : \bot \times \mathtt{dep} : \mathsf{U}(\mathsf{puid}) \times \mathtt{salary} : \mathsf{U}(\mathsf{puid})\ ]^{\bot}$

$$\delta \stackrel{\text{def}}{=} \Sigma[\ \texttt{puid}{:}\bot \times \texttt{cuid}{:}\bot \times \texttt{criteria}{:}\mathsf{P}(\mathsf{puid},\mathsf{cuid}) \times \texttt{test}{:}\mathsf{S}(\top,\mathsf{cuid}) \times \texttt{scores}{:}\gamma^{\bot}\ ]$$
$$\gamma \stackrel{\text{def}}{=} \texttt{ref}(\Sigma[\ \texttt{suid} : \bot\ \times\ \texttt{score} : \mathsf{S}(\mathsf{suid},\mathsf{cuid})\ ]^{*\bot})$$
$$\upsilon \stackrel{\text{def}}{=} \Sigma[\ \texttt{suid} : \bot \times \texttt{cuid} : \bot \times \texttt{grade} : \mathsf{S}(\mathsf{suid},\mathsf{cuid})\ ]^{\bot}$$

Our goal is to statically ensure, by typing, the confidentiality of the data stored in the academic information manager system. So, the security policy that we want to ensure is the following:

- A registered user's personal information (including both students and faculty sensitive data) is *only* observable by *himself/herself*, meaning *no other* registered user can see it;
- The contents of a test's criteria (including the test's solution) can be observable *only* by the professor *of the course it concerns about*;
- The test and its scores are *only* visible to all *enrolled students* – as well as the corresponding *course's professors*;
- The final grade of a course can *only* be observed by the student it concerns.

For our academic information manager, and in order to define policy above, we have the following axioms (quantifiers ranging over natural numbers) that define the security lattice of the system:

$$\forall \texttt{uid}.\ \mathsf{U}(\texttt{uid}) \leq \mathsf{S}(\texttt{uid},\_) \qquad \text{(Axiom 3)}$$
$$\forall \texttt{uid}.\ \mathsf{U}(\texttt{uid}) \leq \mathsf{P}(\texttt{uid},\_) \qquad \text{(Axiom 4)}$$
$$\forall \texttt{cuid}.\ \mathsf{S}(\_,\ \texttt{cuid}) \leq \mathsf{P}(\_,\texttt{cuid}) \qquad \text{(Axiom 5)}$$

Axiom 5 state that information observable by a student of a given course is also observable to a professor of the said course, while Axiom 3 and Axiom 4 denote that data visible to a registered user is also observable by a student or professor, respectively, if the ids match (the id represents the same user).

So, essentially, these axioms together with the defined policy (prescribed in the types), disallow for students to see tests in a course which they are not enrolled in and prevents from any student to see other student's scores or final grades in evaluations and courses, respectively. Moreover, only professors allocated to a course may see its tests, criteria and scores.

Thus, the types presented, together with the security lattice, establish the intended security policy. We will illustrate below with some examples on how these work to disallow insecure programs.

Consider then the following code

Example 33 This code snippet retrieves the average score of student with id `42` for the course with id `70` and then updates the student's final score with the obtained average

```
let grades_val = viewAverageScore(42,70)
in foreach(x in !Grades) with y = skip do
   let t_grade = !x in
     if(t_grade.suid = 42 and t_grade.cuid = 70) then
          let up_rec = [ suid = t_grade.suid,
                         cuid = t_grade.cuid, grade = grades_val ]
     in x := up_rec
```

The type of `Grades` collection is v,

$$\Sigma[\,\texttt{suid} : \texttt{int}^{\perp} \times \texttt{cuid} : \texttt{int}^{\perp} \times \texttt{grade} : \texttt{int}^{\texttt{S(suid,cuid)}}\,]^{\perp}$$

which is a dependent sum type where the security level of some fields depend on the actual values bound to other fields. For instance, notice that the security level of the `grade` field is declared as `S(suid,cuid)` where `suid` and `cuid` are other fields of the (thus dependent) record type.

So, in the first part of the code snippet, we extract the average score associated to student with `suid = 42` for course with `cuid = 70` so, according to type v, the `grades_val` identifier has security label `S(42,70)`.

Then we update the student's final score in mutable collection `Grades` whose `suid` value is `42` and `cuid` `70` with values `t_grade.suid`, `t_grade.cuid`, and `grades_val`, respectively.

Since we are adding a record whose `suid = 42` and `cuid = 70` (which we know from `t_grade.suid = 42` **and** `t_grade.cuid = 70` in the conditional), then the expected type is v where in place of the indexes `suid` and `cuid` we have the runtime values `42` and `70`, respectively. That is, we expect the new record value

```
[suid = t_grade.suid, cuid = t_grade.cuid, grade = grades_val]
```

to be typed as $\Sigma[\texttt{suid} : \perp \times \texttt{cuid} : \perp \times \texttt{grade} : \mathsf{S}(42, 70)]$.

Because we are using the "old" values of fields `suid` and `cuid` for the new record value then, of course, they have the expected type. Finally, since identifier `grade` has security level `S(42,70)`, then the assignment operation of Example 33 is deemed secure.

On the other hand, if we change the last conditional to be **`if`** `t_grade.suid = 666` **`and`** `t_grade.cuid = 70`, then we would be attempting to update record of `suid = 666`, so the record value

```
[ suid = t_grade.suid, cuid = t_grade.cuid, grade = grades_val ]
```

would have type $\Sigma[\texttt{suid} : \perp \times \texttt{cuid} : \perp \times \texttt{grade} : \mathsf{S}(666, 70)]$.

But since we are using identifier `grades_val` for the field `grade`, which we already checked to have security level `S(42,70)`, then this assignment operation is not typeable and thus deemed insecure.

This is intended, of course, since we were using the average score of student with id `42` as the final grade of student with id `666` for the course with id `70`, so the latter student was using private information from the former potentially for his benefit (if, say, the former's grade was greater then the latter's).

Consider now the following operation

```
let viewStudentProfile = λ (uid_a).
  foreach (x in !Studts) with y = {} do
    let t_usr = !x in
      if (t_usr.suid = uid_a) then t_usr::y else y
```

Function `viewStudentProfile` returns a collection of records of dependent sum type whose security labels on fields `CV`, and `tuitions` depend on the value of the parameter `uid_a`. A precise typing for `viewStudentProfile` is

$$\Pi(\texttt{uid\_a}{:}\bot).\Sigma[\texttt{suid}{:}\bot \times \texttt{CV:U(uid\_a)} \times \texttt{tuitions: U(uid\_a)}]^{*\bot}$$

Now say a user with id `10`, so we assume his observational level is then `U(10)`, attempts to observe the result of the `viewStudentProfile(42)`. Then, in order for this attempt to be successful the system tries to establish that `U(42)`$\leq$ `U(10)`, for the fields `CV` and `tuitions`. This, however, is not possible since the security lattice disallows such flow.

Indeed, in fact the security levels are incomparable `U(10)# U(42)`. So, a user with id `10` can observe the record (its structure) and the projection of field `suid` but not the projection of field `CV` and `tuitions`, as intended by the defined policy.

Let us conclude with a final example to illustrate our analysis in this scenario.

Example 34 The `addCriteria` operation is used by a professor to define the criteria of an evaluation test.

```
let addCriteria = λ(p, c).
  foreach (x in !Evals) with y = skip do
    let tuple = !x in
      if(tuple.puid = p and tuple.cuid = c) then
        let up_rec = [ puid = tuple.puid,
                       cuid = tuple.cuid,
                       criteria = defineTestCriteria(c,tuple.test),
                       test = tuple.test,
                       scores = tuple.scores ]
        in x := up_rec
```

Function `defineTestCriteria` returns a given evaluation test's criteria for a given course and has type $(\ \Pi(\texttt{u}{:}\bot,\ \texttt{t}{:}\bot);\ \texttt{S}(\top,\texttt{u})\ )^{\bot}$. Notice that its return type in the call `defineTestCriteria(c,tuple.test)` has security label `S(`$\top$`, c)`.

Additionally, we know that `tuple` has the type of the collection's references (type δ). So, in order to typecheck the assignment expression `x := up_rec`, we need to check that `up_rec` has the same type as the prescribed type for the collection's elements, δ. Namely, we have to check if `defineTestCriteria(c,tuple.test)` has type `P(puid,cuid)`.

As we said, the type for `defineTestCriteria(c,tuple.test)` is `S(`$\top$`, c)` but since it has a dependency, we need to infer a value for it. In this case, because we know by the conditional `tuple.cuid = c`, we can index the security level by field selection `tuple.cuid`, which allows us to type the assignment operation since field `cuid` is bounded by the dependent sum type of the record being used for the assignment.

Then we can type `defineTestCriteria(c, tuple.test)` with type `S(`$\top$`,cuid)` and thus, due to `S(`$\top$`,cuid)` $\leq$ `P(`$\bot$`,cuid)` (Axiom 5), we can up-classify `defineTestCriteria(c,tuple.test)` with `P(puid,cuid)`.

Notice that this up-classification is only possible to professors allocated to the course whose cuid is `tuple.cuid`, so only those professors will be able to see the added criteria for the evaluation test.

So we can, finally, type the record `up_rec` with the dependent sum type

$$\Sigma[\texttt{puid:}\bot\times\texttt{cuid:}\bot\times\texttt{criteria:P(puid,cuid)}\times\texttt{test:S(}\top\texttt{,cuid)}\times\texttt{scores:}v]\bot$$

Thus this program is deemed secure.

We now proceed with how we can reason about confidentiality in Data Manipulation Languages' applications.

5.2 Data Manipulation Languages

We have shown in [35] how value-dependent security labels are useful to express "row-level" security policies for Data Manipulation Language (DML) applications via a typed λ-calculus equipped with SQL-like DML primitives (inspired in proposals such as [7, 14, 38]). We will now show our dependent information flow types can be applied to such applications by encoding DML primitives in our core language and showing how we can derive the typing rules of these primitives [35] with our type system.

5.2.1 Encoding of DML primitives

Let us see how we can derive DML primitives from our core language, given that $\tau = \Sigma[m_1 : \tau_1^{\mathsf{s}_1} \times \ldots \times m_n : \tau_n^{\mathsf{s}_n}]$ and $\mathsf{s} = \sqcap|\mathsf{s}_i|^{\downarrow}$.

We begin with entity declarations,

entity `t(`m_1`:`$\tau_1^{\mathsf{s}_1}$`,...,`m_n`:`$\tau_n^{\mathsf{s}_n}$`)` **in** $e \stackrel{\text{def}}{=}$ **let** `t =` $\mathbf{ref}_{\mathbf{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}}$ `(` ($\mathbf{ref}_{\tau^{\mathsf{s}}}$ `[])::{})` **in** e

so entities can be encoded in our language as collections of (references of) record values, each representing a tuple of the entity.

Let us proceed with entity manipulation primitives, starting with queries to an entity

```
select(t, x.c, x.e) ≝ foreach(x in !t) with y = {} do
                         if c then e::y else y
```

so a query to an entity under condition c corresponds to an iteration of the collection that represents the entity, accumulating the records that satisfy the given condition into a collection.

Insertions to an entity can be encoded as follows

```
insert(t, e) ≝ let new_rec = ref_{τ^s} e in t := new_rec:: !t
```

which essentially consists in creating a new reference with the record value representing the new tuple, and then add it to the collection denoting the entity.

Removing a tuple from an entity given a condition is encoded as

```
delete(t, x.c) ≝ let res = foreach(x in !t) with y = {} do
                     if not c then x::y else y
                  in t ::= res
```

consisting in computing a new collection of records that do not satisfy the condition.

Finally an update operation on an entity is encoded by the following

```
update(t, x.e, x.c) ≝ foreach(x in !t) with y = skip do
                         if c then x := e else skip
```

so, basically, we need to iterate over the collection representing the entity to assign the new record value to all references representing tuples that satisfy the given condition.

We have shown how we can use our expressive core language, λ_{DIFT}, to encode a typical DML language, and next we present insecure information flows that might arise in these DML primitives.

Notice that since we can encode DML primitives into our core language then our dependent information flow types are applicable to programs coded by DML languages. Moreover, we can derive the typing rules presented in [35] from our type system. Therefore, our analysis ensures data-confidentiality in these scenarios.

Next, we will discuss the insecure information flows that may arise via DML primitives.

5.2.2 Information Flow Analysis for DML Primitives

We now discuss, identify, and analyse the insecure information flows that can arise in DML primitives. Value-dependent labels introduces some subtleties in the analysis, which we will point out in this discussion.

As we have seen in previous chapters, types themselves play no role in the information flow analysis so we will omit them in the discussion, focusing on the security level of expressions instead.

We recall entity `Users` declaration from Chapter 1 and declare an additional one, `Temp`, for the sake of this discussion:

```
entity Users(uid:⊥, name: U(uid), univ:U(uid), email: U(uid)) in
entity Temp(a: ⊥, b:⊥) in
let s_email = first(from (x in Users)
                    where x.uid = 42 select x.email) in
let pub = ''my_public_email@gmail.com''
```

Then, similar to what we have seen in Example 33 in the previous section, identifier `s_email` has security level `U(42)` since we are extracting the email of user with id `42`, and `pub` has security level $\bot$ since by default values are public.
Let us assume in the following examples `U(10)# U(42)`.

EXPLICIT FLOWS. A program state is represented by the set of entities (locations) that a program manipulates and the collection of tuples (references of records) they hold. Obviously, the DML primitives that enable modification of entities state pose the same issues that a typical assignment expression would when it comes to explicit flows.

Namely, in a insert operation

```
insert e in t
```

e corresponds to a new record to be added to the collection of (references of) records, that represents the entity located at t.

So we can look at this operation as an assignment to location t of the resulting collection of adding e to the collection stored at t, that is, $t := e :: !t$.

Let us take a look at a couple of examples. In the following insertion

```
insert [a = 42, b = s_email] in Temp
```

an explicit flow occurs because we are storing `s_email`, of security level `U(42)`, in field `b` that has a lower security level, $\bot$, so the operation is equivalent to `b:= s_email`.

The converse, however, is not true:

```
insert [uid = 10, ..., email = pub] in Users
```

we are inserting a record that assigns a value of security level $\bot$ to a field of a higher security level `U(42)`, equivalent to operation `email:= pub`.

This does not violate the non-interference property since we are increasing the value's security level by storing it in a container of a higher security level, and so does not violate data confidentiality.

update (x **in** t) **with** e **where** c

In an update, expression e represents a record to be used to update all tuples that satisfy condition c, say $\bar{r}$, in entity located at t. Roughly, we can see this operation as $r_i := e$ for all tuples (references of records) in $\bar{r}$. So the issues posed by an update, regarding explicit flows, are similar to those of the insert operation.

Thus, the following update

```
update (x in Users) with [uid = 10, ..., email = s_email] where true
```

is insecure since we are updating a field of security label `U(10)` with information of incomparable security label, `U(42)`.

Notice that the declared type for field `email` is, in fact, `U(uid)` and not `U(10)`. That is, the security label of this field depends on the value of field `uid`, so our analysis needs to be able to infer that when updating records with `uid = 10` we require field `email` to have label `U(10)`.

Finally, like in **insert**, the converse is secure:

```
update (x in Users) with [uid = 10, ..., email = pub] where true
```

Notice that condition c plays no part in explicit flows.

IMPLICIT FLOWS. Implicit flows may arise in DML primitives that depend on conditional expressions to filter tuples, the issues are much like the same as in a if-then-else expression. In particular:

from (x **in** t) **where** c **select** e

a select operation filters the collection of tuples located at t and executes expression e if the conditional expression c is satisfied.

So, regarding implicit flows, we can see this primitive as a conditional **if** `c` **then** `e` for each tuple of entity t. That means an implicit flow can occur if the guarded expression e changes the state. For example, the following query

```
from (x in Users) where x.email = pub and x.uid = 42 select leakUpdate

leakUpdate ≜ update (y in Users)
             with [uid = 10, name= y.name, univ = y.univ,
                   email = ''youremailisgone@toobad.org'']
             where y.uid = 10
```

has an implicit flow because we are interfering with information of user with `uid = 10` (fields `uid` and `email` via `leakUpdate`) based on data of incomparable security level (field `email` of user with `uid = 42`). That is, we have something similar to the following insecure expression

```
if (x.email = pub and x.uid = 42 and y.uid = 10) then
        y.uid:= 10; y.email:= ‘‘youremailisgone@toobad.org’’
```

where the conditional expression is classified with security level `U(42)` because of the usage of field `emails` of user with `uid = 42`.

update (*x* **in** *t*) **with** *e* **where** *c*

As stated previously, this operation updates all tuples that satisfy condition c, say $\bar{r}$, in entity located at t. Roughly, it is equivalent to **if** `c` **then** $r_i := e$ for all tuples (references of records) in $\bar{r}$. Then, implicit flows can occur in an update operation via its **where** clause whenever its security label is higher than the security label of the updated fields:

```
update (x in Users)
 with [uid= 10, name = x.name, univ = x.univ, email= x.email]
 where x.email= s_email
```

here condition `x.email= s_email` has security level `U(42)` and the field being updated (`uid`) has security level $\bot$.

delete (*x* **in** *t*) **where** *c*

The delete operation removes all the tuples of a collection of tuples located at t that satisfy the conditional expression c. Since a tuple may have fields with different security levels, namely with lower security level than the condition c, implicit flows may occur after executing a delete operation. For instance

```
delete (x in Users) where x.email = s_email and x.uid = 42
```

is insecure because we remove information of security level $\bot$, value of field `uid` for user with `uid = 42`, based on a condition with a higher security level, `U(42)`. So an attacker at observational level $\bot$ could observe the removal of these tuples.

Notice that, like before, our analysis needs to be able to infer the correct values for the dependent security labels that occur in **select**, **update** and **delete** primitives. In these particular cases, our analysis can extract relevant information from the **where** clauses in order to infer the correct value for the dependent labels (as shown in the examples).

In the following section, we show how typing rules for DML can be derived using the type system for λ_{DIFT}.

5.2.3 Deriving DML Typing Rules

We will now show how the typing rules for DML primitives can be derived from our type system. We start with primitive **entity** $t(m_1 : \tau_1^{\mathsf{s}_1}, \ldots, m_n : \tau_n^{\mathsf{s}_n})$ **in** e, its typing rule is

$$\frac{\Delta, t : [m_1 : \tau_1^{\mathsf{s}_1}, \ldots, m_n : \tau_n^{\mathsf{s}_n}]^{*\sqcap|\mathsf{s}_i|^{\downarrow}} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{entity}\ t(m_1 : \tau_1^{\mathsf{s}_1}, \ldots, m_n : \tau_n^{\mathsf{s}_n})\ \textbf{in}\ e : \tau^{\mathsf{s}'}}\ \text{(T-ENTITY)}$$

Recall our encoding

entity $\mathtt{t}(m_1{:}\tau_1^{\mathsf{s}_1}, \ldots, m_n{:}\tau_n^{\mathsf{s}_n})$ **in** $e \stackrel{\text{def}}{=}$ **let** $\mathtt{t}$ = $\mathbf{ref}_{\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}}$ (($\mathbf{ref}_{\tau^{\mathsf{s}}}$ []):: {}) **in** e

where $\tau = \Sigma[m_1 : \tau_1^{\mathsf{s}_1} \times \ldots \times m_n : \tau_n^{\mathsf{s}_n}]$ and $\mathsf{s} = \sqcap|\mathsf{s}_i|^{\downarrow}$.

So we can derive the following, where $\Delta' = \Delta, \mathtt{t} : \mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$:

1. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ []: τ^{s}
   by (T-RECORD)
2. $\mathsf{r} \leq \mathsf{s}$

---

3. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ $\mathbf{ref}_{\tau^{\mathsf{s}}}$ []: $\mathtt{ref}(\tau^{\mathsf{s}})^{\perp}$
   by (T-REF), 1, 2
4. $\perp \leq \mathsf{s}$

---

5. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ $\mathbf{ref}_{\tau^{\mathsf{s}}}$ []: $\mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}}$
   by (T-SUB), 3, 4
6. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \{\} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
   by (T-EMPTY)

---

7. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ ($\mathbf{ref}_{\tau^{\mathsf{s}}}$ []):: $\{\} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
   by (T-CONS), 5, 6
8. $\mathsf{r} \leq \mathsf{s}$

---

9. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ $\mathbf{ref}_{\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}}$ (($\mathbf{ref}_{\tau^{\mathsf{s}}}$ []):: {}): $\mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$
   by (T-REF), 7, 8
10. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ e: $\tau^{\mathsf{s}'}$

---

11. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ **let** t = $\mathbf{ref}_{\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}}$ (($\mathbf{ref}_{\tau^{\mathsf{s}}}$ []):: {}) **in** e : $\tau^{\mathsf{s}'}$
    by (T-LET), 9, 10

Note that the language in [35] does not have references, instead entities were treated as locations that store mutable collections of mutable records (as encoded in our language). So step 10 is equivalent to the premise of rule (T-ENTITY) since $\Delta' = \Delta, \mathtt{t} : \mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$ and $\tau^{\mathsf{s}} = \Sigma[m_1 : \tau_1^{\mathsf{s}_1} \times \ldots \times m_n : \tau_n^{\mathsf{s}_n}]^{\sqcap|\mathsf{s}_i|^{\downarrow}}$.

Next we have primitive **insert** e **in** t, its typing rule is

$$\frac{\begin{array}{l}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \\ \forall_i\, \mathsf{r} \leq \theta_x(\mathcal{S}\{x \doteq e\}, \mathsf{s}_i)\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{insert}(t,\ e) : \mathtt{cmd}^{\perp}}\ \text{(T-INSERT)}$$

where $\theta_x(\mathcal{S}\{x \doteq e\}, \mathsf{s}_i)$, with x fresh, is used to approximate the concrete values in dependencies occurring in security labels s_i.
So, in our encoding we have

insert(t, e) $\stackrel{\text{def}}{=}$ **let** new_rec = $\mathbf{ref}_{\tau^{\mathsf{s}}}$ e **in** t := new_rec:: !t

We assume for the next derivation: $\Delta(\mathtt{t}) = \mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$, and
$\Delta' = \Delta, \mathtt{new\_rec} : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}}$.
Then, from the above encoding, we derive the following

1. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ e : τ^{s}
2. $\mathsf{r} \leq \mathsf{s}$

---

3. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ $\mathbf{ref}_{\tau^{\mathsf{s}}}$ e : $\mathtt{ref}(\tau^{\mathsf{s}})^{\perp}$
    by (T-REF), 1, 2
4. $\perp \leq \mathsf{s}$

---

5. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ $\mathbf{ref}_{\tau^{\mathsf{s}}}$ e : $\mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}}$
    by (T-SUB), 3, 4
6. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ new_rec: $\mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}}$
    by (T-ID)
7. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ t: $\mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$
    by (T-ID)

8. $\bot \leq \mathsf{s}$

---

9. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ `!t:` $\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
by (T-DEREF), 7, 8

---

10. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ `new_rec ::` `!t:` $\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
by (T-CONS), 6, 9

11. $\mathsf{r} \sqcup \bot \leq \mathsf{s}$

---

12. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ `t := new_rec ::` `!t:` $\mathtt{cmd}^{\bot}$
by (T-ASSIGN), 10, 11

---

13. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ **let** `new_rec =` **ref**$_{\tau^{\mathsf{s}}}$ `e` **in** `t := new_rec ::` `!t :` $\mathtt{cmd}^{\bot}$
by (T-LET), 5, 12

So by step 11, and since $\mathsf{s} = \sqcap|\mathsf{s}_i|^{\downarrow}$, we have $\mathsf{r} \leq \sqcap|\mathsf{s}_i|^{\downarrow}$. This corresponds to the side-condition $\forall_i\, \mathsf{r} \leq \theta_x(\mathcal{S}\{x \doteq e\}, \mathsf{s}_i)$ in the typing rule (T-INSERT) since:

- $\mathsf{r} \leq \sqcap|\mathsf{s}_i|^{\downarrow} \;\equiv\; \forall_i\, \mathsf{r} \leq |\mathsf{s}_i|^{\downarrow}$ and
- $\forall_i\, |\mathsf{s}_i|^{\downarrow} \leq \theta_x(\mathcal{S}\{x \doteq e\}, \mathsf{s}_i)$

Note that the last condition holds because by definition of $|\cdot|^{\downarrow}$ any dependency occurring in label s_i is approximated by $\bot$ so by the security lattice axioms we know that any approximation to a concrete value obtained via $\theta_x(\mathcal{S}\{x \doteq e\}, \mathsf{s}_i)$ will always be greater or equal to $|\mathsf{s}_i|^{\downarrow}$.

We proceed with primitive **from** `(`x **in** t`)` **where** c **select** e, its typing rule is

$$\frac{\begin{array}{c}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \\ \mathcal{S}' = \mathcal{S} \cup \{c \doteq \mathtt{true}\} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{u}} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}'} e : \tau^{\mathsf{u}}\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{select}(t,\ x.c,\ x.e) : \tau^{*\mathsf{u}}}\ \text{(T-SELECT)}$$

Let us recall our encoding of the select primitive

```
select(t, x.c, x.e) ≝ foreach(x in !t) with y = {} do
                        if c then e::y else y
```

We assume in the following derivation:
$\Delta(\mathtt{t}) = \mathtt{ref}(\mathtt{ref}(\tau^{\mathtt{s}})^{*\mathtt{s}})^{\perp}$, and
$\Delta' = \Delta, \mathtt{x} : \mathtt{ref}(\tau^{\mathtt{s}})^{\mathtt{s}}, \mathtt{y} : \tau'^{*\mathtt{u}}$.
So we have the following type derivation for the encoding above

1. $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}} \mathtt{t}: \mathtt{ref}(\mathtt{ref}(\tau^{\mathtt{s}})^{*\mathtt{s}})^{\perp}$
   by (T-ID)

2. $\perp \leq \mathtt{s}$

---

3. $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}} \mathtt{!t}: \mathtt{ref}(\tau^{\mathtt{s}})^{*\mathtt{s}}$
   by (T-DEREF), 1,2

4. $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}} \{\} : \tau'^{*\mathtt{u}}$
   by (T-EMPTY)

5. $\Delta' \vdash^{\mathtt{r}}_{\mathcal{S}} \mathtt{c} : \mathtt{bool}^{\mathtt{u}}$

6. $\Delta' \vdash^{\mathtt{r}\sqcup\mathtt{u}}_{\mathcal{S}\cup\{c \doteq \mathtt{true}\}} \mathtt{e} : \tau'^{\mathtt{u}}$

7. $\Delta' \vdash^{\mathtt{r}\sqcup\mathtt{u}}_{\mathcal{S}\cup\{c \doteq \mathtt{true}\}} \mathtt{y} : \tau'^{*\mathtt{u}}$

---

8. $\Delta' \vdash^{\mathtt{r}\sqcup\mathtt{u}}_{\mathcal{S}\cup\{c \doteq \mathtt{true}\}} \mathtt{e} :: \mathtt{y} : \tau'^{*\mathtt{u}}$
   by (T-CONS), 6, 7

9. $\Delta' \vdash^{\mathtt{r}\sqcup\mathtt{u}}_{\mathcal{S}\cup\{c \doteq \mathtt{false}\}} \mathtt{y} : \tau'^{*\mathtt{u}}$
   by (T-ID)

---

10. $\Delta' \vdash^{\mathtt{r}}_{\mathcal{S}}$ **if** c **then** e :: y **else** y $: \tau'^{*\mathtt{u}}$
    by (T-IF), 5, 8, 9

---

11. $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}}$ **foreach**(x **in** !t) **with** y = $\{\}$ **do**
    if c **then** e :: y **else** y $: \tau'^{*\mathtt{u}}$
    by (T-LET), 3, 10

We point out that expression e, in step 6, which is the query to be applied to the collection of tuples that satisfy condition c, is typed in our system as $\Delta' \vdash^{\mathtt{r}\sqcup\mathtt{u}}_{\mathcal{S}\cup\{c \doteq \mathtt{true}\}} \mathtt{e} : \tau'^{\mathtt{u}}$.
This typing corresponds to the one we find in rule (T-SELECT), which is
$\Delta, x : [\ldots, m_i{:}\tau_i^{\mathtt{s}_i}, \ldots]^{\mathtt{s}} \vdash^{\mathtt{r}\sqcup\mathtt{s}'}_{\mathcal{S}\cup\{c \doteq \mathtt{true}\}} e : \tau^{\mathtt{u}}$, such that in this case we have $\tau = \tau'$.

Let us now see primitive **delete** $(x$ **in** $t)$ **where** c, with typing rule

$$\frac{\begin{array}{c}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}'} \\ \forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq \theta_x(\mathcal{S} \cup \{c \doteq \mathtt{true}\}, \mathsf{s}_i)\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{delete}(t,\ x.c) : \mathtt{cmd}^{\perp}}\ \text{(T-DELETE)}$$

Our encoding of the delete primitive

```
delete(t, x.c) ≝ let res = foreach(x in !t) with y = {} do
                     if not c then x::y else y
                 in t := res
```

Assuming

$\Delta(\mathtt{t}) = \mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$,
$\Delta' = \Delta, \mathtt{x} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}, \mathtt{y} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}\sqcup\mathsf{s}'}$,
$\Delta'' = \Delta, \mathtt{res} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}\sqcup\mathsf{s}'}$.

We derive for the encoding above

1. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ `t`: $\mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$
   by (T-ID)
2. $\perp \leq \mathsf{s}$

---

3. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ `!t`: $\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
   by (T-DEREF), 1,2
4. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \{\} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}\sqcup\mathsf{s}'}$
   by (T-EMPTY)
5. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ **not** `c` : $\mathtt{bool}^{\mathsf{s}'}$
6. $\mathsf{s}' \leq \mathsf{s} \sqcup \mathsf{s}'$

---

7. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ **not** `c` : $\mathtt{bool}^{\mathsf{s}\sqcup\mathsf{s}'}$
   by (T-SUB), 5, 6
8. $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{\mathsf{not}\ \mathsf{c}\doteq\mathtt{true}\}}$ `x` : $\mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}}$
   by (T-ID)
9. $\mathsf{s} \leq \mathsf{s} \sqcup \mathsf{s}'$

---

10. $\Delta' \vdash^{\mathsf{r} \sqcup \mathsf{s}'}_{\mathcal{S} \cup \{\text{not } \mathsf{c} \doteq \mathtt{true}\}} \mathtt{x} : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s} \sqcup \mathsf{s}'}$
by (T-SUB), 8, 9

11. $\Delta' \vdash^{\mathsf{r} \sqcup \mathsf{s}'}_{\mathcal{S} \cup \{\text{not } \mathsf{c} \doteq \mathtt{true}\}} \mathtt{y} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
by (T-ID)

12. $\mathsf{s} \leq \mathsf{s} \sqcup \mathsf{s}'$

---

13. $\Delta' \vdash^{\mathsf{r} \sqcup \mathsf{s}'}_{\mathcal{S} \cup \{\text{not } \mathsf{c} \doteq \mathtt{true}\}} \mathtt{y} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s} \sqcup \mathsf{s}'}$
by (T-SUB), 11, 12

---

14. $\Delta' \vdash^{\mathsf{r} \sqcup \mathsf{s}'}_{\mathcal{S} \cup \{\text{not } \mathsf{c} \doteq \mathtt{true}\}} \mathtt{x::y} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s} \sqcup \mathsf{s}'}$
by (T-CONS), 10, 13

15. $\Delta' \vdash^{\mathsf{r} \sqcup \mathsf{s}'}_{\mathcal{S} \cup \{\text{not } \mathsf{c} \doteq \mathtt{false}\}} \mathtt{y} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
by (T-ID)

16. $\mathsf{s} \leq \mathsf{s} \sqcup \mathsf{s}'$

---

17. $\Delta' \vdash^{\mathsf{r} \sqcup \mathsf{s}'}_{\mathcal{S} \cup \{\text{not } \mathsf{c} \doteq \mathtt{false}\}} \mathtt{y} : \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s} \sqcup \mathsf{s}'}$
by (T-SUB), 15, 16

---

18. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ **if not** c **then** x::y **else** y : $\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s} \sqcup \mathsf{s}'}$
by (T-IF), 14, 17

19. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ **foreach**(x **in** !t) **with** y = $\{\}$ **do**
if not c **then** x::y **else** y : $\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s} \sqcup \mathsf{s}'}$
by (T-FOREACH), 3, 4, 18

20. $\Delta'' \vdash^{\mathsf{r}}_{\mathcal{S}} \mathtt{t}: \mathtt{ref}(\mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$
by (T-ID)

21. $\Delta'' \vdash^{\mathsf{r}}_{\mathcal{S}} \mathtt{res}: \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s} \sqcup \mathsf{s}'}$
by (T-ID)

22. $\mathsf{s} \sqcup \mathsf{s}' \leq \mathsf{s}$

---

23. $\Delta'' \vdash^{\mathsf{r}}_{\mathcal{S}} \mathtt{res}: \mathtt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
by (T-SUB), 21, 22

24. $\mathsf{r} \sqcup \bot \leq \mathsf{s} \sqcup \mathsf{s}'$

---

25. $\Delta'' \vdash^{\mathsf{r}}_{\mathcal{S}}$ `t := res : cmd`$^{\bot}$
by (T-ASSIGN), 20, 23, 24

---

26. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ `let res = foreach(x in !t) with y = {} do`
`if not c then x::y else y`
`in t := res : cmd`$^{\bot}$ by (T-LET), 19, 25

Then by step 9 and step 22 we have $\mathsf{s} \sqcup \mathsf{s}' = \mathsf{s}$.

And by step 6 and step 24 we conclude $\mathsf{s}' \leq \mathsf{s}$ and $\mathsf{r} \leq \mathsf{s}$, respectively.

Therefore, we can infer $\mathsf{r} \sqcup \mathsf{s}' \leq \sqcap |\mathsf{s}_i|^{\downarrow}$ (since $\mathsf{s} = \sqcap |\mathsf{s}_i|^{\downarrow}$) holds. This corresponds to the condition imposed in rule (T-DELETE), $\forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq \theta_x(\mathcal{S} \cup \{c \doteq \texttt{true}\}, \mathsf{s}_i)$, since:

- $\mathsf{r} \sqcup \mathsf{s}' \leq \sqcap |\mathsf{s}_i|^{\downarrow} \ \equiv\ \forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq |\mathsf{s}_i|^{\downarrow}$ and

- $\forall_i\ |\mathsf{s}_i|^{\downarrow} \leq \theta_x(\mathcal{S} \cup \{c \doteq \texttt{true}\}, \mathsf{s}_i)$

Note that the last condition holds because by definition of $|\cdot|^{\downarrow}$ any dependency occurring in label s_i is approximated by $\bot$ so by the security lattice axioms we know that any approximation to a concrete value obtained via $\theta_x(\mathcal{S} \cup \{c \doteq \texttt{true}\}, \mathsf{s}_i)$ will always be greater or equal to $|\mathsf{s}_i|^{\downarrow}$.

We conclude with primitive **update** `(`x **in** t`)` **with** e **where** c, whose typing rule is

$$\frac{\begin{array}{c}\Delta(t) = [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{*\mathsf{s}} \\ \mathcal{S}' = \mathcal{S} \cup \{c \doteq \texttt{true}\} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} c : \texttt{Bool}^{\mathsf{s}'} \\ \Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}'} e : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}'} \\ \forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq \theta_x(\mathcal{S}', \mathsf{s}_i) \sqcap \theta_y(\mathcal{S}\{y \doteq e\}, \mathsf{s}_i)\end{array}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{update}(t,\ x.e,\ x.c) : \texttt{cmd}^{\bot}}\ \text{(T-UPDATE)}$$

Our encoding of the update primitive

```
update(t, x.e, x.c) ≝ foreach(x in !t) with y = skip do
                         if c then x := e else skip
```

For the next derivation, we assume

$\Delta(\texttt{t}) = \texttt{ref}(\texttt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\bot}$,
$\Delta' = \Delta, \texttt{x} : \texttt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}}, \texttt{y} : \texttt{cmd}^{\bot}$.
The type derivation for the encoding above is as follows

1. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \texttt{t}\colon \texttt{ref}(\texttt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}})^{\perp}$
   by (T-ID)
2. $\perp \leq \mathsf{s}$

---

3. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \texttt{!t}\colon \texttt{ref}(\tau^{\mathsf{s}})^{*\mathsf{s}}$
   by (T-DEREF), 1,2
4. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} \texttt{skip} : \texttt{cmd}^{\perp}$
5. $\perp \sqcup \mathsf{s}'$

---

6. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} \texttt{skip} : \texttt{cmd}^{\mathsf{s}'}$
   by (T-SUB), 4, 5
7. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} \texttt{c} : \texttt{bool}^{\mathsf{s}'}$
8. $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{c\doteq\texttt{true}\}} \texttt{x} : \texttt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}}$
   by (T-ID)
9. $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{c\doteq\texttt{true}\}} \texttt{e} : \tau^{\mathsf{s}}$
10. $(\mathsf{r}\sqcup\mathsf{s}')\sqcup\mathsf{s} \leq \mathsf{s}$

---

11. $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{c\doteq\texttt{true}\}} \texttt{x := e} : \texttt{cmd}^{\perp}$ by (T-ASSIGN), 8, 9, 10
12. $\perp \leq \mathsf{s}'$

---

13. $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{c\doteq\texttt{true}\}} \texttt{x := e} : \texttt{cmd}^{\mathsf{s}'}$
    by (T-SUB), 11, 12
14. $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{c\doteq\texttt{false}\}} \texttt{skip} : \texttt{cmd}^{\perp}$
15. $\perp \leq \mathsf{s}'$

---

16. $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{c\doteq\texttt{false}\}} \texttt{skip} : \texttt{cmd}^{\mathsf{s}'}$
    by (T-SUB), 14, 15

---

17. $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}}$ **if** c **then** x := e **else** skip : $\mathtt{cmd}^{\mathsf{s}'}$
by (T-IF), 7, 13, 16

---

18. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ **foreach**(x **in** !t) **with** y = skip **do**
if c **then** x := e **else** skip : $\mathtt{cmd}^{\mathsf{s}'}$
by (T-FOREACH), 3, 6, 17

Note that we type the expression e (step 9), used to update the tuples that satisfy condition c, in our type system as $\Delta' \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}\cup\{c\doteq\mathtt{true}\}} \mathsf{e} : \tau^{\mathsf{s}}$.

This corresponds to the premise in rule (T-UPDATE)

$$\Delta, x : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}\cup\{c\doteq\mathtt{true}\}} e : [\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots]^{\mathsf{s}'}$$

The difference resides in

(i) we require the computational context to be $\mathsf{r} \sqcup \mathsf{s}'$ instead of just r, since in our encoding expression e is typed in an assignment operation under a conditional's branch;

(ii) in our system we type expression e with security level s instead of s'

But (i) is used to prevent implicit flows on writes occurring on expression e, which in this case we know to be a record value, so we could in fact, if not for the conditional rule, type expression e safely under computational context r like it is done in its DML counterpart.

As for (ii) we have to check if the record value matches the type for the entity's tuples but on its DML counterpart rule, we have to check if this record value can be raised the security level of the condition being used for the update, c, because we treat the update operation as a conditional.

Also, by step 10 we have $(\mathsf{r} \sqcup \mathsf{s}') \sqcup \mathsf{s} \leq \mathsf{s}$, that is, $\mathsf{r} \sqcup \mathsf{s}' \leq \sqcap|\mathsf{s}_i|^{\downarrow}$ (since $\mathsf{s} = \sqcap|\mathsf{s}_i|^{\downarrow}$). This corresponds to the condition imposed on rule (T-UPDATE)

$$\forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq \theta_x(\mathcal{S}', \mathsf{s}_i) \sqcap \theta_y(\mathcal{S}\{y \doteq e\}, \mathsf{s}_i)$$

This is so because:

- $\mathsf{r} \sqcup \mathsf{s}' \leq \sqcap|\mathsf{s}_i|^{\downarrow} \ \equiv\ \forall_i\ \mathsf{r} \sqcup \mathsf{s}' \leq |\mathsf{s}_i|^{\downarrow}$ and
- $\forall_i\ |\mathsf{s}_i|^{\downarrow} \leq \theta_x(\mathcal{S}', \mathsf{s}_i) \sqcap \theta_y(\mathcal{S}\{y \doteq e\}, \mathsf{s}_i)$

Note that the last condition holds because by definition of $|\cdot|^{\downarrow}$ any dependency occurring in label s_i is approximated by $\bot$ so by the security lattice axioms we know that any approximation to a concrete value obtained via $\theta_x(\mathcal{S} \cup \{c \doteq \mathtt{true}\}, \mathsf{s}_i)$ or $\theta_y(\mathcal{S}\{y \doteq e\}, \mathsf{s}_i)$ will always be greater or equal to $|\mathsf{s}_i|^{\downarrow}$.

We conclude by taking note that the type given by our type system to our encoding for the **update** primitive is more conservative than it's DML counterpart typing rule since we

require, using our type system, the encoding to be typed as $\texttt{cmd}^{s'}$ instead of typing with security level $\bot$.

Finally, we conclude with an illustrative example using DML primitives.

5.2.4 A Conference Manager using DML primitives

To conclude this chapter we show how to program some of our conference manager system operations using DML primitives.
We begin with the declaration of the entities ("database tables")

```
entity Users(uid:⊥, name:U(uid), univ:U(uid), email:U(uid)) in
entity Submissions(uid:⊥, sid:⊥, title:A(uid,sid),
                   abst:A(uid,sid), paper:A(uid,sid)) in
entity Reviews(uid:⊥, sid:⊥, PC_only:PC(uid,sid),
               review:A(⊤,sid), grade:A(⊤,sid)) in ...
```

and proceed with the presented operations of the conference manager system

```
let viewUserProfile = λ uida.
         ( from(x in Users) where x.uid = uida select x )

let viewAuthorPapers = λ uida. ( from (x in Submissions)
                                 where x.uid = uida
                                 select x)

let viewAssignedPapers = λ uidr.
   ( from (x in Reviews)
     where x.uid = uidr
     select ( from (y in Submissions)
              where y.sid = x.sid
              select y ) )

let addCommentSubmission = λ uid_r, sidr.
     ( foreach(p in viewAssignedPapers(uid_r)) with dummy = skip do
         if(p.sid == sidr ) then
           update (x in Reviews)
           where x.sid = p.sid
           with [uid= x.uid, sid= x.sid,
                 PC_only= comment(p.uid,p.sid,p),
                 review= x.review, grade= x.grade] )
```

So functions `viewUserProfile` and `viewAuthorPapers` are simple queries to entities `Users` and `Submissions`, respectively; function `viewAssignedPapers` is a join operation between entities `Reviews` and `Submissions`; and function `addCommentSubmission`

updates all the tuples that match the submission id of each resulting tuple of query `viewAssignedPapers` for the given input.
Finally, we show some of the code snippets we presented in Chapter 1, we begin with Example 4

```
let t = first( from(x in Submissions)
              where x.uid = 42 and x.sid = 70
              select x.title )
in update (x in Submissions)
   where x.uid =42 and x.sid = 70
   with [uid= x.uid, sid= x.sid, title= t, abs= x.abs, paper= x.paper]

let t = first( from(x in Submissions)
          where x.uid = 42 and x.sid = 70
              select x.title )
in update (x in Submissions)
   where x.uid =32
   with [uid= x.uid, sid= x.sid, title= t, abs= x.abs, paper= x.paper]
```

As we have seen before, the first code snippet is secure while the latter is insecure because we are updating the tuple of an author using another author's information.

Let us see a modification of function `addCommentSubmission` where the update operation is applied to all resulting tuples of query `viewAssignedPapers` for the given input. that match the submission id of each resulting tuple of query `viewAssignedPapers` for the given input.

```
let addCommentSubmission = λ uid_r, sidr.
   ( foreach(p in viewAssignedPapers(uid_r)) with dummy = skip do
        if(p.sid == sidr ) then
           update (x in Reviews)
           where true
           with [uid= x.uid, sid= x.sid,
                 PC_only= comment(p.uid,p.sid,p),
                 review= x.review, grade= x.grade] )
```

That is, we do not filter out those tuples whose submission id does not match the input, therefore this version of the function `addCommentSubmission` is insecure because we might be interfering with other submission's reviews.
We conclude with the following code snippet

```
let p = first(viewUserProfile(42) ) in
insert [uid= 42, name= p.name, univ= p.univ, email= p.email] in Users
```

that inserts a new tuple in entity `Users` using the profile of user with id `42` (retrieved by query `viewUserProfile`). This is deemed secure since we are associating the retrieved information with the author of id `42`.

5.3 Remarks

In this chapter we discussed the applications of this thesis work. First we illustrated, via a toy example, how we can use our type system to reason about data confidentiality in data-centric system. We proceeded, then, to show how our core language can encode typical Data Manipulation Language's (DML) primitives.

By encoding such primitives we can apply our type-based information flow analysis to ensure data confidentiality in typical DML programs. We also discussed the possible information flows that can arise in DML primitives, followed by typing derivations for each encoding. We then established that we can derive from our typing rules, the rules that one would expect for DML primitives (based on those presented in [35]).

Finally, we illustrated how we could use DML primitives to program our conference manager example.

The following chapter presents a typechecking algorithm for our dependent information flow type system and discusses its implementation in a prototype.

CHAPTER 6

ALGORTIHMIC TYPECHECKING

In this chapter we present a typechecker algorithm of our type system, discuss details of its prototype implementation and compare our typechecker prototype with other relevant type-based information-flow tools.

6.1 Algorithm

In this section we discuss a type-checking algorithm for a suitable annotated version of our core language.

The algorithm already allows us to verify many interesting examples, including those presented in this thesis, and lead to a prototype implementation that can be found at `http://ctp.di.fct.unl.pt/DIFTprototype/` (a live version can be found at rise4fun: `http://rise4fun.com/DIFT/`).

For pragmatic reasons, we require type annotations on reference creation, empty collection, record fields, and variant labels, leaving for future work possible inference. We introduce type cast constructs, of the forms $[\tau^{\mathsf{s}}]e$ and $]\mathsf{s}[e$, useful to manually up-classify primitive values and raise the level of the computational context, respectively.

The algorithm depends on subsidiary procedures to check subtyping, which we represent by the $\sigma <: \tau$ tests; on a constraint solving procedure, which we represent by $\mathcal{S} \models V \doteq U$ tests; and on a procedure that checks whether a type is well-formed. The auxiliary procedure $\mathit{unref}(\mathcal{S}, [\ldots, m_i{:}\tau_i^{\mathsf{s}_i} = e_i, \ldots])$ eliminates field dependencies on the given (possibly dependent) record type, and returns an unrefined record type by attempting the most precise possible approximations to the field values v_i given by each expression e_i, using $\mathcal{S}\{x \doteq e_i\} \models x \doteq \mathsf{v}_i$.

The subtyping test essentially implements the subtyping rules, given a suitable security lattice, while the check for well-formedness of types implements the well-formed types rules. For simplicity, we will omit in the algorithm presentation the well-formedness

checks but point out in the discussion when relevant. In our prototype implementation the security lattice can be user-defined, in a preamble to the code to be type checked. As far as constraint solving is concerned, our current prototype relies on an encoding of the required entailment checks on queries for the Z3 SMT solver [41].

Since our typechecking algorithm is syntax-oriented, its efficiency is dependent on the decidability of the SMT solver. That is, the completeness of the algorithm is relative to the completeness of the required constraint solving problem. However, in all the tests made, the programs were always typechecked in efficient time so we believe our typechecking algorithm is, in general, efficient.

We now present the type-checking algorithm, $tc(\Delta, \mathcal{S}, \mathsf{r}, e)$, that given as input a typing environment Δ, a constraint set $\mathcal{S}$ (both initially empty), the current computational context r (begins as $\bot$) and a expression e, returns as output the type of expression e, if successful, or a typing error otherwise.

We begin with unit values, booleans, integers, abstractions, variants, and identifiers. The typechecking procedure is as expected from our typing rules

$tc(\Delta, \mathcal{S}, \mathsf{r}, ()) \stackrel{\text{def}}{=} \mathtt{cmd}^{\bot}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, \mathbf{true}) \stackrel{\text{def}}{=} \mathtt{Bool}^{\bot}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, 1) \stackrel{\text{def}}{=} \mathtt{Int}^{\bot}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, \lambda(x{:}\tau^{\mathsf{s}}).e) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta \cup \{x{:}\tau^{\mathsf{s}}\}, \mathcal{S}, \mathsf{r}, e)\ \mathbf{in}\ (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{t}})^{\bot}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, \#m(e)\ \mathbf{as}\ \{\ldots, m : \tau^{\mathsf{s}}, \ldots\}^{\mathsf{t}}) \stackrel{\text{def}}{=}$

$\quad \mathbf{if}\ tc(\Delta, \mathcal{S}, \mathsf{r}, e) <: \tau^{\mathsf{s}}\ \mathbf{then}\ \{\ldots, m : \tau^{\mathsf{s}}, \ldots\}^{\mathsf{t}}\ \mathbf{else}\ \mathtt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, x) \stackrel{\text{def}}{=} \mathbf{if}\ x \in \Delta\ \mathbf{then}\ \Delta(x)\ \mathbf{else}\ \mathtt{typerror}$

Then we have our type cast constructs,

$tc(\Delta, \mathcal{S}, \mathsf{r}, [\tau^{\mathsf{s}}]e) \stackrel{\text{def}}{=} \mathbf{if}\ tc(\Delta, \mathcal{S}, \mathsf{r}, e) <: \tau^{\mathsf{s}}\ \mathbf{then}\ \tau^{\mathsf{s}}\ \mathbf{else}\ \mathtt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r},]\mathsf{s}[e) \stackrel{\text{def}}{=} \mathbf{if}\ \mathsf{r} \leq \mathsf{s}\ \mathbf{then}\ tc(\Delta, \mathcal{S}, \mathsf{s}, e)\ \mathbf{else}\ \mathtt{typerror}$

where, for the up-cast primitive, we check if the type of expression e is a subtype of the upcast type. While in the context cast operator we check if the current computational context security level is lower or equal than the cast security level, if so then we typecheck expression e under the cast computational context security level, otherwise a type error is returned.

When typechecking an application

$tc(\Delta, \mathcal{S}, \mathsf{r}, e_1(e_2)) \stackrel{\text{def}}{=}$

$\quad \mathbf{if}\ tc(\Delta, \mathcal{S}, \mathsf{r}, e_1) = (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{t}})^{\mathsf{q}}\ \mathbf{and}\ \tau^{\mathsf{s}} <: tc(\Delta, \mathcal{S}, \mathsf{r}, e_2)\ \mathbf{and}\ \mathsf{r} \leq \mathsf{r}'\ \mathbf{then}$

$\qquad \mathbf{if}\ \mathcal{S} \models e_2 \doteq \mathsf{v}\ \mathbf{then}\ \sigma^{\mathsf{t}}\{\mathsf{v}/x\}\ \mathbf{else}\ \sigma^{\mathsf{t}}\{\top/x\}$

$\quad \mathbf{else}\ \mathtt{typerror}$

we match the first expression's type with a dependent product type, check whether the declared type (for the function's parameter) is a subtype of the argument type, and check if its computational context security level is below the function's computational context security level. This is what one would expected but notice that, afterwards, we have to approximate the runtime value for the argument expression via $\mathcal{S} \models e_2 \doteq \mathsf{v}$ and then replace the occurrences of x in the return type with the entailed value v, if it is derivable, otherwise we approximate e_2 to $\top$ and replace all occurrences of x in the return type accordingly.

Let us now take a look at the algorithm for the case of a record expression

$$
\begin{array}{l}
tc(\Delta, \mathcal{S}, \mathsf{r}, [\ldots, m_i{:}\tau_i^{\mathsf{s}_i} = e_i, \ldots]) \stackrel{\text{def}}{=} \texttt{in} \\
\quad \texttt{let}\ \Sigma[\ldots \times m_i{:}\tau_i'^{\mathsf{s}_i'} \times \ldots]^{\mathsf{s}} = \mathit{unref}(\mathcal{S}, [\ldots, m_i{:}\tau_i^{\mathsf{s}_i} = e_i, \ldots]) \\
\qquad \texttt{forall}\ e_i\texttt{.}\ \ \sigma_i^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_i) \\
\qquad\qquad \texttt{if}\ \sigma_i^{\mathsf{t}}\ \texttt{is concrete then} \\
\qquad\qquad\quad \texttt{if}\ \sigma_i^{\mathsf{t}} <: \tau_i'^{\mathsf{s}_i'}\ \texttt{then}\ \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\sqcap|\mathsf{s}_i|^{\downarrow}}\ \texttt{else typerror} \\
\qquad\qquad \texttt{else if}\ \sigma_i^{\mathsf{t}} <: \tau_i^{\mathsf{s}_i}\ \texttt{then}\ \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\sqcap|\mathsf{s}_i|^{\downarrow}}\ \texttt{else typerror}
\end{array}
$$

As we already stated, record fields are type annotated. So we first use procedure *unref* to eliminate field dependencies in the given (possibly dependent) record – this step is equivalent to applying rule (T-UNREFINERECORD) – thus obtaining a dependent sum type whose field's types, $\tau_i'^{\mathsf{s}_i'}$, have no field dependencies.

Then, we typecheck each field expression e_i whose type σ_i^{t} may have dependencies (as long it is well-formed). If it is a concrete type (no field dependencies), then we must check whether it is a subtype of the unrefined (concrete) type. Otherwise, if it has field dependencies, we verify if it is a subtype of the declared (record field), possibly dependent, type. If any of these checks fails the algorithm outputs a type error, otherwise returns the dependent sum type obtained with the given field type annotations – that is, $\Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\sqcap|\mathsf{s}_i|^{\downarrow}}$.

To typecheck a field access expression

$$
\begin{array}{l}
tc(\Delta, \mathcal{S}, \mathsf{r}, e.m) \stackrel{\text{def}}{=} \\
\quad \texttt{let}\ \tau^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e)\ \texttt{in} \\
\quad\ \texttt{if}\ \tau = \Sigma[\ldots \times m{:}\sigma^{\ell(\mathsf{u})} \times \ldots]\ \texttt{then} \\
\qquad \texttt{if}\ \mathsf{u}\ \texttt{is concrete then}\ \sigma^{\ell(\mathsf{u})} \\
\qquad \texttt{else (}\ \texttt{if}\ \mathsf{u} = n\ \texttt{and}\ \mathcal{S}\{x \doteq e\} \models x.n \doteq \mathsf{v}\ \texttt{then}\ \sigma^{\ell(\mathsf{v})}\ \texttt{else}\ \sigma^{\ell(\top)}\ \texttt{)} \\
\quad\ \texttt{else typerror}
\end{array}
$$

the procedure first checks if the type of expression e is a dependent sum type.

Afterwards, for the given field we check whether its security level is concrete. If so, then we return the type for the given field.

If the security level of the given field's type has a field dependency n, then the algorithm attempts to approximate the field value via the constraint solving procedure, using fresh

$$tc(\Delta, \mathcal{S}, \mathsf{r},\ \mathbf{ref}_{\tau^{\mathsf{s}}}\ e) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e)\ \mathbf{in}$$
$$\quad \mathbf{if}\ \sigma^{\mathsf{t}} <: \tau^{\mathsf{s}}\ \mathbf{and}\ \mathsf{r} \leq \mathsf{s}\ \mathbf{then}\ (\mathtt{ref}_{\tau^{\mathsf{s}}})^{\perp}\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta, \mathcal{S}, \mathsf{r}, !e) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e)\ \mathbf{in}$$
$$\quad \mathbf{if}\ \sigma^{\mathsf{t}} <: (\mathtt{ref}_{\tau^{\mathsf{s}}})^{\mathsf{t}}\ \mathbf{and}\ \mathsf{t} \leq \mathsf{s}\ \mathbf{then}\ \tau^{\mathsf{s}}\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta, \mathcal{S}, \mathsf{r}, e_1 := e_2) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_1)\ \mathbf{in}$$
$$\ \mathbf{let}\ \tau^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_2)\ \mathbf{in}$$
$$\quad \mathbf{if}\ \sigma^{\mathsf{t}} <: (\mathtt{ref}_{\tau^{\mathsf{s}}})^{\mathsf{t}}\ \mathbf{and}\ \mathsf{r} \sqcup \mathsf{t} \leq \mathsf{s}\ \mathbf{then}\ \mathtt{cmd}^{\perp}\ \mathbf{else}\ \mathtt{typerror}$$

Figure 6.1: Typechecking algorithm: Imperative expressions

variable x to denote the record expression e. If successful then the output is the concrete field's type $\sigma^{\ell(\mathsf{v})}$(using the entailed value v), otherwise the algorithm returns an upward approximation (that is, $\mathsf{v} = \top$), $\sigma^{\ell(\top)}$.

We conclude with the **case** primitive

$$tc(\Delta, \mathcal{S}, \mathsf{r}, \mathbf{case}\ e(\overline{m \cdot x : \tau^{\mathsf{s}} \Rightarrow e})) \stackrel{\text{def}}{=} \mathbf{let}\ \{\overline{m : \tau'^{\mathsf{s}'}}\}^{\mathsf{q}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e)\ \mathbf{in}$$
$$\mathbf{if}\ \{\overline{m : \tau'^{\mathsf{s}'}}\}^{\mathsf{q}} \not<: \{\overline{m : \tau^{\mathsf{s}}}\}^{\mathsf{t}}\ \mathbf{then}\ \mathtt{typerror}$$
$$\mathbf{else}\ \mathbf{forall}\ e_i.\ \sigma^{\mathsf{q}_i} = tc(\Delta, x_i : \tau_i^{\mathsf{s}_i}, \mathcal{S}, \mathsf{r}, e_i)$$
$$\mathbf{return}\ \sigma^{\sqcup \mathsf{q}_i}$$

The algorithm starts by checking if the expressions being case-analysed is a variant type, namely a subtype of the variant type obtained through the variant label's type annotations. Then, typechecks all the branches of the case, expressions e_i, and checks whether their base type is the same between all branches. The output will then be the base type of the branches expressions with the lowest upper bound of the security levels of all branches.

The remaining cases of the algorithm are a direct translation of the typing rules and can be found in Figure 6.1, and Figure 6.2.

6.2 Implementation

In this section we discuss some implementation details of our prototype typechecker.

Our typechecker is implemented in Scala and uses Z3 SMT solver for constraint solving during the typechecking procedure. As input for Z3 SMT solver we use SMT-LIB 2.0 [1] scripts with Z3's extensions, which we generate with a scala library [2] (a Generic SMT Front-End for Z3) for that purpose. Calls to the solver only occur when typechecking a function application, a record, field access, and upcast operations (since the cast type may be a dependent sum type).

[1] `http://www.smt-lib.org/`
[2] `https://bitbucket.org/tvcsantos/smtlib/overview`

$tc(\Delta, \mathcal{S}, \mathsf{r},\ \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) \stackrel{\text{def}}{=}$
$\quad \textbf{let}\ \tau^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_1)\ \textbf{in}$
$\quad \textbf{if}\ \tau = \texttt{Bool}\ \textbf{then}$
$\quad (\ \textbf{let}\ \tau_2{}^{\mathsf{s}_2} = tc(\Delta, \mathcal{S} \cup \{e_1 \doteq \texttt{true}\}, \mathsf{r} \sqcup \mathsf{s}, e_2)\ \textbf{in}$
$\quad\ \ \textbf{let}\ \tau_3{}^{\mathsf{s}_3} = tc(\Delta, \mathcal{S} \cup \{e_1 \doteq \texttt{false}\}, \mathsf{r} \sqcup \mathsf{s}, e_3)\ \textbf{in}$
$\quad\ \ \textbf{if}\ \tau_2 = \tau_3\ \textbf{then}\ \tau_2^{\mathsf{s} \sqcup \mathsf{s}_2 \sqcup \mathsf{s}_3}\ \textbf{else}\ \texttt{typerror}$
$\quad)\ \textbf{else}\ \texttt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r},\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2) \stackrel{\text{def}}{=}$
$\quad \textbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_1)\ \textbf{in}$
$\quad \textbf{if}\ \sigma^{\mathsf{t}} <: \tau^{\mathsf{s}}\ \textbf{then}\ tc(\Delta \cup \{x{:}\tau^{\mathsf{s}}\}, \mathcal{S}\{x \doteq e_1\}, \mathsf{r}, e_2)$
$\quad \textbf{else}\ \texttt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, \{e_1, \ldots, e_n\}) \stackrel{\text{def}}{=}$
$\quad \textbf{forall}\ e_i\ \textbf{do}$
$\quad (\ \textbf{let}\ \tau_i{}^{\mathsf{s}_i} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_i)\ \textbf{in}$
$\quad\ \ \textbf{if}\ \tau_i{}^{\mathsf{s}_i} \not<: \tau_1{}^{\mathsf{s}_1}\ \textbf{then}\ \texttt{typerror}\ )$
$\quad \textbf{return}\ \tau_1{}^{*\mathsf{s}_1}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, e_1{::}e_2) \stackrel{\text{def}}{=} \textbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_1)\ \textbf{in}$
$\quad \textbf{let}\ \tau'^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_2)\ \textbf{in}$
$\quad \textbf{if}\ \tau' = \tau^*\ \textbf{then}$
$\quad (\textbf{if}\ \sigma^{\mathsf{t}} <: \tau^{\mathsf{s}}\ \textbf{then}\ \tau^{*\mathsf{s}}\ \textbf{else}\ \texttt{typerror})\ \textbf{else}\ \texttt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r},\ \textbf{foreach}(e_1, e_2, x.y.e_3)) \stackrel{\text{def}}{=}$
$\quad \textbf{let}\ \sigma'^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_1)\ \textbf{in}$
$\quad \textbf{if}\ \sigma' = \sigma^*\ \textbf{then}$
$\quad (\ \textbf{let}\ \tau^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_2)\ \textbf{in}$
$\quad\ \ \textbf{let}\ \tau^{\mathsf{q}} = tc(\Delta \cup \{x{:}\sigma^{\mathsf{t}}, y{:}\tau^{\mathsf{s}}\}, \mathcal{S}, \mathsf{r}, e_3)\ \textbf{in}\ \tau^{\mathsf{s} \sqcup \mathsf{t} \sqcup \mathsf{q}}\ )$
$\quad \textbf{else}\ \texttt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, e_1 \vee e_2) \stackrel{\text{def}}{=} \textbf{let}\ \tau^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_1)\ \textbf{in}$
$\quad \textbf{let}\ \sigma^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e_2)\ \textbf{in}$
$\quad \textbf{if}\ \tau = \texttt{Bool}\ \textbf{and}\ \sigma = \texttt{Bool}\ \textbf{then}\ \texttt{Bool}^{\mathsf{t} \sqcup \mathsf{s}}\ \textbf{else}\ \texttt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, \neg e) \stackrel{\text{def}}{=} \textbf{let}\ \tau^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, e)\ \textbf{in}$
$\quad \textbf{if}\ \tau = \texttt{Bool}\ \textbf{then}\ \texttt{Bool}^{\mathsf{s}}\ \textbf{else}\ \texttt{typerror}$

$tc(\Delta, \mathcal{S}, \mathsf{r}, V_1 = V_2) \stackrel{\text{def}}{=} \textbf{let}\ \tau^{\mathsf{t}} = tc(\Delta, \mathcal{S}, \mathsf{r}, V_1)\ \textbf{in}$
$\quad \textbf{let}\ \sigma^{\mathsf{s}} = tc(\Delta, \mathcal{S}, \mathsf{r}, V_2)\ \textbf{in}$
$\quad \textbf{if}\ \tau = \sigma\ \textbf{then}\ \texttt{Bool}^{\mathsf{t} \sqcup \mathsf{s}}\ \textbf{else}\ \texttt{typerror}$

Figure 6.2: Typechecking algorithm: Pure expressions

Dependent sum types are encoded as Z3's recursive datatypes, for instance to encode the type of record `[uid = 1, count = 23]` we have:

```
( declare-datatypes ()
      ( ( Record<uid^Int.count^Int> ( mkRecord (uid Int) (count Int) ) ) ) )
```

Notice that we do not encode security labels since they play no role in this step of our analysis (constraint solving dependencies).
Let us now see some examples (using our prototype's syntax) of calls to the SMT solver.

Example 35 Suppose we have the following program:

```
(fun x: Sigma[ uid: int^BOT, count: int^U(uid) ]^BOT =>
                  if x.uid == 1 then x.count else [int^U(1)] 0);;
```

Then we need to make a call to the SMT solver in the field access `x.count` since it has a dependency in its security label. To do so, we first add to the current constraint set, $\{$`x.uid == 1` $\doteq$ `true`$\}$ (because we are typing under the then-branch), followed by the constraint $\{y \doteq$`x`$\}$, for a fresh y, since we are going to "unrefine" record `x`.

Then, to generate the intended SMT script, we first declare a constant `fconst` to entail the value of the field dependency, that is $\{y \doteq$`x`, `x.uid == 1`$\doteq$ `true`$\} \models y.$`uid`$\doteq$`fconst`.

This constraint set is encoded as the universal closure of the formula

$$((x.uid = 1) \equiv \texttt{true}) \wedge ((x = y) \equiv \texttt{true})$$

so to derive knowledge from it we generate the following logical implication:

$$\forall_{x,y}\ ((x.uid = 1) \equiv \texttt{true}) \wedge ((x = y) \equiv \texttt{true}) \implies y.uid = \texttt{fconst}$$

Thus we generate the following SMT script:

```
(declare-datatypes ()
      ((Record<uid^Int.count^Int> (mkRecord (uid Int) (count Int)))))

(declare-const fconst Int)
(declare-const y Record<uid^Int.count^Int>)
(declare-const x Record<uid^Int.count^Int>)

(assert
   (forall ( (x Record<uid^Int.count^Int>) (y Record<uid^Int.count^Int>) )
      (=> (and (equiv (= ((as uid (Int)) x) 1) true) (equiv (= y x) true) )
          (= ((as uid (Int)) y) (as fconst (Int)))) ) )
(check-sat)
(get-model)
```

and then ask the solver for a model, obtaining the value 1 for the constant `fconst`.

This example shows how we encode record values (and dependent sum types), field projection and the entailment that allows our analysis to "unrefine" record `x`, obtaining the concrete dependent sum type Σ`[uid: int`$^{\perp}$ $\times$ `count: int`$^{\mathsf{U}(1)}$`]`.
Next we show how we entail functional dependencies.

Example 36 In the following function we retrieve the field `count` of a dependent record given two parameters of the function

```
fun u: int^BOT, s: int^BOT,
    r: Sigma[uid:int^BOT, sid:int^BOT, count:int^U(uid,sid)]^BOT =>
     [int^U(u,s)] (if(r.uid == u and r.sid == s) then r.count else 0 );;
```

So when we call the solver for the field access operation `r.count`, we attempt to "unrefine" record `r` like we did in the previous example.

However, the formula generated will be `unsat` since no constant can be entailed from the constraint set, that is from $\{$`r.uid == u and r.sid == s`$, y \doteq$`r`$\}$ $\models$`y.uid`$=$ v_1 and $\{$`r.uid == u and r.sid == s`$, y \doteq$`r`$\}$ $\models$`y.sid`$=$ v_2 we cannot entail constants for v_1 and v_2.

For these cases, we declare an uninterpreted function symbol whose parameters match all constraints free variables that have the same type as the dependency we are attempting to eliminate. That is, for instance, to eliminate dependency `sid`, we declare function symbol `f_y` with two parameters of type `Int`, corresponding to the free variables `u` and `s` of constraints in $\{$`r.uid == u and r.sid == s`$, y \doteq$`r`$\}$.

Then, we add axioms for the projection of each parameter of the uninterpreted function symbol, in this example we add: $\forall_{u,s}\ f\_y(u,s) = u\ \vee\ \forall_{u,s}\ f\_y(u,s) = s$.

Finally, we generate the same kind of formula we did in the previous example but instead of entailing the value of the field dependency via a constant, we use the uninterpreted function symbol `f_y`:

$$(\forall_{u,s}\ f\_y(u,s) = u\ \vee\ \forall_{u,s}\ f\_y(u,s) = s)$$

$$\wedge$$

$$(\forall_{r,u,s,y}\ ((r.uid = u\ \wedge\ r.sid = s) \equiv true\ \wedge\ (y = r) \equiv true)\ \Longrightarrow\ y.sid = f\_y(u,s))$$

Thus the generated SMT-LIB script to eliminate field dependency `sid` is

```
(declare-datatypes ()
      ( ( Record<uid^Int.sid^Int.count^Int> ( mkRecord (uid Int) (sid Int)
         (count Int) ) ) ) )

(declare-const s Int)
(declare-const r Record<uid^Int.sid^Int.count^Int>)
```

```
(declare-const u Int)
(declare-const y Record<uid^Int.sid^Int.count^Int>)

(declare-fun f_y (Int Int) (Int) )

(assert
 (and (or (forall ( (u Int) (s Int) ) (= ( (as f_y (Int) ) u s) u))
          (forall ( (u Int) (s Int) ) (= ( (as f_y (Int) ) u s) s)) )
   (forall ( (r Record<uid^Int.sid^Int.count^Int>) (u Int) (s Int)
             (y Record<uid^Int.sid^Int.count^Int>) )
                (=> ( and (equiv (and
                            (= ((as uid (Int)) r) u)
                            (= ((as sid (Int)) r) s)) true)
                     (equiv (= y r) true) )
                  (= ( (as sid (Int) ) y) ( (as f_y (Int) ) u s) ) ) )
) )
(check-sat)
(get-model)
```

which will return a model with the brujin index 2 that represents the second parameter of the uninterpreted function symbol `f_y`. This means we eliminate dependency `sid` with the function parameter `s`, as intended.

We now point out some of our prototype's open problems.

In the next section, we illustrate with some examples that can be run in our prototype.

6.3 Examples

We now present some examples using the prototype typechecker's syntax. The full set of examples can be found in Appendix A.

We use the following lattice definition, part of the prototype's configuration, in the following examples

```
forall [x] A(_,x) ~> PC(_,x)
forall [x] U(x) ~> A(x,_)
forall [x] U(x) ~> PC(x,_)
```

which corresponds to the axioms presented in Chapter 1.

6.3.1 Simple Examples

We start with some simple examples to illustrate our typechecker.

Input:

```
(fun x: Sigma[usr: int^BOT, counter: int^U(usr)]^BOT =>
      if x.usr == 1 then x.counter else [int^U(1)] 0);;
```

Output:

```
Type: ( Pi(x: Sigma[usr: int^BOT, counter: int^U(usr)]^BOT).BOT;
        int^U(1) )^BOT
```

Here we declare a function that given a record (storing the counter of a user) retrieves the counter of user `1` or returns `0` if the record does not corresponds to user's `1`.

So while typechecking the body of the function, and more concretely the conditional's then-branch, we obtain security level `U(usr)` from the projection of field `counter` but this is not well-formed (field dependencies only occur in the scope of a dependent sum type), so we have to eliminate the field dependency `usr`.

Since we know `x.usr == 1`, we obtain security level `U(1)` in the then-branch. So, since we are raising the security level of expression `0` to `U(1)`, the function's body is typed with security level `U(1)`.

Now suppose we have
Input:

```
(fun x:Sigma[usr: int^BOT, counter: int^U(usr)]^BOT =>
      if x.usr == 1 then x.counter else [int^U(2)] 0);;
```

Output:

```
Type: ( Pi(x: Sigma[usr: int^BOT, counter: int^U(usr)]^BOT).BOT;
        int^U(TOP) )^BOT
```

As before, the then-branch is typed with security level `U(1)`. However, we are up-casting the else-branch to security level `U(2)`.

So the security level of the conditional is the least upper bound of the security level of its branches, that is `U(1)`$\sqcup$`U(2) = U(`$\top$`)`, hence the security level of the function's result type is `U(TOP)`.

Let us look further into dependent functions
Input:

```
let f = (fun x:int^BOT => [int^U(x)] x) in f ;;
```

Output: `Type: (Pi(x: int^BOT).BOT; int^U(x))^BOT`

We upcast the body of the function so we can type `f` as a dependent function type.
So if we call function `f` with integer `1`.

Input:

```
let f = (fun x:int^BOT => [int^U(x)] x) in f(1) ;;
```

Output: `Type: int^U(1)`

Then we obtain a result of security level `U(1)`.
However, if we sum the results of invoking `f` with integers `1` and `2`

Input:

```
let f = (fun x:int^BOT => [int^U(x)] x) in f(1) + f(2) ;;
```

Output: `Type: int^U(TOP)`
Then we obtain a result of security level `U(TOP)`.

Let us now see some examples of dependent records
Input:

```
[Sigma[usr: int^BOT, counter: int^U(usr)]^BOT]
    [usr: int^BOT = 1, counter :int^U(1) = 2] ;;
```

Output:

```
Type: Sigma[usr: int^BOT, counter: int^U(usr)]^BOT
```

In this snippet, we are attempting to cast the record value with a dependent sum type where field `counter`'s security level depends on field `usr`.

Thus, the typechecker verifies if the record value can be refined into the cast type, which in this case is true since the value of field `usr` in the record value matches the value of the security level of field `counter` in the record value.

Now suppose the value of field `usr` changes
Input:

```
[Sigma[usr: int^BOT, counter: int^U(usr)]^BOT]
   [usr: int^BOT = 2, counter :int^U(1) = 2] ;;
```

Output:

```
Wrong type:
Expected declared type Sigma[usr: int^BOT, counter: int^U(usr)]^BOT
but found type Sigma[usr: int^BOT, counter: int^U(1)]^BOT
```

Then the typechecker will not be able to refine the record value, hence it gives a type error stating the types do not match.

We now refer back to Example 12 in Chapter 3.
Input:

```
let topSecrets = ( let h = [bool^TOP] true in
                   if h then {1,2,3,4,5}
                   else {} : { int^TOP } ) ;;

foreach (x in topSecrets) with count = 0 do count + 1 ;;
```

Output: `Type: int^TOP`

As we have seen, the result of counting the elements of a collection has its elements security level. So in this case, since we are counting a collections of integers classified at $\top$, the result is only observable at security level $\top$.
However,
Input:

```
let boxed = { [usr: int^BOT = 42, pwd :int^TOP = 1234],
              [usr: int^BOT = 24, pwd :int^TOP = 4321] };;

foreach (x in boxed) with count = 0 do count + 1 ;;
```

Output: `Type: int^BOT`
we can observe the boundaries of a collection of records, and the boundaries of the records themselves, since they are both classified at $\bot$.

This is secure because we still cannot observe the value of field `pwd` (classified at $\top$) even if we can see the record containing it.

To conclude this set of examples, we now discuss some examples with references.
Input:

```
let low = ref 0 in
  if [bool^TOP] true then
     low := 1 ;;
```

Output: `Insecure flow detected from label TOP to label BOT!`
As one would expect, this is an insecure assignment operation because data stored in `low` will depend on the value of a higher classified value.

The converse, as seen before, is secure.
Input:

```
let high = ref [int^TOP] 0 in
  if true then
    high := 1 ;;
```

Output: `Type: cmd^BOT`

Now suppose we try to circumvent explicit flows with implicit flows, by defining a write operation that writes on a low container.

Input:

```
let low = ref 0 in
 let write = (fun r: ref(int^BOT)^BOT, x:int^BOT => r := x)
  if [bool^TOP] true then
      write(low,1) ;;
```

Output: `Insecure flow detected from label TOP to label BOT!`

However, this implicit flow is detected because function `write` was typed under computational context $\bot$ so it can only be invoked in computational contexts that are lower or equal than $\bot$.

On the other hand, this condition leads to some false negatives like the following

Input:

```
let high = ref [int^TOP] 0 in
  let write = ( fun r: ref(int^TOP)^BOT, x: int^BOT => r := x )
  in if [bool^TOP] true then
          write(high,1) ;;
```

Output: `Insecure flow detected from label TOP to label BOT!`

The snippet above is secure since we are writing on a container with security level $\top$ under a computational context with the same security level. But since we are performing the assignment via function `write`, then our typechecker conservatively rejects this program as being insecure.

So in order to typecheck this write operation we must raise the computational context under which we type the function `write`, we achieve this with primitive $]\mathsf{s}[e$:

Input:

```
let high = ref [int^TOP] 0 in
 let write = ( ]TOP[ (fun r: ref(int^TOP)^BOT, x: int^BOT => r := x ) )
  in if [bool^TOP] true then
          write(high,1) ;;
```

Output: `Type: cmd^TOP`

Now the typechecker accepts the above program as secure.

6.3.2 A Conference Manager System

Let us see how we encode our conference manager system from Chapter 1. We begin with type declarations for the collections `Users`, `Submissions`, and `Reviews` as well the declaration of the collections themselves.

Input:

```
typedef usr_type =
   { ref (Sigma[ uid: int^BOT, name: int^U(uid),
                 univ: int^U(uid), email: int^U(uid) ]^BOT)^BOT };;

typedef sub_type =
   { ref (Sigma[ uid: int^BOT, sid: int^BOT, title: int^A(uid,sid),
                 abst: int^A(uid,sid), paper: int^A(uid,sid) ]^BOT)^BOT };;

typedef rev_type =
   { ref (Sigma[ uid: int^BOT, sid: int^BOT, PC_only: int^PC(uid,sid),
                 review: int^A(TOP,sid), grade: int^A(TOP,sid)]^BOT)^BOT };;

let Users = ref {}: usr_type ;;
let Submissions = ref {}: sub_type ;;
let Reviews = ref {}: rev_type ;;
```

Next we encode `viewAuthorPapers` (Example 2 from Chapter 1):

Input:

```
typedef ret_type =
   { Sigma[ uid: int^BOT, sid: int^BOT, title: int^A(uida,sid),
            abst:int^A(uida,sid), paper:int^A(uida,sid) ]^BOT } ;;

let viewAuthorPapers = fun uida: int^BOT =>
     [ ret_type ]( foreach(x in !Submissions) with y = {}: ret_type do
                       let tuple = !x in
                         if(tuple.uid == uida) then tuple::y else y ) ;;

let n = 42 in (viewAuthorPapers(n)) ;;
```

Output:

```
Type: { Sigma[uid: int^BOT, sid: int^BOT, title: int^A(42, sid),
              abst: int^A(42, sid), paper: int^A(42, sid)]^BOT }
```

As we have seen, function `viewAuthorPapers` is a dependent function since its return type (declared as `ret_type` above) depends on parameter `uida`.

So if we invoke function `viewAuthorPapers` with identifier `n`, the typechecker will determine the value of `n` – given the knowledge obtained from the declaration of identifier `n`– and type the call with type $\texttt{ret\_type}\{^{42}/_{\texttt{uida}}\}$.

Likewise for function `viewAssignedPapers` (Example 3 of Chapter 1):
Input:

```
typedef sub_elem = Sigma[uid: int^BOT, sid: int^BOT, title: int^A(uid,sid),
    abst:int^A(uid,sid), paper:int^A(uid,sid) ]^BOT ;;

typedef sub = { sub_elem } ;;

let viewAssignedPapers = fun uidr: int^BOT =>
     ( foreach(x in !Reviews) with res_x = {}:sub do
         let tuple_rev = !x in
          if(tuple_rev.uid == uidr ) then
            ( foreach(y in !Submissions) with res_y = {}:sub do
                let tuple_sub = !y in
                 if(tuple_sub.sid == tuple_rev.sid) then
                   tuple_sub::res_y
                 else res_y )
          else res_x ) ;;

let r = first(viewAssignedPapers(42)) in r ;;
```

Output:

```
Type: Sigma[uid: int^BOT, sid: int^BOT, title: int^A(uid, sid),
             abst: int^A(uid, sid), paper: int^A(uid, sid)]^BOT
```

Let us now see the code snippet (Example 4):
Input:

```
let t = first(
  ( foreach(x in !Submissions) with y = {}: { int^A(42,70) } do
      let t_sub = !x in
        if(t_sub.uid == 42 and t_sub.sid == 70 ) then t_sub.title::y else y) )
in foreach(x in !Submissions) with y = skip do
    let t_sub = !x in
     if(t_sub.uid == 42 and t_sub.sid == 70) then
        let new_rec = [uid: int^BOT = t_sub.uid, sid: int^BOT = t_sub.sid,
                       title: int^A(uid, sid) = t,
                       abst: int^A(uid, sid) = t_sub.abst,
                       paper: int^A(uid, sid) = t_sub.paper ]
        in x := new_rec ;;
```

Output: `Type: cmd^BOT`
which we saw in Chapter 1 to be secure.

And now a slightly modified version of the same code snippet, where we attempt to associate to author with id 32 information of author with id 42:
Input:

```
let t = first(
  ( foreach(x in !Submissions) with y = {}: { int^A(42,70) } do
     let t_sub = !x in
       if(t_sub.uid == 42 and t_sub.sid == 70 ) then t_sub.title::y else y) )
in foreach(x in !Submissions) with y = skip do
    let t_sub = !x in
     if(t_sub.uid == 32) then
        let new_rec = [uid: int^BOT = t_sub.uid, sid: int^BOT = t_sub.sid,
                       title: int^A(uid, sid) = t,
                       abst: int^A(uid, sid) = t_sub.abst,
                       paper: int^A(uid, sid) = t_sub.paper ]
        in x := new_rec ;;
```

Output:

```
Wrong type: Expected declared type
Sigma[uid: int^BOT, sid: int^BOT, title: int^A(uid, sid),
      abst: int^A(uid, sid), paper: int^A(uid, sid)]^BOT
but found type
Sigma[uid: int^BOT, sid: int^BOT, title: int^A(42, 70),
     abst: int^A(32, sid), paper: int^A(32, sid)]^BOT
```

As expected, our typechecker deems the above code insecure because the declared dependent sum type (obtained from the declared fields types) does not match the expressions `new_rec`'s dependent sum type. In particular, the type for field `title` does not match.

Recall the operation `addCommentSubmission` (Example 5 from Chapter 1)
Input:

```
let comment = fun u: int^BOT, s: int^BOT, r: sub_elem =>
    [ int^A(u,s) ] (if(r.uid == u and r.sid == s) then r.paper else 1) in

let addCommentSubmission = fun uid_r: int^BOT, sidr: int^BOT =>
   ( foreach(p in viewAssignedPapers(uid_r)) with dummy = skip do
      if(p.sid == sidr) then
       ( foreach(y in !Reviews) with dummy2 = skip do
          let trev = !y in
           if(trev.sid == p.sid) then
            ( let up_rec = [uid: int^BOT = trev.uid,
                            sid: int^BOT = trev.sid,
                            PC_only:int^PC(uid,sid) = comment(p.uid,p.sid,p),
                            review: int^A(TOP, sid) = trev.review,
                            grade: int^A(TOP, sid) = trev.grade ]
              in y := up_rec ) ) )
in addCommentSubmission;;
```

Output: `Type: ( Pi(uid_r: int^BOT, sidr: int^BOT).(cmd^BOT))^BOT`

This program is secure, as we have seen in Chapter 1, because we can raise the security level of expression `comment(p.uid,p.sid,p)` to the declared type for field `PC_only` via subtyping.

However, if we remove the last conditional, **`if`**`(trev.sid == p.sid)`:
Input:

```
let comment = fun u: int^BOT, s: int^BOT, r: sub_elem =>
    [ int^A(u,s) ] (if(r.uid == u and r.sid == s) then r.paper else 1) in

let addCommentSubmission = fun uid_r: int^BOT, sidr: int^BOT =>
   ( foreach(p in viewAssignedPapers(uid_r)) with dummy = skip do
      if(p.sid == sidr ) then
       ( foreach(y in !Reviews) with dummy2 = skip do
          let trev = !y in
           if(trev.sid == p.sid ) then
            ( let up_rec = [uid: int^BOT = trev.uid,
                            sid: int^BOT = trev.sid,
                            PC_only:int^PC(uid,sid) = comment(p.uid,p.sid,p),
                            review: int^A(TOP, sid) = trev.review,
                            grade: int^A(TOP, sid) = trev.grade ]
              in y := up_rec ) ) )
in addCommentSubmission;;
```

Output:

```
Wrong type: Expected declared type
Sigma[uid: int^BOT, sid: int^BOT, PC_only: int^PC(uid, sid),
       review: int^A(TOP, sid), grade: int^A(TOP, sid)]^BOT
but found type
Sigma[uid: int^BOT, sid: int^BOT, PC_only: int^A(TOP, sidr),
       review: int^A(TOP, sid), grade: int^A(TOP, sid)]^BOT
```

Then we are not able to raise the security level of `comment(p.uid,p.sid,p)` (which would be `A(`$\top$`,sidr)` because of the first conditional) to the required type. This is, of course, detected by our typechecker.

We give more examples tested in our prototype typechecker in Appendix A.

6.4 Remarks

In this chapter, we presented a typechecking algorithm that lead to the implementation of a prototype typechecker. We proceeded with the discussion of some technical details regarding the prototype's implementation, followed by a set of examples tested in our prototype.

We now discuss some of the open problems of our implementation.

6.4.1 Open Problems

Our prototype is a *proof-of-concept* typechecker of our dependent information flow types, and not a full-fledge programming language. As such, it naturally has some open problems or unimplemented features.

For instance, we do not have variant values and types implemented nor do we have records and collections indexes in our prototype. This was essentially due to time restrictions since these features were not necessary in our illustrative examples they were left to be implemented later.

We also do not have strings in our language, this simplifies the analysis to be done via the SMT solver since we would have to encode strings and string equality in Z3 SMT Solver (which are not natively supported).

To conclude, we make a comparison to implementations of other type-based information flow approaches. While not exhaustive, this comparison establishes the main differences between different approaches that lead to language's implementations (ours, however, is just a typechecker).

6.4.2 Comparison to Other Tools

In the past years, there has been some some effort in implementations of programming languages that ensure noninterference via a type-based information flow analysis. While we developed a prototype typechecker, in this section we will discuss some of these implementations and how they relate to our prototype with respect to the expressiveness of the security policies.

We start with Flow Caml [57], an extension of Objective Caml with a type system to trace information flow [51].

Types in Flow Caml have an associated security label, forming a lattice of security levels, such that a typechecked program complies with the non-interference property. An interesting feature of this language is its type inference algorithm.

Jif [42, 45] extends Java with static analysis of information flow with a decentralised label model [44] (DLM) so it has the notion of principals, principal hierarchy, integrity and confidentiality constraints and robust declassification. The key idea in DLM is to have labels classify data such that it denotes the principal that owns the data and a list of principals to whom the owner gives reading permissions. If a principal is not listed as a reader in a data's label then it cannot read the data.

Both previous tools are unable to express fine-grained security policies like ours, where labels may be dependent on runtime values. Jif, however, is able to express dynamic policies through dynamic labels, which we do not have in our setting.

Other implementations of programming languages for verifying system security have emerged: Jeeves [67], a DSL library for Scala that enforces noninterference during execution time; Fabric [34], a high-level programming language for open distributed applications

based on Jif; Paragon [9], a programming language that uses Flow Locks-based policy language [8] to enforce security policies; F* [61], a dependently typed ML-style programming language based on prior work [59].

In Paragon, we have security policies through Flow Locks policy language and types instead of labelled types. On one hand this could be an advantage to a programmer by allowing him/her to focus on security policies as orthogonal to the program being developed; but on the other hand, it can be cumbersome to write expressive security policies while with labelled types one can effortlessly write such policies.

Moreover, although it is possible to have policies dependent on runtime principals (dynamic policies, much like in Jif) it does not seem possible to express value-dependent security policies as we do with our dependent information flow types.

Regarding F*, it can arguably encode the same type of value-dependent security policies as we do in our approach however we argue that in our approach doing so is more lightweight and simple since it does not involve axiomatizing security labels, lattice and its operators via logic formulae. Instead we simply require data to be annotated with value-dependent security labels that express security concerns.

The following chapter closes this thesis with some concluding remarks and possible directions for this thesis work.

CHAPTER 7

CONCLUSIONS

In this thesis we introduced and studied a novel theory of dependent information flow types, which provide a direct, natural and elegant way to express and statically enforce fine grained security policies on programs (Chapter 3).

In our framework, the security level of data types, rather than just the data types themselves, may depend on runtime values, unlike in traditional dependent type systems. We have illustrated, including by means of many examples, how the proposed approach provides a general, expressive and fine grained way to formulate realistic, yet challenging, security policies (Chapter 1, Chapter 4). Namely, we have showed how we can reason about data confidentiality in data-centric systems by means of DML primitives encodings in our approach (Chapter 5).

Our development is carried out on top of a minimalistic λ-calculus with general references and collections (Chapter 2), thus adding generality and application scope to the approach. Our main technical results are expressed by type safety and non-interference theorems (Chapter 4), which ensure the soundness of our value dependent information flow analysis: well-typed programs do not disclose information in ways violating the prescribed security polices.

Lastly, we have presented a typechecking algorithm that lead to the implementation of a *proof-of-concept* typechecker prototype (`http://ctp.di.fct.unl.pt/DIFTprototype/`) that already allows us to verify many interesting examples (Chapter 6). A live version of the tool is also avaiable at Microsoft's rise4fun `http://rise4fun.com/DIFT/`.

7.1 Future Work

We point out some possible directions for this thesis work.

7.1.1 Pure Constraints

Reasoning about the identity of dependent security labels, e.g, necessary to eliminate dependent sum type field dependencies or approximate dependent function argument values, requires runtime values to be approximated by a given constraint system.

It may also be the case that the elimination of a dependent sum type results in replacing field dependencies with another dependent sum type's field identifier (e.g., an assignment operation in a conditional's then branch whose logical condition relates the field dependency with another record field, instead of a value), as long the final type remains well-formed.

The constraint system used to deduce approximations of runtime values can only contain pure expressions thus disallowing constraints containing dereferences, however, this is a natural restriction that in our experience does not seem to limit much the expressiveness of the approach, but that deserves further study.

7.1.2 Refinement Types

Also, it would be interesting to investigate formulations of our type system integrating notions of type refinement (e.g, [66]), and type inference. The former would increase the expressiveness of the security policies, namely with refinement types one could be conceive some form of declassification of information.

For instance, going back to our conference manager system, one could prevent authors from reading submission's reviews (initially classified at `PC` level) until the author's notification process started. One could express such concern by having reviews typed as

$$\{x : \mathtt{str}^{\mathsf{PC(uid,sid)}} \mid \mathrm{authorNotification}(\mathsf{sid}) => \mathsf{A}(\top, \mathsf{sid})\}$$

such that predicate $\mathrm{authorNotification}(\mathsf{sid})$ only held whenever the process of notifying authors of submission `sid` started, then the security level would be declassified to `A(⊤,sid)` so all submission's authors could see the reviews.

Notice that this is not a typical refinement type that usually take the form $\{x : \tau \mid \phi(x)\}$, here we would associate to the type a security label and a logical formulae that also includes a security label (for declassification purposes), so it would be something like $\{x : \tau^{\mathsf{s}} \mid \phi(x, \mathsf{v}) => \mathtt{t}\}$ (where v is a label index) to state that whenever ϕ holds, the security type associated to τ is $\mathtt{t}$.

One interesting point of this approach would be that one could have predicates depending on the security label dependencies instead of just the language's terms. To the best of our knowledge, declassification has not been studied within the context of value-dependent security labels.

7.1.3 Access Control

As another follow up topic, since information flow analysis per se is not enough to ensure full data security guarantees, we would like to investigate the combination of our

dependent information flow types with an adequate form of role-based access control. Again, as explored previously by [10], refinement types could play a key role. Another path could be to integrate our dependent information flow types into a DLM-style (which allows for access control) type-based information flow analysis.

7.1.4 Hybrid Type-based Information Flow

Static type-based information flow system can be very conservative and dismiss as insecure programs that are actually secure. On the other hand, a purely dynamic type-based information flow analysis is limited to the execution paths the program takes, nor foreseeing other possible insecure paths (for e.g., implicit flows may not be detected).

As such, it would be interesting to study combinations of static and dynamic typing in the context of our dependent type system in order to increase the precision (i.e., permissiveness) of the analysis.

BIBLIOGRAPHY

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. ACM, 1999, pp. 147–160.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security* 13.1 (2009).

[3] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. "Sharing Mobile Code Securely with Information Flow Control". In: *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 191–205.

[4] T. H. Austin and C. Flanagan. "Multiple Facets for Dynamic Information Flow". In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM, 2012, pp. 165–178.

[5] G. Barthe, C. Fournet, B. Grégoire, P.-Y. Strub, N. Swamy, and S. Z. Béguelin. "Probabilistic relational verification for cryptographic implementations". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by S. Jagannathan and P. Sewell. ACM, 2014, pp. 193–206.

[6] M. Y. Becker, C. Fournet, and A. D. Gordon. "SecPAL: Design and semantics of a decentralized authorization language". In: *Journal of Computer Security* 18.4 (2010).

[7] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. "Semantic subtyping with an SMT solver". In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. Ed. by P. Hudak and S. Weirich. ACM, 2010, pp. 105–116.

[8] N. Broberg and D. Sands. "Paralocks: role-based information flow control and beyond". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by M. V. Hermenegildo and J. Palsberg. ACM, 2010, pp. 431–444.

[9] N. Broberg, B. van Delft, and D. Sands. "Paragon for Practical Programming with Information-Flow Control". In: *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*. Ed. by C. Shan. Vol. 8301. Lecture Notes in Computer Science. Springer, 2013, pp. 217–232.

[10] L. Caires, J. A. Pérez, J. C. Seco, H. T. Vieira, and L. Ferrão. "Type-Based Access Control in Data-Centric Systems". In: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by G. Barthe. Vol. 6602. Lecture Notes in Computer Science. Springer, 2011, pp. 136–155.

[11] W. Cheng, D. R. K. Ports, D. A. Schultz, V. Popic, A. Blankstein, J. A. Cowling, D. Curtis, L. Shrira, and B. Liskov. "Abstractions for Usable Information Flow Control in Aeolus". In: *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. USENIX Association, 2012, pp. 139–151.

[12] A. Chlipala. "Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications". In: *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association, 2010, pp. 105–118.

[13] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. "A certifying compiler for Java". In: *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI*. 2000.

[14] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. "Links: Web Programming Without Tiers". In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Springer, 2006, pp. 266–296.

[15] B. J. Corcoran, N. Swamy, and M. W. Hicks. "Cross-tier, Label-based Security Enforcement for Web Applications". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. ACM, 2009, pp. 269–282.

[16] B. Davis and H. Chen. "DBTaint: Cross-Application Information Flow Tracking via Databases". In: *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*. 2010.

[17] D. E. Denning and P. J. Denning. "Certification of Programs for Secure Information Flow". In: *Communications of the ACM* 20.7 (1977).

[18] *Department of Defense Trusted Computer System Evaluation Criteria*. DOD 5200.28-STD (supersedes CSC-STD-001-83). Department of Defense. Dec. 1985.

[19] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. USENIX Association, 2010, pp. 393–407.

[20] Ú. Erlingsson. "The inlined reference monitor approach to security policy enforcement". PhD thesis. Ithaca, NY, USA, 2004.

[21] Ú. Erlingsson and F. B. Schneider. "IRM Enforcement of Java Stack Inspection". In: *IEEE Symposium on Security and Privacy*. 2000.

[22] Ú. Erlingsson and F. B. Schneider. "SASI enforcement of security policies: a retrospective". In: *Proceedings of the 1999 workshop on New security paradigms*. NSPW '99. Caledon Hills, Ontario, Canada: ACM, 2000, pp. 87–95.

[23] D. F. Ferraiolo and D. R. Kuhn. "Role-based access controls". In: *15th NIST-NCSC National Computer Security Conference* (1992), pp. 554–563.

[24] C. Fournet, A. D. Gordon, and S. Maffeis. "A type discipline for authorization policies". In: *ACM Transactions on Programming Languages and Systems* 29.5 (2007).

[25] N. Glew and J. G. Morrisett. "Type-Safe Linking and Modular Assembly Language". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL 1999*. 1999, pp. 250–261.

[26] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *IEEE Symposium on Security and Privacy*. 1982.

[27] D. Hedin and A. Sabelfeld. "Information-Flow Security for a Core of JavaScript". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. IEEE, 2012, pp. 3–18.

[28] N. Heintze and J. G. Riecke. "The SLam Calculus: Programming with Secrecy and Integrity". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19-21, 1998, San Diego, CA, USA*. 1998.

[29] K. Honda, V. T. Vasconcelos, and N. Yoshida. "Secure Information Flow as Typed Process Behaviour". In: *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*. LNCS. Springer, 2000, pp. 180–199.

[30] D. Kozen. *Efficient Code Certification*. Tech. rep. Ithaca, NY, USA, 1998.

[31] B. W. Lampson. "A Note on the Confinement Problem". In: *Communications of the ACM* 16.10 (1973).

[32] B. W. Lampson. "Protection". In: *SIGOPS Oper. Syst. Rev.* 8.1 (1974), pp. 18–24.

[33] Z. Li and X. Wang. "FIRM: capability-based inline mediation of Flash behaviors". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. Austin, Texas: ACM, 2010, pp. 181–190.

[34] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. "Fabric: a platform for secure distributed computation and storage". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by J. N. Matthews and T. E. Anderson. ACM, 2009, pp. 321–334.

[35] L. Lourenço and L. Caires. "Information Flow Analysis for Valued-Indexed Data Security Compartments". In: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*. Ed. by M. Abadi and A. Lluch-Lafuente. Springer, 2013, pp. 180–198.

[36] L. Lourenço and L. Caires. "Dependent Information Flow Types". In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: ACM, 2015, pp. 317–328.

[37] S. McCamant and G. Morrisett. "Evaluating SFI for a CISC architecture". In: *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*. Vancouver, B.C., Canada: USENIX Association, 2006.

[38] E. Meijer, B. Beckman, and G. M. Bierman. "LINQ: reconciling object, relations and XML in the .NET framework". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 2006, p. 706.

[39] J. G. Morrisett, K. Crary, N. Glew, and D. Walker. "Stack-Based Typed Assembly Language". In: *Types in Compilation, Second International Workshop, TIC '98, Kyoto, Japan, March 25-27, 1998, Proceedings*. Ed. by X. Leroy and A. Ohori. Lecture Notes in Computer Science. Springer, 1998, pp. 28–52.

[40] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. "From system F to typed assembly language". In: *ACM Transactions on Programming Languages and Systems* 21.3 (1999), pp. 527–568.

[41] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Springer, 2008, pp. 337–340.

[42] A. C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20-22, 1999, San Antonio, TX*. 1999.

[43] A. C. Myers and B. Liskov. "A Decentralized Model for Information Flow Control". In: *SOSP*. 1997, pp. 129–142.

[44] A. C. Myers and B. Liskov. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology* 9.4 (2000).

[45] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. "Jif: Java Information Flow". Software release. http://www.cs.cornell.edu/jif. 2001.

[46] A. Nanevski, A. Banerjee, and D. Garg. "Verification of Information Flow and Access Control Policies with Dependent Types". In: *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*. IEEE Computer Society, 2011, pp. 165–179.

[47] G. C. Necula. "Proof-Carrying Code". In: *Proceedings of 24th ACM Symposium on Principles of Programming Languages, POPL 1997*. 1997.

[48] G. C. Necula and P. Lee. "The Design and Implementation of a Certifying Compiler". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI*. 1998.

[49] P. H. Phung, D. Sands, and A. Chudnov. "Lightweight self-protecting JavaScript". In: *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ASIACCS '09. Sydney, Australia: ACM, 2009, pp. 47–60.

[50] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. "A survey of static analysis methods for identifying security vulnerabilities in software systems". In: *IBM Systems Journal* 46.2 (2007), pp. 265–288.

[51] F. Pottier and V. Simonet. "Information flow inference for ML". In: *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. ACM, 2002, pp. 319–330.

[52] A. Sabelfeld and A. C. Myers. "Language-Based Information-Flow Security". In: *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security* (2003).

[53] A. Sabelfeld and D. Sands. "A Per Model of Secure Information Flow in Sequential Programs". In: *Higher-Order and Symbolic Computation* (2001).

[54] A. Sabelfeld and D. Sands. "Dimensions and Principles of Declassification". In: *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. 2005.

[55] F. B. Schneider. "Enforceable security policies". In: *ACM Transactions on Information and System Security* 3.1 (2000), pp. 30–50.

[56] D. Schultz. "Decentralized Information Flow Control for Databases". Ph.D. MIT, 2012.

[57] V. Simonet. "Flow Caml in a Nutshell". In: *Proceedings of the first APPSEM-II workshop*. Ed. by G. Hutton. 2003, pp. 152–165.

[58] M. Sridhar and K. W. Hamlen. "ActionScript In-Lined Reference Monitoring in Prolog". In: *Proceedings of the Twelfth Symposium on Practical Aspects of Declarative Languages, PADL 2010*. 2010.

[59] N. Swamy, B. J. Corcoran, and M. Hicks. "Fable: A Language for Enforcing User-defined Security Policies". In: *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 2008, pp. 369–383.

[60] N. Swamy, J. Chen, and R. Chugh. "Enforcing Stateful Authorization and Information Flow Policies in Fine". In: *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Springer, 2010, pp. 529–549.

[61] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. "Secure Distributed Programming with Value-dependent Types". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. ACM, 2011, pp. 266–278.

[62] D. Tarditi, J. G. Morrisett, P. Cheng, C. A. Stone, R. Harper, and P. Lee. "TIL: A Type-Directed Optimizing Compiler for ML". In: *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, PLDI*. 1996, pp. 181–192.

[63] S. Tse and S. Zdancewic. "Run-time Principals in Information-flow Type Systems". In: *ACM Trans. Program. Lang. Syst.* 30.1 (2007).

[64] D. M. Volpano, C. E. Irvine, and G. Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2–3 (1996).

[65] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. "Efficient Software-Based Fault Isolation". In: *Proceedings of the fourteenth ACM symposium on Operating systems principles, SOSP 1993*. Vol. 27. 5. 1993, pp. 203–216.

[66] H. Xi and F. Pfenning. "Dependent Types in Practical Programming". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by A. W. Appel and A. Aiken. ACM, 1999, pp. 214–227.

[67] J. Yang, K. Yessenov, and A. Solar-Lezama. "A language for automatically enforcing privacy policies". In: (2012), pp. 85–96.

[68] S. Zdancewic and A. C. Myers. "Observational Determinism for Concurrent Program Security". In: *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*. IEEE Computer Society, 2003, p. 29.

[69] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. "Securing Distributed Systems with Information Flow Control". In: *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. USENIX Association, 2008, pp. 293–308.

[70] L. Zheng and A. C. Myers. "Dynamic Security Labels and Static Information Flow Control". In: *Int. J. Inf. Sec.* 6.2-3 (2007), pp. 67–84.

APPENDIX A

PROTOTYPE TYPECHECKER EXAMPLES

In this appendix we show the remaining examples verified by our prototype typechecker.

A.1 An Academic Information Manager System

We start with our academic information manager system, presented in Section 5.1.1 of Chapter 5. We use the following lattice definition for this example:

```
forall [x] U(x) ~> P(x,_)
forall [x] U(x) ~> S(x,_)
forall [x] S(_,x) ~> P(_,x)
```

which corresponds to the axioms presented in Section 5.1.1.

Next we declare the collections and their types (via type declarations) used by the system: `Students`, `Faculty`, `Evals` and `Grades`.

Input:

```
typedef student_type =
   { ref (Sigma[ suid: int^BOT, curriculum: int^U(suid),
                 tuition_balance: int^U(suid) ]^BOT )^BOT } ;;
typedef faculty_type =
   { ref (Sigma[ puid: int^BOT, department: int^BOT,
                 salary: int^U(puid) ]^BOT)^BOT } ;;
typedef evaluation_type =
   { ref (Sigma[ puid: int^BOT, cuid: int^BOT,
                 criteria: int^P(puid,cuid), test: int^S(TOP,cuid),
                 scores: ref ( { Sigma[suid: int^BOT,
                                       score: int^S(suid,cuid)]^BOT } )^BOT
               ]^BOT)^BOT } ;;
```

```
typedef grade_type =
   { ref (Sigma[ suid: int^BOT, cuid: int^BOT,
                 grade: int^S(suid,cuid) ]^BOT)^BOT } ;;

let Students = ref {}: student_type ;;
let Faculty = ref {}: faculty_type ;;
let Evals= ref {}: evaluation_type ;;
let Grades= ref {}: grade_type ;;
```

We can now encode the first operations, `enrollStudent2Course` and `viewAverageScore` (Example 30 and Example 31 from Chapter 5):

Input:

```
let enrollStudent2Course = fun s: int^BOT, c: int^BOT =>
  Grades := ref [suid: int^BOT = s, cuid: int^BOT = c, grade: int^S(suid,
      cuid) = 0] :: !Grades ;;

let viewAverageScore = fun suid: int^BOT, cuid: int^BOT =>
  let counter = ref 0 in
    ( foreach (x in !Evals) with avg = 0 do
        let tuple = !x in
         if( tuple.cuid == cuid) then
           foreach (y in !(tuple.scores)) with sum = 0 do
             ( if (y.suid == suid) then
                ( counter := !counter + 1;
                  y.score ) + sum
               else sum
             ) + avg
         else avg
    )/!counter ;;
```

As we have seen, function `enrollStudent2Course` adds a new student record to a given course's enrolled student records and function `viewAverageScore` computes a given student's average on a given course.

Next we encode `addCriteria` and `defineTestCriteria` (Example 34) from Chapter 5):

Input:

```
let defineTestCriteria = fun u:int^BOT, t: int^BOT => [ int^S(TOP,u) ] t+10 ;;

defineTestCriteria;;
let std = 42 in (defineTestCriteria(std, 10)) ;;
```

Output:

```
Type: (Pi(u: int^BOT, t: int^BOT).BOT; int^S(TOP, u))^BOT
Type: int^S(TOP, 42)
```

So function `defineTestCriteria` is a dependent function since its return type depends on parameter `u`. So if we input the function's identifier we obtain its type `(Pi(u: int^BOT, t: int^BOT).BOT; int^S(TOP, u))^BOT`. Now if invoke function `defineTestCriteria` with identifier `std`, the typechecker will determine the value of `std` – given the knowledge obtained from the declaration of identifier `std`– and type the call with type `S(`$\top$`,u)`$\{^{42}/_{u}\}$.

Function `addCriteria` adds, for a given professor, a given course's evaluation criteria (here represented as an integer)

Input:

```
typedef scores_type = ref ( { Sigma[ suid: int^BOT,
                                     score: int^S(suid,cuid) ]^BOT } )^BOT;;

let addCriteria = fun p: int^BOT, c: int^BOT =>
  foreach (x in !Evals) with y = skip do
   let tuple = !x in
    if(tuple.puid == p and tuple.cuid == c) then
      let new_rec = [ puid: int^BOT = tuple.puid,
                      cuid: int^BOT = tuple.cuid,
                      criteria: int^P(puid,cuid) = defineTestCriteria(c,
                          tuple.test),
                      test: int^S(TOP,cuid) = tuple.test,
                      scores: scores_type = tuple.scores ]
      in x := new_rec ;;
```

Output:

```
Type: (Pi(p: int^BOT, c: int^BOT).BOT; cmd^BOT)^BOT
```

This program is secure, as we have seen in Chapter 5, because we can raise the security level of expression `defineTestCriteria(tuple.cuid, tuple.test)` to the declared type for field `criteria` via subtyping. However, if we remove the last conditional, `if(tuple.puid == p and tuple.cuid == c)`:

Input:

```
let addCriteria = fun p: int^BOT, c: int^BOT =>
  foreach (x in !Evals) with y = skip do
   let tuple = !x in
      let new_rec = [ puid: int^BOT = tuple.puid,
                      cuid: int^BOT = tuple.cuid,
                      criteria: int^P(puid,cuid) = defineTestCriteria(c,
                          tuple.test),
                      test: int^S(TOP,cuid) = tuple.test,
                      scores: scores_type = tuple.scores ]
      in x := new_rec ;;
```

Output:

```
Wrong type: Expected declared type
Sigma[ puid: int^BOT, cuid: int^BOT, criteria: int^P(puid,cuid),
       test: int^S(TOP,cuid),
       scores: ref( { Sigma[ suid: int^BOT,
                             score: int^S(suid,cuid)]^BOT } )^BOT ]^BOT
but found type
Sigma[ puid: int^BOT, cuid: int^BOT, criteria: int^S(TOP,c),
       test: int^S(TOP,cuid),
       scores: ref( { Sigma[suid: int^BOT,
                             score: int^S(suid,cuid)]^BOT } )^BOT ]^BOT
```

Then we are not able to raise the security level of `defineTestCriteria(tuple.cuid, tuple.test)` (which is `S(TOP,c)`) to the required type. This is, of course, detected by our typechecker.

We end this toy example with the following code snippet from Example 33 of Chapter 5:

Input:

```
let grades_val = viewAverageScore(42,70)
in foreach(x in !Grades) with y = skip do
    let t_grade = !x
    in if(t_grade.suid == 42 and t_grade.cuid == 70) then
         let new_rec = [ suid: int^BOT = t_grade.suid,
                         cuid: int^BOT = t_grade.cuid,
                         grade: int^S(suid,cuid) = grades_val ]
         in x := new_rec ;;
```

Output: `Type: cmd^BOT`

which we saw in Chapter 5 to be secure. And now a slightly modified version of the same code snippet, where we attempt to associate to student with id `666` the grade, for a given course, of student with id `42`:

Input:

```
let grades_val = viewAverageScore(42,70)
in foreach(x in !Grades) with y = skip do
    let t_grade = !x
    in if(t_grade.suid == 666 and t_grade.cuid == 70) then
         let new_rec = [ suid: int^BOT = t_grade.suid,
                         cuid: int^BOT = t_grade.cuid,
                         grade: int^S(suid,cuid) = grades_val ]
         in x := new_rec ;;
```

Output:

```
Wrong type: Expected declared type
Sigma[suid: int^BOT, cuid: int^BOT, grade: int^S(suid, cuid)]^BOT
```

```
but found type
Sigma[suid: int^BOT, cuid: int^BOT, grade: int^S(42, 70)]^BOT
```

As expected, our typechecker deems the above code insecure because the declared dependent sum type (obtained from the declared fields types) does not match the expressions `new_rec`'s dependent sum type. In particular, the type for field `grade` does not match.

A.2 A Cloud Storage Service

We now illustrate a cloud storage service. In this scenario, the system associates a storage space in the cloud for each user, which is reffered to as the user's "box". So we begin with the declaration of the types for the cloud store as a collection of mutable "box" interfaces. A "box" interface has associated its user's uid, a drop operation to store new data to the user's "box" and a fetch operation to retrieve data from the user's "box".

Input:

```
typedef intf_type = Sigma[ uid:int^BOT,
                           drop: (int^U(uid) => cmd^BOT)^BOT,
                           fetch: (cmd^BOT => int^U(uid))^BOT ]^BOT ;;

typedef store_type = { ref (intf_type)^BOT } ;;

let store = ref ( { } : store_type ) ;;
let usr_uid = ref [int^BOT] 0 ;;
```

Reference `usr_uid` is global to ensure each new user gets a unique identifier. We now define the operation `new_box` that registers a new user in the cloud storage service returning his uid:

Input:

```
let new_box = fun x: cmd^BOT =>
  ( usr_uid := !usr_uid + 1 ;
    let u = !usr_uid in
     let refr = ref [ int^U(u) ] 0 in
      let stub = [ uid:int^BOT = u,
                   drop: ( int^U(u) => cmd^BOT )^BOT =
                      ( fun d: int^U(u) => ( refr := d + !refr ) ),
                   fetch: ( cmd^BOT => int^U(u) )^BOT =
                      ( fun x: cmd^BOT => !refr ) ]
        in let usr_intf = [ intf_type ] stub in
          let new_usr = ref usr_intf in
           ( store := (new_usr :: !store) ; u ) ) ;;
```

Output:

```
Type: (Pi(x: cmd^BOT).BOT; int^BOT)^BOT
```

Before being able to interact with its "box" a user must first open it via operation `open_box`. This operation essentially retrieves the user's "box" interface:

Input:

```
typedef open_type = { Sigma[ uid:int^BOT,
                             drop: ( int^U(u) => cmd^BOT )^BOT,
                             fetch: ( cmd^BOT => int^U(u) )^BOT ]^BOT } ;;

let open_box = fun u:int^BOT =>
    ( let drops = !store in
        let r = (foreach (dr in drops) with acum = {}: open_type do
                   let d = !dr in
                      if (d.uid == u) then
                         d::acum
                      else acum )
        in first(r) ) ;;
```

Output:

```
Type: (Pi(u: int^BOT).BOT; Sigma[uid: int^BOT,
                drop: (Pi(_: int^U(u)).BOT; cmd^BOT)^BOT,
                fetch: (Pi(_: cmd^BOT).BOT; int^U(u))^BOT]^BOT)^BOT
```

Notice that the type of `open_box` is a dependent function type whose return type is a record type where some of its fields dependent on the function's parameter.

We can now encode the following program:

Input:

```
let main = fun a: cmd^BOT =>
  ( let my_usr = new_box ( skip ) in
      let my_box = open_box (my_usr) in
       (
         my_box.drop ( [int^U(my_usr)] 10 );
         let my_data = my_box.fetch (skip) in
           my_box.drop (my_data) ) ) ;;

main(skip);;
```

Output:

```
Type: cmd^BOT
```

In this code snippet, a user registers into the cloud storage service, obtaining his uid, and retrieves his "box" interface by calling `open_box` with the given uid. Then he stores the value `10` into his "box", fetch all the contents stored in his "box" and stores them again in the "box".

Now suppose we had instead the following code snippet:

Input:

```
let main_err = fun a: cmd^BOT =>
  ( let my_usr = new_box( skip ) in
     let other_usr = new_box( skip ) in
      let my_box = open_box (my_usr) in
       let other_box = open_box (other_usr) in
       ( my_box.drop ( [ int^U(my_usr) ] 10);
         let my_data = my_box.fetch (skip) in
          other_box.drop ( my_data ) ) ) ;;
```

Output:

```
Wrong type on arguments: Expected declared type int^U(other_usr)
but found type int^U(my_usr)!
```

Two users, `my_usr` and `other_usr`, register in the cloud storage service, retrieving their "box" interfaces respectively. Then user with uid `my_usr` stores the value `10` in his "box", and all the contents on user `my_usr`'s "box" are kept in identifier `my_data`. Finally, an attempt to store `my_data` on user `other_usr`'s "box" is made but our typechecker deems this operation insecure since we would be storing in a user's "box" another user's data, clearly violating data confidentiality.

APPENDIX B

PROOFS

In this appendix we show the proofs of our main results and the necessary auxiliary results. We omit some standard proofs.

B.1 Type Safety

We start with the lemmas used to prove type safety.

Lemma 6 (Weakening)
Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, then $\Delta \cup \Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$
Proof: By induction on the statement $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$.

Lemma 7 (Constraint Cut Lemma)
If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\cup\{t\doteq t'\}} e : \tau^{\mathsf{s}}$ and $\mathcal{S} \models t \doteq t'$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$.
Proof: Induction on the derivation of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\cup\{t\doteq t'\}} e : \tau^{\mathsf{s}}$, using deduction closure of $\models$.

Lemma 8 (Inversion Lemma for Subtyping)

1. If $\tau^{\mathsf{s}} <: \Sigma[\overline{m : \tau^{\mathsf{s}'}}]^{\mathsf{s}'}$, then
   $\tau^{\mathsf{s}} = \Sigma[\overline{m : \tau^{\mathsf{s}}}]^{\mathsf{t}}$ such that $\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau_i^{\mathsf{s}'_i}$, and $\mathsf{t} = \sqcap|\mathsf{s}_i|^{\downarrow}$

2. If $\tau^{\mathsf{s}} <: \{\overline{m : \tau^{\mathsf{s}'}}\}^{\mathsf{t}'}$, then
   $\tau^{\mathsf{s}} = \{\overline{m : \tau^{\mathsf{s}}}\}^{\mathsf{t}}$ such that $\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau_i^{\mathsf{s}'_i}$, and $\mathsf{t} = \sqcap\mathsf{s}_i$

3. If $\tau^{\mathsf{s}} <: (\Pi x{:}\tau'^{\mathsf{s}'}.\mathsf{q}'; \sigma^{\mathsf{r}'})^{\mathsf{t}}$, then

   a) $\tau^{\mathsf{s}} = (\Pi x{:}\tau'^{\mathsf{s}''}.\mathsf{q}; \sigma^{\mathsf{r}})^{\mathsf{t}}$ such that $\tau'^{\mathsf{s}'} <: \tau'^{\mathsf{s}''}$, $\sigma^{\mathsf{r}} <: \sigma^{\mathsf{r}'}$, and $\mathsf{q}' \leq \mathsf{q}$

b) $\tau^{\mathsf{s}} = (\Pi x{:}\tau'^{\mathsf{s}'}.\mathsf{q}'; \sigma^{\mathsf{r}'})^{\mathsf{s}}$ and $\mathsf{s} \leq \mathsf{t}$

c) $\tau^{\mathsf{s}} = (\Pi x{:}\tau'^{\mathsf{s}'}.\mathsf{q}; \sigma^{\mathsf{r}'})^{\ell(y)}$ and $\mathsf{t} = \ell(v)$ for some v

d) $\tau^{\mathsf{s}} = (\Pi x{:}\tau'^{\mathsf{s}'}.\mathsf{q}; \sigma^{\mathsf{r}'})^{\ell(v)}$ for some v, $\ell(\top) < \ell'(\top)$, and $\mathsf{t} = \ell'(y)$

Proof: By induction on the relation $\tau^{\mathsf{s}} <: \tau'^{\mathsf{s}'}$.

Lemma 9 (Inversion Lemma for Typing)

1. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$, then
   $\Delta, x{:}\tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\sigma^{\mathsf{q}'}$, $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$, $\mathsf{r} \leq \mathsf{r}'$, and $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$.

2. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \#m_i(v)$ **as** $\{\ldots, m_i : \tau_i'^{\mathsf{s}_i'}, \ldots\}^{\mathsf{t}'} : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$, then
   $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau_i'^{\mathsf{s}_i'}$ and $\tau_i'^{\mathsf{s}_i'} <: \tau_i^{\mathsf{s}_i}$.

3. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = v_i, \ldots] : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$, then if:

   a) s_i has no field identifiers nor variables, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i{:}\tau_i^{\mathsf{s}_i}$

   b) $\mathsf{s}_i = \ell_i(m_j)$ and $\mathcal{S}\{x \doteq [\ldots, m_i = v_i, \ldots]\} \models x.m_j \doteq \mathsf{v}$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\ell_i(\mathsf{v})}$

   c) $\mathsf{s}_i = \ell_i(\mathsf{v})$ and $\tau_i^{\ell_i(\mathsf{v})} <: \tau_i^{\mathsf{t}}$, such that $\ell_i(\top) \leq \mathsf{t}$, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\mathsf{t}}$

   d) $\mathsf{s}_i = \mathsf{t}$ and $\tau_i^{\mathsf{t}} \leq \tau_i^{\ell_i(\mathsf{v})}$, such that $\mathsf{t} \leq \ell_i(\bot)$, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\ell_i(\mathsf{v})}$

Proof By induction on the relation $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, using Lemma 8.

1. $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e{:}(\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$, then
   $\Delta, x{:}\tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\sigma^{\mathsf{q}'}$, $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$, $\mathsf{r} \leq \mathsf{r}'$, and $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$.

   Case (T-LAMBDA):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}}).e{:}(\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\mathsf{t}}$	(1) - hyp.
$\Delta, x{:}\tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\sigma^{\mathsf{q}}$	(2) - inv. of (T-LAMBDA) with (1)
$\tau^{\mathsf{s}} <: \tau^{\mathsf{s}}$	by (S-REFLEX) with (2)
$\sigma^{\mathsf{q}} <: \sigma^{\mathsf{q}}$	by (S-REFLEX) with (2)
$\mathsf{r} \leq \mathsf{r}$	by def. of $\leq$

   Case (T-SUB):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}''; \sigma^{\mathsf{q}})^{\mathsf{t}}$	(1) - hyp.
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e : \tau^{\mathsf{s}}$	(2) - inv. of (T-SUB) with (1)
$\tau^{\mathsf{s}} <: (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}''; \sigma^{\mathsf{q}})^{\mathsf{t}}$	(3) - inv. of (T-SUB) with (1)
$\mathsf{r} \leq \mathsf{r}'$	(4) - inv. of (T-SUB) with (1)

- $\tau^{\mathsf{s}} = (\Pi x{:}\tau^{\mathsf{s}''}.\mathsf{r}'''\sigma^{\mathsf{q}''})^{\mathsf{t}}$ (5) - by Lem. 8 with (3)
  $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}''}$ (6) - by Lem. 8 with (3)
  $\sigma^{\mathsf{q}''} <: \sigma^{\mathsf{q}}$ (7) - by Lem. 8 with (3)
  $\mathsf{r}'' \leq \mathsf{r}'''$ (7) - by Lem. 8 with (3)
  $\Delta, x{:}\tau^{\mathsf{s}'} \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \sigma^{\mathsf{q}'}$ (8) - by I.H. with (2)
  $\tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'}$ (9) - by I.H. with (2)
  $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}''}$ (10) - by I.H. with (2)
  $\Delta, x{:}\tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} v : \sigma^{\mathsf{q}'}$ by (T-SUB) with (4,8)
  $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ by (S-TRANS) with (6,9)
  $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$ by (S-TRANS) with (7,10)

- $\tau^{\mathsf{s}} = (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\mathsf{t}'}$ (11) - by Lem. 8 with (3)
  $\mathsf{t}' \leq \mathsf{t}$ (12) - by Lem. 8 with (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e{:}(\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\mathsf{t}'}$ (13) - by (11,2)
  $\Delta, x{:}\tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\sigma^{\mathsf{q}'}$ by I.H. with (13)
  $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ by I.H. with (13)
  $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$ by I.H. with (13)

- $\tau^{\mathsf{s}} = (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\ell(y)}$ (14) - by Lem. 8 with (3)
  $\mathsf{t} = \ell(v)$ for some v (15) - by Lem. 8 with (3)
  $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e : (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\ell(y)}$ (16) - by (2,14)
  $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e : (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\ell(y)}$ (17) - by (T-SUB) with (16,4)
  $\Delta, x{:}\tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\sigma^{\mathsf{q}'}$ by I.H. with (17)
  $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ by I.H. with (17)
  $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$ by I.H. with (17)

- $\tau^{\mathsf{s}} = (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\ell(v)}$ (18) - by Lem. 8 with (3), for some v
  $\mathsf{t} = \ell(y)$ (19) - by Lem. 8 with (3)
  $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e : (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\ell(v)}$ (20) - by (2,18)
  $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} \lambda(x{:}\tau^{\mathsf{s}'}).e : (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\ell(v)}$ (21) - by (T-SUB) with (20,4)
  $\Delta, x{:}\tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\sigma^{\mathsf{q}'}$ by I.H. with (21)
  $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ by I.H. with (21)
  $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$ by I.H. with (21)

2. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \#m(v)$ **as** $\{\ldots, m_i : \tau_i'^{\mathsf{s}_i'}, \ldots\}^{\mathsf{t}'} : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$, then
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau_i'^{\mathsf{s}_i'}$ and $\tau_i'^{\mathsf{s}_i'} <: \tau_i^{\mathsf{s}_i}$

Case (T-INJ):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \#m_i(v)$ **as** $\{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\sqcap \mathsf{s}_i} : \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\sqcap \mathsf{s}_i}$	(1) - hyp.
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau_i^{\mathsf{s}_i}$	(2) - inv. of (T-INJ) with (1)
$\tau_i^{\mathsf{s}_i} <: \tau_i^{\mathsf{s}_i}$	by (S-REFLEX) with (2)

Case (T-SUB):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \#m(v)$ **as** $\{\ldots, m_i : \tau_i'^{\mathsf{s}_i'}, \ldots\}^{\mathsf{t}'} : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$	(1) - hyp.
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \#m(v)$ **as** $\{\ldots, m_i : \tau_i'^{\mathsf{s}_i'}, \ldots\}^{\mathsf{t}'} : \tau^{\mathsf{s}}$	(2) - inv. of (T-SUB) with (1)
$\tau^{\mathsf{s}} <: \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$	(3) - inv. of (T-SUB) with (1)
$\mathsf{r} \leq \mathsf{r}'$	(4) - inv. of (T-SUB) with (1)
$\tau^{\mathsf{s}} = \{\overline{m : \tau''^{\mathsf{s}''}}\}^{\mathsf{t}''}$	(5) - by Lem. 8 with (3)
$\forall_i\ \tau_i''^{\mathsf{s}_i''} <: \tau_i^{\mathsf{s}_i}$	(6) - by Lem. 8 with (3)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau_i'^{\mathsf{s}_i'}$	by I.H. with (2)
$\tau_i'^{\mathsf{s}_i'} <: \tau_i''^{\mathsf{s}_i''}$	(7) - by I.H. with (2)
$\tau_i'^{\mathsf{s}_i'} <: \tau_i^{\mathsf{s}_i}$	by (S-TRANS) with (7,6)

3. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = v_i, \ldots] : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$, then if:

   a) s_i has no field identifiers nor variables, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i{:}\tau_i^{\mathsf{s}_i}$

   b) $\mathsf{s}_i = \ell_i(m_j)$ and $\mathcal{S}\{x \doteq [\ldots, m_i = v_i, \ldots]\} \models x.m_j \doteq \mathsf{v}$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\ell_i(\mathsf{v})}$

   c) $\mathsf{s}_i = \ell_i(\mathsf{v})$ and $\tau_i^{\ell_i(\mathsf{v})} <: \tau_i^{\mathsf{t}}$, such that $\ell_i(\top) \leq \mathsf{t}$, then
   $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\mathsf{t}}$

   d) $\mathsf{s}_i = \mathsf{t}$ and $\tau_i^{\mathsf{t}} \leq \tau_i^{\ell_i(\mathsf{v})}$, such that $\mathsf{t} \leq \ell_i(\bot)$, then
   $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\ell_i(\mathsf{v})}$

Case (T-RECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = v_i, \ldots] : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$	hyp.
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\mathsf{s}_i}$	(2) - inv. of (T-RECORD) with (1)

Case (T-SUB):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = v_i, \ldots] : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$	(1) - hyp.
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} [\ldots, m_i = v_i, \ldots] : \tau^{\mathsf{s}}$	(2) - inv. of (T-SUB) with (1)
$\tau^{\mathsf{s}} <: \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$	(3) - inv. of (T-SUB) with (1)
$\mathsf{r} <: \mathsf{r}'$	(4) - inv. of (T-SUB) with (1)

$\tau^{\mathsf{s}} = \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}'_i} \times \ldots]^{\mathsf{t}'}$ (5) - by Lem. 8 with (3)
$\forall_i\ \tau_i^{\mathsf{s}'_i} <: \tau_i^{\mathsf{s}_i}$ (6) - by Lem. 8 with (3)
$\sqcap|\mathsf{s}'_i|^{\downarrow} = \mathsf{t}'$ (7) - by Lem. 8 with (3)

a) s'_i has no field identifiers nor variables — by I.H. with (2,5)
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v_i{:}\tau_i^{\mathsf{s}'_i}$ (8) - by I.H. with (2,5)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i{:}\tau_i^{\mathsf{s}_i}$ (9) - by (T-SUB) with (8,6,4)

b) not applicable

c) not applicable

d) not applicable

Case (T-UNREFINERECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = v_i, \ldots] : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots, m_i{:}\tau_i^{\ell_i(\mathsf{v})}, \ldots]^{\mathsf{s}}$ (1) - hyp.
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = v_i, \ldots] : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots, m_i{:}\tau_i^{\ell_i(m_j)}, \ldots]^{\mathsf{s}}$
(2) - inv. (T-UNREFINERECORD) of (1)
$\mathcal{S}\{x \doteq [\ldots, m_i = v_i, \ldots]\} \models x.m_j \doteq \mathsf{v}$ (3) - inv. (T-UNREFINERECORD) of (1)

a) not applicable because $\mathsf{s}_i = \ell_i(m_j)$ — by I.H. with (2)

b) $\mathsf{s}_i = \ell_i(m_j)$ and $\mathcal{S}\{x \doteq [\ldots, m_i = v_i, \ldots]\} \models x.m_j \doteq \mathsf{v}$ — by I.H. with (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\ell_i(\mathsf{v})}$ — by I.H. with (2)

c) $\mathsf{s}_i = \ell_i(m_j)$ and $\tau_i^{\ell_i(m_j)} <: \tau_i^{\ell_i(\mathsf{v})}$, such that $\ell_i(\top) \leq \ell_i(\mathsf{v})$ — by I.H. with (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\ell_i(\top)}$ — since by $\ell_i(\top) \leq \ell_i(\mathsf{v})$ implies $\mathsf{v} = \top$

d) not applicable because $\mathsf{s}_i = \ell_i(m_j)$

□

Lemma 10 (Canonical Forms Lemma)

1. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\mathsf{t}}$, then $\exists_{x,\mathsf{s}',e}\ v = \lambda(x : \tau^{\mathsf{s}'}).e$.
2. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \mathtt{Bool}^{\mathsf{s}}$, then $v = \mathtt{true}$ or $v = \mathtt{false}$.
3. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \mathtt{cmd}^{\mathsf{s}}$, then $v = ()$.
4. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}}$, then $v = l$.
5. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \Sigma[\overline{m{:}\tau^{\mathsf{s}}}]^{\mathsf{t}}$, then $v = [\overline{m = v'}]$, and $\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v'_i{:}\tau_i^{\mathsf{s}'_i}$ such that $\mathsf{t} = \sqcap|\mathsf{s}'_i|^{\downarrow}$.

6. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$, then $v = \#m_i(u)$, and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} u : \tau_i^{\mathsf{s}_i}$ such that $\mathsf{t} = \sqcap \mathsf{s}_i$.

7. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{*\mathsf{s}}$, then $v = \overline{v'}$, and $\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v'_i : \tau^{\mathsf{s}}$.

Proof: By induction on the relation $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$.

Lemma 11 (Substitution Lemma for Subtyping)
Let $\tau^{\mathsf{s}} <: \tau'^{\mathsf{s}'}$ and $\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}} <: \tau'^{\mathsf{s}'}$. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\gamma^{\mathsf{p}}$ then
$(\tau^{\mathsf{s}})\{v/x\} <: (\tau'^{\mathsf{s}'})\{v/x\}$, and $\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\} <: (\tau'^{\mathsf{s}'})\{v/x\}$.

Proof Induction on the derivation of $\tau^{\mathsf{s}} <: \tau'^{\mathsf{s}'}$.

Case (S-REFLEX):

$\tau^{\mathsf{s}} <: \tau^{\mathsf{s}}$	hyp
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}} <: \tau^{\mathsf{s}}$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\gamma^{\mathsf{p}}$	(2) - hyp
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}}$	(3) - inv. of (W-SUBTYPE) of (1)
$x \notin fv((\tau^{\mathsf{s}})\{v/x\})$	(4) - by def. of substitution
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\}$	(5) - by Def. 19 with (3,4)
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\} <: (\tau^{\mathsf{s}})\{v/x\}$	by (W-SUBTYPE) with (5)

Case (S-TRANS):

$\tau^{\mathsf{s}} <: \tau^{\mathsf{r}}$	(1) - hyp
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}} <: \tau^{\mathsf{r}}$	(2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\gamma^{\mathsf{p}}$	(3) - hyp
$\tau^{\mathsf{s}} <: \tau^{\mathsf{t}}$	(6) - inv. of (S-TRANS) of (1)
$\tau^{\mathsf{t}} <: \tau^{\mathsf{r}}$	(7) - inv. of (S-TRANS) of (1)
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}} <: \tau^{\mathsf{t}}$	(8) - by (2,6)
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{t}} <: \tau^{\mathsf{r}}$	(9) - by (3,7)
$(\tau^{\mathsf{s}})\{v/x\} <: (\tau^{\mathsf{t}})\{v/x\}$	(10) - by I.H. with (6,8,3)
$(\tau^{\mathsf{t}})\{v/x\} <: (\tau^{\mathsf{r}})\{v/x\}$	(11) - by I.H. with (7,9,3)
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\} <: (\tau^{\mathsf{t}})\{v/x\}$	(12) - by I.H. with (6,8,3)
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{t}})\{v/x\} <: (\tau^{\mathsf{r}})\{v/x\}$	(13) - by I.H. with (7,9,3)
$(\tau^{\mathsf{s}})\{v/x\} <: (\tau^{\mathsf{r}})\{v/x\}$	by (S-TRANS) with (10,11)
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\}$	(14) - inv. of (W-SUBTYPE) of (12)
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{r}})\{v/x\}$	(15) - inv. of (W-SUBTYPE) of (13)
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\} <: (\tau^{\mathsf{r}})\{v/x\}$	by (W-SUBTYPE) with (14,15)

Case (S-ARROW):

$(\Pi y{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{r}})^{\mathsf{t}} <: (\Pi y'{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{r}'})^{\mathsf{t}}$	(1) - hyp
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} (\Pi y{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{r}})^{\mathsf{t}} <: (\Pi y'{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{r}'})^{\mathsf{t}}$	(2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\gamma^{\mathsf{p}}$	(3) - hyp

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} (\Pi y{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{r}})^{\mathsf{t}}$ (4) - inv. of (W-SUBTYPE) of (2)

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} (\Pi y'{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{r}'})^{\mathsf{t}}$ (5) - inv. of (W-SUBTYPE) of (2)

$\tau^{\mathsf{s}'} <: \tau^{\mathsf{s}}$ (6) - inv. of (S-ARROW) of (1)

$\sigma^{\mathsf{r}} <: \sigma^{\mathsf{r}'}$ (7) - inv. of (S-ARROW) of (1)

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}}$ (8) - inv. of (W-ARROW) of (4)

$\Delta, x{:}\gamma^{\mathsf{p}}, y : \tau^{\mathsf{s}} \vdash^{\mathcal{N}} \sigma^{\mathsf{r}}$ (9) - inv. of (W-ARROW) of (4)

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}'}$ (10) - inv. of (W-ARROW) of (5)

$\Delta, x{:}\gamma^{\mathsf{p}}, y' : \tau^{\mathsf{s}'} \vdash^{\mathcal{N}} \sigma^{\mathsf{r}'}$ (11) - inv. of (W-ARROW) of (5)

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}'} <: \tau^{\mathsf{s}}$ (12) - by (W-SUBTYPE) with (8,10)

$\Delta, x{:}\gamma^{\mathsf{p}}, y : \tau^{\mathsf{s}}, y' : \tau^{\mathsf{s}'} \vdash^{\mathcal{N}} \sigma^{\mathsf{r}} <: \sigma^{\mathsf{r}'}$ (13) - by (W-SUBTYPE) with (9,11) and Lem. 6

$(\tau^{\mathsf{s}'})\{v/x\} <: (\tau^{\mathsf{s}})\{v/x\}$ (14) - by I.H. with (6,12,3)

$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}'})\{v/x\} <: (\tau^{\mathsf{s}})\{v/x\}$ (15) - by I.H. with (6,12,3)

$(\sigma^{\mathsf{r}})\{v/x\} <: (\sigma^{\mathsf{r}'})\{v/x\}$ (16) - by I.H. with (7,13,3)

$\Delta, y : \tau^{\mathsf{s}}, y' : \tau^{\mathsf{s}'} \vdash^{\mathcal{N}} (\sigma^{\mathsf{r}})\{v/x\} <: (\sigma^{\mathsf{r}'})\{v/x\}$ (17) - by I.H. with (7,13,3)

$(\Pi y{:}(\tau^{\mathsf{s}})\{v/x\}.(\sigma^{\mathsf{r}})\{v/x\})^{\mathsf{t}\{v/x\}} <: (\Pi y'{:}(\tau^{\mathsf{s}'})\{v/x\}.(\sigma^{\mathsf{r}'})\{v/x\})^{\mathsf{t}\{v/x\}}$
by (S-ARROW) with (14,16)

$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}'})\{v/x\}$ (18) - inv. (W-SUBTYPE) of (15)

$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\}$ (19) - inv. (W-SUBTYPE) of (15)

$\Delta, y : \tau^{\mathsf{s}}, y' : \tau^{\mathsf{s}'} \vdash^{\mathcal{N}} (\sigma^{\mathsf{r}})\{v/x\}$ (20) - inv. (W-SUBTYPE) of (17)

$\Delta, y : \tau^{\mathsf{s}}, y' : \tau^{\mathsf{s}'} \vdash^{\mathcal{N}} (\sigma^{\mathsf{r}'})\{v/x\}$ (21) - inv. (W-SUBTYPE) of (17)

$\Delta, y : \tau^{\mathsf{s}} \vdash^{\mathcal{N}} (\sigma^{\mathsf{r}})\{v/x\}$ (22) - by (20) since y' is fresh in σ^{r} by (9)

$\Delta, y' : \tau^{\mathsf{s}'} \vdash^{\mathcal{N}} (\sigma^{\mathsf{r}'})\{v/x\}$ (23) - by (21) since y is fresh in $\sigma'^{\mathsf{r}'}$ by (11)

$fv(\mathsf{t}\{v/x\}) \subseteq lv(\mathcal{S})$ (24) - since $fv(\mathsf{t}\{v/x\}) \subseteq fv(\mathsf{t})$ by def. of subst.

$fv(\mathsf{t}\{v/x\}) \subseteq dom(\Delta)$ (25) - since $fv(\mathsf{t}\{v/x\}) \subseteq fv(\mathsf{t})$ by def. of subst.

$\Delta \vdash^{\mathcal{N}} (\Pi y{:}(\tau^{\mathsf{s}})\{v/x\}.(\sigma^{\mathsf{r}})\{v/x\})^{\mathsf{t}\{v/x\}}$ (26) - by (W-ARROW) with (18,22,24,25)

$\Delta \vdash^{\mathcal{N}} (\Pi y'{:}(\tau^{\mathsf{s}'})\{v/x\}.(\sigma^{\mathsf{r}'})\{v/x\})^{\mathsf{t}\{v/x\}}$ (27) - by (W-ARROW) with (19,23,24,25)

$\Delta \vdash^{\mathcal{N}} (\Pi y{:}(\tau^{\mathsf{s}})\{v/x\}.(\sigma^{\mathsf{r}})\{v/x\})^{\mathsf{t}\{v/x\}} <: (\Pi y'{:}(\tau^{\mathsf{s}'})\{v/x\}.(\sigma^{\mathsf{r}'})\{v/x\})^{\mathsf{t}\{v/x\}}$
by (W-SUBTYPE) with (26,27)

Case (S-RECORD):

$\Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}} <: \Sigma[\ldots \times m_i : \tau'^{\mathsf{s}'_i}_i \times \ldots]^{\mathsf{s}'}$ (1) - hyp

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}} <: \Sigma[\ldots \times m_i : \tau'^{\mathsf{s}'_i}_i \times \ldots]^{\mathsf{s}'}$ (2) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\gamma^{\mathsf{p}}$ (3) - hyp

$\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau'^{\mathsf{s}'_i}_i$ (4) - inv. (S-RECORD) of (1)

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$ (6) - inv. (W-SUBTYPE) of (2)

$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \Sigma[\ldots \times m_i : \tau'^{\mathsf{s}'_i}_i \times \ldots]^{\mathsf{s}'}$ (7) - inv. (W-SUBTYPE) of (2)

$\forall_i\ \Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N} \cup \{m_1,\ldots,m_{i-1}\}} \tau_i^{\mathsf{s}_i}$ (8) - inv. (W-RECORD) of (6)

$fv(\mathsf{s}) \in lv(\mathcal{N})$ (9) - inv. (W-RECORD) of (6)

$fv(\mathsf{s}) \subseteq dom(\Delta, x{:}\gamma^{\mathsf{p}})$ (10) - inv. (W-RECORD) of (6)

$\forall_i\ \Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\{m_1,\ldots,m_{i-1}\}} \tau'^{\mathsf{s}'_i}_i$ (11) - inv. (W-RECORD) of (7)

$fv(\mathsf{s}') \in lv(\mathcal{N})$ (12) - inv. (W-RECORD) of (7)

$fv(\mathsf{s}') \subseteq dom(\Delta, x{:}\gamma^{\mathsf{p}})$ (13) - inv. (W-RECORD) of (7)
$\forall_i\ \Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N} \cup \{m_1,\ldots,m_{i-1}\}} \tau_i^{\mathsf{s}_i} <: \tau'^{\mathsf{s}'_i}_i$ (14) - by (W-SUBTYPE) with (8,11)
$\forall_i\ (\tau_i^{\mathsf{s}_i})\{v/x\} <: (\tau'^{\mathsf{s}'_i}_i)\{v/x\}$ (15) - by I.H. with (4,14,3)
$\forall_i\ \Delta \vdash^{\mathcal{N} \cup \{m_1,\ldots,m_{i-1}\}} (\tau_i^{\mathsf{s}_i})\{v/x\} <: (\tau'^{\mathsf{s}'_i}_i)\{v/x\}$ (16) - by I.H. with (4,14,3)
$\forall_i\ \Delta \vdash^{\mathcal{N} \cup \{m_1,\ldots,m_{i-1}\}} (\tau_i^{\mathsf{s}_i})\{v/x\}$ (17) - inv. of (W-SUBTYPE) with (16)
$\forall_i\ \Delta \vdash^{\mathcal{N} \cup \{m_1,\ldots,m_{i-1}\}} (\tau'^{\mathsf{s}'_i}_i)\{v/x\}$ (18) - inv. of (W-SUBTYPE) with (16)
$fv(\mathsf{s}\{v/x\}) \in lv(\mathcal{N})$ (19) - by (9)
$fv(\mathsf{s}\{v/x\}) \subseteq dom(\Delta)$ (20) - by (10) and def. of substitution
$fv(\mathsf{s}\{v/x\}) \in lv(\mathcal{N})$ (21) - by (12)
$fv(\mathsf{s}\{v/x\}) \subseteq dom(\Delta)$ (22) - by (13) and def. of substitution
$\Delta \vdash^{\mathcal{N}} (\Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}})\{v/x\}$ (23) - by (W-RECORD) with(19,20,17)
$\Delta \vdash^{\mathcal{N}} (\Sigma[\ldots \times m_i : \tau'^{\mathsf{s}'_i}_i \times \ldots]^{\mathsf{s}'})\{v/x\}$ (24) - by (W-RECORD) of (18,21,22)
$(\Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}})\{v/x\} <: (\Sigma[\ldots \times m_i : \tau'^{\mathsf{s}'_i}_i \times \ldots]^{\mathsf{s}'})\{v/x\}$ by (S-RECORD) with (15)
$\Delta \vdash^{\mathcal{N}} (\Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}})\{v/x\} <: (\Sigma[\ldots \times m_i : \tau'^{\mathsf{s}'_i}_i \times \ldots]^{\mathsf{s}'})\{v/x\}$ by (W-RECORD) with (23,24)

Case (S-VARIANT):

$\{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}} <: \{\ldots, m_i : \tau'^{\mathsf{s}'_i}_i, \ldots\}^{\mathsf{s}'}$ (1) - hyp
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}} <: \{\ldots, m_i : \tau'^{\mathsf{s}'_i}_i, \ldots\}^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \gamma^{\mathsf{p}}$ (3) - hyp
$\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau'^{\mathsf{s}'_i}_i$ (4) - inv. (S-VARIANT) of (1)
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}}$ (6) - inv. (W-SUBTYPE) of (2)
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \{\ldots, m_i : \tau'^{\mathsf{s}'_i}_i, \ldots\}^{\mathsf{s}'}$ (7) - inv. (W-SUBTYPE) of (2)
$\forall_i\ \Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau_i^{\mathsf{s}_i}$ (8) - inv. (W-VARIANT) of (6)
$fv(\mathsf{s}) \in lv(\mathcal{N})$ (9) - inv. (W-VARIANT) of (6)
$fv(\mathsf{s}) \subseteq dom(\Delta, x{:}\gamma^{\mathsf{p}})$ (10) - inv. (W-VARIANT) of (6)
$\forall_i\ \Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau'^{\mathsf{s}'_i}_i$ (11) - inv. (W-VARIANT) of (7)
$fv(\mathsf{s}') \in lv(\mathcal{N})$ (12) - inv. (W-VARIANT) of (7)
$fv(\mathsf{s}') \subseteq dom(\Delta, x{:}\gamma^{\mathsf{p}})$ (13) - inv. (W-VARIANT) of (7)
$\forall_i\ \Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau_i^{\mathsf{s}_i} <: \tau'^{\mathsf{s}'_i}_i$ (14) - by (W-SUBTYPE) with (8,11)
$\forall_i\ (\tau_i^{\mathsf{s}_i})\{v/x\} <: (\tau'^{\mathsf{s}'_i}_i)\{v/x\}$ (15) - by I.H. with (4,14,3)
$\forall_i\ \Delta \vdash^{\mathcal{N}} (\tau_i^{\mathsf{s}_i})\{v/x\} <: (\tau'^{\mathsf{s}'_i}_i)\{v/x\}$ (16) - by I.H. with (4,14,3)
$\forall_i\ \Delta \vdash^{\mathcal{N}} (\tau_i^{\mathsf{s}_i})\{v/x\}$ (17) - inv. of (W-SUBTYPE) with (16)
$\forall_i\ \Delta \vdash^{\mathcal{N}} (\tau'^{\mathsf{s}'_i}_i)\{v/x\}$ (18) - inv. of (W-SUBTYPE) with (16)
$fv(\mathsf{s}\{v/x\}) \in lv(\mathcal{N})$ (19) - by (9)
$fv(\mathsf{s}\{v/x\}) \subseteq dom(\Delta)$ (20) - by (10) and def. of substitution
$fv(\mathsf{s}\{v/x\}) \in lv(\mathcal{N})$ (21) - by (12)
$fv(\mathsf{s}\{v/x\}) \subseteq dom(\Delta)$ (22) - by (13) and def. of substitution
$\Delta \vdash^{\mathcal{N}} (\{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}})\{v/x\}$ (23) - by (W-VARIANT) with(19,20,17)
$\Delta \vdash^{\mathcal{N}} (\{\ldots, m_i : \tau'^{\mathsf{s}'_i}_i, \ldots\}^{\mathsf{s}'})\{v/x\}$ (24) - by (W-VARIANT) of (18,21,22)

$(\{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}})\{v/x\} <: (\{\ldots, m_i : \tau'^{\mathsf{s}'_i}_i, \ldots\}^{\mathsf{s}'})\{v/x\}$ by (S-VARIANT) with (15)
$\Delta \vdash^{\mathcal{N}} (\{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}})\{v/x\} <: (\{\ldots, m_i : \tau'^{\mathsf{s}'_i}_i, \ldots\}^{\mathsf{s}'})\{v/x\}$
by (W-VARIANT) with (23,24)

Case (S-EXPR):

$\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ (1) - hyp
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\gamma^{\mathsf{p}}$ (3) - hyp
$\mathsf{s} \leq \mathsf{s}'$ (4) - by inv. (S-EXPR) of (1)
$\mathsf{s}, \mathsf{s}' \in \mathcal{L}$ (5) - by (4) since $\mathcal{L}$ only contains concrete labels
$(\tau^{\mathsf{s}})\{v/x\} <: (\tau^{\mathsf{s}'})\{v/x\}$ by (5)
$\Delta \vdash^{\mathcal{N}} (\tau^{\mathsf{s}})\{v/x\} <: (\tau^{\mathsf{s}'})\{v/x\}$ by (W-SUBTYPE)

Case (S-INDEXLEFT):

$\tau^{\ell(\mathsf{v})} <: \tau^{\mathsf{s}}$ (1) - hyp
$\Delta, x{:}\gamma^{\mathsf{p}} \vdash^{\mathcal{N}} \tau^{\ell(\mathsf{v})} <: \tau^{\mathsf{s}}$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v{:}\gamma^{\mathsf{p}}$ (3) - hyp
$\ell(\top) \leq \mathsf{s}$ (4) - by inv. (S-INDEXLEFT) of (1)

- $\tau^{\ell(\mathsf{v})}\{v/x\} = \tau^{\ell(v)}$
  $\ell(v) < \ell(\top) \leq \mathsf{s}$ (5) - by def, security lattice with (4)
  $\ell(\top) \leq \mathsf{s}\{v/x\}$ (6) - by (4) we know s is a concrete label, so $\mathsf{s}\{v/x\} = \mathsf{s}$
  $\tau\{v/x\}^{\ell(\mathsf{v})} <: \tau\{v/x\}^{\mathsf{s}\{v/x\}}$ by (5) and by (s-trans) and (S-INDEXLEFT) with (5,6)
  $\Delta \vdash^{\mathcal{N}} \tau\{v/x\}^{\ell(\mathsf{v})} <: \tau\{v/x\}^{\mathsf{s}\{v/x\}}$ by (W-SUBTYPE)

- $\tau^{\ell(\mathsf{v})}\{v/x\} = \tau^{\ell(\mathsf{v})}$
  $\ell(\top) \leq \mathsf{s}\{v/x\}$ (7) - by (4) we know s is a concrete label, so $\mathsf{s}\{v/x\} = \mathsf{s}$
  $\tau\{v/x\}^{\ell(\mathsf{v})} <: \tau\{v/x\}^{\mathsf{s}\{v/x\}}$ by (S-INDEXLEFT) with (4,7)
  $\Delta \vdash^{\mathcal{N}} \tau\{v/x\}^{\ell(\mathsf{v})} <: \tau\{v/x\}^{\mathsf{s}\{v/x\}}$ by (W-SUBTYPE)

Case (S-VARRIGHT):

similar to previous case □

We now prove substitution lemma which uses the subtyping substitution lemma to prove the case for subsumption rule.

Lemma 12 (Substitution Lemma)
If $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$ and $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$.

Proof Induction on the derivation of $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$.

Cases (T-TRUE), (T-FALSE), and (T-UNIT):

$\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} u{:}\tau^{\mathsf{s}}$ (1) - hyp.
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau^{\mathsf{s}}$ (2) - hyp.
$u\{v/x\} = u$ (3) - by def. of substitution
$\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} u\{v/x\}{:}\tau^{\mathsf{s}}$ (4) - by (1,3)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} u\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ since x fresh in $u, \mathcal{S}, \mathsf{s}, \tau$ by (2)

Case (T-ID):

$\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} y{:}\tau^{\mathsf{s}}$ (1) - hyp.
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau^{\mathsf{s}}$ (2) - hyp.

- $x \neq y$ (3)
  $y\{v/x\} = y$ (4) - by def. of substitution
  $\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} y\{v/x\}{:}\tau^{\mathsf{s}}$ (5) - by (4),(1)
  $x \notin fv(y)$ (6) - by def. of fv
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} y\{v/x\}{:}\tau^{\mathsf{s}}$ (7) - by (6),(5)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} y\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ x is fresh in $\mathcal{S}, \Delta$ by (2)
- $x = y$
  $y\{v/x\} = v$ (8) - by def.
  $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} y\{v/x\}{:}\tau^{\mathsf{s}}$ by (2,8)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} y\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ x is fresh in $\mathcal{S}, \Delta$ by (2)

Case (T-FIELD):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e.m_i{:}\tau_i^{\mathsf{s}_i}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$ (2) - hyp.
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$ (3) - inv. (T-FIELD) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\}{:}(\Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}})\{v/x\}$ (5) - I.H. by (2), (3)
$e.m_i\{v/x\} = e\{v/x\}.m_i$ (6) - by def.
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\}.m_i{:}(\tau_i^{\mathsf{s}_i})\{v/x\}$ by rule (T-FIELD) from (5,6,7)

Case (T-LET):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{let}\ y = e_1\ \mathbf{in}\ e_2{:}\tau_2^{\mathsf{s}_2}$ (1) - hyp.
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$ (2) - hyp.
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1{:}\tau_1^{\mathsf{s}_1}$ (3) - inv. (T-LET) of (1)
$\Delta, x : \tau'^{\mathsf{s}'}, y : \tau_1^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}\{y \doteq e_1\}} e_2{:}\tau_2^{\mathsf{s}_2}$ (4) - inv. (T-LET) of (1)

- $x = y$
  $(\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v/x\} = (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)$ (5) - by def.
  $\Delta, x : \tau_1^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v/x\} : \tau_2^{\mathsf{s}_2}$ (6) - by (5),(1)
  $x \notin fv(\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)$ (7) - by def. of fv
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v/x\}{:}\tau_2^{\mathsf{s}_2}$ by (7),(6)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v/x\}{:}(\tau_2^{\mathsf{s}_2})\{v/x\}$ x is fresh in $\Delta, \mathcal{S}, \mathsf{s}_2, \tau_2$

- $x \neq y$

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_1\{v/x\}{:}(\tau_1^{\mathsf{s}_1})\{v/x\}$ (8) - I.H. with (2), (3)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{y \doteq e_1\}} v{:}\tau'^{\mathsf{s}'}$ (9) - y is fresh in $\Delta, \mathcal{S}, v$ by (2)

  $\Delta, y : \tau_1^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}\{y \doteq e_1\{v/x\}\}} e_2\{v/x\}{:}(\tau_2^{\mathsf{s}_2})\{v/x\}$ (10) - I.H. with (9), (4)

  $(\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v/x\} = (\mathbf{let}\ y = e_1\{v/x\}\ \mathbf{in}\ e_2\{v/x\})$ (11) - by def.

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \mathbf{let}\ x = e_1\{v/x\}\ \mathbf{in}\ e_2\{v/x\}{:}(\tau_2^{\mathsf{s}_2})\{v/x\}$

  by rule (T-LET) with (8),(10), and by (11)

Case (T-CASE):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots) : \tau^{\mathsf{s}}$ (1) - hyp.

$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp.

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$ (3) - inv. (T-CASE) of (1)

$\forall_i\ \Delta, x : \tau'^{\mathsf{s}'}, y_i : \tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (4) - inv. (T-CASE) of (1)

- $x = y$

  $(\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} = \mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots)$ (5) - by def.

  $\Delta, x : \tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}} (\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} : \tau^{\mathsf{s}}$ (6) - by (5),(1)

  $x \notin fv(\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))$ (7) - by def. of fv

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} (\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} : \tau^{\mathsf{s}}$ by (7),(6)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} (\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ x is fresh in $\Delta, \mathcal{S}, \mathsf{s}_2, \tau_2$

- $x \neq y$

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}})\{v/x\}$ (8) - I.H. with (2), (3)

  $\forall_i\ \Delta, y : \tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_i\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ (9) - I.H. with (2), (4)

  $(\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} = \mathbf{case}\ e\{v/x\}(\ldots, m_i \cdot y_i \Rightarrow e_i\{v/x\}, \ldots)$

  (10) - by def. of substitution

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \mathbf{case}\ e\{v/x\}(\ldots, m_i \cdot y_i \Rightarrow e_i\{v/x\}, \ldots) : (\tau^{\mathsf{s}})\{v/x\}$

  by rule (T-CASE) with (8),(9), and by (10)

Case (T-APP):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1(e_2) : \sigma\{u/y\}^{\mathsf{q}\{u/y\} \sqcup \mathsf{t}}$ (1) - hyp.

$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\Pi y{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$ (3) - inv. (T-APP) of (1)

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (4) - inv. (T-APP) of (1)

$\mathsf{r} \leq \mathsf{r}'$ inv. (T-APP) of (1)

$\mathcal{S} \cup \{y \doteq e_2\} \models y \doteq \mathsf{v}$ (5) - inv. (T-APP) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_1\{v/x\} : ((\Pi y : \tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}})\{v/x\}$ (6) - I.H. with (3), (2)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_2\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ (7) - I.H. with (4), (2)

$\mathcal{S}\{v/x\} \cup \{y \doteq e_2\{v/x\}\} \models y \doteq \mathsf{v}$ (8) - by subst closure of $\doteq$ with (5)

$(e_1(e_2))\{v/x\} = (e_1\{v/x\}(e_2\{v/x\}))$ (9) - by def.

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_1\{v/x\}(e_2\{v/x\}) : (\sigma\{\mathsf{v}/y\}^{(\mathsf{q}\{\mathsf{v}/y\}\sqcup\mathsf{t})})\{v/x\}$
by rule (t-app) with (6,7,8), and by (9)

Case (T-SUB):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$ (1) - hyp.
$\Delta \vdash^{\mathsf{r}''}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp.
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}'}_{\mathcal{S}} e{:}\tau^{\mathsf{s}''}$ (3) - inv. (T-SUB) of (1)
$\tau^{\mathsf{s}''} <: \tau^{\mathsf{s}}$ (4) - inv. (T-SUB) of (1)
$\mathsf{r}' \leq \mathsf{r}$ (5) - inv. (T-SUB) of (1)
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\tau^{\mathsf{s}''})\{v/x\}$ (6) - I.H. with (3), (2)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathcal{N}} \tau^{\mathsf{s}}$ (7) - by Def. 19 with (1), for some name set $\mathcal{N}$
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}''}$ (8) - by (T-SUB) with (3,5)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathcal{N}} \tau^{\mathsf{s}''}$ (9) - by Def. 19 with (8), for some name set $\mathcal{N}$
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathcal{N}} \tau^{\mathsf{s}} <: \tau^{\mathsf{s}''}$ (10) - by (W-SUBTYPE) with (7,9)
$(\tau^{\mathsf{s}''})\{v/x\} <: (\tau^{\mathsf{s}})\{v/x\}$ (11) - Lem. 11 with (4,10,2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ by rule (T-SUB) with (5,6,11)

Case (T-IF):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}}$ (3) - inv. (T-IF) of (1)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}\sqcup\mathsf{s}}_{\mathcal{S}\cup\{c\doteq\mathtt{true}\}} e_1 : \tau^{\mathsf{s}}$ (4) - inv. (T-IF) of (1)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}\sqcup\mathsf{s}}_{\mathcal{S}\cup\{c\doteq\mathtt{false}\}} e_2 : \tau^{\mathsf{s}}$ (5) - inv. (T-IF) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} c\{v/x\} : \mathtt{Bool}^{\mathsf{s}\{v/x\}}$ (6) - I.H. with (3,2)
$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}}_{\mathcal{S}\{v/x\}\cup\{c\{v/x\}\doteq\mathtt{true}\}} e_1\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ (7) - I.H. with (4,2)
$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}}_{\mathcal{S}\{v/x\}\cup\{c\{v/x\}\doteq\mathtt{false}\}} e_2\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ (8) - I.H. with (5,2)
$(\textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2\{v/x\}) = (\textbf{if } c\{v/x\} \textbf{ then } e_1\{v/x\} \textbf{ else } e_2\{v/x\})$
(9) - by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \textbf{if } c\{v/x\} \textbf{ then } e_1\{v/x\} \textbf{ else } e_2\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$
by rule (t-if) with (6,7,8), and by (9)

Case (T-CONS):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1{::}e_2 : \tau^{*\mathsf{s}}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}}$ (3) - inv. (T-CONS) of (1)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{*\mathsf{s}}$ (4) - inv. (T-CONS) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_1\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ (5) - I.H. with (3), (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_2\{v/x\}{:}(\tau^{*\mathsf{s}})\{v/x\}$ (6) - I.H. with (4), (2)
$(e_1{::}e_2)\{v/x\} = (e_1\{v/x\}{::}e_2\{v/x\})$ (7) - by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_1\{v/x\}{::}e_2\{v/x\} : (\tau^{*\mathsf{s}})\{v/x\}$ by rule (T-CONS) with (5,6), and by (7)

Case (T-COLLECTION):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \{e_1, \ldots, e_n\} : \tau^{*\mathsf{s}}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp
$\forall_i\ \Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (3) - inv. (T-COLLECTION) of (1)
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_i\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$ (4) - I.H. with (3), (2)
$\{e_1, \ldots, e_n\}\{v/x\} = \{e_1\{v/x\}, \ldots, e_n\{v/x\}\}$ (5) - by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \{e_1\{v/x\}, \ldots, e_n\{v/x\}\} : (\tau^{*\mathsf{s}})\{v/x\}$ by rule (T-COLLECTION) with (4) and (5)

Case (T-RECORD):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i{=}e_i, \ldots] : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\sqcap|\mathsf{s}_i|^{\downarrow}}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$ (2) - hyp
$\forall_i\ \Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}$ (3) - inv. (T-RECORD) with (1)
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_i\{v/x\}{:}(\tau_i^{\mathsf{s}_i})\{v/x\}$ (4) - I.H. with (3), (2)
$[\ldots, m_i = e_i, \ldots]\{v/x\} = [\ldots \times m_i{:}e_i\{v/x\} \times \ldots]$ (5) - by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} [\ldots, m_i = e_i\{v/x\}, \ldots] : \Sigma[\ldots \times m_i : (\tau_i^{\mathsf{s}_i})\{v/x\} \times \ldots]^{\sqcap|\mathsf{s}_i|^{\downarrow}}$
by rule (T-RECORD) with (4) and by (5)

Case (T-OR):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} c_1 \vee c_2{:}\mathtt{Bool}^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} c_1{:}\mathtt{Bool}^{\mathsf{s}}$ (3) - inv. (T-OR) with (1)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} c_2{:}\mathtt{Bool}^{\mathsf{s}}$ (4) - inv. (T-OR) with (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} c_1\{v/x\}{:}\mathtt{Bool}^{\mathsf{s}\{v/x\}}$ (5) - I.H. with (3), (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} c_2\{v/x\}{:}\mathtt{Bool}^{\mathsf{s}\{v/x\}}$ (6) - I.H. with (4), (2)
$(c_1 \vee c_2)\{v/x\} = c_1\{v/x\} \vee c_2\{v/x\}$ (7) - by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} c_1\{v/x\} \vee c_2\{v/x\}{:}\mathtt{Bool}^{\mathsf{s}\{v/x\}}$ by rule (T-OR) with (5,6), and by (7)

Case (T-NOT):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \neg c : \mathtt{Bool}^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash v : \tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} c{:}\mathtt{Bool}^{\mathsf{s}}$ (3) - inv. (T-NOT) with (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} c\{v/x\}{:}\mathtt{Bool}^{\mathsf{s}\{v/x\}}$ (4) - I.H. with (3), (2)
$(\neg c)\{v/x\} = \neg c\{v/x\}$ (5) -by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \neg c\{v/x\} : \mathtt{Bool}^{\mathsf{s}\{v/x\}}$ by rule (T-NOT) with (4) and by (5)

Case (T-EQUAL):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} V_1 = V_2 : \mathtt{Bool}^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} V_1 : \tau^{\mathsf{s}}$ (3) - inv. (T-EQUAL) with (1)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} V_2 : \tau^{\mathsf{s}}$ (4) - inv. (T-EQUAL) with (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} V_1\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ (5) - I.H. with (3), (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} V_2\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ (6) - I.H. with (4), (2)

$(V_1 = V_2)\{v/x\} = V_1\{v/x\} = V_2\{v/x\}$ by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} V_1\{v/x\} = V_2\{v/x\} : \texttt{Bool}^{\mathsf{s}\{v/x\}}$ by rule (T-EQUAL) with (5,6), (7)

Case (T-REFINERECORD):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}}$ (1) - hyp.
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp.
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}}$ (3) - inv. (T-REFINERECORD) of (1)
$\mathcal{S}\{y \doteq e\} \models y.m_j \doteq \mathsf{v}$ (4) - inv. (T-REFINERECORD) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}})\{v/x\}$ (5) - I.H. with (3), (2)
$\mathcal{S}\{v/x\}\{y \doteq e\{v/x\}\} \models y.m_j \doteq \mathsf{v}$ (6) - from (4), subst closure of $\doteq$
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}})\{v/x\}$
by rule (T-REFINERECORD) with (5,6)

Case (T-UNREFINERECORD):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}}$ (1) - hyp.
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp.
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}}$ (3) - inv. (T-UNREFINERECORD) of (1)
$\mathcal{S}\{y \doteq e\} \models y.m_j \doteq \mathsf{v}$ (4) - inv. (T-UNREFINERECORD) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}})\{v/x\}$ (5) - I.H. with (3), (2)
$\mathcal{S}\{v/x\}\{y \doteq e\{v/x\}\} \models y.m_j \doteq \mathsf{v}$ (6) - from (4), subst closure of $\doteq$
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}})\{v/x\}$
by rule (T-UNREFINERECORD) with (5,6)

Case (T-FOREACH):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, y.z.e_3) : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (2) - hyp

- $x = y$ or $x = z$
  $(\mathbf{foreach}(e_1, e_2, y.z.e_3))\{v/x\} = \mathbf{foreach}(e_1, e_2, y.z.e_3)$ (3) - by def.
  $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, y.z.e_3)\{v/x\} : \tau^{\mathsf{s}}$ (4) - by (1,3)
  $x \notin fv(\mathbf{foreach}(e_1, e_2, y.z.e_3))$ (5) - by def of fv
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, y.z.e_3)\{v/x\} : \tau^{\mathsf{s}}$ by (4,5)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \mathbf{foreach}(e_1, e_2, y.z.e_3)\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ x is fresh is $\Delta, \mathcal{S}$ by (2)
- $x \neq y$ and $x \neq z$
  $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau''^{*\mathsf{s}}$ (6) - inv. (T-FOREACH) of (1)
  $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (7) - inv. (T-FOREACH) of (1)
  $\Delta, x : \tau'^{\mathsf{s}'}, y : \tau''^{\mathsf{s}}, z : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} e_3 : \tau^{\mathsf{s}}$ (8) - inv. (T-FOREACH) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_1\{v/x\} : (\tau''^{*\mathsf{s}})\{v/x\}$ (9) - I.H. with (6), (2)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_2\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ (10) - I.H. with (7), (2)
  $\Delta, y : \tau''^{\mathsf{s}}, z : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_3\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ (11) - I.H. with (8), (2)
  $(\mathbf{foreach}(e_1, e_2, y.z.e_3))\{v/x\} = \mathbf{foreach}(e_1\{v/x\}, e_2\{v/x\}, y.z.e_3\{v/x\})$

(12) - by def. of substitution

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \mathbf{foreach}(e_1\{v/x\}, e_2\{v/x\}, y.z.e_3\{v/x\}) : (\tau^{\mathsf{s}})\{v/x\}$

by rule (T-FOREACH) with (9,10,11), and by (12)

Case (T-REF):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{ref}_{\tau^{\mathsf{s}}}\ e : \mathtt{ref}(\tau^{\mathsf{s}})^{\perp}$	(1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$	(2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$	(3) - inv. (T-REF) with (1)
$\mathsf{r} \leq \mathsf{s}$	(4) - inv. (T-REF) with (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\}{:}(\tau^{\mathsf{s}})\{v/x\}$	(5) - I.H. with (3), (2)
$(\mathbf{ref}_{\tau^{\mathsf{s}}}\ e)\{v/x\} = \mathbf{ref}_{\tau^{\mathsf{s}}}\ e\{v/x\}$	(6) - by def. of substitution
$\mathsf{r} \leq \mathsf{s}\{v/x\}$	(7) - by (4) we have $\mathsf{s}\{v/x\} = \mathsf{s}$
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} \mathbf{ref}_{\tau^{\mathsf{s}}}\ e\{v/x\} : \mathtt{ref}(\tau^{\mathsf{s}})\{v/x\}^{\perp}$	by rule (T-REF) with (5,7), by (6)

Case (T-DEREF):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} !e : \tau^{\mathsf{s}}$	(1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$	(2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}}$	(3) - inv. (T-DEREF) with (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}})\{v/x\}$	(4) - I.H. with (3), (2)
$(\mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}})\{v/x\} = \mathtt{ref}(\tau^{\mathsf{s}}\{v/x\})^{\mathsf{t}\{v/x\}}$	(5) - by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} !e\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}^{\mathsf{t}\{v/x\}}$	by rule (T-DEREF) with (4), by (5)

Case (T-ASSIGN):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 := e_2 : \mathtt{cmd}^{\perp}$	(1) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} v{:}\tau'^{\mathsf{s}'}$	(2) - hyp
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}}$	(3) - inv. (T-ASSIGN) with (1)
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$	(4) - inv. (T-ASSIGN) with (1)
$\mathsf{r} \sqcup \mathsf{t} \leq \mathsf{s}$	(5) - inv. (T-ASSIGN) with (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e_1\{v/x\} : (\mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}})\{v/x\}$	(6) - I.H. with (3), (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/\S\}} e_2\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$	(7) - I.H. with (4), (2)
$\mathsf{r} \sqcup \mathsf{t}\{v/x\} \leq \mathsf{s}\{v/x\}$	(8) - by (5) we have $\mathsf{s}\{v/x\} = \mathsf{s}$ and $\mathsf{t}\{v/x\} = \mathsf{t}$
$(e_1 := e_2)\{v/x\} = e_1\{v/x\} := e_2\{v/x\}$	(9) - by def. of substitution
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} (e_1 := e_2)\{v/x\} : \mathtt{cmd}^{\perp}$	by rule (T-ASSIGN) with (6,7,8), by (9)

□

We now prove type preservation, which states well-typed configurations remain well-typed after a reduction step, and possibly the final configuration is extended with new locations in the state.

Theorem 6 (Type Preservation)

Let $fv(e) \cup fv(\tau^{\mathsf{s}}) = \emptyset$, $vars(\Delta) = \emptyset$, $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$.
If $(S; e) \longrightarrow (S'; e')$ then there is Δ' such that $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau^{\mathsf{s}}$, $\Delta' \vdash S'$, and $\Delta \subseteq \Delta'$.

Proof By induction on the relation $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$.

Case (T-FIELD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e.m_i : \tau_i^{\mathsf{s}_i}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$fv(e) \cup fnfv(\tau^{\mathsf{s}}) = \emptyset$ and $vars(\Delta) = \emptyset$ (3) - hyp

- Sub-case **(FIELD-LEFT)**: $(S; e.m_i) \longrightarrow (S'; e'.m_i)$ (4) - hyp
  $(S; e) \longrightarrow (S'; e')$ (5) - inv. (FIELD-LEFT) of (4)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$ (6) - inv. (T-FIELD) of (1)
  $\Delta \subseteq \Delta'$ (7) - I.H. with (2,5,6)
  $\Delta' \vdash S'$ (8) - I.H. with (2,5,6)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$ (9) - I.H. with (2,5,6)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e'.m_i : \tau_i^{\mathsf{s}_i}$ by rule (T-FIELD) with (9)

- Sub-case **(FIELD-RIGHT)**: $(S; [m_1 = v_1, \ldots, m_n = v_n].m_i) \longrightarrow (S; v_i)$ (4) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [m_1 = v_1, \ldots, m_n = v_n] : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}}$ (5) - inv. (T-FIELD) of (1)

  1. s_i has no field identifiers nor variables — by Lem. 9 with (4)
     $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i{:}\tau_i^{\mathsf{s}_i}$ — by Lem. 9 with (4)
  2. $\mathsf{s}_i = \ell_i(m_j)$
     not applicable because of (1) and Def. 19
  3. not applicable
  4. not applicable

Case (T-RECORD):

$(S; [\ldots, m_i = e, \ldots]) \longrightarrow (S'; [\ldots, m_i = e', \ldots])$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = e, \ldots] : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$ (2) - hyp
$\Delta \vdash S$ (3) -hyp
$(S; e) \longrightarrow (S'; e')$ (4) - inv. (RECORD) of (1)
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}$ (5) - inv. (T-RECORD) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau_i^{\mathsf{s}_i}$ (6) - by (5)
$\Delta \subseteq \Delta'$ (7) - I.H. with (3), (4), (6)
$\Delta' \vdash S'$ (8) - I.H. with (3), (4), (6)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau_i^{\mathsf{s}_i}$ (9) - I.H. with (3), (4), (6)
$\forall_i\ \Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}$ (11) - Lem. 6 with (5)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = e', \ldots] : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$ by rule (T-RECORD) with (9,11)

Case (T-REFINERECORD):

$(S; e) \longrightarrow (S'; e')$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (4) - inv. (T-REFINERECORD) of (2)
$\mathcal{S}\{x \doteq e\} \models x.m_j \doteq \mathsf{v}$ (5) - inv. (T-REFINERECORD) of (2)
$\Delta \subseteq \Delta'$ (6) - I.H. with (1,3,4)
$\Delta' \vdash S'$ (7) - I.H. with (1,3,4)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (8) - I.H. with (1,3,4)
$\mathcal{S}\{x \doteq e'\} \models x.m_j \doteq \mathsf{v}$ (9) - by (1) since reduction preserves $\doteq$, so $e \doteq e'$
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ by rule (T-REFINERECORD) with (8,9)

Case (T-UNREFINERECORD):

$(S; e) \longrightarrow (S'; e')$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (4) - inv. (T-UNREFINERECORD) of (2)
$\mathcal{S}\{x \doteq e\} \models x.m_j \doteq \mathsf{v}$ (5) - inv. (T-UNREFINERECORD) of (2)
$\Delta \subseteq \Delta'$ (6) - I.H. with (1,3,4)
$\Delta' \vdash S'$ (7) - I.H. with (1,3,4)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (8) - I.H. with (1,3,4)
$\mathcal{S}\{x \doteq e'\} \models x.m_j \doteq \mathsf{v}$ (9) - by (1) since reduction perserves $\doteq$, so $e \doteq e'$
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ by rule (T-UNREFINERECORD) with (8,9)

Case (T-COLLECTION):

$(S; \{\ldots, e, \ldots\}) \longrightarrow (S'; \{\ldots, e', \ldots\})$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \{\ldots, e, \ldots\} : \tau^{*\mathsf{s}}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$(S; e) \longrightarrow (S'; e')$ (4) - inv. (list) of (1)
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (5) - inv. (T-COLLECTION) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$ (6) - by (5)
$\Delta \subseteq \Delta'$ (7) - I.H. with (3), (4), (6)
$\Delta' \vdash S'$ (8) - I.H. with (3), (4), (6)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau^{\mathsf{s}}$ (9) - I.H. with (3), (4), (6)
$\forall_i\ \Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (11) - Lem. 6 with (5)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} \{\ldots, e', \ldots\} : \tau^{*\mathsf{s}}$ by rule (T-COLLECTION) with (9,11)

Case (T-LET)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp

- Sub-case **(LET-LEFT)**: $(S; \textbf{let } x = e_1 \textbf{ in } e_2) \longrightarrow (S'; \textbf{let } x = e_1' \textbf{ in } e_2)$ (3) - hyp
  $(S; e_1) \longrightarrow (S'; e_1')$ (4) - inv. (LET-LEFT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau'^{\mathsf{s}'}$ (5) - inv. (T-LET) of (1)
  $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq e_1\}} e_2 : \tau^{\mathsf{s}}$ (6) - inv. (T-LET) of (1)
  $\Delta \subseteq \Delta'$ (7) - I.H. with (2,4,5)
  $\Delta' \vdash S'$ (8) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_1' : \tau'^{\mathsf{s}'}$ (9) - I.H. with (2,4,5)
  $\Delta', x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq e_1\}} e_2 : \tau^{\mathsf{s}}$ (10) - Lem. 6 with (6)
  $\Delta', x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq e_1'\}} e_2 : \tau^{\mathsf{s}}$ (11) - by (4) since reduction preserves $\doteq$, so $e_1 \doteq e_1'$
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{let } x = e_1' \textbf{ in } e_2 : \tau^{\mathsf{s}}$ by rule (T-LET) with (9) and (11)

- Sub-case **(LET-RIGHT)**: $(S; \textbf{let } x = v \textbf{ in } e_2) \longrightarrow (S; e_2\{v/x\})$ (3) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau'^{\mathsf{s}'}$ (4) - inv. (T-LET) of (1)
  $\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq v\}} e_2 : \tau^{\mathsf{s}}$ (5) - inv. (T-LET) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq v\}} v : \tau'^{\mathsf{s}'}$ (6) - x fresh in Δ, S, v from (4)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}\{v \doteq v\}} e_2\{v/x\} : (\tau^{\mathsf{s}})\{v/x\}$ (7) - **??** from (5,6)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2\{v/x\} : \tau^{\mathsf{s}}$ $\mathcal{S} = \mathcal{S}\{v/x\}$ and $\mathcal{S} \models v \doteq v$ from (7), and x fresh in τ^{s} by (1)

Case (T-APP):
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1(e_2) : \sigma\{v/x\}^{\mathsf{q}\{v/x\} \sqcup \mathsf{t}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp

- Sub-case **(APP-LEFT)**: $(S; e_1(e_2)) \longrightarrow (S'; e_1'(e_2))$ (3) - hyp
  $(S; e_1) \longrightarrow (S'; e_1')$ (4) - inv. (APP-LEFT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$ (5) - inv. (T-APP) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (6) - inv. (T-APP) of (1)
  $\mathcal{S} \cup \{x \doteq e_2\} \models x \doteq \mathsf{v}$ (7) - inv. (T-APP) of (1)
  $\mathsf{r} \leq \mathsf{r}'$ inv. (t-app) of (1)
  $\Delta \subseteq \Delta'$ (8) - I.H. with (2,4,5)
  $\Delta' \vdash S'$ (9) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_1' : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$ (10) - I.H. with (2,4,5)
  $\mathcal{S}\{v/x\} \cup \{x \doteq e_2\{v/x\}\} \models x \doteq \mathsf{v}$ (11) - from (7), subst closure of $\doteq$
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (12) - Lem. 6 with (6)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_1'(e_2) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ by rule (T-APP) with (10,11,12)

- Sub-case **(APP-RIGHT)**: $(S; (\lambda(x{:}\tau^{\mathsf{s}'}).e)(e_2)) \to (S'; (\lambda(x{:}\tau^{\mathsf{s}'}).e)(e_2'))$ (3) - hyp
  $(S; e_2) \longrightarrow (S'; e_2')$ (4) - inv. (APP-RIGHT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} (\lambda(x : \tau^{\mathsf{s}'}).e) : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$ (5) - inv. (T-APP) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (6) - inv. (T-APP) of (1)
  $\mathcal{S} \cup \{x \doteq e_2\} \models x \doteq \mathsf{v}$ (7) - inv. (T-APP) of (1)
  $\mathsf{r} \leq \mathsf{r}'$ inv. (T-APP) of (1)

$\Delta \subseteq \Delta'$ (8) - I.H. with (2,4,6)
$\Delta' \vdash S'$ (9) - I.H. with (2,4,6)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_2' : \tau^{\mathsf{s}}$ (10) - I.H. with (2,4,6)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} (\lambda(x : \tau^{\mathsf{s}'}).e) : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$ (11) - Lem. 6 with (5)
$\mathcal{S} \cup \{x \doteq e_2'\} \models x \doteq \mathsf{v}$ (12) - since reduction perserves $\doteq$, so $e \doteq e'$
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} (\lambda(x : \tau^{\mathsf{s}'}).e)(e_2) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ by rule (T-APP) with (10,11,12)

- Sub-case **(APP)**: $(S; (\lambda(x : \tau^{\mathsf{s}'}).e)(v)) \longrightarrow (S; e\{v/x\})$ (3) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} (\lambda(x : \tau^{\mathsf{s}'}).e) : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}}$ (4) - inv. (T-APP) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$ (5) - inv. (T-APP) of (1)
  $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ (6) - by Lem. 9 with (4)
  $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$ (7) - by Lem. 9 with (4)
  $\Delta, x : \tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e : \sigma^{\mathsf{q}'}$ (8) - by Lem. 9 with (4)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}'}$ (9) - by rule (t-sub) with (5), (6)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{v/x\}} e\{v/x\} : (\sigma^{\mathsf{q}'})\{v/x\}$ (10) - **??** with (8), (9)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e\{v/x\} : (\sigma^{\mathsf{q}'})\{v/x\}$ (11) - $\mathcal{S} = \mathcal{S}\{v/x\}$ since x is fresh in $\Delta, \mathcal{S}$ by (9)
  $(\sigma^{\mathsf{q}'})\{v/x\} <: (\sigma^{\mathsf{q}})\{v/x\}$ (12) - by Lem. 11 with (7)
  $(\sigma^{\mathsf{q}})\{v/x\} <: \sigma\{v/x\}^{\mathsf{q}\{v/x\} \sqcup \mathsf{t}}$ (13) - since $\mathsf{q}\{v/x\} \leq \mathsf{q}\{v/x\} \sqcup \mathsf{t}$ by def. of $\sqcup$
  $(\sigma^{\mathsf{q}'})\{v/x\} <: \sigma\{v/x\}^{\mathsf{q}\{v/x\} \sqcup \mathsf{t}}$ (14) - by (s-trans) with (12,13)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e\{v/x\} : \sigma\{v/x\}^{\mathsf{q}\{v/x\} \sqcup \mathsf{t}}$ by rule (T-SUB) with (11), (14)

Case (T-IF):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ **if** c **then** e_1 **else** $e_2 : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}}$ (3) - inv. (T-IF) of (1)
$\Delta \vdash^{\mathsf{r} \sqcup \mathsf{s}}_{\mathcal{S} \cup \{c \doteq \mathtt{true}\}} e_1 : \tau^{\mathsf{s}}$ (4) - inv. (T-IF) of (1)
$\Delta \vdash^{\mathsf{r} \sqcup \mathsf{s}}_{\mathcal{S} \cup \{c \doteq \mathtt{false}\}} e_2 : \tau^{\mathsf{s}}$ (5) - inv. (T-IF) of (1)
$\mathsf{r} \leq \mathsf{r} \sqcup \mathsf{s}$ (6) - by def. of $\sqcup$

- Sub-case **(IF-TRUE)**: $(S;$ **if** c **then** e_1 **else** $e_2) \longrightarrow (S; e_1)$ (7) - hyp
  $\mathcal{C}[\![c]\!] = \mathtt{true}$ (8) - inv. (IF-TRUE) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S} \cup \{c \doteq \mathtt{true}\}} e_1 : \tau^{\mathsf{s}}$ (9) - by rule (T-SUB) with (4,6)
  $\mathcal{S} \models \mathtt{true} \doteq \mathtt{true}$ (10) - by (9,8)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}}$ by Lem. 7 with (9,10)

- Sub-case **(IF-FALSE)**: $(S;$ **if** c **then** e_1 **else** $e_2) \longrightarrow (S; e_2)$ (7) - hyp
  $\mathcal{C}[\![c]\!] = \mathtt{false}$ (8) - inv. (IF-TRUE) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S} \cup \{c \doteq \mathtt{false}\}} e_2 : \tau^{\mathsf{s}}$ (9) - by rule (T-SUB) with (5,6)
  $\mathcal{S} \models \mathtt{false} \doteq \mathtt{false}$ (10) - by (9,8)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ by Lem. 7 with (9,10)

Case (T-CASE):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{case}\ e(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$ (3) - inv. (T-CASE) of (1)
$\forall_i\ \ \Delta, x_i{:}\tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (4) - inv. (T-CASE) of (1)

- Sub-case **(CASE-LEFT)**: $(S; \textbf{case}\ e(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S'; \textbf{case}\ e'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots))$ (5) - hyp
  $(S; e) \longrightarrow (S'; e')$ (6) - inv. of CASE-LEFT with (5)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$ (7) - by I.H. with (2,3,6)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{case}\ e'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) : \tau^{\mathsf{s}}$ by rule (T-CASE) with (7,4)

- Sub-case **(CASE-RIGHT)**: $(S; \textbf{case}\ \#m_i(v)(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S'; e_i\{^v/_{x_i}\})$ (8) - hyp
  $\Delta, x_i{:}\tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (9) - by (4)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau_i'^{\mathsf{s}_i'}$ (10) - by Lem. 9 with (3) since $e = \#m_i(v)$
  $\tau_i'^{\mathsf{s}_i'} <: \tau_i^{\mathsf{s}_i}$ (11) - by Lem. 9 with (3) since $e = \#m_i(v)$
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau_i^{\mathsf{s}_i}$ (12) - rule T-SUB with (10,11)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i\{^v/_{x_i}\} : \tau^{\mathsf{s}}$ (9) - by **??** with (9,12)

Case (T-VARIANT):

$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} \#m_i(e)\ \textbf{as}\ \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\sqcap \mathsf{s}_i} : \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\sqcap \mathsf{s}_i}$ (1) - hyp
$(S; e) \longrightarrow (S'; e')$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau_i^{\mathsf{s}_i}$ (4) - inv. (T-VARIANT) of (1)
$\Delta \subseteq \Delta'$ (5) - I.H. with (2,3,4)
$\Delta' \vdash S'$ (6) - I.H. with (2,3,4)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau_i^{\mathsf{s}_i}$ (7) - I.H. with (2,3,4)
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} \#m_i(e')\ \textbf{as}\ \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\sqcap \mathsf{s}_i} : \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\sqcap \mathsf{s}_i}$ by rule (T-VARIANT) with (7)

Case (T-CONS):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1{::}e_2 : \tau^{*\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp

- Sub-case **(CONS-LEFT)**: $(S; e_1{::}e_2) \longrightarrow (S'; e_1'{::}e_2)$ (3) - hyp
  $(S; e_1) \longrightarrow (S'; e_1')$ (4) - inv. (CONS-LEFT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}}$ (5) - inv. (T-CONS) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{*\mathsf{s}}$ (6) - inv. (T-CONS) of (1)
  $\Delta \subseteq \Delta'$ (7) - I.H. with (2,4,5)
  $\Delta' \vdash S'$ (8) - I.H. with (2,4,5)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e'_1 : \tau^{\mathsf{s}}$ (9) - I.H. with (2,4,5)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{*\mathsf{s}}$ (10) - Lem. 6 with (6)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e'_1 :: e_2 : \tau^{*\mathsf{s}}$ by rule (T-CONS) with (9,10)

- Sub-case **(CONS-RIGHT)**: $(S; v :: e_2) \longrightarrow (S'; v :: e'_2)$ (3) - hyp
  $(S; e_2) \longrightarrow (S'; e'_2)$ (4) - inv. (CONS-RIGHT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$ (5) - inv. (T-CONS) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{*\mathsf{s}}$ (6) - inv. (T-CONS) of (1)
  $\Delta \subseteq \Delta'$ (7) - I.H. with (2,4,6)
  $\Delta' \vdash S'$ (8) - I.H. with (2,4,6)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e'_2 : \tau^{*\mathsf{s}}$ (9) - I.H. with (2,4,6)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$ (10) - Lem. 6 with (5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} v :: e'_2 : \tau^{*\mathsf{s}}$ by rule (T-CONS) with (9,10)

- Sub-case **(CONS)**: $(S; v :: \{v_1, \ldots, v_n\}) \longrightarrow (S; \{v, v_1, \ldots, v_n\})$
  (3) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$ (4) - inv. (T-CONS) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \{v_1, \ldots, v_n\} : \tau^{*\mathsf{s}}$ (5) - inv. (T-CONS) of (1)
  $\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau^{\mathsf{s}}$ (6) - inv. (T-COLLECTION) of (5)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \{v, v_1, \ldots, v_n\} : \tau^{*\mathsf{s}}$ by rule (T-COLLECTION) with (4,6)

Case (T-SUB):
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} e : \tau^{\mathsf{s}'}$ (1) - hyp
$(S; e) \longrightarrow (S'; e')$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$ (4) - inv. (T-SUB) of (1)
$\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ and $\mathsf{r}' \leq \mathsf{r}$ (6) - inv. (T-SUB) of (1)
$\Delta \subseteq \Delta'$ (7) - I.H. with (2,3,4)
$\Delta' \vdash S'$ (8) - I.H. with (2,3,4)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau^{\mathsf{s}}$ (9) - I.H. with (2,3,4)
$\Delta' \vdash^{\mathsf{r}'}_{\mathcal{S}} e' : \tau^{\mathsf{s}'}$ by rule (T-SUB) with (9,6)

Case (T-FOREACH):
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, x.y.e_3)) : \tau'^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp

- Sub-case **(FOREACH-LEFT)**: $(S; \mathbf{foreach}(e_1, e_2, x.y.e_3)) \longrightarrow$
  $(S'; \mathbf{foreach}(e'_1, e_2, x.y.e_3))$ (3) - hyp
  $(S; e_1) \longrightarrow (S'; e'_1)$ (4) - inv. (FOREACH-LEFT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{*\mathsf{s}}$ (5) - inv. (T-FOREACH) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau'^{\mathsf{s}}$ (6) - inv. (T-FOREACH) of (1)

$\Delta, x : \tau^{s}, y : \tau'^{s} \vdash^{r}_{\mathcal{S}} e_3 : \tau'^{s}$ (7) - inv. (T-FOREACH) of (1)
$\Delta \subseteq \Delta'$ (8) - I.H. with (2,4,5)
$\Delta' \vdash S'$ (9) - I.H. with (2,4,5)
$\Delta' \vdash^{r}_{\mathcal{S}} e'_1 : \tau^{*s}$ (10) - I.H. with (2,4,5)
$\Delta' \vdash^{r}_{\mathcal{S}} e_2 : \tau'^{s}$ (11) - Lem. 6 with (6)
$\Delta', x : \tau^{s}, y : \tau'^{s} \vdash^{r}_{\mathcal{S}} e_3 : \tau'^{s}$ (12) - Lem. 6 with (7)
$\Delta' \vdash^{r}_{\mathcal{S}} \mathbf{foreach}(e'_1, e_2, x.y.e_3)) : \tau'^{s}$ by rule (T-FOREACH) with (10,11,12)

- Sub-case **(FOREACH-RIGHT)**: $(S; \mathbf{foreach}(v, e_2, x.y.e_3))) \longrightarrow (S'; \mathbf{foreach}(v, e'_2, x.y.e_3)))$ (3) - hyp
$(S; e_2) \longrightarrow (S'; e'_2)$ (4) - inv. (FOREACH-RIGHT) of (3)
$\Delta \vdash^{r}_{\mathcal{S}} v : \tau^{*s}$ (5) - inv. (T-FOREACH) of (1)
$\Delta \vdash^{r}_{\mathcal{S}} e_2 : \tau'^{s}$ (6) - inv. (T-FOREACH) of (1)
$\Delta, x : \tau^{s}, y : \tau'^{s} \vdash^{r}_{\mathcal{S}} e_3 : \tau'^{s}$ (7) - inv. (T-FOREACH) of (1)
$\Delta \subseteq \Delta'$ (8) - I.H. with (2,4,6)
$\Delta' \vdash S'$ (9) - I.H. with (2,4,6)
$\Delta' \vdash^{r}_{\mathcal{S}} e'_2 : \tau'^{s}$ (10) - I.H. with (2,4,6)
$\Delta' \vdash^{r}_{\mathcal{S}} v : \tau^{*s}$ (11) - Lem. 6 with (5)
$\Delta', x : \tau^{s}, y : \tau'^{s} \vdash^{r}_{\mathcal{S}} e_3 : \tau'^{s}$ (12) - Lem. 6 with (7)
$\Delta' \vdash^{r}_{\mathcal{S}} \mathbf{foreach}(v, e'_2, x.y.e_3) : \tau'^{s}$ by rule (T-FOREACH) with (13,14,15)

- Sub-case **(FOREACH)**: $(S; \mathbf{foreach}(l, v, x.y.e_3)) \longrightarrow (S'; \mathbf{foreach}(hs, e_3\{h/x\}\{v/y\}, x.y.e_3))$ (3) - hyp
$l = h::hs$ (4) - inv. (FOREACH) of (3)
$\Delta \vdash^{r}_{\mathcal{S}} l : \tau^{*s}$ (5) - inv. (T-FOREACH) of (1)
$\Delta \vdash^{r}_{\mathcal{S}} v : \tau'^{s}$ (6) - inv. (T-FOREACH) of (1)
$\Delta, x : \tau^{s}, y : \tau'^{s} \vdash^{r}_{\mathcal{S}} e_3 : \tau'^{s}$ (7) - inv. (T-FOREACH) of (1)
$\Delta \vdash^{r}_{\mathcal{S}} h : \tau^{s}$ (8) - inv. (T-CONS) of (5)
$\Delta \vdash^{r}_{\mathcal{S}} hs : \tau^{*s}$ (9) - inv. (T-CONS) of (5)
$\Delta \vdash^{r}_{\mathcal{S}'} e_3\{h/x\}\{v/y\} : (\tau'^{s})\{h/x\}\{v/y\}$ (10) - **??** with (6,7,8)
$\mathcal{S}' = \mathcal{S}\{h/x\}\{v/y\}$ (11) - **??** with (6,7,8)
$\Delta \vdash^{r}_{\mathcal{S}} e_3\{h/x\}\{v/y\} : (\tau'^{s})\{h/x\}\{v/y\}$ (12) - $\mathcal{S}' = \mathcal{S}$, since x, y fresh in $\Delta, \mathcal{S}, e_3\{h/x\}\{v/y\}$ by (5,6)
$\Delta \vdash^{r}_{\mathcal{S}} \mathbf{foreach}(hs, e_3\{h/x\}\{v/y\}, x.y.e_3) : (\tau'^{s})\{h/x\}\{v/y\}$ by rule (T-FOREACH) with (7,9,12)
$\Delta \vdash^{r}_{\mathcal{S}} \mathbf{foreach}(hs, e_3\{h/x\}\{v/y\}, x.y.e_3) : \tau'^{s}$ x, y fresh in τ'^{s} by (1,6) and by Def. 19

- Sub-case **(FOREACH-BASE)**: $(S; \mathbf{foreach}(\{\}, v, x.y.e_3)) \longrightarrow (S; v)$ (3) - hyp
$\Delta \vdash^{r}_{\mathcal{S}} \{\} : \tau^{*s}$ (4) - inv. (T-FOREACH) of (1)
$\Delta \vdash^{r}_{\mathcal{S}} v : \tau'^{s}$ (5) - inv. (T-FOREACH) of (1)
$\Delta, x : \tau^{s}, y : \tau'^{s} \vdash^{r}_{\mathcal{S}} e_3 : \tau'^{s}$ (6) - inv. (T-FOREACH) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau'^{\mathsf{s}}$ by (5)

Case (T-REF):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{ref}_{\tau^{\mathsf{s}}}\ e : \mathtt{ref}(\tau^{\mathsf{s}})^{\perp}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp

- Sub-case **(REF-LEFT)**: $(S; \mathbf{ref}_{\tau^{\mathsf{s}}}\ e) \longrightarrow (S'; \mathbf{ref}_{\tau^{\mathsf{s}}}\ e')$ (3) - hyp
  $(S; e) \longrightarrow (S'; e')$ (4) - inv. (REF-LEFT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$ (5) - inv. (T-REF) of (1)
  $\Delta \subseteq \Delta'$ (6) - I.H. with (2,4,5)
  $\Delta' \vdash S'$ (7) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \tau^{\mathsf{s}}$ (8) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{ref}_{\tau^{\mathsf{s}}}\ e' : \mathtt{ref}(\tau^{\mathsf{s}})^{\perp}$ by rule (T-REF) with (8)

- Sub-case **(REF-RIGHT)**: $(S; \mathbf{ref}_{\tau^{\mathsf{s}}}\ v) \longrightarrow (S \cup \{l \mapsto v\};\ l)$ (3) - hyp
  $l \notin dom(S) \cup fn(e)$ (4) - inv. (REF-RIGHT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$ (5) - inv. (T-REF) of (1)
  $\Delta, l : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}})^{\perp}$ (6) - by (T-LOC)
  $\Delta, l : \tau^{\mathsf{s}} \vdash S \cup \{l \mapsto v\}$ (7) - by Def. 22 with (5,6)

Case (T-DEREF):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} !e : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp

- Sub-case **(DEREF-LEFT)**: $(S; !e) \longrightarrow (S'; !e')$ (3) - hyp
  $(S; e) \longrightarrow (S'; e')$ (4) - inv. (DEREF-LEFT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (5) - inv. (T-DEREF) of (1)
  $\Delta \subseteq \Delta'$ (6) - I.H. with (2,4,5)
  $\Delta' \vdash S'$ (7) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e' : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (8) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} !e' : \tau^{\mathsf{s}}$ by rule (T-DEREF) with (8)

- Sub-case **(DEREF)**: $(S; !l) \longrightarrow (S; v)$ (3) - hyp
  $S(l) = v$ (4) - inv. (DEREF) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (5) - inv. (T-DEREF) of (1)
  $\Delta(l) = \tau^{\mathsf{s}}$ (6) - by (T-LOC) with (5)
  $\Delta \vdash S$ (7) - by Def. 22 with (4,6)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$ by (4,6,7)

Case (T-ASSIGN):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 := e_2 : \mathtt{cmd}^{\perp}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp

- Sub-case **(ASSIGN-LEFT)**: $(S; e_1) \longrightarrow (S'; e_1')$ (3) - hyp
  $(S; e_1) \longrightarrow (S'; e_1')$ (4) - inv. (ASSIGN-LEFT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (5) - inv. (T-ASSIGN) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (6) - inv. (T-ASSIGN) of (1)
  $\Delta \subseteq \Delta'$ (7) - I.H. with (2,4,5)
  $\Delta' \vdash S'$ (8) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_1' : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (9) - I.H. with (2,4,5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (10) - by Lem. 6 with (6)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_1' := e_2 : \mathtt{cmd}^{\perp}$ by (T-ASSIGN) with (9,10)

- Sub-case **(ASSIGN-RIGHT)**: $(S; e_2) \longrightarrow (S'; e_2')$ (3) - hyp
  $(S; e_1) \longrightarrow (S'; e_1')$ (4) - inv. (ASSIGN-RIGHT) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (5) - inv. (T-ASSIGN) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (6) - inv. (T-ASSIGN) of (1)
  $\Delta \subseteq \Delta'$ (7) - I.H. with (2,4,6)
  $\Delta' \vdash S'$ (8) - I.H. with (2,4,6)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_2' : \tau^{\mathsf{s}}$ (9) - I.H. with (2,4,6
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (10) - by Lem. 6 with (5)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} l := e_2' : \mathtt{cmd}^{\perp}$ by (T-ASSIGN) with (9,10)

- Sub-case **(ASSIGN)**: $(S; l := v) \longrightarrow (S[l \mapsto v]; ())$ (3) - hyp
  $l \in dom(S)$ (4) - inv. (ASSIGN) of (3)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (5) - inv. (T-ASSIGN) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}}$ (6) - inv. (T-ASSIGN) of (1)
  $\Delta(l) = \tau^{\mathsf{s}}$ (7) - by (T-LOC) with (5)
  $\Delta \vdash S[l \mapsto v]$ by (5,6,7)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} () : \mathtt{cmd}^{\perp}$ by (T-UNIT)

□

Finally, we conclude the proof of type safety with the proof of our progress result that ensures well-typed programs never get stuck.

Theorem 7 (Progress)
Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$, and $\Delta \vdash S$, then e is either a value or $(S; e) \longrightarrow (S'; e')$.

Proof By induction on the statement $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$.

Cases (T-TRUE), (T-FALSE), (T-UNIT), (T-LAMBDA), (T-LOC), (T-INJ)

For any of these cases, the expression is a value.

Case (T-OR), (T-NOT), (T-EQUAL)

In these cases we have a conditional expression that is evaluated through an interpretation function $\mathcal{C}$. Since $\mathcal{C}$ is a total function, then we know the evaluation of these expressions terminates with a boolean value.

Case (T-FIELD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e.m_i : \tau_i^{\mathsf{s}_i}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_i{:}\tau^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}$ (3) - inv. (T-FIELD) of (1)

- $(S; e) \longrightarrow (S'; e')$ (5) - by I.H. with (3),(2)
  $(S; e.m_i) \longrightarrow (S'; e'.m_i)$ by rule (FIELD-LEFT) with (5)
- e is a value (6) - by I.H. with (3),(2)
  $e = [\overline{m : v}]$ (7) - Lem. 10 with (3),(6)
  $\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i : \tau_i^{\mathsf{s}'_i}$ Lem. 10 with (3),(6)
  $(S; e.m_i) \longrightarrow (S; v_i)$ by rule (FIELD-RIGHT) with (7)

Case (T-LET):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau'^{\mathsf{s}'}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}}$ (3) - inv. (T-LET) of (1)
$\Delta, x : \tau^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq e_1\}} e_2 : \tau'^{\mathsf{s}'}$ inv. (T-LET) of (1)

- $(S; e_1) \longrightarrow (S'; e'_1)$ (4) - by I.H. with (3),(2)
  $(S; \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \longrightarrow (S'; \mathbf{let}\ x = e'_1\ \mathbf{in}\ e_2)$ (6) - by rule (LET-LEFT) with (4)
- e_1 is a value (5) - by I.H. with (3),(2)
  $(S; \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \longrightarrow (S; e_2\{^{x}/_{e_1}\})$ by rule (LET-RIGHT) with (5)

Case (T-CASE):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{case}\ e(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$ (3) - inv. (T-CASE) of (1)
$\forall_i\ \ \Delta, x_i{:}\tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (4) - inv. (T-CASE) of (1)

- $(S; e) \longrightarrow (S'; e')$ (5) - by I.H. with (3),(2)
  $(S; \mathbf{case}\ e(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S'; \mathbf{case}\ e'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots))$
  (6) - by rule (CASE-LEFT) with (5)
- e is a value (7) - by I.H. with (3),(2)
  $(S; \mathbf{case}\ \#m(v)(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S; e_i\{^{v}/_{x_i}\})$
  by rule (CASE-RIGHT) with (7)

Case (T-APP):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1(e_2) : \sigma\{^{v}/_{x}\}^{\mathsf{q}\{^{v}/_{x}\}\sqcup\mathsf{t}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r};\sigma^{\mathsf{q}})^{\mathsf{t}}$ (3) - inv. (T-APP) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (4) - inv. (T-APP) of (1)

- $(S; e_1) \longrightarrow (S'; e_1')$ (5) - by I.H. with (3),(2)
  $(S; e_1(e_2)) \longrightarrow (S'; e_1'(e_2))$ by (APP-LEFT) with (5)
- e_1 is a value (6) - by I.H. with (3),(2)
  $e_1 = \lambda(x : \tau^{\mathsf{s}'}).e$ (7) - Lem. 10 with (3),(6)
  - $(S; e_2) \longrightarrow (S'; e_2')$ (8) - by I.H. with (4),(2)
    $(S; e_1(e_2)) \longrightarrow (S'; e_1(e_2'))$ by rule (APP-RIGHT) with (8,7)
  - e_2 is a value (9) - by I.H. with (4),(2)
    $(S; (\lambda(x : \tau^{\mathsf{s}'}).e)(e_2)) \longrightarrow (S; e\{^{x}/_{e_2}\})$
    by rule (APP) with (9),(7)

Case (T-SUB):

$\Delta^{\mathsf{r}'}_{\mathcal{S}} \vdash e : \tau^{\mathsf{s}'}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$ (3) - inv. (T-SUB) with (1)
$\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$ inv. (T-SUB) with (1)
$\mathsf{r}' \leq \mathsf{r}$ inv. (T-SUB) with (1)
$(S; e) \longrightarrow (S'; e')$ or e is a value by I.H. with (3),(2)

Case (T-IF):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}}$ **if** c **then** e_1 **else** $e_2 : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} c : \mathtt{Bool}^{\mathsf{s}}$ inv. (T-IF) of (1)
$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}}_{\mathcal{S}\cup\{c\doteq\mathtt{true}\}} e_1 : \tau^{\mathsf{s}}$ inv. (T-IF) of (1)
$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}}_{\mathcal{S}\cup\{c\doteq\mathtt{false}\}} e_2 : \tau^{\mathsf{s}}$ inv. (T-IF) of (1)
$\mathcal{C}$ is a total function (3) - by def.
$\mathcal{C}[\![c]\!]$ is either `true` or `false` (4) - by def. and (3)
$(S;$ **if** c **then** e_1 **else** $e_2) \longrightarrow (S; e_1)$ by rule (IF-TRUE), (4) with `true`
$(S;$ **if** c **then** e_1 **else** $e_2) \longrightarrow (S; e_2)$ by rule (IF-FALSE), (4) with`false`

Case (T-CONS):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1{::}e_2 : \tau^{*\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}}$ (3) - inv. (T-CONS) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{*\mathsf{s}}$ (4) - inv. (T-CONS) of (1)

- $(S; e_1) \longrightarrow (S'; e_1')$ (5) - by I.H. with (3),(2)
  $(S; e_1 :: e_2) \longrightarrow (S; e_1' :: e_2)$ by (CONS-LEFT) with (5)
- e_1 is a value (6) - by I.H. with (3),(2)
  - $(S; e_2) \longrightarrow (S'; e_2')$ (7) - by I.H. with (4),(2)
    $(S; e_1 :: e_2) \longrightarrow (S; e_1 :: e_2')$ by rule (CONS-RIGHT) with (6,7)
  - e_2 is a value (8) - by I.H. with (4),(2)
    $e_2 = \{v_1, \ldots, v_n\}$ such that $\forall_i\ \Delta \vdash v_i : \tau^{\mathsf{s}'}$ (9) - Lem. 10 with (4),(8)
    $(S; e_1 :: \{v_1, \ldots, v_n\}) \longrightarrow (S; \{e_1, v_1, \ldots, v_n\})$ by rule (CONS) with (9),(6)

Case (T-RECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i{=}e_i, \ldots] : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\sqcap|\mathsf{s}_i|^{\downarrow}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}$ (3) - inv. (T-RECORD) of (1)

- $(S; e_i) \longrightarrow (S'; e_i')$ (4) - by I.H. with (3),(2)
  $(S; [\ldots, m_i{=}e_i, \ldots]) \longrightarrow (S'; [\ldots, m_i{=}e_i', \ldots])$ by rule (RECORD) with (4)
- $\forall_i\ e_i$ is a value (5) - by I.H. with (3),(2)
  $[\ldots, m_i{=}e_i, \ldots]$ is value

Case (T-COLLECTION):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \{e_1, \ldots, e_n\} : \tau^{*\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}}$ (3) - inv. (T-COLLECTION) of (1)

- $(S; e_i) \longrightarrow (S'; e_i')$ (4) - by I.H. with (3),(2)
  $(S; \{\ldots, e_i, \ldots\}) \longrightarrow (S'; \{\ldots, e_i', \ldots\})$ by rule (COLLECTION) with (4)
- $\forall_i\ e_i$ is a value (5) - by I.H. with (3),(2)
  $\{e_1, \ldots, e_n\}$ is value

Case (T-REFINERECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}}$ (3) - inv. (T-REFINERECORD) of (1)
$(S; e) \longrightarrow (S'; e')$ or e is a value (4) - by I.H. with (3),(2)

Case (T-UNREFINERECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}}$ (3) - inv. (T-UNREFINERECORD) of (1)
$(S; e) \longrightarrow (S'; e')$ or e is a value (4) - by I.H. with (3),(2)

Case (T-FOREACH):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, x.y.e_3) : \tau'^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{*\mathsf{s}}$ (3) - inv. (T-FOREACH) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau'^{\mathsf{s}}$ (4) - inv. (T-FOREACH) of (1)
$\Delta, x : \tau^{\mathsf{s}}, y : \tau'^{\mathsf{s}} \vdash^{\mathsf{r}}_{\mathcal{S}} e_3 : \tau'^{\mathsf{s}}$ inv. (T-FOREACH) of (1)

- $(S; e_1) \longrightarrow (S'; e_1')$ (5) - by I.H. with (3),(2)
  $(S; \mathbf{foreach}(e_1, e_2, x.y.e_3)) \longrightarrow (S; \mathbf{foreach}(e_1', e_2, x.y.e_3))$
  by rule (FOREACH-LEFT) with (5)
- e_1 is a value (6) - by I.H. with (3),(2)
  $e_1 = \{v_1, \ldots, v_n\}$ such that $\forall_i\ \Delta \vdash v_i : \tau^{\mathsf{s}}$ (7) - Lem. 10 with (3),(6)
  - $(S; e_2) \longrightarrow (S'; e_2')$ (8) - by I.H. with (4),(2)
    $(S; \mathbf{foreach}(e_1, e_2, x.y.e_3)) \longrightarrow (S; \mathbf{foreach}(e_1, e_2', x.y.e_3))$
    by rule (FOREACH-RIGHT) with (6,8)
  - e_2 is a value (9) - by I.H. with (4),(2)
    $(S; \mathbf{foreach}(\{v_1, \ldots, v_n\}, e_2, x.y.e_3)) \longrightarrow$
    $(S; \mathbf{foreach}(\{v_2, \ldots, v_n\}, e_3\{^{x}/_{v_1}\}\{^{y}/_{e_2}\}, x.y.e_3))$ by rule (FOREACH) if $n \geq 1$
    $(S; \mathbf{foreach}(\{\}, e_2, x.y.e_3)) \longrightarrow (S; e_2)$ by rule (FOREACH-BASE) if $n = 0$

Case (T-REF):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{ref}_{\tau^{\mathsf{s}}}\ e : \mathtt{ref}(\tau^{\mathsf{s}})^{\perp}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}}$ (3) - inv. (T-REF) of (1)

- $(S; e) \longrightarrow (S'; e')$ (4) - by I.H. with (3),(2)
  $(S; \mathbf{ref}_{\tau^{\mathsf{s}}}\ e) \longrightarrow (S'; \mathbf{ref}_{\tau^{\mathsf{s}}}\ e')$ by rule (REF-LEFT) with (4)
- e is a value (5) - by I.H. with (3),(2)
  $(S; \mathbf{ref}_{\tau^{\mathsf{s}}}\ e) \longrightarrow (S \cup \{l \mapsto e\};\ l)$ by rule (REF-RIGHT) with (5)

Case (T-DEREF):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} !e : \tau^{\mathsf{s}}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (3) - inv. (T-DEREF) of (1)

- $(S; e) \longrightarrow (S'; e')$ (4) - by I.H. with (3),(2)
  $(S; !e) \longrightarrow (S'; !e')$ by rule (DEREF-LEFT) with (4)
- e is a value (5) - by I.H. with (3),(2)
  $e = l$ (6) - Lem. 10 with (3)
  $S(l) = v$ such that $\Delta \vdash v : \tau^{\mathsf{s}}$ (7) - by Def. 22 with (2,3,6)
  $(S; !l) \longrightarrow (S; v)$ by rule (DEREF) with (7)

Case (T-ASSIGN):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 := e_2 : \mathtt{cmd}^{\perp}$ (1) - hyp
$\Delta \vdash S$ (2) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{s}'}$ (3) - inv. (T-ASSIGN) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}}$ (4) - inv. (T-ASSIGN) of (1)

- $(S; e_1) \longrightarrow (S'; e_1')$ (5) - by I.H. with (3),(2)
  $(S; e_1 := e_2) \longrightarrow (S'; e_1' := e_2)$ by rule (ASSIGN-LEFT) with (5)
- e_1 is a value (6) - by I.H. with (3),(2)
  $e_1 = l$ (7) - Lem. 10 with (3)
  - $(S; e_2) \longrightarrow (S'; e_2')$ (8) - by I.H. with (4),(2)
    $(S; l := e_2) \longrightarrow (S'; l := e_2')$ by rule (ASSIGN-RIGHT) with (6,8)
  - e_2 is a value (9) - by I.H. with (4),(2)
    $(S; l := v) \longrightarrow (S[l \mapsto v]; ())$ by rule (ASSIGN)

□

B.2 Noninterference

We have shown the proof of our main result, noninterference, in Chapter 4 but deferred the proofs of the main lemmas used to prove it. We shall now show their proofs, as well as of other auxiliary lemmas.

Before presenting the technical results, we show the full set of rules that define our expression equivalence relation.

Definition 27 (Expression Equivalence) Let expressions e_1 and e_2 be well-typed under typing environment Δ, computational security level r and constrain sets $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. We define expression equivalence of e_1 and e_2 up to s, asserted by $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2{:}\tau^{\mathsf{s}'}$, as inductively defined by the rules in Figure B.1 and Figure B.2.

We begin with an auxiliary lemma similar to Lemma 7 in Appendix B.1 but applied to the expression equivalence relation.

Lemma 13 (Constraint Cut Lemma)
Let $\mathcal{S}, \mathcal{S}'$ be constraint sets, and $e, e', t_1, t_1', t_2, t_2'$ expressions.
If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\cup\{t_1 \doteq t_1'\},\mathcal{S}'\cup\{t_2 \doteq t_2'\}} e \cong_{\mathsf{s}} e'{:}\tau^{\mathsf{s}'}$, $\mathcal{S} \models t_1 \doteq t_1'$ and $\mathcal{S}' \models t_2 \doteq t_2'$.
Then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S},\mathcal{S}'} e \cong_{\mathsf{s}} e'{:}\tau^{\mathsf{s}'}$.
Proof: By induction on the statement $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\cup\{t_1 \doteq t_1'\},\mathcal{S}'\cup\{t_2 \doteq t_2'\}} e \cong_{\mathsf{s}} e'{:}\tau^{\mathsf{s}'}$, using deduction closure of $\models$.

Next, our store consistency lemma states that given two equivalent stores then the stored values are also equivalent under the expression equivalence relation.

$$\text{(E-VAL)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} v : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} v : \tau^{\mathsf{s}'}}$$

$$\text{(E-VALOPAQUE)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v_1 : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} v_2 : \tau^{\mathsf{s}'} \quad \mathsf{s} < \mathsf{s}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau^{\mathsf{s}'}}$$

$$\text{(E-EXPROPAQUE)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} e_1 : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} e_2 : \tau^{\mathsf{s}'} \quad \mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}}$$

$$\text{(E-REFINERECORD)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}'} \quad \mathcal{S}_1\{x \doteq e\} \models x.m_j \doteq \mathsf{v} \quad \mathcal{S}_2\{x \doteq e'\} \models x.m_j \doteq \mathsf{v}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}'}}$$

$$\text{(E-SUB)}\quad \frac{\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''} \quad \tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'} \quad \mathsf{r} \leq \mathsf{r}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'}}$$

$$\text{(E-UNREFINERECORD)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{s}'} \quad \mathcal{S}_1\{x \doteq e\} \models x.m_j \doteq \mathsf{v} \quad \mathcal{S}_2\{x \doteq e'\} \models x.m_j \doteq \mathsf{v}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{s}'}}$$

$$\text{(E-FOREACH)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : \tau^{*\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_2' : \tau'^{\mathsf{s}'} \quad \Delta, x : \tau^{\mathsf{s}'}, y : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_3 \cong_{\mathsf{s}} e_3' : \tau'^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(e_1,\ e_2,\ x.y.e_3) \ \cong_{\mathsf{s}} \mathbf{foreach}(e_1',\ e_2',\ x.y.e_3') : \tau'^{\mathsf{s}'}}$$

$$\text{(E-APP)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}'; \sigma^{\mathsf{q}})^{\mathsf{t}} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}} \quad \mathsf{r} \leq \mathsf{r}' \quad (\mathsf{v} = \top) \vee (\mathcal{S}_1 \cup \{x \doteq e_2\} \models x \doteq \mathsf{v} \ \wedge \ \mathcal{S}_2 \cup \{x \doteq e_2'\} \models x \doteq \mathsf{v})}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1(e_2) \ \cong_{\mathsf{s}} e_1'(e_2') : (\sigma^{\mathsf{q}})\{v/x\}}$$

$$\text{(E-RECORD)}\quad \frac{\forall_i \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau_i^{\mathsf{s}_i}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} [\overline{m{=}e}] \cong_{\mathsf{s}} [\overline{m{=}e'}] : \Sigma[\overline{m_i{:}\tau'^{\mathsf{s}'}}]^{\sqcap |\mathsf{s}_i|^{\downarrow}}}$$

$$\text{(E-FIELD)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1.m_i \cong_{\mathsf{s}} e_2.m_i : \tau_i^{\mathsf{s}_i}}$$

$$\text{(E-VARIANT)}\quad \frac{\forall_i \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau_i^{\mathsf{s}_i} \quad \tau^{\mathsf{t}} = \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\sqcap \mathsf{s}_i}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \#m_i(e) \ \mathbf{as}\ \tau^{\mathsf{t}} \cong_{\mathsf{s}} \#m_i(e') \ \mathbf{as}\ \tau^{\mathsf{t}} : \tau^{\mathsf{t}}}$$

$$\text{(E-CASE)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}} \quad \forall_i \ \Delta, x_i{:}\tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{case}\ e(\overline{m \cdot x \Rightarrow e}) \cong_{\mathsf{s}} \mathbf{case}\ e'(\overline{m \cdot x \Rightarrow e'}) : \tau^{\mathsf{s}'}}$$

Figure B.1: Equivalence of expressions up to level s (Part 1)

Lemma 14 (Store Consistency Lemma)
Let S_1, S_2 be stores such that $\Delta \vdash S_1$, and $\Delta \vdash S_2$.
If $S_1 =_{\mathsf{s}} S_2$, then $\forall_l\, l{\in}dom(S_1)\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} S_1(l) \cong_{\mathsf{s}} S_2(l) : \Delta(l)$

Proof

$\Delta \vdash S_1$	(1) - hyp
$\Delta \vdash S_2$	(2) - hyp
$S_1 =_{\mathsf{s}} S_2$	(3) - hyp
$dom(S_1) \subseteq dom(\Delta)$	(4) - by Def. 22 with (1)
$\forall l \in dom(S_1)$	(5) - by Def. 22 with (1)

$$\text{(E-LET)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : \tau^{\mathsf{s}_1} \quad \Delta, x:\tau^{\mathsf{s}_1} \vdash^{\mathsf{r}}_{\mathcal{S}_1\{x\doteq e_1\},\mathcal{S}_2\{x\doteq e_1'\}} e_2 \cong_{\mathsf{s}} e_2' : \tau'^{\mathsf{s}_2}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \textbf{let } x = e_1 \textbf{ in } e_2 \;\cong_{\mathsf{s}}\; \textbf{let } x = e_1' \textbf{ in } e_2' : \tau'^{\mathsf{s}_2}}$$

$$\text{(E-IF)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c \cong_{\mathsf{s}} c' : \texttt{Bool}^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{c\doteq\texttt{true}\},\mathcal{S}_2\cup\{c\doteq\texttt{true}\}} e_1 \cong_{\mathsf{s}} e_1' : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{c\doteq\texttt{false}\},\mathcal{S}_2\cup\{c\doteq\texttt{false}\}} e_2 \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 \;\cong_{\mathsf{s}}\; \textbf{if } c' \textbf{ then } e_1' \textbf{ else } e_2' : \tau^{\mathsf{s}'}}$$

$$\text{(E-CONS)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_2' : \tau^{*\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 :: e_2 \;\cong_{\mathsf{s}}\; e_1' :: e_2' : \tau^{*\mathsf{s}'}}$$

$$\text{(E-COLLECTION)}\quad \frac{\forall_i \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \{e_1,\ldots,e_n\} \;\cong_{\mathsf{s}}\; \{e_1',\ldots,e_n'\} : \tau^{*\mathsf{s}'}}$$

$$\text{(E-EQUAL)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} V_1 \cong_{\mathsf{s}} V_1' : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} V_2 \cong_{\mathsf{s}} V_2' : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} V_1 = V_2 \;\cong_{\mathsf{s}}\; V_1' = V_2' : \texttt{Bool}^{\mathsf{s}'}}$$

$$\text{(E-OR)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c_1 \cong_{\mathsf{s}} c_1' : \texttt{Bool}^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c_2 \cong_{\mathsf{s}} c_2 : \texttt{Bool}^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c_1 \vee c_2 \;\cong_{\mathsf{s}}\; c_1' \vee c_2' : \texttt{Bool}^{\mathsf{s}'}}$$

$$\text{(E-NOT)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c \cong_{\mathsf{s}} c' : \texttt{Bool}^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \neg c \;\cong_{\mathsf{s}}\; \neg c' : \texttt{Bool}^{\mathsf{s}'}}$$

$$\text{(E-REF)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'} \quad \mathsf{r} \le \mathsf{s}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \textbf{ref}_{\tau^{\mathsf{s}}}\, e \cong_{\mathsf{s}} \textbf{ref}_{\tau^{\mathsf{s}}}\, e' : \texttt{ref}(\tau^{\mathsf{s}'})^{\perp}}$$

$$\text{(E-DEREF)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \texttt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}} \quad \mathsf{t} \le \mathsf{s}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} !e \cong_{\mathsf{s}} !e' : \tau^{\mathsf{s}'}}$$

$$\text{(E-ASSIGN)}\quad \frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : \texttt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}'} \quad \mathsf{r}\sqcup\mathsf{t} \le \mathsf{s}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 := e_2 \cong_{\mathsf{s}} e_1' := e_2' : \texttt{cmd}^{\perp}}$$

Figure B.2: Equivalence of expressions up to level s (Part 2)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} S_1(l) : \Delta(l)$	(6) - by Def. 22 with (1)
$dom(S_2) \subseteq dom(\Delta)$	(7) - by Def. 22 with (2)
$\forall l' \in dom(S_2)$	(8) - by Def. 22 with (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} S_2(l') : \Delta(l')$	(9) - by Def. 22 with (2)
$redact(\Delta, S_1, \mathsf{s}) = redact(\Delta, S_2, \mathsf{s})$	(10) - by Def. 26 with (3)
$dom(S_1) = dom(S_2)$	(11) - by (4,7,10)
$redact(\Delta(l), S_1(l), \mathsf{s}) = redact(\Delta(l), S_2(l), \mathsf{s})$	(12) - by (10,11)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} S_2(l) : \Delta(l)$	(13) - by (9,11,12)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} S_1(l) \cong_{\mathsf{s}} S_2(l) : \Delta(l)$	by (6,13) either using rule (E-VAL) or (E-VALOPAQUE)

□

We also have an inversion lemma for expression equivalence.

Lemma 15 (Inversion Lemma for Expression Equivalence)

1. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \lambda(x{:}\tau^{\mathsf{s}'}).e \;\cong_{\mathsf{s}}\; \lambda(x{:}\tau^{\mathsf{s}'}).e' {:} (\Pi x{:}\tau^{\mathsf{s}}.\sigma^{\mathsf{q}})^{\mathsf{t}}$, then $\Delta, x:\tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \;\cong_{\mathsf{s}}\; e' : \sigma^{\mathsf{q}'}$, $\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}$, and $\sigma^{\mathsf{q}'} <: \sigma^{\mathsf{q}}$.

2. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \#m_i(v)$ **as** $\{\ldots, m_i : \tau_i'^{\mathsf{s}_i'}, \ldots\}^{\mathsf{t}'} \;\cong_{\mathsf{s}}$ $\#m_i(u)$ **as** $\{\ldots, m_i : \tau_i'^{\mathsf{s}_i'}, \ldots\}^{\mathsf{t}'} : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \;\cong_{\mathsf{s}}\; u : \tau_i'^{\mathsf{s}_i'}$ and $\tau_i'^{\mathsf{s}_i'} <: \tau_i^{\mathsf{s}_i}$.

3. If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} [\ldots, m_i{=}v_i, \ldots] \cong_{\mathsf{s}} [\ldots, m_i{=}u_i, \ldots] : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$, then if:

a) s_i has no field identifiers nor variables, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i \cong_{\mathsf{s}} u_i{:}\tau_i^{\mathsf{s}_i}$

b) $\mathsf{s}_i = \ell_i(m_j)$ and $\mathcal{S}_{i\in\{1,2\}}\{x \doteq [\ldots, m_i = v_i, \ldots]\} \models x.m_j \doteq \mathsf{v}$ then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i \cong_{\mathsf{s}} u_i : \tau_i^{\ell_i(\mathsf{v})}$

c) $\mathsf{s}_i = \ell_i(\mathsf{v})$ and $\tau_i^{\ell_i(\mathsf{v})} <: \tau_i^{\mathsf{t}}$, such that $\ell_i(\top) \leq \mathsf{t}$, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i \cong_{\mathsf{s}} u_i : \tau_i^{\mathsf{t}}$

d) $\mathsf{s}_i = \mathsf{t}$ and $\tau_i^{\mathsf{t}} \leq \tau_i^{\ell_i(\mathsf{v})}$, such that $\mathsf{t} \leq \ell_i(\bot)$, then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v_i \cong_{\mathsf{s}} u_i : \tau_i^{\ell_i(\mathsf{v})}$

Proof By induction on the relation $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}}$, using Lemma 9.

The following couple of lemmas, Lemma 16 and Lemma 17, states that the evaluation of equivalent "observable" closed terms/conditions outputs the same result.

Lemma 16 (Term Evaluation Lemma)
Let S_1, S_2 be stores and s a security level such that
$\Delta \vdash S_1$, $\Delta \vdash S_2$, $S_1 =_{\mathsf{s}} S_2$, $\{r_1, \ldots, r_n\} \in S_1$, and $\{r'_1, \ldots, r'_n\} \in S_2$.
If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} r_i \cong_{\mathsf{s}} r'_i : \tau_r^{\mathsf{s}''}$, and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\},\mathcal{S}_2\{r'_i/x\}} V\{r_i/x\} \cong_{\mathsf{s}} V'\{r'_i/x\}{:}\tau^{\mathsf{s}'}$, such that
$fv(V\{r_i/x\}) \cup fv(V'\{r'_i/x\}) = \emptyset$, and $\mathsf{s}' \leq \mathsf{s}$.
Then $\mathcal{T}[\![V\{r_i/x\}]\!] = \mathcal{T}[\![V'\{r'_i/x\}]\!]$

Proof By induction on $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\},\mathcal{S}_2\{r'_i/x\}} V\{r_i/x\} \cong_{\mathsf{s}} V'\{r'_i/x\}{:}\tau^{\mathsf{s}'}$.

For instance, for **Case (E-FIELD)** we have:

$\Delta \vdash S_1$ (1) - hyp
$\Delta \vdash S_2$ (2) - hyp
$S_1 =_{\mathsf{s}} S_2$ (3) - hyp
$\{r_1, \ldots, r_n\} \in S_1$ (4) - hyp
$\{r'_1, \ldots, r'_n\} \in S_2$ (5) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} r_i \cong_{\mathsf{s}} r'_i : \tau_r^{\mathsf{s}''}$ (6) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\},\mathcal{S}_2\{r'_i/x\}} (e_1.m_i)\{r_i/x\} \cong_{\mathsf{s}} (e_2.m_i)\{r'_i/x\} : \tau_i^{\mathsf{s}_i}$ (7) - hyp
$\mathsf{s}_i \leq \mathsf{s}$ (8) - hyp
$fv((e_1.m_i)\{r_i/x\}) \cup fv((e_2.m_i)\{r'_i/x\}) = \emptyset$ (9) - hyp
$(e.m)\{v/x\} = e\{v/x\}.m$ (10) - by def. of substitution with (7)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\}_1,\mathcal{S}_2\{r'_i/x\}} e_1\{r_i/x\}.m_i \cong_{\mathsf{s}} e_2\{r'_i/x\}.m_i{:}\tau_i^{\mathsf{s}_i}$ (11) - by (7,10)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\}_1,\mathcal{S}_2\{r'_i/x\}} e_1\{r_i/x\} \cong_{\mathsf{s}} e_2\{r'_i/x\}{:}\Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}$ (12) - by inv. of (E-FIELD) with (11)
$\mathcal{T}[\![e_1\{r_i/x\}]\!] = \mathcal{T}[\![e_2\{r'_i/x\}]\!]$ (13) - by I.H. with (1,2,3,4,5,6,12,8,9)
$\mathcal{T}[\![V.m]\!] = field(\mathcal{T}[\![V]\!], m)$ with $field([\ldots, m : v, \ldots], m) = v$ (14) - by Def. 2
$field(\mathcal{T}[\![e_1\{r_i/x\}]\!], m) = field(\mathcal{T}[\![e_2\{r'_i/x\}]\!], m)$ (15) - by (13,14)
$\mathcal{T}[\![e_1\{r_i/x\}.m]\!] = \mathcal{T}[\![e_2\{r'_i/x\}.m]\!]$ by (14,15)

□

Lemma 17 (Condition Evaluation Lemma)
Let $\Delta \vdash S_1$, $\Delta \vdash S_2$, $S_1 =_{\mathsf{s}} S_2$, $\{r_1, \ldots, r_n\} \in S_1$, and $\{r'_1, \ldots, r'_n\} \in S_2$.
If $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} r_i \cong_{\mathsf{s}} r'_i : \tau_r^{\mathsf{s}''}$, and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{r_i/x\},\mathcal{S}_2\{r'_i/x\}} c\{r_i/x\} \cong_{\mathsf{s}} c'\{r'_i/x\} : \texttt{Bool}^{\mathsf{s}'}$, such that
$fv(c\{r_i/x\}) \cup fv(c'\{r'_i/x\}) = \emptyset$, and $\mathsf{s}' \leq \mathsf{s}$.
Then $\mathcal{C}[\![c\{r_i/x\}]\!] = \mathcal{C}[\![c'\{r'_i/x\}]\!]$.

Proof By induction on the definition of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{r_i/x\},\mathcal{S}_2\{r'_i/x\}} c\{r_i/x\} \cong_{\mathsf{s}} c'\{r'_i/x\} : \texttt{Bool}^{\mathsf{s}'}$, using Lemma 16.
For instance, for **Case (E-EQUAL)** we have:

$\Delta \vdash S_1$	(1) - hyp
$\Delta \vdash S_2$	(2) - hyp
$S_1 =_{\mathsf{s}} S_2$	(3) - hyp
$\{r_1, \ldots, r_n\} \in S_1$	(4) - hyp
$\{r'_1, \ldots, r'_n\} \in S_2$	(5) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} r_i \cong_{\mathsf{s}} r'_i : \tau_r^{\mathsf{s}''}$	(6) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\}_1,\mathcal{S}_2\{r'_i/x\}} (V_1 = V_2)\{r_i/x\} \cong_{\mathsf{s}} (V'_1 = V'_2)\{r'_i/x\}:\texttt{Bool}^{\mathsf{s}'}$	(7) - hyp
$\mathsf{s}' \leq \mathsf{s}$	(8) - hyp
$fv((V_1 = V_2)\{r_i/x\}) \cup fv((V'_1 = V'_2)\{r'_i/x\}) = \emptyset$	(9) - hyp
$(V_1 = V_2)\{v/x\} = V_1\{v/x\} = V_2\{v/x\}$	(10) - by def. of substitution with (7)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\}_1,\mathcal{S}_2\{r'_i/x\}} V_1\{r_i/x\}=V_2\{r_i/x\} \cong_{\mathsf{s}} V'_1\{r'_i/x\}=V'_2\{r'_i/x\}:\texttt{Bool}^{\mathsf{s}'}$	(11) - by (7,10)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\}_1,\mathcal{S}_2\{r'_i/x\}} V_1\{r_i/x\} \cong_{\mathsf{s}} V'_1\{r'_i/x\} : \tau^{\mathsf{s}'}$	(12) - by inv. of (E-EQUAL) with (11)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}\{r_i/x\}_1,\mathcal{S}_2\{r'_i/x\}} V_2\{r_i/x\} \cong_{\mathsf{s}} V'_2\{r'_i/x\} : \tau^{\mathsf{s}'}$	(13) - by inv. of (E-EQUAL) with (11)
$\mathcal{T}[\![V_1\{r_i/x\}]\!] = \mathcal{T}[\![V'_1\{r'_i/x\}]\!]$	(14) - by Lem. 16 with (1,2,3,4,5,6,12,8,9)
$\mathcal{T}[\![V_2\{r_i/x\}]\!] = \mathcal{T}[\![V'_2\{r'_i/x\}]\!]$	(15) - by Lem. 16 with (1,2,3,4,5,6,13,8,9)
$\mathcal{C}[\![V_1 = V_2]\!] = (\mathcal{T}[\![V_1]\!] = \mathcal{T}[\![V_2]\!])$	(16) - by def. of Def. 2
$\mathcal{C}[\![V_1\{r_i/x\} = V_2\{r_i/x\}]\!] = \mathcal{C}[\![V'_1\{r'_i/x\} = V'_2\{r'_i/x\}]\!]$	by (14,15,16)

□

We have a substitution lemma for our expression equivalence relation, necessary to prove our main lemmas.

Lemma 18 (Substitution Lemma for Expression Equivalence)
Let $\Delta \vdash S_1$, $\Delta \vdash S_2$, and $S_1 =_{\mathsf{s}} S_2$.
If $\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e'{:}\tau^{\mathsf{s}''}$, and $\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2{:}\tau'^{\mathsf{s}'}$.
Then $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v_1/x\},\mathcal{S}_2\{v_2/x\}} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\}{:}(\tau^{\mathsf{s}''})\{v_1/x\}$.

Proof By induction on the definition of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'}$.

Case (E-TRUE), (E-FALSE), (E-UNIT), (E-VAL), (E-VALOPAQUE)

$\Delta, x : \tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''}$	(1) - hyp
$\Delta \vdash S_1$	(2) - hyp

$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (5) - hyp
$e\{v_1/x\} = e$ (6) - by def. of substitution
$e'\{v_2/x\} = e'$ (7) - by def. of substitution
$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : \tau^{\mathsf{s}''}$ (8) - by (1),(6),(7)
$fv(e) = \emptyset$ (9) - by def. of fv for values
$fv(e') = \emptyset$ (10) - by def. of fv for values
$\mathcal{S}_1 = \mathcal{S}_1\{v_1/x\}$ (11) - x is fresh in $\mathcal{S}_1$ by (5)
$\mathcal{S}_2 = \mathcal{S}_2\{v_2/x\}$ (12) - x is fresh in $\mathcal{S}_2$ by (5)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v_1/x\},\mathcal{S}_2\{v_2/x\}} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ by (8,9,10,11,12) and by Def. 19

Case (E-EXPROPAQUE)

$\Delta, x : \tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (5) - hyp
$\Delta, x : \tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1} e_1 : \tau^{\mathsf{s}''}$ (6) - inv. (E-EXPROPAQUE) of (1)
$\Delta, x : \tau^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_2} e_2 : \tau^{\mathsf{s}''}$ (7) - inv. (E-EXPROPAQUE) of (1)
$\mathsf{s} < \mathsf{s}'' \sqcap \mathsf{r}$ (8) - inv. (E-EXPROPAQUE) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v_1/x\}} e_1\{v_1/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ (9) - by **??** with (2,5,6)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2\{v_2/x\}} e_2\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_2/x\}$ (10) - by **??** with (3,5,7)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v_1/x\},\mathcal{S}_2\{v_2/x\}} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ by (E-EXPROPAQUE) with (9,10,8)

Case (E-LET)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{let}\ y = e_1\ \mathbf{in}\ e_2 \ \cong_{\mathsf{s}}\ \mathbf{let}\ y = e'_1\ \mathbf{in}\ e'_2{:}\tau_2^{\mathsf{s}_2}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (4) - hyp
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e'_1 : \tau_1^{\mathsf{s}_1}$ (5) - inv. (E-LET) of (1)
$\Delta, x{:}\tau'^{\mathsf{s}'}, y : \tau_1^{\mathsf{s}_1} \vdash^{\mathsf{r}}_{\mathcal{S}_1\{y\doteq e_1\},\mathcal{S}_2\{y\doteq e'_1\}} e_2 \cong_{\mathsf{s}} e'_2 : \tau_2^{\mathsf{s}_2}$ (6) - inv. (E-LET) of (1)

- $x = y$ (7)
  $(\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v/x\} = \mathbf{let}\ y = e_1\ \mathbf{in}\ e_2$ (8) - by def. of substitution
  $\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v_1/x\} \ \cong_{\mathsf{s}}\ (\mathbf{let}\ y = e'_1\ \mathbf{in}\ e'_2)\{v_2/x\} : \tau_2^{\mathsf{s}_2}$ (9) - by (1),(8)
  $x \notin fv(\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)$ (10) - by def. of fv with (7)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v_1/x\} \ \cong_{\mathsf{s}}\ (\mathbf{let}\ y = e'_1\ \mathbf{in}\ e'_2)\{v_2/x\}{:}\tau_2^{\mathsf{s}_2}$ by (9,10)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v_1/x\},\mathcal{S}_2\{v_2/x\}} (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v_1/x\} \cong_{\mathsf{s}}\ (\mathbf{let}\ y = e'_1\ \mathbf{in}\ e'_2)\{v_2/x\} : (\tau_2^{\mathsf{s}_2})\{v_1/x\}$
  x is fresh in $\Delta, \mathcal{S}_1\mathcal{S}_2, v_1, v_2$ by (4)

- $x \neq y$ (11)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_1\{v_1/x\} \cong_{\mathsf{s}} e_1'\{v_2/x\} : (\tau_1^{\mathsf{s}_1})\{v_1/x\}$ (12) - by I.H. with (4,5)

  $\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ (13) - by I.H. with (4,5)

  $\Delta, y{:}\tau_1^{\mathsf{s}_1} \vdash S_1$ (14) - by Def. 22 with (2)

  $\Delta, y{:}\tau_1^{\mathsf{s}_1} \vdash S_2$ (15) - by Def. 22 with (3)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{y\doteq e_1\},\mathcal{S}_2\{y\doteq e_1'\}} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (16) - y is fresh in $\mathcal{S}_1, \mathcal{S}_2, \Delta, v_1, v_2$ by (4)

  $\Delta, y{:}\tau_1^{\mathsf{s}_1} \vdash^{\mathsf{r}}_{\mathcal{S}_1'\{y\doteq e_1\},\mathcal{S}_2'\{y\doteq e_1'\}} e_2\{v_1/x\} \cong_{\mathsf{s}} e_2'\{v_2/x\} : (\tau_2^{\mathsf{s}_2})\{v_1/x\}$ (17) - by I.H. with (6,16)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\{v_1/x\} \cong_{\mathsf{s}} (\mathbf{let}\ y = e_1'\ \mathbf{in}\ e_2')\{v_2/x\} : (\tau_2^{\mathsf{s}_2})\{v_1/x\}$ by rule (E-LET) with (12,17)

Case (E-CASE):

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots) \cong_{\mathsf{s}} \mathbf{case}\ e'(\ldots, m_i \cdot y_i \Rightarrow e_i', \ldots) : \sigma^{\mathsf{q}}$ (1) - hyp.

$\Delta \vdash S_1$ (2) - hyp

$\Delta \vdash S_2$ (3) - hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (4) - hyp

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$ (5) - inv. (E-CASE) of (1)

$\forall_i\ \Delta, x : \tau'^{\mathsf{s}'}, y_i : \tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \sigma^{\mathsf{q}}$ (6) - inv. (E-CASE) of (1)

- $x = y$

  $(\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} = \mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots)$ (7) - by def. of substitution

  $\Delta, x : \tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} (\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} \cong_{\mathsf{s}} (\mathbf{case}\ e'(\ldots, m_i \cdot y_i \Rightarrow e_i', \ldots))\{v/x\} : \sigma^{\mathsf{q}}$ (8) - by (7),(1)

  $x \notin fv(\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))$ (9) - by def. of fv

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} (\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} \cong_{\mathsf{s}} (\mathbf{case}\ e'(\ldots, m_i \cdot y_i \Rightarrow e_i', \ldots))\{v/x\} : \sigma^{\mathsf{q}}$ by (8),(9)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v/x\},\mathcal{S}_2\{v/x\}} (\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} \cong_{\mathsf{s}} (\mathbf{case}\ e'(\ldots, m_i \cdot y_i \Rightarrow e_i', \ldots))\{v/x\} : (\sigma^{\mathsf{q}})\{v/x\}$ x is fresh in $\Delta, \mathcal{S}, \mathsf{s}_2, \tau_2$

- $x \neq y$

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v/x\},\mathcal{S}_2\{v/x\}} e\{v/x\} \cong_{\mathsf{s}} e'\{v/x\} : (\{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}})\{v/x\}$ (10) - I.H. with (2), (3)

  $\forall_i\ \Delta, y : \tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v/x\},\mathcal{S}_2\{v/x\}} e_i\{v/x\} \cong_{\mathsf{s}} e_i'\{v/x\} : (\sigma^{\mathsf{q}})\{v/x\}$ (11) - I.H. with (2), (4)

  $(\mathbf{case}\ e(\ldots, m_i \cdot y_i \Rightarrow e_i, \ldots))\{v/x\} = \mathbf{case}\ e\{v/x\}(\ldots, m_i \cdot y_i \Rightarrow e_i\{v/x\}, \ldots)$ (12) - by def. of substitution

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v/x\},\mathcal{S}_2\{v/x\}} \mathbf{case}\ e\{v/x\}(\ldots, m_i \cdot y_i \Rightarrow e_i\{v/x\}, \ldots) \cong_{\mathsf{s}} \mathbf{case}\ e'\{v/x\}(\ldots, m_i \cdot y_i \Rightarrow e_i'\{v/x\}, \ldots) : (\sigma^{\mathsf{q}})\{v/x\}$ by rule (E-CASE) with (10),(11), and by (12)

Case (E-SUB)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'''}$ (3) - inv. (E-SUB) of (1)

$\tau^{\mathsf{s}'''} <: \tau^{\mathsf{s}''}$ (4) - inv. (E-SUB) of (1)

$\mathsf{r} \leq \mathsf{r}'$ (5) - inv. (E-SUB) of (1)

$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}'_1,\mathcal{S}'_2} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\tau^{\mathsf{s}'''})\{v_1/x\}$ (6) - by I.H. with (2,3)

$\mathcal{S}'_1 = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}'_2 = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1} \tau^{\mathsf{s}''}$ (7) - by Def. 19 with (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'''}$ (8) - by (E-SUB) with (3,5)

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1} \tau^{\mathsf{s}'''}$ (9) - by Def. 19 with (8)

$\Delta, x : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1} \tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'''}$ (10) - by (W-SUBTYPE) with (7,9)

$(\tau^{\mathsf{s}'''})\{v_1/x\} <: (\tau^{\mathsf{s}''})\{v_1/x\}$ (11) - Lem. 11 with (4,10,2)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}'_1,\mathcal{S}'_2} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ by rule (E-SUB) with (5,6,11)

Case (E-REFINERECORD):

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$
(3) - inv. (E-REFINERECORD) of (1)

$\mathcal{S}_1\{y \doteq e\} \models y.m_j \doteq \mathsf{v}$ (4) - inv. (E-REFINERECORD) of (1)

$\mathcal{S}_2\{y \doteq e'\} \models y.m_j \doteq \mathsf{v}$ (5) - inv. (E-REFINERECORD) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}'_1,\mathcal{S}'_2} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}})\{v_1/x\}$
(6) - I.H. with (3), (2)

$\mathcal{S}'_1 = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}'_2 = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)

$\mathcal{S}'_1\{y \doteq e\{v_1/x\}\} \models y.m_j \doteq \mathsf{v}$ (7) - from (4), subst closure of $\doteq$

$\mathcal{S}'_2\{y \doteq e'\{v_2/x\}\} \models y.m_j \doteq \mathsf{v}$ (8) - from (5), subst closure of $\doteq$

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}'_1,\mathcal{S}'_2} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}})\{v_1/x\}$
by rule (E-REFINERECORD) with (6,7,8)

Case (e-unrefineRecord):

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$
(3) - inv. (E-UNREFINERECORD) of (1)

$\mathcal{S}_1\{y \doteq e\} \models y.m_j \doteq \mathsf{v}$ (4) - inv. (E-UNREFINERECORD) of (1)

$\mathcal{S}_2\{y \doteq e'\} \models y.m_j \doteq \mathsf{v}$ (5) - inv. (E-UNREFINERECORD) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}})\{v_1/x\}$
(6) - I.H. with (3), (2)

$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)

$\mathcal{S}_1'\{y \doteq e\{v_1/x\}\} \models y.m_j \doteq \mathsf{v}$ (7) - from (4), subst closure of $\doteq$

$\mathcal{S}_2'\{y \doteq e'\{v_2/x\}\} \models y.m_j \doteq \mathsf{v}$ (8) - from (5), subst closure of $\doteq$

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}})\{v_1/x\}$
by rule (E-UNREFINERECORD) with (6,7,8)

Case (E-APP)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1(e_2) \;\cong_{\mathsf{s}}\; e_1'(e_2') : \sigma\{v/y\}^{\mathsf{q}\{v/y\}\sqcup\mathsf{t}}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \;\cong_{\mathsf{s}}\; e_1' : (\Pi y{:}\tau^{\mathsf{s}''}.\sigma^{\mathsf{q}})^{\mathsf{t}}$ (3) - inv. (E-APP) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \;\cong_{\mathsf{s}}\; e_2' : \tau^{\mathsf{s}''}$ (4) - inv. (E-APP) of (1)

$\mathcal{S}_1 \cup \{y \doteq e_2\} \models y \doteq \mathsf{v}$ and $\mathcal{S}_2 \cup \{y \doteq e_2'\} \models y \doteq \mathsf{v}$
(5) - inv. (E-APP) of (1), where e_2, e_2' are pure

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_1\{v_1/x\} \;\cong_{\mathsf{s}}\; e_1'\{v_2/x\} : ((\Pi y{:}\tau^{\mathsf{s}''}.\sigma^{\mathsf{q}})^{\mathsf{t}})\{v_1/x\}$ (6) - by I.H. with (2,3)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_2\{v_1/x\} \;\cong_{\mathsf{s}}\; e_2'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ (7) - by I.H. with (2,4)

$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_1/x\}$ by I.H. with (2,3,4)

$\mathcal{S}_1' \cup \{y \doteq e_2\{v_1/x\}\} \models y \doteq \mathsf{v}$ and $\mathcal{S}_2' \cup \{y \doteq e_2'\{v_2/x\}\} \models y \doteq \mathsf{v}$
(8) - by subst closure with (5)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (e_1(e_2))\{v_1/x\} \;\cong_{\mathsf{s}}\; (e_1'(e_2'))\{v_2/x\}{:}(\sigma\{\mathsf{v}/y\}^{\mathsf{q}\{\mathsf{v}/y\}\sqcup\mathsf{t}})\{v_1/x\}$
by rule (E-APP) with (6),(7), (8)

Case (E-IF)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 \;\cong_{\mathsf{s}}\; \textbf{if } c' \textbf{ then } e_1' \textbf{ else } e_2'{:}\tau^{\mathsf{s}''}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c \;\cong_{\mathsf{s}}\; c' : \mathtt{Bool}^{\mathsf{s}''}$ (3) - inv. (E-IF) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}\sqcup\mathsf{s}''}_{\mathcal{S}_1\cup\{c\doteq\mathtt{true}\},\mathcal{S}_2\cup\{c'\doteq\mathtt{true}\}} e_1 \;\cong_{\mathsf{s}}\; e_1' : \tau^{\mathsf{s}''}$ (4) - inv. (E-IF) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}\sqcup\mathsf{s}''}_{\mathcal{S}_1\cup\{c\doteq\mathtt{false}\},\mathcal{S}_2\cup\{c'\doteq\mathtt{false}\}} e_2 \;\cong_{\mathsf{s}}\; e_2' : \tau^{\mathsf{s}''}$ (5) - inv. (E-IF) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} c\{v_1/x\} \cong_{\mathsf{s}} c'\{v_2/x\} : \mathtt{Bool}^{\mathsf{s}''\{v_1/x\}}$ (6) - by I.H. with (2,3)

$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1 \cup \{c \doteq \mathtt{true}\}, \mathcal{S}_2 \cup \{c' \doteq \mathtt{true}\}} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (7) - $vars(c) \cap vars(c')$ are fresh in v_1, v_2 by (2)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r} \sqcup \mathsf{s}''}_{\mathcal{S}_1' \cup \{c\{v_1/x\} \doteq \mathtt{true}\}, \mathcal{S}_2' \cup \{c'\{v_2/x\} \doteq \mathtt{true}\}} e_1\{v_1/x\} \cong_{\mathsf{s}} e_1'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$
(8) - by I.H. with (7,4)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r} \sqcup \mathsf{s}''}_{\mathcal{S}_1' \cup \{c\{v_1/x\} \doteq \mathtt{false}\}, \mathcal{S}_2' \cup \{c'\{v_2/x\} \doteq \mathtt{false}\}} e_2\{v_1/x\} \cong_{\mathsf{s}} e_2'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$
(9) - by I.H. with (7,5)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (\mathbf{if}\ c\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2)\{v_1/x\} \cong_{\mathsf{s}} (\mathbf{if}\ c'\ \mathbf{then}\ e_1'\ \mathbf{else}\ e_2')\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$
by rule (E-IF) with (6,8,9)

Case (E-FIELD)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e.m_i \cong_{\mathsf{s}} e'.m_i : \tau_i^{\mathsf{s}_i}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$ (3) - inv. (E-FIELD) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e\{v_1/x\} \cong_{\mathsf{s}} e'\{v_2/x\} : (\Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}})\{v_1/x\}$ (5) - by I.H. with (2,3)

$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (e.m_i)\{v_1/x\} \cong_{\mathsf{s}} (e'.m_i)\{v_2/x\} : (\tau_i^{\mathsf{s}_i})\{v_1/x\}$ by rule (E-FIELD) with (5,6)

Case (E-RECORD)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} [\ldots, m_i{=}e_i, \ldots] \cong_{\mathsf{s}} [\ldots, m_i{=}e_i', \ldots] : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\forall_i\ \Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau_i^{\mathsf{s}_i}$ (3) - inv. (E-RECORD) of (1)

$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_i\{v_1/x\} \cong_{\mathsf{s}} e_i'\{v_2/x\} : (\tau_i^{\mathsf{s}_i})\{v_1/x\}$ (4) - by I.H. with (2,3)

$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} [\ldots, m_i = e_i, \ldots]\{v_1/x\} \cong_{\mathsf{s}} [\ldots, m_i = e_i', \ldots]\{v_2/x\} : \Sigma[\ldots \times m_i{:}(\tau_i^{\mathsf{s}_i})\{v_1/x\} \times \ldots]^{\mathsf{t}}$
by rule (E-RECORD) with (4)

Case (E-CONS)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1{::}e_2 \cong_{\mathsf{s}} e_1'{::}e_2' : \tau^{*\mathsf{s}''}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : \tau^{\mathsf{s}''}$ (3) - inv. (E-CONS) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_2' : \tau^{*\mathsf{s}''}$ (4) - inv. (E-CONS) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_1\{v_1/x\} \cong_{\mathsf{s}} e_1'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ (5) - by I.H. with (2,3)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_2\{v_1/x\} \cong_{\mathsf{s}} e_2'\{v_2/x\} : (\tau^{*\mathsf{s}''})\{v_1/x\}$ (6) - by I.H. with (2,4)
$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_1/x\}$ by I.H. with (2,3,4)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (e_1::e_2)\{v_1/x\} \cong_{\mathsf{s}} (e_1'::e_2')\{v_2/x\} : (\tau^{*\mathsf{s}''})\{v_1/x\}$ by rule (E-CONS) with (5,6)

Case (E-COLLECTION)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \{e_1,\ldots,e_n\} \cong_{\mathsf{s}} \{e_1',\ldots,e_n'\} : \tau^{*\mathsf{s}''}$ (1) - hyp
$\Delta \vdash S_1$ hyp
$\Delta \vdash S_2$ hyp
$S_1 =_{\mathsf{s}} S_2$ hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp
$\forall_i\ \Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau^{\mathsf{s}''}$ (3) - inv. (E-COLLECTION) of (1)
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_i\{v_1/x\} \cong_{\mathsf{s}} e_i'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ (4) - by I.H. with (2,3)
$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (\{e_1,\ldots,e_n\})\{v_1/x\} \cong_{\mathsf{s}} (\{e_1',\ldots,e_n'\})\{v_2/x\}{:}(\tau^{*\mathsf{s}''})\{v_1/x\}$ by rule (E-COLLECTION) with (4)

Case (E-NOT)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \neg c \cong_{\mathsf{s}} \neg c' : \mathtt{Bool}^{\mathsf{s}''}$ (1) - hyp
$\Delta \vdash S_1$ hyp
$\Delta \vdash S_2$ hyp
$S_1 =_{\mathsf{s}} S_2$ hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c \cong_{\mathsf{s}} c' : \mathtt{Bool}^{\mathsf{s}''}$ (3) - inv. (E-NOT) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} c\{v_1/x\} \cong_{\mathsf{s}} c'\{v_2/x\} : \mathtt{Bool}^{\mathsf{s}''\{v_1/x\}}$ (4) - by I.H. with (2,3)
$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (\neg c)\{v_1/x\} \cong_{\mathsf{s}} (\neg c')\{v_2/x\} : \mathtt{Bool}^{\mathsf{s}''\{v_1/x\}}$ by rule (E-NOT) with (4)

Case (E-OR)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c_1 \vee c_2 \cong_{\mathsf{s}} c_1' \vee c_2' : \mathtt{Bool}^{\mathsf{s}''}$ (1) - hyp
$\Delta \vdash S_1$ hyp
$\Delta \vdash S_2$ hyp
$S_1 =_{\mathsf{s}} S_2$ hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c_1 \cong_{\mathsf{s}} c_1' : \mathtt{Bool}^{\mathsf{s}''}$ (3) - inv. (E-OR) of (1)
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c_2 \cong_{\mathsf{s}} c_2' : \mathtt{Bool}^{\mathsf{s}''}$ (4) - inv. (E-OR) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} c_1\{v_1/x\} \cong_{\mathsf{s}} c_1'\{v_2/x\} : \mathtt{Bool}^{\mathsf{s}''\{v_1/x\}}$ (5) - by I.H. with (2,3)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} c_2\{v_1/x\} \cong_{\mathsf{s}} c_2'\{v_2/x\} : \mathtt{Bool}^{\mathsf{s}''\{v_1/x\}}$ (6) - by I.H. with (2,4)
$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_1/x\}$ by I.H. with (2,3,4)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (c_1 \vee c_2)\{v_1/x\} \cong_{\mathsf{s}} (c_1' \vee c_2')\{v_2/x\} : \mathtt{Bool}^{\mathsf{s}''\{v_1/x\}}$ by rule (E-OR) with (5,6)

Case (E-EQUAL)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} V_1 = V_2 \cong_{\mathsf{s}} V_1' = V_2' : \mathtt{Bool}^{\mathsf{s}''}$ (1) - hyp

$\Delta \vdash S_1$ hyp

$\Delta \vdash S_2$ hyp

$S_1 =_{\mathsf{s}} S_2$ hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (2) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} V_1 \cong_{\mathsf{s}} V_1' : \tau^{\mathsf{s}''}$ (3) - inv. (E-EQUAL) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} V_2 \cong_{\mathsf{s}} V_2' : \tau^{\mathsf{s}''}$ (4) - inv. (E-EQUAL) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} V_1\{v_1/x\} \cong_{\mathsf{s}} V_1'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ (5) - by I.H. with (2,3)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} V_2\{v_1/x\} \cong_{\mathsf{s}} V_2'\{v_2/x\} : (\tau^{\mathsf{s}''})\{v_1/x\}$ (6) - by I.H. with (2,4)

$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3,4)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (V_1 = V_2)\{v_1/x\} \cong_{\mathsf{s}} (V_1' = V_2')\{v_2/x\} : \mathtt{Bool}^{\mathsf{s}''\{v_1/x\}}$ by rule (E-EQUAL) with (5,6)

Case (E-FOREACH)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(e_1,\ e_2,\ y.z.e_3) \cong_{\mathsf{s}} \mathbf{foreach}(e_1',\ e_2',\ y.z..e_3') : \tau''^{\mathsf{s}''}$ (1) - hyp

$\Delta \vdash S_1$ (2) - hyp

$\Delta \vdash S_2$ (3) - hyp

$S_1 =_{\mathsf{s}} S_2$ (4) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (5) - hyp

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_1' : \tau^{*\mathsf{s}''}$ (6) - inv. (E-FOREACH) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_2' : \tau''^{\mathsf{s}''}$ (7) - inv. (E-FOREACH) of (1)

$\Delta, x{:}\tau'^{\mathsf{s}'}, y : \tau^{\mathsf{s}''}, z : \tau''^{\mathsf{s}''} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_3 \cong_{\mathsf{s}} e_3' : \tau''^{\mathsf{s}''}$ (8) - inv. (E-FOREACH) of (1)

- $x \in \{y, z\}$

  $\mathbf{foreach}(e_1,\ e_2,\ y.z.e_3)\{v/x\} = \mathbf{foreach}(e_1,\ e_2,\ y.z.e_3)$ (9) - by def. of substitution

  $\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(e_1,\ e_2,\ y.z.e_3)\{v_1/x\} \cong_{\mathsf{s}} \mathbf{foreach}(e_1',\ e_2',\ y.z.e_3')\{v_2/x\} : \tau''^{\mathsf{s}''}$ (10) - by (1,9)

  $x \notin fv(\mathbf{foreach}(e_1,\ e_2,\ y.z.e_3))$ (11) - by def. of fv

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(e_1,\ e_2,\ y.z.e_3)\{v_1/x\} \cong_{\mathsf{s}} \mathbf{foreach}(e_1',\ e_2',\ y.z..e_3')\{v_2/x\} : \tau''^{\mathsf{s}''}$ by (10,11)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{v_1/x\},\mathcal{S}_2\{v_2/x\}} \mathbf{foreach}(e_1,\ e_2,\ y.z.e_3)\{v_1/x\} \cong_{\mathsf{s}} \mathbf{foreach}(e_1',\ e_2',\ y.z.e_3')\{v_2/x\} : \tau''^{\mathsf{s}''}$ x fresh in $\Delta, \mathcal{S}_1, \mathcal{S}_2$ by (5)

- $x \notin \{y, z\}$

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_1\{v_1/x\} \cong_{\mathsf{s}} e_1'\{v_2/x\} : (\tau^{*\mathsf{s}''})\{v_1/x\}$ (12) - by I.H. with (2,3,4,5,6)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_2\{v_1/x\} \cong_{\mathsf{s}} e_2'\{v_2/x\} : (\tau''^{\mathsf{s}''})\{v_1/x\}$ (13) - by I.H. with (2,3,4,5,7)

  $\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$ by I.H. with (2,3,4,5,6,7)

  $\Delta, y{:}\tau^{\mathsf{s}''}, z{:}\tau''^{\mathsf{s}''} \vdash S_1$ (14) - by Def. 22 with (2)

  $\Delta, y{:}\tau^{\mathsf{s}''}, z{:}\tau''^{\mathsf{s}''} \vdash S_2$ (15) - by Def. 22 with (3)

  $\Delta, y{:}\tau^{\mathsf{s}''}, z{:}\tau''^{\mathsf{s}''} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$ (16) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (5)

$\Delta, y{:}\tau^{\mathsf{s}''}, z{:}\tau'^{\mathsf{s}''} \vdash^{\mathsf{r}}_{S_1',S_2'} e_3\{v_1/x\} \cong_{\mathsf{s}} e_3'\{v_2/x\} : (\tau'^{\mathsf{s}''})\{v_1/x\}$
(17) - by I.H. with (8,14,15,16)

$\Delta \vdash^{\mathsf{r}}_{S_1',S_2'} \mathbf{foreach}(e_1,\ e_2,\ y.z.e_3)\{v_1/x\} \cong_{\mathsf{s}} \mathbf{foreach}(e_1',\ e_2',\ y.z..e_3')\{v_2/x\} : (\tau'^{\mathsf{s}''})\{v_1/x\}$
by rule (E-FOREACH) with (12,13,17)

Case (E-REF)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{S_1,S_2} \mathbf{ref}_{\tau^{\mathsf{t}}}\ e \cong_{\mathsf{s}} \mathbf{ref}_{\tau^{\mathsf{t}}}\ e' : \mathtt{ref}(\tau^{\mathsf{t}})^{\perp}$	(1) - hyp
$\Delta \vdash S_1$	hyp
$\Delta \vdash S_2$	hyp
$S_1 =_{\mathsf{s}} S_2$	hyp
$\Delta \vdash^{\mathsf{r}}_{S_1,S_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$	(2) - hyp
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{S_1,S_2} e \ \cong_{\mathsf{s}}\ e' : \tau^{\mathsf{t}}$	(3) - inv. (E-REF) of (1)
$\Delta \vdash^{\mathsf{r}}_{S_1',S_2'} e\{v_1/x\} \ \cong_{\mathsf{s}}\ e'\{v_2/x\} : (\tau^{\mathsf{t}})\{v_1/x\}$	(4) - by I.H. with (2,3)
$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$	by I.H. with (2,3)
$\mathsf{r} \leq \mathsf{t}$	(5) - inv. (T-ASSIGN) with (1)
$\mathsf{r} \leq \mathsf{t}\{v/x\}$	(6) - by (5) we have $\mathsf{t}\{v/x\} = \mathsf{t}$
$\Delta \vdash^{\mathsf{r}}_{S_1',S_2'} \mathbf{ref}_{\tau^{\mathsf{t}}}\ e\{v_1/x\} \cong_{\mathsf{s}} \mathbf{ref}_{\tau^{\mathsf{t}}}\ e'\{v_2/x\} : \mathtt{ref}(\tau^{\mathsf{t}})^{\perp}$	by rule (E-REF) with (4,6)

Case (E-DEREF)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{S_1,S_2} !e \cong_{\mathsf{s}} !e' : \tau^{\mathsf{t}}$	(1) - hyp
$\Delta \vdash S_1$	hyp
$\Delta \vdash S_2$	hyp
$S_1 =_{\mathsf{s}} S_2$	hyp
$\Delta \vdash^{\mathsf{r}}_{S_1,S_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$	(2) - hyp
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{S_1,S_2} e \ \cong_{\mathsf{s}}\ e' : \mathtt{ref}(\tau^{\mathsf{t}})^{\mathsf{q}}$	(3) - inv. (E-DEREF) of (1)
$\Delta \vdash^{\mathsf{r}}_{S_1',S_2'} e\{v_1/x\} \ \cong_{\mathsf{s}}\ e'\{v_2/x\} : (\mathtt{ref}(\tau^{\mathsf{t}})^{\mathsf{q}})\{v_1/x\}$	(4) - by I.H. with (2,3)
$\mathcal{S}_1' = \mathcal{S}_1\{v_1/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v_2/x\}$	by I.H. with (2,3)
$\mathsf{q} \leq \mathsf{t}$	(5) - inv. (E-DEREF) of (1)
$\mathsf{q}\{v/x\} \leq \mathsf{t}\{v/x\}$	by (5) we have $\mathsf{q}\{v/x\} = \mathsf{q}$ and $\mathsf{t}\{v/x\} = \mathsf{t}$
$\Delta \vdash^{\mathsf{r}}_{S_1',S_2'} !e\{v_1/x\} \cong_{\mathsf{s}} !e'\{v_2/x\} : (\tau^{\mathsf{t}})\{v_1/x\}$	by rule (e-deref) with (4)

Case (E-ASSIGN)

$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{S_1,S_2} e_1 := e_2 \ \cong_{\mathsf{s}}\ e_1' := e_2' : \mathtt{cmd}^{\perp}$	(1) - hyp
$\Delta \vdash S_1$	hyp
$\Delta \vdash S_2$	hyp
$S_1 =_{\mathsf{s}} S_2$	hyp
$\Delta \vdash^{\mathsf{r}}_{S_1,S_2} v_1 \cong_{\mathsf{s}} v_2 : \tau'^{\mathsf{s}'}$	(2) - hyp
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{S_1,S_2} e_1 \ \cong_{\mathsf{s}}\ e_1' : \mathtt{ref}(\tau^{\mathsf{t}})^{\mathsf{q}}$	(3) - inv. (E-ASSIGN) of (1)
$\Delta, x{:}\tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{S_1,S_2} e_2 \ \cong_{\mathsf{s}}\ e_2' : \tau^{\mathsf{t}}$	(4) - inv. (E-ASSIGN) of (1)
$\Delta \vdash^{\mathsf{r}}_{S_1',S_2'} e_1\{v_1/x\} \ \cong_{\mathsf{s}}\ e_1'\{v_2/x\} : (\mathtt{ref}(\tau^{\mathsf{t}})^{\mathsf{q}})\{v_1/x\}$	(5) - by I.H. with (2,3)
$\Delta \vdash^{\mathsf{r}}_{S_1',S_2'} e_2\{v_1/x\} \ \cong_{\mathsf{s}}\ e_2'\{v_2/x\} : (\tau^{\mathsf{t}})\{v_1/x\}$	(6) - by I.H. with (2,4)

$\mathcal{S}_1' = \mathcal{S}_1\{^{v_1}/_x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{^{v_1}/_x\}$ by I.H. with (2,3,4)
$\mathsf{r} \sqcup \mathsf{q} \leq \mathsf{t}$ (7) - inv. (E-ASSIGN) with (1)
$\mathsf{r} \sqcup \mathsf{q}\{^{v}/_x\} \leq \mathsf{t}\{^{v}/_x\}$ (8) - by (7) we have $\mathsf{q}\{^{v}/_x\} = \mathsf{q}$ and $\mathsf{t}\{^{v}/_x\} = \mathsf{t}$
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} (e_1 := e_2)\{^{v_1}/_x\} \cong_{\mathsf{s}} (e_1' := e_2')\{^{v_2}/_x\} : (\mathtt{cmd}^{\perp})\{^{v_1}/_x\}$
by rule (E-ASSIGN) with (5,6)
□

The following lemma is used in the proof of Lemma 20. It states that the resulting store of a reduction of an expression whose security level is above the observational level – and whose effects are also not observable – is equivalent to the initial store under the observational level.

Lemma 19
Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}'}$, $\Delta \vdash S$, and $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$.
If $(S, e) \longrightarrow (S', e')$, then there is Δ' such that $\Delta \subseteq \Delta'$, $\Delta' \vdash S'$ and $S =_{\mathsf{s}} S'$.

Proof By induction on $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}'}$.

Case (T-INJ):
$(S; \#m_i(e)\ \mathbf{as}\ \tau^{\mathsf{s}'}) \longrightarrow (S'; \#m_i(e')\ \mathbf{as}\ \tau^{\mathsf{s}'})$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \#m_i(e)\ \mathbf{as}\ \tau^{\mathsf{s}'} : \tau^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$(S; e) \longrightarrow (S'; e')$ (6) - inv. (VARIANT) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau_i^{\mathsf{s}_i}$ (7) - inv. (T-INJ) of (2)
$\mathsf{s} < \sqcap \mathsf{s}_i \sqcap \mathsf{r}$ (8) - by (W-VARIANT) with (2) and (4)
$\mathsf{s}' \leq \mathsf{s}_i$ (9) - by (8)
$\mathsf{s} < \mathsf{s}_i \sqcap \mathsf{r}$ (10) - by (4,9)
$\Delta \subseteq \Delta'$ by I.H. with (6,7,3,4,10)
$\Delta' \vdash S'$ by I.H. with (6,7,3,4,10)
$S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,4,10)

Case **(T-CASE)**:

- Sub-case **(CASE-LEFT)**:
  $(S; \mathbf{case}\ e(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S'; \mathbf{case}\ e'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots))$ (1) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{case}\ e(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) : \tau^{\mathsf{s}'}$ (2) - hyp
  $\Delta \vdash S$ (3) - hyp
  $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
  $(S; e) \longrightarrow (S'; e')$ (6) - inv. (CASE-LEFT) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}}$ (7) - inv. (T-CASE) of (2)

– $\mathsf{t} < \mathsf{s}'$ (8)
$\{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{t}} <: \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}'}$ (9) - by (S-VARIANT) with (8)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}'}$ (10) - by (T-SUB) with (7,9)
$\Delta \subseteq \Delta'$ by I.H. with (6,10,3,4)
$\Delta' \vdash S'$ by I.H. with (6,10,3,4)
$S =_{\mathsf{s}} S'$ by I.H. with (6,10,3,4)

– $\mathsf{t} \geq \mathsf{s}'$ (11)
$\mathsf{t} > \mathsf{s}$ (12) - by (4,11)
$\Delta \subseteq \Delta'$ by I.H. with (6,7,3,4,12)
$\Delta' \vdash S'$ by I.H. with (6,7,3,4,12)
$S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,4,12)

- Sub-case **(CASE-RIGHT)**:
$(S; \textbf{case}\ \#m_i(v)(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S; e_i\{{}^{v}/_{x_i}\})$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{case}\ \#m_i(v)(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) : \tau^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

Case **(T-FIELD)**:

- Sub-case **(FIELD-LEFT)**:
$(S; e.m_i) \longrightarrow (S'; e'.m_i)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e.m_i : \tau_i^{\mathsf{s}_i}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}_i \sqcap \mathsf{r}$ (4) - hyp
$(S; e) \longrightarrow (S'; e')$ (6) - inv. (FIELD-LEFT) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}$ (7) - inv. (T-FIELD) of (2)

– $\mathsf{s}' < \mathsf{s}_i$ (8)
$\Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'} <: \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}_i}$ (9) - by (S-RECORD) with (8)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}_i}$ (10) - by (T-SUB) with (7,9)
$\Delta \subseteq \Delta'$ by I.H. with (6,10,3,4)
$\Delta' \vdash S'$ by I.H. with (6,10,3,4)
$S =_{\mathsf{s}} S'$ by I.H. with (6,10,3,4)

– $\mathsf{s}' \geq \mathsf{s}_i$ (11)
$\mathsf{s}' > \mathsf{s}$ (12) - by (4,11)
$\Delta \subseteq \Delta'$ by I.H. with (6,7,3,4,12)
$\Delta' \vdash S'$ by I.H. with (6,7,3,4,12)
$S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,4,12)

- Sub-case (FIELD-RIGHT):

$(S; [m_1 = v_1, \ldots, m_n = v_n].m_i) \longrightarrow (S; v_i)$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [m_1 = v_1, \ldots, m_n = v_n].m_i : \tau_i^{\mathsf{s}_i}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{s}_i \sqcap \mathsf{r}$	(4) - hyp
$S =_{\mathsf{s}} S$	by Definition 22

Case (T-RECORD):

$(S; [\ldots, m_i = e, \ldots]) \longrightarrow (S'; [\ldots, m_i = e', \ldots])$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} [\ldots, m_i = e, \ldots] : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{t}}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$	(4) - hyp
$(S; e) \longrightarrow (S'; e')$	(6) - inv. (RECORD) of (1)
$\forall_i\, \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau_i^{\mathsf{s}_i}$	(7) - inv. (T-RECORD) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau_i^{\mathsf{s}_i}$	(8) - by (7)
$\mathsf{s} < \sqcap \lvert \mathsf{s}_j \rvert^{\downarrow} \sqcap \mathsf{r}$	(9) - by (W-RECORD) with (2) and (4)
$\mathsf{t} \leq \mathsf{s}_i$	(10) - by (9)
$\mathsf{s} < \mathsf{s}_i \sqcap \mathsf{r}$	(11) - by (4,10)
$\Delta \subseteq \Delta'$	by I.H. with (6,8,3,4,11)
$\Delta' \vdash S'$	by I.H. with (6,8,3,4,11)
$S =_{\mathsf{s}} S'$	by I.H. with (6,8,3,4,11)

Case (T-REFINERECORD):

$(S; e) \longrightarrow (S'; e')$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$	(4) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$	(6) - inv. (T-REFINERECORD) of (2)
$\Delta \subseteq \Delta'$	by I.H. with (1,6,3,4)
$\Delta' \vdash S'$	by I.H. with (1,6,3,4)
$S =_{\mathsf{s}} S'$	by I.H. with (1,6,3,4)

Case (T-UNREFINERECORD):

$(S; e) \longrightarrow (S'; e')$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$	(4) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$	(6) - inv. (T-UNREFINERECORD) of (2)
$\Delta \subseteq \Delta'$	by I.H. with (1,6,3,4)
$\Delta' \vdash S'$	by I.H. with (1,6,3,4)
$S =_{\mathsf{s}} S'$	by I.H. with (1,6,3,4)

Case (T-COLLECTION):

$(S; \{\ldots, e, \ldots\}) \longrightarrow (S'; \{\ldots, e', \ldots\})$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \{\ldots, e, \ldots\} : \tau^{*\mathsf{s}'}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$	(4) - hyp
$(S; e) \longrightarrow (S'; e')$	(5) - inv. (COLLECTION) of (1)
$\forall_i\, \Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_i : \tau^{\mathsf{s}'}$	(6) - inv. (T-COLLECTION) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}'}$	(7) - by (6)
$\Delta \subseteq \Delta'$	by I.H. with (7,3,4,5)
$\Delta' \vdash S'$	by I.H. with (7,3,4,5)
$S =_{\mathsf{s}} S'$	by I.H. with (7,3,4,5)

Case **(T-LET)**:

- Sub-case **(LET-LEFT)**:

$(S; \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \longrightarrow (S'; \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2)$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau^{\mathsf{s}'}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$	(4) - hyp
$(S; e_1) \longrightarrow (S'; e_1')$	(6) - inv. (LET-LEFT) of (3)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau'^{\mathsf{s}''}$	(7) - inv. (T-LET) of (1)
$\Delta, x : \tau'^{\mathsf{s}''} \vdash^{\mathsf{r}}_{\mathcal{S}\{x \doteq e_1\}} e_2 : \tau^{\mathsf{s}'}$	(8) - inv. (T-LET) of (1)

  - $\mathsf{s}'' < \mathsf{s}'$ (9)
    $\tau'^{\mathsf{s}''} <: \tau'^{\mathsf{s}'}$ (10) - by (S-EXPR) with (9)
    $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau'^{\mathsf{s}'}$ (11) - by (T-SUB) with (7,10)
    $\Delta \subseteq \Delta'$ by I.H. with (6,11,3,4)
    $\Delta' \vdash S'$ by I.H. with (6,11,3,4)
    $S =_{\mathsf{s}} S'$ by I.H. with (6,11,3,4)

  - $\mathsf{s}'' \geq \mathsf{s}'$ (12)
    $\mathsf{s} < \mathsf{s}'' \sqcap \mathsf{r}$ (13) - by (4,12)
    $\Delta \subseteq \Delta'$ by I.H. with (6,7,3,13)
    $\Delta' \vdash S'$ by I.H. with (6,7,3,13)
    $S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,13)

- Sub-case **(LET-RIGHT)**:

$(S; \mathbf{let}\ x = v\ \mathbf{in}\ e_2) \longrightarrow (S; e_2\{^v/_x\})$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau^{\mathsf{s}'}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp

$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

Case **(T-APP)**:

- Sub-case **(APP-LEFT)**:

  $(S; e_1(e_2)) \longrightarrow (S'; e_1'(e_2))$ (1) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1(e_2) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ (2) - hyp
  $\Delta \vdash S$ (3) - hyp
  $\mathsf{s} < (\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}) \sqcap \mathsf{r}$ (4) - hyp
  $(S; e_1) \longrightarrow (S'; e_1')$ (6) - inv. (APP-LEFT) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\Pi x{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{q}})^{\mathsf{t}}$ (7) - inv. (T-APP) of (2)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}'}$ (8) - inv. (T-APP) of (2)

  - $\mathsf{t} < \mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}$ (9)
    $(\Pi x{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{q}})^{\mathsf{t}} <: (\Pi x{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{q}})^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ (10) - by (S-EXPR) with (9)
    $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : (\Pi x{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{q}})^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ (11) - by (T-SUB) with (7,10)
    $\Delta \subseteq \Delta'$ by I.H. with (6,11,3,4)
    $\Delta' \vdash S'$ by I.H. with (6,11,3,4)
    $S =_{\mathsf{s}} S'$ by I.H. with (6,11,3,4)

  - $\mathsf{t} \geq \mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}$ (12)
    $\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$ (13) - by (4,12)
    $\Delta \subseteq \Delta'$ by I.H. with (6,7,3,13)
    $\Delta' \vdash S'$ by I.H. with (6,7,3,13)
    $S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,13)

- Sub-case **(APP-RIGHT)**:

  $(S; (\lambda(x : \tau^{\mathsf{s}''}).e)(e_2)) \longrightarrow (S'; (\lambda(x : \tau^{\mathsf{s}''}).e)(e_2'))$ (3) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \lambda(x : \tau^{\mathsf{s}''}).e)(e_2) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ (1) - hyp
  $\Delta \vdash S$ (3) - hyp
  $\mathsf{s} < (\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}) \sqcap \mathsf{r}$ (4) - hyp
  $(S; e_2) \longrightarrow (S'; e_2')$ (6) - inv. (APP-RIGHT) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} (\lambda(x : \tau^{\mathsf{s}''}).e) : (\Pi x{:}\tau^{\mathsf{s}''}.\sigma^{\mathsf{q}})^{\mathsf{t}}$ (7) - inv. (T-APP) of (2)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}''}$ (8) - inv. (T-APP) of (2)

  - $\mathsf{s}'' < \mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}$ (9)
    $\tau^{\mathsf{s}''} <: \tau^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ (10) - by (S-EXPR) with (9)
    $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{q}\{\mathsf{v}/x\} \sqcup \mathsf{t}}$ (11) - by (T-SUB) with (7,10)
    $\Delta \subseteq \Delta'$ by I.H. with (6,11,3,4)
    $\Delta' \vdash S'$ by I.H. with (6,11,3,4)

$S =_{\mathsf{s}} S'$ by I.H. with (6,11,3,4)

– $\mathsf{s}'' \geq \mathsf{q}\{^{\mathsf{v}}/_{x}\} \sqcup \mathsf{t}$ (12)
$\mathsf{s} < \mathsf{s}'' \sqcap \mathsf{r}$ (13) - by (4,12)
$\Delta \subseteq \Delta'$ by I.H. with (6,7,3,13)
$\Delta' \vdash S'$ by I.H. with (6,7,3,13)
$S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,13)

- Sub-case **(APP)**:

$(S; (\lambda(x : \tau^{\mathsf{s}''}).e)(v)) \longrightarrow (S; e\{^{v}/_{x}\})$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} (\lambda(x : \tau^{\mathsf{s}''}).e)(v) : \sigma^{\mathsf{s}'}$ (1) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

Case **(T-IF)**:

- Sub-case **(IF-TRUE)**:

$(S; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \longrightarrow (S; e_1)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

- Sub-case **(IF-FALSE)**:

$(S; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \longrightarrow (S; e_2)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 : \tau^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

Case (T-CONS):

- Sub-case **(CONS-LEFT)**:

$(S; e_1 :: e_2) \longrightarrow (S'; e_1' :: e_2)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 :: e_2 : \tau^{*\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$(S; e_1) \longrightarrow (S'; e_1')$ (6) - inv. (CONS-LEFT) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{\mathsf{s}'}$ (7) - inv. (T-CONS) of (2)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{*\mathsf{s}'}$ (8) - inv. (T-CONS) of (2)
$\Delta \subseteq \Delta'$ by I.H. with (6,7,3,4)
$\Delta' \vdash S'$ by I.H. with (6,7,3,4)
$S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,4)

- Sub-case **(CONS-RIGHT)**:

$(S; v::e_2) \longrightarrow (S'; v::e'_2)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1::e_2 : \tau^{*\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$(S; e_2) \longrightarrow (S'; e'_2)$ (6) - inv. (CONS-RIGHT) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}'}$ (7) - inv. (T-CONS) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{*\mathsf{s}'}$ (8) - inv. (T-CONS) of (2)
$\Delta \subseteq \Delta'$ by I.H. with (6,8,3,4)
$\Delta' \vdash S'$ by I.H. with (6,8,3,4)
$S =_{\mathsf{s}} S'$ by I.H. with (6,8,3,4)

- Sub-case **(CONS)**:

$(S; v::\{v_1, \ldots, v_n\}) \longrightarrow (S; \{v, v_1, \ldots, v_n\})$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1::e_2 : \tau^{*\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

Case (T-SUB):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}'}$ (1) - hyp
$(S; e) \longrightarrow (S'; e')$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} e : \tau^{\mathsf{s}''}$ (6) - inv. (T-SUB) of (1)
$\tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'}$ and $\mathsf{r} \leq \mathsf{r}'$ (7) - inv. (T-SUB) of (1)
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}'$ (8) - by (4,7)

- $\mathsf{s}'' > \mathsf{s}$ (9)
$\Delta \subseteq \Delta'$ by I.H. with (6,2,3,8,9)
$\Delta' \vdash S'$ by I.H. with (6,2,3,8,9)
$S =_{\mathsf{s}} S'$ by I.H. with (6,2,3,8,9)

- $\mathsf{s}'' \leq \mathsf{s}$ (10)
$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}} e : \tau^{\mathsf{s}'}$ (11) - by (T-SUB) with (6,7,10)

$\Delta \subseteq \Delta'$ by I.H. with (11,2,3,8)
$\Delta' \vdash S'$ by I.H. with (11,2,3,8)
$S =_{\mathsf{s}} S'$ by I.H. with (11,2,3,8)

Case **(T-FOREACH)**:

- Sub-case **(FOREACH-LEFT)**:

  $(S; \mathbf{foreach}(e_1, e_2, x.y.e_3)) \longrightarrow (S'; \mathbf{foreach}(e_1', e_2, x.y.e_3))$ (1) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, x.y.e_3)) : \tau'^{\mathsf{s}'}$ (2) - hyp
  $\Delta \vdash S$ (3) - hyp
  $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
  $(S; e_1) \longrightarrow (S'; e_1')$ (6) - inv. (FOREACH-LEFT) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \tau^{*\mathsf{s}'}$ (7) - inv. (T-FOREACH) of (2)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau'^{\mathsf{s}'}$ (8) - inv. (T-FOREACH) of (2)
  $\Delta, x : \tau^{\mathsf{s}'}, y : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_3 : \tau'^{\mathsf{s}'}$ (9) - inv. (T-FOREACH) of (2)
  $\Delta \subseteq \Delta'$ by I.H. with (6,7,3,4)
  $\Delta' \vdash S'$ by I.H. with (6,7,3,4)
  $S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,4)

- Sub-case **(FOREACH-RIGHT)**:

  $(S; \mathbf{foreach}(v, e_2, x.y.e_3))) \longrightarrow (S'; \mathbf{foreach}(v, e_2', x.y.e_3)))$ (1) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, x.y.e_3)) : \tau'^{\mathsf{s}'}$ (2) - hyp
  $\Delta \vdash S$ (3) - hyp
  $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
  $(S; e_2) \longrightarrow (S'; e_2')$ (6) - inv. (FOREACH-RIGHT) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{*\mathsf{s}'}$ (7) - inv. (T-FOREACH) of (2)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau'^{\mathsf{s}'}$ (9) - inv. (T-FOREACH) of (2)
  $\Delta, x : \tau^{\mathsf{s}'}, y : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}} e_3 : \tau'^{\mathsf{s}'}$ (7) - inv. (T-FOREACH) of (2)
  $\Delta \subseteq \Delta'$ by I.H. with (6,8,3,4)
  $\Delta' \vdash S'$ by I.H. with (6,8,3,4)
  $S =_{\mathsf{s}} S'$ by I.H. with (6,8,3,4)

- Sub-case **(FOREACH)**:

  $(S; \mathbf{foreach}(l, v, x.y.e_3)) \longrightarrow (S; \mathbf{foreach}(hs, e_3\{^{h}/_{x}\}\{^{v}/_{y}\}, x.y.e_3))$ (1) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, x.y.e_3)) : \tau'^{\mathsf{s}'}$ (2) - hyp
  $\Delta \vdash S$ (3) - hyp

$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

- Sub-case **(FOREACH-BASE)**:

$(S; \mathbf{foreach}(\{\}, v, x.y.e_3)) \longrightarrow (S; v)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{foreach}(e_1, e_2, x.y.e_3)) : \tau'^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

Case **(T-REF)**:

- Sub-case **(REF-LEFT)**:

$(S; \mathbf{ref}_{\tau^{\mathsf{s}}}\ e) \longrightarrow (S'; \mathbf{ref}_{\tau^{\mathsf{s}}}\ e')$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{ref}_{\tau^{\mathsf{s}}}\ e : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (1) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$ (4) - hyp
$(S; e) \longrightarrow (S'; e')$ (6) - inv. (REF-LEFT) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{s}'}$ (7) - inv. (T-REF) of (1)

- $\mathsf{s}' < \mathsf{t}$ (8)
  $\tau^{\mathsf{s}'} <: \tau'^{\mathsf{t}}$ (9) - by (S-EXPR) with (8)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \tau^{\mathsf{t}}$ (10) - by (T-SUB) with (7,)
  $\Delta \subseteq \Delta'$ by I.H. with (6,10,3,4)
  $\Delta' \vdash S'$ by I.H. with (6,10,3,4)
  $S =_{\mathsf{s}} S'$ by I.H. with (6,10,3,4)

- $\mathsf{s}' \geq \mathsf{t}$ (11)
  $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (12) - by (4,11)
  $\Delta \subseteq \Delta'$ by I.H. with (6,7,3,12)
  $\Delta' \vdash S'$ by I.H. with (6,7,3,12)
  $S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,12)

- Sub-case **(REF-RIGHT)**:

$(S; \mathbf{ref}_{\tau^{\mathsf{s}'}}\ v) \longrightarrow (S \cup \{l \mapsto v\};\ l)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} \mathbf{ref}_{\tau^{\mathsf{s}'}}\ v : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$ (4) - hyp
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (6) - Theorem 6 with (1,2,3)

$\Delta \subseteq \Delta'$ (7) - Theorem 6 with (1,2,3)
$\Delta' \vdash S \cup \{l \mapsto v\}$ (8) - Theorem 6 with (1,2,3)
$S =_{\mathsf{s}} S \cup \{l \mapsto v\}$ by Definition 22 with (4,6,8)

Case **(T-DEREF)**:

- Sub-case **(DEREF-LEFT)**:

$(S; !e) \longrightarrow (S'; !e')$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} !e : \tau^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$(S; e) \longrightarrow (S'; e')$ (6) - inv. (DEREF-LEFT) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (7) - inv. (T-DEREF) of (2)

  - $\mathsf{t} < \mathsf{s}'$ (8)
    $\mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}} <: \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{s}'}$ (9) - by (S-EXPR) with (8)
    $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{s}'}$ (10) - by (T-SUB) with (7)
    $\Delta \subseteq \Delta'$ by I.H. with (6,10,3,4)
    $\Delta' \vdash S'$ by I.H. with (6,10,3,4)
    $S =_{\mathsf{s}} S'$ by I.H. with (6,10,3,4)

  - $\mathsf{t} \geq \mathsf{s}'$ (11)
    $\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$ (12) - by (4,11)
    $\Delta \subseteq \Delta'$ by I.H. with (6,7,3,12)
    $\Delta' \vdash S'$ by I.H. with (6,7,3,12)
    $S =_{\mathsf{s}} S'$ by I.H. with (6,7,3,12)

- Sub-case **(DEREF)**:

$(S; !l) \longrightarrow (S; v)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} !e : \tau^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$S =_{\mathsf{s}} S$ by Definition 22

Case **(T-ASSIGN)**:

- Sub-case **(ASSIGN-LEFT)**:

$(S; e_1 := e_2) \longrightarrow (S'; e_1' := e_2)$ (1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 := e_2 : \mathtt{cmd}^{\mathsf{s}'}$ (2) - hyp
$\Delta \vdash S$ (3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (4) - hyp
$(S; e_1) \longrightarrow (S'; e_1')$ (6) - inv. (ASSIGN-LEFT) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_1 : \mathtt{ref}(\tau^{\mathsf{s}''})^{\mathsf{t}}$	(7) - inv. (T-ASSIGN) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}''}$	(8) - inv. (T-ASSIGN) of (2)
$\mathsf{r} \sqcup \mathsf{t} \leq \mathsf{s}''$	(9) - inv. (T-ASSIGN) of (2)
$\mathsf{r} \leq \mathsf{t}$	(10) - inv. (T-REF) of (7)
$\mathsf{s} < \mathsf{t} \sqcap \mathsf{r}$	(11) - by (4,10)
$\Delta \subseteq \Delta'$	by I.H. with (6,7,11)
$\Delta' \vdash S'$	by I.H. with (6,7,11)
$S =_{\mathsf{s}} S'$	by I.H. with (6,7,11)

- Sub-case **(ASSIGN-RIGHT)**:

$(S; l := e_2) \longrightarrow (S'; l := e_2')$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} l := e_2 : \mathtt{cmd}^{\mathsf{s}'}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$	(4) - hyp
$(S; e_2) \longrightarrow (S'; e_2')$	(6) - inv. (ASSIGN-RIGHT) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}''})^{\mathsf{t}}$	(7) - inv. (T-ASSIGN) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e_2 : \tau^{\mathsf{s}''}$	(8) - inv. (T-ASSIGN) of (2)
$\mathsf{r} \sqcup \mathsf{t} \leq \mathsf{s}''$	(9) - inv. (T-ASSIGN) of (2)
$\mathsf{s} < \mathsf{r} \leq \mathsf{s}''$	(10) - by (4,9)
$\mathsf{s} < \mathsf{s}''$	(11) - by (10)
$\mathsf{s} < \mathsf{s}'' \sqcap \mathsf{r}$	(12) - by (4,11)
$\Delta \subseteq \Delta'$	by I.H. with (7,3,12)
$\Delta' \vdash S'$	by I.H. with (7,3,12)
$S =_{\mathsf{s}} S'$	by I.H. with (7,3,12)

- Sub-case **(ASSIGN)**:

$(S; l := v) \longrightarrow (S[l \mapsto v]; ())$	(1) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} l := v : \mathtt{cmd}^{\mathsf{s}'}$	(2) - hyp
$\Delta \vdash S$	(3) - hyp
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$	(4) - hyp
$l \in dom(S)$	(6) - inv. (ASSIGN) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} l : \mathtt{ref}(\tau^{\mathsf{s}''})^{\mathsf{t}}$	(7) - inv. (T-ASSIGN) of (2)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} v : \tau^{\mathsf{s}''}$	(8) - inv. (T-ASSIGN) of (2)
$\mathsf{r} \sqcup \mathsf{t} \leq \mathsf{s}''$	(9) - inv. (T-ASSIGN) of (2)
$\Delta(l) = \tau^{\mathsf{s}''}$	(10) - by (T-LOC) with (2)
$\Delta \vdash S[l \mapsto v]$	by (7,8,10)
$\mathsf{s} < \mathsf{r} \leq \mathsf{s}''$	(11) - by (4,9)
$\mathsf{s} < \mathsf{s}''$	(12) - by (11)
$S =_{\mathsf{s}} S'$	Definition 22 with (12)

□

We now prove the first of our main lemmas for the noninterference theorem.

Lemma 20 (Non-interference Step)

Let $fv(e_1) \cup fv(e_2) \cup fv(\tau^{s'}) = \emptyset$, $vars(\Delta) = \emptyset$,
$\Delta \vdash S_1$, $\Delta \vdash S_2$, $S_1 =_s S_2$, $\Delta \vdash^r_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_s e_2{:}\tau^{s'}$.
If $(S_1, e_1) \longrightarrow (S_1', e_1')$, and $(S_2, e_2) \longrightarrow (S_2', e_2')$, then there is Δ' such that
$\Delta \subseteq \Delta'$, $\Delta' \vdash S_1'$, $\Delta' \vdash S_2'$, $S_1' =_s S_2'$, $\Delta' \vdash^r_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_s e_2'{:}\tau^{s'}$.

Proof By induction on the relation $\Delta \vdash^r_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_s e_2{:}\tau^{s'}$.

Case (E-EXPROPAQUE):

$\Delta \vdash^r_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_s e_2 : \tau^{s'}$	(1) - hyp
$\Delta \vdash S_1$	(2) - hyp
$\Delta \vdash S_2$	(3) - hyp
$S_1 =_s S_2$	(4) - hyp
$fv(e_1.m_i) \cup fv(e_2.m_i) \cup fv(\tau_i^{s_i}) = \emptyset$ and $vars(\Delta) = \emptyset$	(5) - hyp
$(S_1; e_1) \longrightarrow (S_1'; e_1')$	(6) - hyp
$(S_2; e_2) \longrightarrow (S_2'; e_2')$	(7) - hyp
$\Delta \vdash^r_{\mathcal{S}_1} e_1 : \tau^{s'}$	(8) - inv. (E-EXPROPAQUE) of (1)
$\Delta \vdash^r_{\mathcal{S}_2} e_2 : \tau^{s'}$	(9) - inv. (E-EXPROPAQUE) of (1)
$s < s' \sqcap r$	(10) - inv. (E-EXPROPAQUE) of (1)
$\Delta \subseteq \Delta'$	by Theorem 6 with (2,6,8)
$\Delta' \vdash^r_{\mathcal{S}_1} e_1' : \tau^{s'}$	(12) - by Theorem 6 with (2,6,8)
$\Delta' \vdash^r_{\mathcal{S}_2} e_2' : \tau^{s'}$	(13) - by Theorem 6 with (3,7,9)
$\Delta' \vdash S_1'$	(14) - by Theorem 6 with (2,6,8)
$\Delta' \vdash S_2'$	(15) - by Theorem 6 with (3,7,9)
$\Delta' \vdash^r_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_s e_2' : \tau^{s'}$	by (E-EXPROPAQUE) with (12,13,10)
$S_1 =_s S_1'$	(16) - by Lemma 19 with (2,6,8,10)
$S_2 =_s S_2'$	(17) - by Lemma 19 with (3,7,9,10)
$S_1' =_s S_2'$	by (4,16,17)

Case (E-FIELD):

$\Delta \vdash^r_{\mathcal{S}_1,\mathcal{S}_2} e_1.m_i \cong_s e_2.m_i : \tau_i^{s_i}$	(1) - hyp
$\Delta \vdash S_1$	(2) - hyp
$\Delta \vdash S_2$	(3) - hyp
$S_1 =_s S_2$	(4) - hyp
$fv(e_1.m_i) \cup fv(e_2.m_i) \cup fv(\tau_i^{s_i}) = \emptyset$ and $vars(\Delta) = \emptyset$	(5) - hyp

- Sub-case **(FIELD-LEFT)**:

  $(S_1; e_1.m_i) \longrightarrow (S_1'; e_1'.m_i)$ (6) - hyp

  $(S_2; e_2.m_i) \longrightarrow (S_2'; e_2'.m_i)$ (7) - hyp

  $(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (FIELD-LEFT) of (6)

  $(S_2; e_2) \longrightarrow (S_2'; e_2')$ (9) - inv. (FIELD-LEFT) of (7)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}$ (10) - inv. (E-FIELD) of (1)

  $fv(e_1) \cup fv(e_2) \cup fv(\Sigma[\ldots \times m_i{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}) = \emptyset$ and $vars(\Delta) = \emptyset$ (11) - by def. of fv and by (W-RECORD) with (5,10)

  $\Delta \subseteq \Delta'$ by I.H. with (8,9,10,11)

  $\Delta' \vdash S_1'$ by I.H. with (8,9,10,11)

  $\Delta' \vdash S_2'$ by I.H. with (8,9,10,11)

  $S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,10,11)

  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \Sigma[\ldots \times m_i : \tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}$ (12) - by I.H. with (8,9,10,11)

  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1'.m \cong_{\mathsf{s}} e_2'.m : \tau_i^{\mathsf{s}_i}$ by rule (E-FIELD) with (12)

- Sub-case **(FIELD-RIGHT)**:

  $(S_1; [m_1{=}v_1, \ldots, m_n{=}v_n].m_i) \longrightarrow (S_1; v_i)$ (6) - hyp

  $(S_2; [m_1{=}v_1', \ldots, m_n{=}v_n'].m_i) \longrightarrow (S_2; v_i')$ (7) - hyp

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} [m_1{=}v_1, \ldots, m_n{=}v_n] \cong_{\mathsf{s}} [m_1{=}v_1', \ldots, m_n{=}v_n'] : \Sigma[\ldots \times m_1{:}\tau_i^{\mathsf{s}_i} \times \ldots]^{\mathsf{s}'}$ (8) - inv. (E-FIELD) of (1)

  $\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_i \cong_{\mathsf{s}} v_i' : \tau_i^{\mathsf{s}_i}$ (10) - by Lem. 15 with (8,1) and by (7)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_i \cong_{\mathsf{s}} v_i' : \tau_i^{\mathsf{s}_i}$ by (10)

Case (E-RECORD):

$\Delta \vdash S_1$ (1) - hyp

$\Delta \vdash S_2$ (2) - hyp

$S_1 =_{\mathsf{s}} S_2$ (3) - hyp

$(S_1; [\ldots, m_k{=}e_1, \ldots]) \longrightarrow (S_1'; [\ldots, m_k{=}e_1', \ldots])$ (4) - hyp

$(S_2; [\ldots, m_k{=}e_2, \ldots]) \longrightarrow (S_2'; [\ldots, m_k{=}e_2', \ldots])$ (5) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} [\ldots, m_k{=}e_1, \ldots] \cong_{\mathsf{s}} [\ldots, m_k{=}e_2, \ldots] : \Sigma[\ldots \times m_k{:}\tau_k^{\mathsf{s}_k} \times \ldots]^{\mathsf{t}}$ (6) - hyp, where $k \in \{1, \ldots, n\}$

$fv([\ldots, m_k{=}e_1, \ldots]) \cup fv([\ldots, m_k{=}e_2, \ldots]) \cup fv(\Sigma[\ldots \times m_k{:}\tau_k^{\mathsf{s}_k} \times \ldots]^{\mathsf{t}}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp

$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (RECORD) of (4)

$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (9) - inv. (RECORD) of (5)

$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau_i^{\mathsf{s}_i}$ (10) - inv. (E-RECORD) of (6)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau_k^{\mathsf{s}_k}$ (11) - by (10)

$fv(e_1) \cup fv(e_2) \cup fv(\tau_k^{\mathsf{s}_k}) = \emptyset$ and $vars(\Delta) = \emptyset$ (12) - by def. of fv with (7,11)

$\Delta \subseteq \Delta'$ by I.H. with (8,9,11,12)

$\Delta' \vdash S_1'$ by I.H. with (8,9,11,12)

$\Delta' \vdash S_2'$ by I.H. with (8,9,11,12)
$S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,11,12)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau_k^{\mathsf{s}_k}$ (13) - by I.H. with (8,9,11,12)
$\forall_i\ \Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau_i^{\mathsf{s}_i}$ (14) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} [\ldots, m_k{=}e_1', \ldots] \cong_{\mathsf{s}} [\ldots, m_k{=}e_2', \ldots] : \Sigma[\ldots \times m_k{:}\tau_k^{\mathsf{s}_k} \times \ldots]^{\mathsf{t}}$
by rule (E-RECORD) with (13,14)

Case (E-CASE):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{case}\ e_1(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) \cong_{\mathsf{s}} \mathbf{case}\ e_2(\ldots, m_i \cdot x_i \Rightarrow e_i', \ldots) : \sigma^{\mathsf{t}}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp
$fv(\mathbf{case}\ e_1(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \cup fv(\mathbf{case}\ e_2(\ldots, m_i \cdot x_i \Rightarrow e_i', \ldots)) \cup fv(\sigma^{\mathsf{t}}) = \emptyset$
and $vars(\Delta) = \emptyset$ (5) - hyp

- Sub-case **(CASE-LEFT)**:
  $(S_1; \mathbf{case}\ e_1(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S_1'; \mathbf{case}\ e_1'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots))$ (6) - hyp
  $(S_2; \mathbf{case}\ e_2(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S_2'; \mathbf{case}\ e_2'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots))$ (7) - hyp
  $(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (CASE-LEFT) of (6)
  $(S_2; e_2) \longrightarrow (S_2'; e_2')$ (9) - inv. (CASE-LEFT) of (7)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}'}$ (10) - inv. (E-CASE) of (1)
  $fv(e_1) \cup fv(e_2) \cup fv(\{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}'}) = \emptyset$ and $vars(\Delta) = \emptyset$
  (11) - by def. of fv and by (W-VARIANT) with (5,10)
  $\Delta \subseteq \Delta'$ by I.H. with (8,9,10,11)
  $\Delta' \vdash S_1'$ by I.H. with (8,9,10,11)
  $\Delta' \vdash S_2'$ by I.H. with (8,9,10,11)
  $S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,10,11)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \{\ldots, m_i : \tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}'}$ (12) - by I.H. with (8,9,10,11)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{case}\ e_1'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) \cong_{\mathsf{s}} \mathbf{case}\ e_2'(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots) : \sigma^{\mathsf{t}}$
  by rule (E-CASE) with (12)

- Sub-case **(CASE-RIGHT)**:
  $(S_1; \mathbf{case}\ \#m_i(v)(\ldots, m_i \cdot x_i \Rightarrow e_i, \ldots)) \longrightarrow (S_1; e_i\{v/x_i\})$ (6) - hyp
  $(S_2; \mathbf{case}\ \#m_i(u)(\ldots, m_i \cdot x_i \Rightarrow e_i') \longrightarrow (S_2; e_i'\{u/x_i\})$ (7) - hyp
  $\Delta, x_i : \tau_i^{\mathsf{s}_i} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \sigma^{\mathsf{t}}$ (8) - inv. (E-CASE) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \#m_i(v) \cong_{\mathsf{s}} \#m_i(u) : \{\ldots, m_i{:}\tau_i^{\mathsf{s}_i}, \ldots\}^{\mathsf{s}'}$ (9) - inv. (E-CASE) of (1)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} u : \tau_i'^{\mathsf{s}_i'}$ (10) - by Lem. 15 with (9,1) and by (7)
  $\tau_i'^{\mathsf{s}_i'} <: \tau_i^{\mathsf{s}_i}$ (11) - by Lem. 15 with (9,1) and by (7)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} u : \tau_i^{\mathsf{s}_i}$ (12) - by (E-SUB) (10,11)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} e_i\{v/x_i\} \cong_{\mathsf{s}} e_i'\{u/x_i\} : (\sigma^{\mathsf{t}})\{v/x_i\}$ (13) - Lem. 18 with (8,12)
  $\mathcal{S}_1' = \mathcal{S}_1\{v/x_i\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{u/x_i\}$ Lem. 18 with (8,12)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i\{^v/_{x_i}\} \cong_{\mathsf{s}} e_i'\{^u/_{x_i}\} : (\sigma^{\mathsf{t}})\{^v/_{x_i}\}$ x fresh in $\Delta, \mathcal{S}_1, \mathcal{S}_2$ by (10) with (13)

Case (E-INJ):

$\Delta \vdash S_1$ (1) - hyp
$\Delta \vdash S_2$ (2) - hyp
$S_1 =_{\mathsf{s}} S_2$ (3) - hyp
$(S_1; \#m(e_1)) \longrightarrow (S_1'; \#m(e_1'))$ (4) - hyp
$(S_2; \#m(e_2)) \longrightarrow (S_2'; \#m(e_2'))$ (5) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \#m(e_1) \cong_{\mathsf{s}} \#m(e_2) : \tau^{\mathsf{t}}$ (6) - hyp
$fv(\#m(e_1)) \cup fv(\#m(e_2)) \cup fv(\tau^{\mathsf{t}}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp
$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (VARIANT) of (4)
$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (9) - inv. (VARIANT) of (5)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau_i^{\mathsf{s}_i}$ (10) - inv. (E-VARIANT) of (6)
$fv(e_1) \cup fv(e_2) \cup fv(\tau_i^{\mathsf{s}_i}) = \emptyset$ and $vars(\Delta) = \emptyset$ (11) - by def. of fv with (7,10)
$\Delta \subseteq \Delta'$ by I.H. with (8,9,10,11)
$\Delta' \vdash S_1'$ by I.H. with (8,9,10,11)
$\Delta' \vdash S_2'$ by I.H. with (8,9,10,11)
$S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,10,11)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau_i^{\mathsf{s}_i}$ (12) - by I.H. with (8,9,10,11)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} \#m(e_1') \cong_{\mathsf{s}} \#m(e_2') : \tau^{\mathsf{t}}$ by rule (E-VARIANT) with (12)

Case (E-REFINERECORD):

$\Delta \vdash S_1$ (1) - hyp
$\Delta \vdash S_2$ (2) - hyp
$S_1 =_{\mathsf{s}} S_2$ (3) - hyp
$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (4) - hyp
$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (5) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (6) - hyp
$fv(e_1) \cup fv(e_2) \cup fv(\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (8) - inv. (E-REFINERECORD) of (6)
$\mathcal{S}_1\{x \doteq e_1\} \models x.m_j \doteq \mathsf{v}$ and $\mathcal{S}_2\{x \doteq e_2\} \models x.m_j \doteq \mathsf{v}$ (9) - inv. (E-REFINERECORD) of (6)
$fv(\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}) = \emptyset$ (10) - by (7)
$\Delta \subseteq \Delta'$ I.H. with (4,5,8,10)
$\Delta' \vdash S'$ I.H. with (4,5,8,10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (11) - I.H. with (4,5,8,10)
$\mathcal{S}_1\{x \doteq e_1'\} \models x.m_j \doteq \mathsf{v}$ (12) - by (4) since reduction preserves $\doteq$, so $e_1 \doteq e_1'$
$\mathcal{S}_2\{x \doteq e_2'\} \models x.m_j \doteq \mathsf{v}$ (13) - by (5) since reduction preserves $\doteq$, so $e_2 \doteq e_2'$
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ by rule (E-REFINERECORD) with (11,12,13)

Case (E-UNREFINERECORD):

$\Delta \vdash S_1$ (1) - hyp

$\Delta \vdash S_2$ (2) - hyp

$S_1 =_{\mathsf{s}} S_2$ (3) - hyp

$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (4) - hyp

$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (5) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (6) - hyp

$fv(e_1) \cup fv(e_2) \cup fv(\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (8) - inv. (E-UNREFINERECORD) of (6)

$\mathcal{S}_1\{x \doteq e_1\} \models x.m_j \doteq \mathsf{v}$ and $\mathcal{S}_2\{x \doteq e_2\} \models x.m_j \doteq \mathsf{v}$ (9) - inv. (E-UNREFINERECORD) of (6)

$fv(\Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}) = \emptyset$ (10) - by (7)

$\Delta \subseteq \Delta'$ I.H. with (4,5,8,10)

$\Delta' \vdash S'$ I.H. with (4,5,8,10)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (11) - I.H. with (4,5,8,10)

$\mathcal{S}_1\{x \doteq e_1'\} \models x.m_j \doteq \mathsf{v}$ (12) - by (4) since reduction preserves $\doteq$, so $e_1 \doteq e_1'$

$\mathcal{S}_2\{x \doteq e_2'\} \models x.m_j \doteq \mathsf{v}$ (13) - by (5) since reduction preserves $\doteq$, so $e_2 \doteq e_2'$

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ by rule (E-UNREFINERECORD) with (11,12,13)

Case (E-SUB):

$\Delta \vdash S_1$ (1) - hyp

$\Delta \vdash S_2$ (2) - hyp

$S_1 =_{\mathsf{s}} S_2$ (3) - hyp

$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (4) - hyp

$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (5) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$ (6) - hyp

$fv(e_1) \cup fv(e_2) \cup fv(\tau^{\mathsf{s}'}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp

$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}''}$ (8) - inv. (E-SUB) of (6)

$\tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'}$ and $\mathsf{r} \leq \mathsf{r}'$ (9) - inv. (E-SUB) of (6)

$fv(\tau^{\mathsf{s}''}) = \emptyset$ (10) - by (7,8)

$\Delta \subseteq \Delta'$ I.H. with (4,5,8,10)

$\Delta' \vdash S'$ I.H. with (4,5,8,10)

$\Delta' \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}''}$ (11) - I.H. with (4,5,8,10)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau^{\mathsf{s}'}$ by rule (E-SUB) with (11,9)

Case (E-COLLECTION):

$\Delta \vdash S_1$ (1) - hyp

$\Delta \vdash S_2$ (2) - hyp
$S_1 =_{\mathsf{s}} S_2$ (3) - hyp
$(S_1; \{\ldots, e_1, \ldots\}) \longrightarrow (S_1'; \{\ldots, e_1', \ldots\})$ (4) - hyp
$(S_2; \{\ldots, e_2, \ldots\}) \longrightarrow (S_2'; \{\ldots, e_2', \ldots\})$ (5) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \{\ldots, e_1, \ldots\} \cong_{\mathsf{s}} \{\ldots, e_2, \ldots\} : \tau'^{*\mathsf{s}'}$ (6) - hyp
$fv(\{\ldots, e_1, \ldots\}) \cup fv(\{\ldots, e_2, \ldots\}) \cup fv(\tau'^{*\mathsf{s}'}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp
$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (COLLECTION) of (4)
$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (9) - inv. (COLLECTION) of (5)
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau'^{\mathsf{s}'}$ (10) - inv. (E-COLLECTION) of (6)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau'^{\mathsf{s}'}$ (11) - by (10)
$fv(e_1) \cup fv(e_2) \cup fv(\tau'^{\mathsf{s}'}) = \emptyset$ (12) - by def. of fv with (7,11)
$\Delta \subseteq \Delta'$ by I.H. with (8,9,11,12)
$\Delta' \vdash S_1'$ by I.H. with (8,9,11,12)
$\Delta' \vdash S_2'$ by I.H. with (8,9,11,12)
$S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,11,12)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_2' : \tau'^{\mathsf{s}'}$ (13) - by I.H. with (8,9,11,12)
$\forall_i\ \Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_i \cong_{\mathsf{s}} e_i' : \tau'^{\mathsf{s}'}$ (14) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \{\ldots, e_1', \ldots\} \cong_{\mathsf{s}} \{\ldots, e_2', \ldots\} : \tau'^{*\mathsf{s}'}$ by rule (E-COLLECTION) with (13),(14)

Case (E-LET):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \cong_{\mathsf{s}} \mathbf{let}\ x = e_3\ \mathbf{in}\ e_4 : \tau'^{\mathsf{s}_2}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp

- Sub-case **(LET-LEFT)**:
  $(S_1; \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \longrightarrow (S_1'; \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2)$ (5) - hyp
  $(S_2; \mathbf{let}\ x = e_3\ \mathbf{in}\ e_4) \longrightarrow (S_2'; \mathbf{let}\ x = e_3'\ \mathbf{in}\ e_4)$ (6) - hyp
  $fv(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \cup fv(\mathbf{let}\ x = e_3\ \mathbf{in}\ e_4) \cup fv(\tau'^{\mathsf{s}_2}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp
  $(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (LET-LEFT) of (5)
  $(S_2; e_3) \longrightarrow (S_2'; e_3')$ (9) - inv. (LET-LEFT) of (6)
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_3 : \tau^{\mathsf{s}_1}$ (10) - inv. (E-LET) of (1)
  $\Delta, x : \tau^{\mathsf{s}_1} \vdash^{\mathsf{r}}_{\mathcal{S}_1\{x \doteq e_1\},\mathcal{S}_2\{x \doteq e_3\}} e_2 \cong_{\mathsf{s}} e_4 : \tau'^{\mathsf{s}_2}$ (11) - inv. (E-LET) of (1)
  $fv(e_1) \cup fv(e_3) \cup fv(\tau^{\mathsf{s}_1}) = \emptyset$ (12) - by def. of fv with (7,10,11) and by Def. 19
  $\Delta \subseteq \Delta'$ by I.H. with (8,9,10,12)
  $\Delta' \vdash S_1'$ by I.H. with (8,9,10,12)
  $\Delta' \vdash S_2'$ by I.H. with (8,9,10,12)
  $S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,10,12)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_3' : \tau^{\mathsf{s}_1}$ (13) - by I.H. with (8,9,10,12)
  $\Delta', x{:}\tau^{\mathsf{s}_1} \vdash^{\mathsf{r}}_{\mathcal{S}_1\{x \doteq e_1\},\mathcal{S}_2\{x \doteq e_3\}} e_2 \cong_{\mathsf{s}} e_4{:}\tau'^{\mathsf{s}_2}$ (14) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (11)
  $\Delta', x{:}\tau^{\mathsf{s}_1} \vdash^{\mathsf{r}}_{\mathcal{S}_1\{x \doteq e_1'\},\mathcal{S}_2\{x \doteq e_3'\}} e_2 \cong_{\mathsf{s}} e_4 : \tau'^{\mathsf{s}_2}$

(15) - by (8,9) with (14) since reduction preserves $\doteq$, so $e_1 \doteq e_1'$ and $e_3 \doteq e_3'$

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \textbf{let } x = e_1' \textbf{ in } e_2 \;\cong_{\mathsf{s}}\; \textbf{let } x = e_3' \textbf{ in } e_4 : \tau'^{\mathsf{s}_2}$ by rule (E-LET) with (13,15)

- **Case (LET-RIGHT)**:

$(S_1; \textbf{let } x = v \textbf{ in } e) \longrightarrow (S_1'; e\{v/x\})$ (5) - hyp

$(S_2; \textbf{let } x = v' \textbf{ in } e') \longrightarrow (S_2'; e'\{v'/x\})$ (6) - hyp

$S_1 = S_1'$ (7) - by rule (let-right) of (5)

$S_2 = S_2'$ (8) - by rule (let-right) of (6)

$S_1' =_{\mathsf{s}} S_2'$ by (2,7,8)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \;\cong_{\mathsf{s}}\; v' : \tau^{\mathsf{s}_1}$ (9) - inv. (E-LET) of (1)

$\Delta, x : \tau^{\mathsf{s}_1} \vdash^{\mathsf{r}}_{\mathcal{S}_1\{x\doteq v\},\mathcal{S}_2\{x\doteq v'\}} e \cong_{\mathsf{s}} e' : \tau'^{\mathsf{s}_2}$ (10) - inv. (E-LET) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{x\doteq v\},\mathcal{S}_2\{x\doteq v'\}} v \;\cong_{\mathsf{s}}\; v' : \tau^{\mathsf{s}_1}$ (11) - x fresh in $\Delta, \mathcal{S}_1, \mathcal{S}_2, v, v'$ by (9)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1'\{x\doteq v\},\mathcal{S}_2'\{x\doteq v'\}} e\{v/x\} \;\cong_{\mathsf{s}}\; e'\{v'/x\} : (\tau'^{\mathsf{s}_2})\{v/x\}$ (12) - by Lem. 18 with (10,11)

$\mathcal{S}_1' = \mathcal{S}_1\{v/x\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{v'/x\}$ by Lem. 18 with (10,11)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\{x\doteq v\},\mathcal{S}_2\{x\doteq v'\}} e\{v/x\} \;\cong_{\mathsf{s}}\; e'\{v'/x\} : \tau'^{\mathsf{s}_2}$

x fresh in $\Delta, \mathcal{S}_1, \mathcal{S}_2$ by (9), and x fresh in τ'^{s_2} by (1)

Case (E-APP):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1(e_2) \;\cong_{\mathsf{s}}\; e_3(e_4) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\}\sqcup\mathsf{t}}$ (1) - hyp

$\Delta \vdash S_1$ (2) - hyp

$\Delta \vdash S_2$ (3) - hyp

$S_1 =_{\mathsf{s}} S_2$ (4) - hyp

- Sub-case **(APP-LEFT)**:

$fv(e_1(e_2)) \cup fv(e_3(e_4)) \cup fv(\sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\}\sqcup\mathsf{t}}) = \emptyset$ and $vars(\Delta) = \emptyset$ (5) - hyp

$(S_1; e_1(e_2)) \longrightarrow (S_1'; e_1'(e_2))$ (6) - hyp

$(S_2; e_3(e_4)) \longrightarrow (S_2'; e_3'(e_4))$ (7) - hyp

$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (APP-LEFT) of (6)

$(S_2; e_3) \longrightarrow (S_2'; e_3')$ (9) - inv. (APP-LEFT) of (7)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \;\cong_{\mathsf{s}}\; e_3 : (\Pi x{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{q}})^{\mathsf{t}}$ (10) - inv. (E-APP) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \;\cong_{\mathsf{s}}\; e_4 : \tau^{\mathsf{s}'}$ (11) - inv. (E-APP) of (1)

$\mathcal{S}_1 \cup \{x \doteq e_2\} \models x \doteq \mathsf{v}$ and $\mathcal{S}_2 \cup \{x \doteq e_2'\} \models x \doteq \mathsf{v}$ (12) - inv. (E-APP) of (1)

$fv(e_1) \cup fv(e_3) \cup fv((\Pi x{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{q}})^{\mathsf{t}}) = \emptyset$ (13) - by def. of fv with (5,10) and by Def. 19

$\Delta \subseteq \Delta'$ by I.H. with (8,9,10,13)

$\Delta' \vdash S_1'$ by I.H. with (8,9,10,13)

$\Delta' \vdash S_2'$ by I.H. with (8,9,10,13)

$S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,10,13)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \;\cong_{\mathsf{s}}\; e_3' : (\Pi x{:}\tau^{\mathsf{s}'}.\sigma^{\mathsf{q}})^{\mathsf{t}}$ (14) - by I.H. with (8,9,10,13)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \;\cong_{\mathsf{s}}\; e_4 : \tau^{\mathsf{s}'}$ (15) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (11)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1'(e_2) \;\cong_{\mathsf{s}}\; e_3'(e_4) : \sigma\{\mathsf{v}/x\}^{\mathsf{q}\{\mathsf{v}/x\}\sqcup\mathsf{t}}$ by rule (E-APP) with (12,14,15)

- Sub-case **(APP-RIGHT)**:

$(S_1; (\lambda(x : \tau^{s'}).e)(e_1)) \longrightarrow (S_1'; (\lambda(x : \tau^{s'}).e)(e_1'))$ (6) - hyp

$(S_2; (\lambda(x : \tau^{s'}).e')(e_2)) \longrightarrow (S_2'; (\lambda(x : \tau^{s'}).e')(e_2'))$ (7) - hyp

$fv((\lambda(x{:}\tau^{s'}).e)(e_1)) \cup fv((\lambda(x{:}\tau^{s'}).e)(e_2)) \cup fv(\sigma\{^{v}/_{x}\}^{q\{^{v}/_{x}\}\sqcup t}) = \emptyset$ and $vars(\Delta) = \emptyset$ (8) - hyp

$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (9) - inv. (APP-RIGHT) of (6)

$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (10) - inv. (APP-RIGHT) of (7)

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} \lambda(x : \tau^{s'}).e \cong_s \lambda(x : \tau^{s'}).e' : (\Pi x{:}\tau^{s'}.\sigma^{q})^{t}$ (11) - inv. (E-APP) of (1)

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_s e_2 : \tau^{s'}$ (12) - inv. (E-APP) of (1)

$\mathcal{S}_1 \cup \{x \doteq e_2\} \models x \doteq v$ and $\mathcal{S}_2 \cup \{x \doteq e_2'\} \models x \doteq v$ (13) - inv. (E-APP) of (1)

$fv(e_1) \cup fv(e_2) \cup fv(\tau^{s'}) = \emptyset$ (14) - by def. of fv with (5,11) and by Def. 19

$\Delta \subseteq \Delta'$ by I.H. with (9,10,12,14)

$\Delta' \vdash S_1'$ by I.H. with (9,10,12,14)

$\Delta' \vdash S_2'$ by I.H. with (9,10,12,14)

$S_1' =_s S_2'$ by I.H. with (9,10,12,14)

$\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_s e_2' : \tau^{s'}$ (15) - by I.H. with (9,10,12,14)

$\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} \lambda(x : \tau^{s'}).e \cong_s \lambda(x : \tau^{s'}).e' : (\Pi x{:}\tau^{s'}.\sigma^{q})^{t}$ (16) - by Weakining Lemma for $\cong_s$ with (11)

$\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} (\lambda(x{:}\tau^{s'}).e)(e_1') \cong_s (\lambda(x{:}\tau^{s'}).e')(e_2') : \sigma\{^{v}/_{x}\}^{q\{^{v}/_{x}\}\sqcup t}$ by rule (E-APP) with (13,15,16)

- Sub-case **(APP)**:

$(S_1; (\lambda(x : \tau^{s''}).e)(v)) \longrightarrow (S_1; e\{^{x}/_{v}\})$ (5) - hyp

$(S_2; (\lambda(x : \tau^{s''}).e')(v')) \longrightarrow (S_2; e'\{^{x}/_{v'}\})$ (6) - hyp

$S_1 = S_1'$ (7) - by rule (APP) of (5)

$S_2 = S_2'$ (8) - by rule (APP) of (6)

$S_1' =_s S_2'$ by (4,7,8)

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} (\lambda(x : \tau^{s''}).e) \cong_s (\lambda(x : \tau^{s''}).e') : (\Pi x{:}\tau^{s'}.\sigma^{q})^{t}$ (9) - inv. (E-APP) of (1)

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s v' : \tau^{s'}$ (10) - inv. (E-APP) of (1)

$\mathcal{S}_1 \cup \{x \doteq v\} \models x \doteq v$ and $\mathcal{S}_2 \cup \{x \doteq v\} \models x \doteq v$ (11) - inv. (E-APP) of (1)

$\tau^{s'} <: \tau^{s''}$ (12) - by Lem. 15 with (9)

$\sigma^{q'} <: \sigma^{q}$ (13) - by Lem. 15 with (9)

$\Delta, x : \tau^{s''} \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_s e' : \sigma^{q'}$ (14) - by Lem. 15 with (9)

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s v' : \tau^{s''}$ (15) - by rule (E-SUB) with (10,12)

$\Delta \vdash^{r}_{\mathcal{S}_1',\mathcal{S}_2'} e\{^{v}/_{x}\} \cong_s e'\{^{v'}/_{x}\} : (\sigma^{q'})\{^{v}/_{x}\}$ (16) - Lem. 18 with (14,15)

$\mathcal{S}_1' = \mathcal{S}_1\{^{v}/_{x}\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{^{v'}/_{x}\}$ (17) - Lem. 18 with (14,15)

$(\sigma^{q'})\{^{v}/_{x}\} <: (\sigma^{q})\{^{v}/_{x}\}$ (18) - by Lem. 11 with (13)

$(\sigma^{q})\{^{v}/_{x}\} <: \sigma\{^{v}/_{x}\}^{q\{^{v}/_{x}\}\sqcup t}$ (19) - since $q\{^{v}/_{x}\} \leq q\{^{v}/_{x}\} \sqcup t$ by def. of $\sqcup$

$(\sigma^{q'})\{^{v}/_{x}\} <: \sigma\{^{v}/_{x}\}^{q\{^{v}/_{x}\}\sqcup t}$ (20) - by (S-TRANS) with (18,19)

$\Delta \vdash^{r}_{\mathcal{S}_1',\mathcal{S}_2'} e\{^{v}/_{x}\} \cong_s e'\{^{v'}/_{x}\} : \sigma\{^{v}/_{x}\}^{q\{^{v}/_{x}\}\sqcup t}$ by rule (E-SUB) with (16,20)

$\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e\{^{v}/_{x}\} \cong_s e'\{^{v'}/_{x}\} : \sigma\{^{v}/_{x}\}^{q\{^{v}/_{x}\}\sqcup t}$ x fresh in $\Delta, \mathcal{S}_1, \mathcal{S}_2$ by (10)

Case (E-IF):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 \cong_{\mathsf{s}} \textbf{if } c' \textbf{ then } e_3 \textbf{ else } e_4 : \tau^{\mathsf{s}'}$ (1) - hyp

$\Delta \vdash S_1$ (2) - hyp

$\Delta \vdash S_2$ (3) - hyp

$S_1 =_{\mathsf{s}} S_2$ (4) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} c \cong_{\mathsf{s}} c' : \mathtt{Bool}^{\mathsf{s}'}$ (5) - inv. (E-IF) of (1)

$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{c\doteq\mathtt{true}\},\mathcal{S}_2\cup\{c'\doteq\mathtt{true}\}} e_1 \cong_{\mathsf{s}} e_3 : \tau^{\mathsf{s}'}$ (6) - inv. (E-IF) of (1)

$\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{c\doteq\mathtt{false}\},\mathcal{S}_2\cup\{c'\doteq\mathtt{false}\}} e_2 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$ (7) - inv. (E-IF) of (1)

$\mathsf{r} \leq \mathsf{r} \sqcup \mathsf{s}'$ (8) - by def. of $\sqcup$

- Sub-case **(IF-TRUE)**:

  $(S_1; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \longrightarrow (S_1; e_1)$ (9) - hyp

  $(S_2; \textbf{if } c' \textbf{ then } e_3 \textbf{ else } e_4) \longrightarrow (S_2; e_3)$ (10) - hyp

  $\mathcal{C}[\![c]\!] = \mathtt{true}$ (11) - inv. (IF-TRUE) of (9)

  $\mathcal{C}[\![c']\!] = \mathtt{true}$ (12) - inv. (IF-TRUE) of (10)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\cup\{c\doteq\mathtt{true}\},\mathcal{S}_2\cup\{c'\doteq\mathtt{true}\}} e_1 \cong_{\mathsf{s}} e_3 : \tau^{\mathsf{s}'}$ (13) - by rule (E-SUB) with (8,6)

  $\mathcal{S}_1 \models \mathtt{true} \doteq \mathtt{true}$ and $\mathcal{S}_2 \models \mathtt{true} \doteq \mathtt{true}$ (14) - by (11,12,13)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_3 : \tau^{\mathsf{s}'}$ by Lem. 13 with (13,14)

- Sub-case **(IF-FALSE)**:

  $(S_1; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \longrightarrow (S_1; e_2)$ (9) - hyp

  $(S_2; \textbf{if } c' \textbf{ then } e_3 \textbf{ else } e_4) \longrightarrow (S_2; e_4)$ (10) - hyp

  $\mathcal{C}[\![c]\!] = \mathtt{false}$ (11) - inv. (IF-FALSE) of (9)

  $\mathcal{C}[\![c']\!] = \mathtt{false}$ (12) - inv. (IF-FALSE) of (10)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1\cup\{c\doteq\mathtt{false}\},\mathcal{S}_2\cup\{c'\doteq\mathtt{false}\}} e_2 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$ (13) - by rule (E-SUB) with (8,7)

  $\mathcal{S}_1 \models \mathtt{false} \doteq \mathtt{false}$ and $\mathcal{S}_2 \models \mathtt{false} \doteq \mathtt{false}$ (14) - by (11,12,13)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$ by Lem. 13 with (13,14)

- Sub-case **(IF-TRUE)** and **(IF-FALSE)**:

  $(S_1; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2) \longrightarrow (S_1; e_1)$ (9) - hyp

  $(S_2; \textbf{if } c' \textbf{ then } e_3 \textbf{ else } e_4) \longrightarrow (S_2; e_4)$ (10) - hyp

  $\mathcal{C}[\![c]\!] = \mathtt{true}$ (11) - inv. (IF-TRUE) of (9)

  $\mathcal{C}[\![c']\!] = \mathtt{false}$ (12) - inv. (IF-FALSE) of (10)

  $\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{c\doteq\mathtt{true}\}} e_1 : \tau^{\mathsf{s}'}$ (13) - by def. of expression equivalence with (6)

  $\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_2\cup\{c'\doteq\mathtt{false}\}} e_4 : \tau^{\mathsf{s}'}$ (14) - by def. of expression equivalence with (7)

  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathtt{true} \cong_{\mathsf{s}} \mathtt{false} : \mathtt{Bool}^{\mathsf{s}'}$ (15) - by (5,11,12)

  $\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$ (16) - by inv. of (E-EXPROPAQUE) with (15)

  $\mathsf{s} < \mathsf{s}' \sqcap (\mathsf{r} \sqcup \mathsf{s}')$ (17) - by (13,14,16)

  $\Delta \vdash^{\mathsf{r}\sqcup\mathsf{s}'}_{\mathcal{S}_1\cup\{c\doteq\mathtt{true}\},\mathcal{S}_2\cup\{c'\doteq\mathtt{false}\}} e_1 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$
  (18) - by rule (E-EXPROPAQUE) with (13,14,15,17)

$\mathcal{S}_1 \models \texttt{true} \doteq \texttt{true}$ and $\mathcal{S}_2 \models \texttt{false} \doteq \texttt{false}$ (19) - by (11,12,18)
$\Delta \vdash^{\mathsf{r} \sqcup \mathsf{s}'}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$ (20) - by Lem. 13 with (18,19)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_4 : \tau^{\mathsf{s}'}$ by rule (E-SUB) with (20,8)

- Sub-case **(IF-FALSE)** and **(IF-TRUE)**: symmetric previous subcase

Case (E-CONS):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1::e_2 \cong_{\mathsf{s}} e_3::e_4 : \tau^{*\mathsf{s}'}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp

- Sub-case **(CONS-LEFT)**:

$(S_1; e_1::e_2) \longrightarrow (S_1'; e_1'::e_2)$ (5) - hyp
$(S_2; e_3::e_4) \longrightarrow (S_2'; e_3'::e_4)$ (6) - hyp
$fv(e_1::e_2) \cup fv(e_3::e_4) \cup fv(\tau^{*\mathsf{s}'}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp
$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (CONS-LEFT) of (5)
$(S_2; e_3) \longrightarrow (S_2'; e_3')$ (9) - inv. (CONS-LEFT) of (6)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_3\ \tau^{\mathsf{s}'}$ (10) - inv. (E-CONS) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_4 : \tau^{*\mathsf{s}'}$ (11) - inv. (E-CONS) of (1)
$fv(e_1) \cup fv(e_3) \cup fv(\tau^{\mathsf{s}'}) = \emptyset$ (12) - by def. of fv with (7,10)
$\Delta \subseteq \Delta'$ by I.H. with (8,9,10,12)
$\Delta' \vdash S_1'$ by I.H. with (8,9,10,12)
$\Delta' \vdash S_2'$ by I.H. with (8,9,10,12)
$S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,10,12)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_{\mathsf{s}} e_3' : \tau^{\mathsf{s}'}$ (13) - by I.H. with (8,9,10,12)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_{\mathsf{s}} e_4 : \tau^{*\mathsf{s}'}$ (14) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1'::e_2 \cong_{\mathsf{s}} e_3'::e_4 : \tau'^{*\mathsf{s}'}$ by rule (E-CONS) with (13,14)

- Sub-case **(CONS-RIGHT)**:

$(S_1; v::e_1) \longrightarrow (S_1'; v::e_1')$ (5) - hyp
$(S_2; u::e_2) \longrightarrow (S_2'; u::e_2')$ (6) - hyp
$fv(v::e_1) \cup fv(u::e_2) \cup fv(\tau^{*\mathsf{s}'}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp
$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (CONS-RIGHT) of (5)
$(S_2; e_2) \longrightarrow (S_2'; e_2')$ (9) - inv. (CONS-RIGHT) of (6)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} u : \tau^{\mathsf{s}'}$ (10) - inv. (E-CONS) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{*\mathsf{s}'}$ (11) - inv. (E-CONS) of (1)
$fv(e_1) \cup fv(e_2) \cup fv(\tau^{*\mathsf{s}'}) = \emptyset$ (12) - by def. of fv with (7,11)
$\Delta \subseteq \Delta'$ by I.H. with (8,9,11,12)
$\Delta' \vdash S_1'$ by I.H. with (8,9,11,12)

$\Delta' \vdash S_2'$ by I.H. with (8,9,11,12)
$S_1' =_\mathsf{s} S_2'$ by I.H. with (8,9,11,12)
$\Delta' \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_\mathsf{s} e_2' : \tau^{*\mathsf{s}'}$ (13) - by I.H. with (8,9,11,12)
$\Delta' \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_\mathsf{s} u{:}\tau^{\mathsf{s}'}$ (14) - by Weakining Lemma for $\cong_\mathsf{s}$ with (10)
$\Delta' \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} v{::}e_1' \cong_\mathsf{s} u{::}e_2' : \tau^{*\mathsf{s}'}$ by rule (E-CONS) with (13,14)

- Sub-case **(CONS)**:
  $(S; v{::}\{v_1, \ldots, v_n\}) \longrightarrow (S; \{v, v_1, \ldots, v_n\})$ (5) - hyp
  $(S; v'{::}\{v_1', \ldots, v_n'\}) \longrightarrow (S; \{v', v_1', \ldots, v_n'\})$ (6) - hyp
  $\Delta \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_\mathsf{s} v' : \tau^{\mathsf{s}'}$ (7) - inv. ([e-cons]) of (1)
  $\Delta \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} \{v_1, \ldots, v_n\} \cong_\mathsf{s} \{v_1', \ldots, v_n'\} : \tau^{*\mathsf{s}'}$ (8) - inv. (E-CONS) of (1)
  $\Delta \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} \forall_i\, v_i \cong_\mathsf{s} v_i' : \tau^{\mathsf{s}'}$ (9) - inv. (E-COLLECTION) of (8)
  $\Delta \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} \{v, v_1, \ldots, v_n\} \cong_\mathsf{s} \{v', v_1', \ldots, v_n'\} : \tau^{*\mathsf{s}'}$ by rule (E-COLLECTION) with (7,9)

Case (E-FOREACH):
$\Delta \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(e_1, e_2, x.y.e_3) \cong_\mathsf{s} \mathbf{foreach}(e_4, e_5, x.y.e_6){:}\tau'^{\mathsf{s}'}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_\mathsf{s} S_2$ (4) - hyp

- Sub-case **(FOREACH-LEFT)**:
  $(S_1; \mathbf{foreach}(e_1, e_2, x.y.e_3)) \longrightarrow (S_1'; \mathbf{foreach}(e_1', e_2, x.y.e_3))$ (5) - hyp
  $(S_2; \mathbf{foreach}(e_4, e_5, x.y.e_6)) \longrightarrow (S_2'; \mathbf{foreach}(e_4', e_5, x.y.e_6))$ (6) - hyp
  $fv(\mathbf{foreach}(e_1, e_2, x.y.e_3)) \cup fv(\mathbf{foreach}(e_4, e_5, x.y.e_6) \cup fv(\tau'^{\mathsf{s}'}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp
  $(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. (FOREACH-LEFT) of (5)
  $(S_2; e_4) \longrightarrow (S_2'; e_4')$ (9) - inv. (FOREACH-LEFT) of (6)
  $\Delta \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_\mathsf{s} e_4 : \tau^{*\mathsf{s}'}$ (10) - inv. (E-FOREACH) of (1)
  $\Delta \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_\mathsf{s} e_5 : \tau'^{\mathsf{s}'}$ (11) - inv. (E-FOREACH) of (1)
  $\Delta, x : \tau^{\mathsf{s}'}, y : \tau'^{\mathsf{s}'} \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} e_3 \cong_\mathsf{s} e_6 : \tau'^{\mathsf{s}'}$ (12) - inv. (E-FOREACH) of (1)
  $fv(e_1) \cup fv(e_4) \cup fv(\tau^{*\mathsf{s}'}) = \emptyset$ (13) - by def. of fv with (7,10)
  $\Delta \subseteq \Delta'$ by I.H. with (8,9,10,13)
  $\Delta' \vdash S_1'$ by I.H. with (8,9,10,13)
  $\Delta' \vdash S_2'$ by I.H. with (8,9,10,13)
  $S_1' =_\mathsf{s} S_2'$ by I.H. with (8,9,10,13)
  $\Delta' \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1' \cong_\mathsf{s} e_4' : \tau^{*\mathsf{s}'}$ (14) - by I.H. with (8,9,10,13)
  $\Delta' \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_\mathsf{s} e_5 : \tau'^{\mathsf{s}'}$ (15) - by Weakining Lemma for $\cong_\mathsf{s}$ with (11)
  $\Delta', x : \tau^{\mathsf{s}'}, y : \tau'^{\mathsf{s}'} \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} e_3 \cong_\mathsf{s} e_6 : \tau'^{\mathsf{s}'}$ (16) - by Weakining Lemma for $\cong_\mathsf{s}$ with (12)
  $\Delta' \vdash^\mathsf{r}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(e_1', e_2, x.y.e_3) \cong_\mathsf{s} \mathbf{foreach}(e_4', e_5, x.y.e_6){:}\tau'^{\mathsf{s}'}$ by rule (E-FOREACH) with (14,15,16)

- Sub-case **(FOREACH-RIGHT)**:

  $(S_1; \mathbf{foreach}(v, e_2, x.y.e_3)) \longrightarrow (S_1'; \mathbf{foreach}(v, e_2', x.y.e_3))$ (5) - hyp

  $(S_2; \mathbf{foreach}(u, e_5, x.y.e_6)) \longrightarrow (S_2'; \mathbf{foreach}(u, e_5', x.y.e_6))$ (6) - hyp

  $fv(\mathbf{foreach}(v, e_2, x.y.e_3)) \cup fv(\mathbf{foreach}(u, e_5, x.y.e_6) \cup fv(\tau'^{s'}) = \emptyset$ and $vars(\Delta) = \emptyset$ (7) - hyp

  $(S_1; e_2) \longrightarrow (S_1'; e_2')$ (8) - inv. (FOREACH-RIGHT) of (5)

  $(S_2; e_5) \longrightarrow (S_2'; e_5')$ (9) - inv. (FOREACH-RIGHT) of (6)

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s u : \tau^{*s'}$ (10) - inv. (E-FOREACH) of (1)

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_2 \cong_s e_5 : \tau'^{s'}$ (11) - inv. (E-FOREACH) of (1)

  $\Delta, x : \tau^{s'}, y : \tau'^{s'} \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_3 \cong_s e_6 : \tau'^{s'}$ (12) - inv. (E-FOREACH) of (1)

  $fv(e_2) \cup fv(e_5) \cup fv(\tau^{s'}) = \emptyset$ (13) - by def. of fv with (7,11)

  $\Delta \subseteq \Delta'$ by I.H. with (8,9,11,13)

  $\Delta' \vdash S_1'$ by I.H. with (8,9,11,13)

  $\Delta' \vdash S_2'$ by I.H. with (8,9,11,13)

  $S_1' =_s S_2'$ by I.H. with (8,9,11,13)

  $\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_2' \cong_s e_5' : \tau'^{s'}$ (14) - by I.H. with (8,9,11,13)

  $\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s u : \tau^{*s'}$ (15) - by Weakining Lemma for $\cong_s$ with (10)

  $\Delta', x : \tau^{s'}, y : \tau'^{s'} \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_3 \cong_s e_6 : \tau'^{s'}$ (16) - by Weakining Lemma for $\cong_s$ with (12)

  $\Delta' \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(v, e_2', x.y.e_3) \cong_s \mathbf{foreach}(u, e_5', x.y.e_6) : \tau'^{s'}$ by rule (E-FOREACH) with (14,15,16)

- Sub-case **(FOREACH)**:

  $(S_1; \mathbf{foreach}(h::hs, v, x.y.e_1)) \longrightarrow (S_1'; \mathbf{foreach}(hs, e_1\{h/x\}\{v/y\}, x.y.e_1))$ (5) - hyp

  $(S_2; \mathbf{foreach}(h'::hs', u, x.y.e_2)) \longrightarrow (S_2'; \mathbf{foreach}(hs', e_2\{h'/x\}\{u/y\}, x.y.e_2))$ (6) - hyp

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} h::hs \cong_s h'::hs' : \tau^{*s'}$ (7) - inv. (E-FOREACH) of (1)

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_s u : \tau'^{s'}$ (8) - inv. (E-FOREACH) of (1)

  $\Delta, x : \tau^{s'}, y : \tau'^{s'} \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_s e_2 : \tau'^{s'}$ (9) - inv. (E-FOREACH) of (1)

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} h \cong_s h' : \tau^{s'}$ (10) - inv. (E-CONS) of (7)

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} hs \cong_s hs' : \tau^{*s'}$ (11) - inv. (E-CONS) of (7)

  $\Delta \vdash^{r}_{\mathcal{S}_1',\mathcal{S}_2'} e_1\{h/x\}\{v/y\} \cong_s e_2\{h'/x\}\{u/y\} : (\tau'^{s'})\{h/x\}\{v/y\}$ (12) - by Lem. 18 with (8,9,10)

  $\mathcal{S}_1' = \mathcal{S}_1\{h/x\}\{v/y\}$ and $\mathcal{S}_2' = \mathcal{S}_2\{h'/x\}\{u/y\}$ by Lem. 18 with (8,9,10)

  $\Delta \vdash^{r}_{\mathcal{S}_1',\mathcal{S}_2'} \mathbf{foreach}(hs, e_1\{h/x\}\{v/y\}, x.y.e_1) \cong_s \mathbf{foreach}(hs', e_2\{h'/x\}\{u/y\}, x.y.e_2) : (\tau'^{s'})\{h/x\}\{v/y\}$ by rule (e-foreach) with (9,11,12)

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(hs, e_1\{h/x\}\{v/y\}, x.y.e_1) \cong_s \mathbf{foreach}(hs', e_2\{h'/x\}\{u/y\}, x.y.e_2) : (\tau'^{s'})\{h/x\}\{v/y\}$ x, y, fresh in $\Delta, \mathcal{S}_1, \mathcal{S}_2$ by (8,10)

  $\Delta \vdash^{r}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{foreach}(hs, e_1\{h/x\}\{v/y\}, x.y.e_1) \cong_s \mathbf{foreach}(hs', e_2\{h'/x\}\{u/y\}, x.y.e_2) : \tau'^{s'}$ x, y fresh in $\tau'^{s'}$ by (1,8) and by Def. 19

- Sub-case **(FOREACH-BASE)**:

  $(S; \mathbf{foreach}(\{\}, v, x.y.e_3)) \longrightarrow (S; v)$ (5) - hyp

$(S; \mathbf{foreach}(\{\}, v', x.y.e'_3)) \longrightarrow (S; v')$ (6) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \{\} \cong_{\mathsf{s}} \{\} : \tau^{*\mathsf{s}'}$ (7) - inv. (E-FOREACH) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} v' : \tau'^{\mathsf{s}'}$ (8) - inv. (E-FOREACH) of (1)
$\Delta, x : \tau^{\mathsf{s}'}, y : \tau'^{\mathsf{s}'} \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_3 \cong_{\mathsf{s}} e'_3 : \tau'^{\mathsf{s}}$ (9) - inv. (E-FOREACH) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} v' : \tau'^{\mathsf{s}'}$ by (8)

Case (E-REF):

$\Delta \vdash S_1$ (1) - hyp
$\Delta \vdash S_2$ (2) - hyp
$S_1 =_{\mathsf{s}} S_2$ (3) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{ref}_{\tau^{\mathsf{s}}} e_1 \cong_{\mathsf{s}} \mathbf{ref}_{\tau^{\mathsf{s}}} e_2 : \mathtt{ref}(\tau^{\mathsf{s}'})^{\perp}$ (4) - hyp

- Sub-case **(REF-LEFT)**:

  $(S_1; \mathbf{ref}_{\tau^{\mathsf{s}}} e_1) \longrightarrow (S'_1; \mathbf{ref}_{\tau^{\mathsf{s}}} e'_1)$ (5) - hyp
  $(S_2; \mathbf{ref}_{\tau^{\mathsf{s}}} e_2) \longrightarrow (S'_2; \mathbf{ref}_{\tau^{\mathsf{s}}} e'_2)$ (6) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$ (7) - inv. of (E-REF) with (4)
  $\mathsf{r} \leq \mathsf{s}'$ inv. of (E-REF) with (4)
  $(S_1; e_1) \longrightarrow (S'_1; e'_1)$ (8) - inv. of (REF-LEFT) with (5)
  $(S_2; e_2) \longrightarrow (S'_2; e'_2)$ (9) - inv. of (REF-LEFT) with (6)
  $\Delta \subseteq \Delta'$ by I.H. with (7,8,9)
  $\Delta' \vdash S'_1$ by I.H. with (7,8,9)
  $\Delta' \vdash S'_2$ by I.H. with (7,8,9)
  $S'_1 =_{\mathsf{s}} S'_2$ by I.H. with (7,8,9)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e'_1 \cong_{\mathsf{s}} e'_2 : \tau^{\mathsf{s}'}$ by (10) - I.H. with (7,8,9)
  $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \mathbf{ref}_{\tau^{\mathsf{s}}} e'_1 \cong_{\mathsf{s}} \mathbf{ref}_{\tau^{\mathsf{s}}} e'_2 : \mathtt{ref}(\tau^{\mathsf{s}'})^{\perp}$ by (E-REF) with (10)

- Sub-case **(REF-RIGHT)**:

  $(S_1; \mathbf{ref}_{\tau^{\mathsf{s}}} e_1) \longrightarrow (S_1 \cup \{l \mapsto e_1\};\ l)$ (5) - hyp
  $(S_2; \mathbf{ref}_{\tau^{\mathsf{s}}} e_2) \longrightarrow (S_2 \cup \{l \mapsto e_2\};\ l)$ (6) - hyp
  $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}$ (7) - inv. of (E-REF) with (4)
  $l \notin dom(S_1) \cup fn(e_1)$ (8) - inv. of (REF-RIGHT) with (5)
  $l \notin dom(S_2) \cup fn(e_2)$ (9) - inv. of (REF-RIGHT) with (6)
  $\Delta' = \Delta, l : \tau^{\mathsf{s}'}$
  $\Delta' \vdash S_1 \cup \{l \mapsto e_1\}$ (10) - by Def. 22 with (1)
  $\Delta' \vdash S_2 \cup \{l \mapsto e_2\}$ (11) - by Def. 22 with (2)
  $redact(\Delta, S_1, \mathsf{s}) = redact(\Delta, S_2, \mathsf{s})$ (12) - by Def. 26 with (3)
  $redact(\Delta', S'_1, \mathsf{s}) = redact(\Delta, S_1, \mathsf{s}) \cup redact(\{l : \Delta'(l)\}, \{l \mapsto e_1\}, \mathsf{s})$
  (13) - by Def. 22 with (10)
  $redact(\Delta', S'_2, \mathsf{s}) = redact(\Delta, S_2, \mathsf{s}) \cup redact(\{l : \Delta'(l)\}, \{l \mapsto e_2\}, \mathsf{s})$
  (14) - by Def. 22 with (11)
  $redact(\{l : \Delta'(l)\}, \{l \mapsto e_1\}, \mathsf{s}) = redact(\{l : \Delta'(l)\}, \{l \mapsto e_2\}, \mathsf{s})$

(15) - by Def. 24 with (7)

$redact(\Delta', S_1', \mathtt{s}) = redact(\Delta', S_2', \mathtt{s})$ (16) - by (12,15)

$S_1 \cup \{l \mapsto e_1\} =_{\mathtt{s}} S_2 \cup \{l \mapsto e_2\}$ Def. 26 with (16)

Case (E-DEREF):

$\Delta \vdash S_1$ (1) - hyp

$\Delta \vdash S_2$ (2) - hyp

$S_1 =_{\mathtt{s}} S_2$ (3) - hyp

$\Delta \vdash^{\mathtt{r}}_{\mathcal{S}_1,\mathcal{S}_2} !e_1 \cong_{\mathtt{s}} !e_2 : \tau^{\mathtt{s}'}$ (4) - hyp

- Sub-case **(DEREF-LEFT)**:

  $(S_1;\ !e_1) \longrightarrow (S_1';\ !e_1')$ (5) - hyp

  $(S_2;\ !e_2) \longrightarrow (S_2';\ !e_2')$ (6) - hyp

  $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}_1,\mathcal{S}_2}\ e_1 \cong_{\mathtt{s}}\ e_2 : \mathtt{ref}(\tau^{\mathtt{s}'})^{\mathtt{t}}$ (7) - inv. of (E-REF) with (4)

  $(S_1; e_1) \longrightarrow (S_1'; e_1')$ (8) - inv. of (REF-LEFT) with (5)

  $(S_2; e_2) \longrightarrow (S_2'; e_2')$ (9) - inv. of (REF-LEFT) with (6)

  $\Delta \subseteq \Delta'$ by I.H. with (7,8,9)

  $\Delta' \vdash S_1'$ by I.H. with (7,8,9)

  $\Delta' \vdash S_2'$ by I.H. with (7,8,9)

  $S_1' =_{\mathtt{s}} S_2'$ by I.H. with (7,8,9)

  $\Delta' \vdash^{\mathtt{r}}_{\mathcal{S}_1,\mathcal{S}_2}\ e_1' \cong_{\mathtt{s}}\ e_2' : \mathtt{ref}(\tau^{\mathtt{s}'})^{\mathtt{t}}$ (10) - by I.H. with (7,8,9)

  $\Delta' \vdash^{\mathtt{r}}_{\mathcal{S}_1,\mathcal{S}_2} !e_1' \cong_{\mathtt{s}} !e_2' : \tau^{\mathtt{s}'}$ by (E-DEREF) with (10)

- Sub-case **(DEREF)**:

  $(S_1; !e_1) \longrightarrow (S_1; v_1)$ (5) - hyp

  $(S_2; !e_2) \longrightarrow (S_2; v_2)$ (6) - hyp

  $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}_1,\mathcal{S}_2}\ e_1 \cong_{\mathtt{s}}\ e_2 : \mathtt{ref}(\tau^{\mathtt{s}'})^{\mathtt{t}}$ (7) - inv. of (E-DEREF) with (4)

  $S_1(l) = v_1$ (8) - inv. of (DEREF) with (5)

  $S_2(l) = v_2$ (9) - inv. of (DEREF) with (6)

  $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}_1,\mathcal{S}_2}\ v_1 \cong_{\mathtt{s}}\ v_2 : \tau^{\mathtt{s}'}$ by Def. 26 with (4)

Case (E-ASSIGN):

$\Delta \vdash S_1$ (1) - hyp

$\Delta \vdash S_2$ (2) - hyp

$S_1 =_{\mathtt{s}} S_2$ (3) - hyp

$\Delta \vdash^{\mathtt{r}}_{\mathcal{S}}\ e_1 := e_2\ \cong_{\mathtt{s}}\ e_3 := e_4 : \mathtt{cmd}^{\mathtt{s}'}$ (4) - hyp

- Sub-case **(ASSIGN-LEFT)**:

  $(S_1; e_1 := e_2) \longrightarrow (S_1'; e_1' := e_2)$ (5) - hyp

  $(S_2; e_3 := e_4) \longrightarrow (S_2'; e_3' := e_4)$ (6) - hyp

  $\Delta \vdash^{\mathtt{r}}_{\mathcal{S}_1,\mathcal{S}_2}\ e_1 \cong_{\mathtt{s}}\ e_3 : \mathtt{ref}(\tau^{\mathtt{s}'})^{\mathtt{t}}$ (7) - inv. of (E-ASSIGN) with (4)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ e_2 \cong_{\mathsf{s}} \ e_4 : \tau^{\mathsf{s}'}$ (8) - inv. of (E-ASSIGN) with (4)
$(S_1; e_1) \longrightarrow (S_1'; e_1')$ (9) - inv. of (ASSIGN-LEFT) with (5)
$(S_2; e_3) \longrightarrow (S_2'; e_3')$ (10) - inv. of (ASSIGN-LEFT) with (6)
$\Delta \subseteq \Delta'$ by I.H. with (7,9,10)
$\Delta' \vdash S_1'$ by I.H. with (7,9,10)
$\Delta' \vdash S_2'$ by I.H. with (7,9,10)
$S_1' =_{\mathsf{s}} S_2'$ by I.H. with (7,9,10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ e_1' \cong_{\mathsf{s}} \ e_3' : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (11) - by I.H. with (7,9,10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ e_2 \cong_{\mathsf{s}} \ e_4 : \tau^{\mathsf{s}'}$ (12) - by Lem. 6 with (8)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} e_1' := e_2 \ \cong_{\mathsf{s}} \ e_3' := e_4 : \mathtt{cmd}^{\mathsf{s}'}$ by (E-ASSIGN) with (11,12)

- Sub-case **(ASSIGN-RIGHT)**:

$(S_1; l := e_2) \longrightarrow (S_1'; l := e_2')$ (5) - hyp
$(S_2; l := e_4) \longrightarrow (S_2'; l := e_4')$ (6) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ l \cong_{\mathsf{s}} \ l : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (7) - inv. of (E-ASSIGN) with (4)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ e_2 \cong_{\mathsf{s}} \ e_4 : \tau^{\mathsf{s}'}$ (8) - inv. of (E-ASSIGN) with (4)
$(S_1; e_2) \longrightarrow (S_1'; e_2')$ (9) - inv. of (ASSIGN-RIGHT) with (5)
$(S_2; e_4) \longrightarrow (S_2'; e_4')$ (10) - inv. of (ASSIGN-RIGHT) with (6)
$\Delta \subseteq \Delta'$ by I.H. with (8,9,10)
$\Delta' \vdash S_1'$ by I.H. with (8,9,10)
$\Delta' \vdash S_2'$ by I.H. with (8,9,10)
$S_1' =_{\mathsf{s}} S_2'$ by I.H. with (8,9,10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ e_2' \cong_{\mathsf{s}} \ e_4' : \tau^{\mathsf{s}'}$ (11) - by I.H. with (8,9,10)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ l \cong_{\mathsf{s}} \ l : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (12) - by Lem. 6 with (7)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}} l := e_2' \ \cong_{\mathsf{s}} \ l := e_4' : \mathtt{cmd}^{\mathsf{s}'}$ by (E-ASSIGN) with (11,12)

- Sub-case **(ASSIGN)**:

$(S_1; l := v_1) \longrightarrow (S_1[l \mapsto v_1]; ())$ (5) - hyp
$(S_2; l := v_2) \longrightarrow (S_2[l \mapsto v_2]; ())$ (6) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ l \cong_{\mathsf{s}} \ l : \mathtt{ref}(\tau^{\mathsf{s}'})^{\mathsf{t}}$ (7) - inv. of (E-ASSIGN) with (4)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \ v_1 \cong_{\mathsf{s}} \ v_2 : \tau^{\mathsf{s}'}$ (8) - inv. of (E-ASSIGN) with (4)
$l \in dom(S_1)$ (9) - inv. of (ASSIGN) with (5)
$l \in dom(S_2)$ (10) - inv. of (ASSIGN) with (6)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} () \ \cong_{\mathsf{s}} \ () : \mathtt{cmd}^{\mathsf{s}'}$ by (E-VAL) and (E-SUB)
$\Delta \vdash S_1[l \mapsto v_1]$ by (1,2,3,8)
$\Delta \vdash S_2[l \mapsto v_2]$ by (1,2,3,8)
$S_1[l \mapsto v_1] =_{\mathsf{s}} S_2[l \mapsto v_2]$ by Def. 26 with (8)

□

Lemma 21 (Value-Step-Equivalence)
Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e{:}\tau^{\mathsf{s}'}$, $vars(\Delta) = \emptyset$, $\Delta \vdash S_1$, $\Delta \vdash S_2$ and $S_1 =_{\mathsf{s}} S_2$.
If $(S_2, e) \longrightarrow (S_2', e')$ then there is Δ' such that
$\Delta \subseteq \Delta'$, $\Delta' \vdash S_1$, $\Delta' \vdash S_2'$, $S_1 =_{\mathsf{s}} S_2'$ and $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e'{:}\tau^{\mathsf{s}'}$.

Proof Induction on the derivation of $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e{:}\tau^{\mathsf{s}'}$.

Case (E-EXPROPAQUE):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \tau^{\mathsf{s}'}$	(1) - hyp
$\Delta \vdash S_1$	(2) - hyp
$\Delta \vdash S_2$	(3) - hyp
$S_1 =_{\mathsf{s}} S_2$	(4) - hyp
$(S_2; e) \longrightarrow (S_2'; e')$	(5) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v : \tau^{\mathsf{s}'}$	(6) - inv. (E-EXPROPAQUE) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} e : \tau^{\mathsf{s}'}$	(7) - inv. (E-EXPROPAQUE) of (1)
$\mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}$	(8) - inv. (E-EXPROPAQUE) of (1)
$\Delta \subseteq \Delta'$	by Theorem 6 with (3,5,7)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_2} e' : \tau^{\mathsf{s}'}$	(9) - by Theorem 6 with (3,5,7)
$\Delta' \vdash S_2'$	(10) - by Theorem 6 with (3,5,7)
$\Delta' \vdash S_1$	(11) - by Definition 22
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'}$	by (E-EXPROPAQUE) with (6,8,10)
$S_2 =_{\mathsf{s}} S_2'$	(12) - by Lemma 19 with (3,5,7,8)
$S_1 =_{\mathsf{s}} S_2'$	by (5,12)

Case (E-RECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} [\ldots, m_k{=}v, \ldots] \cong_{\mathsf{s}} [\ldots, m_k{=}e, \ldots] : \Sigma[\ldots \times m_k{:}\tau_k^{\mathsf{s}_k} \times \ldots]^{\mathsf{t}}$	(1) - hyp, where $k \in \{1, \ldots, n\}$
$\Delta \vdash S_1$	(2) - hyp
$\Delta \vdash S_2$	(3) - hyp
$S_1 =_{\mathsf{s}} S_2$	(4) - hyp
$(S_2; [\ldots, m_k{=}e, \ldots]) \longrightarrow (S_2'; [\ldots, m_k{=}e', \ldots])$	(5) - hyp
$(S_2; e) \longrightarrow (S_2'; e')$	(5) - inv. (RECORD) of (5)
$\forall_i\ \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_i \cong_{\mathsf{s}} e_i : \tau_i^{\mathsf{s}_i}$	(6) - inv. (E-RECORD) of (1)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \tau_k^{\mathsf{s}_k}$	(7) - by (6)
$\Delta \subseteq \Delta'$	by I.H. with (5,7,4,2,3)
$\Delta' \vdash S_1$	by I.H. with (5,7,4,2,3)
$\Delta' \vdash S_2'$	by I.H. with (5,7,4,2,3)
$S_1 =_{\mathsf{s}} S_2'$	by I.H. with (5,7,4,2,3)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \tau_k^{\mathsf{s}_k}$	(8) - by I.H. with (5,7,4,2,3)
$\forall_i\ \Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_i \cong_{\mathsf{s}} e_i{:}\tau_i^{\mathsf{s}_i}$	(9) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (6)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} [\ldots, m_k{=}v, \ldots] \cong_{\mathsf{s}} [\ldots, m_k{=}e', \ldots] : \Sigma[\ldots \times m_k{:}\tau_k^{\mathsf{s}_k} \times \ldots]^{\mathsf{t}}$	

by rule (E-RECORD) with (8,9)

Case (E-INJ):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \#m(v) \cong_{\mathsf{s}} \#m(e) : \tau^{\mathsf{t}}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp
$(S_2; \#m(e)) \longrightarrow (S_2'; \#m(e'))$ (5) - hyp
$(S_2; e) \longrightarrow (S_2'; e')$ (5) - inv. (VARAINT) of (5)
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \tau_i^{\mathsf{s}_i}$ (6) - inv. (E-VARIANT) of (1)
$\Delta \subseteq \Delta'$ by I.H. with (5,6,4,2,3)
$\Delta' \vdash S_1$ by I.H. with (5,6,4,2,3)
$\Delta' \vdash S_2'$ by I.H. with (5,6,4,2,3)
$S_1 =_{\mathsf{s}} S_2'$ by I.H. with (5,6,4,2,3)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \tau_i^{\mathsf{s}_i}$ (7) - by I.H. with (5,6,4,2,3)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1',\mathcal{S}_2'} \#m(v) \cong_{\mathsf{s}} \#m(e') : \tau^{\mathsf{t}}$ by rule (E-VARIANT) with (7)

Case (E-REFINERECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp
$(S_2; e) \longrightarrow (S_2'; e')$ (5) - hyp
$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (6) - inv. (E-REFINERECORD) of (1)
$\mathcal{S}_1\{x \doteq v\} \models x.m_j \doteq \mathsf{v}$ and $\mathcal{S}_2\{x \doteq e\} \models x.m_j \doteq \mathsf{v}$ (7) - inv. (E-REFINERECORD) of (1)
$\mathcal{S}_2\{x \doteq e'\} \models x.m_j \doteq \mathsf{v}$ (8) - by (7,5) since reduction preserves $\doteq$, so $e \doteq e'$
$\Delta \subseteq \Delta'$ by I.H. with (2,3,4,5,6)
$\Delta' \vdash S_1$ by I.H. with (2,3,4,5,6)
$\Delta' \vdash S_2'$ by I.H. with (2,3,4,5,6)
$S_1 =_{\mathsf{s}} S_2'$ by I.H. with (2,3,4,5,6)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (9) - by I.H. with (5,7,4,2,3)
$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$
by rule (E-REFINERECORD) with (7,8,9)

Case (E-UNREFINERECORD):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$ (1) - hyp
$\Delta \vdash S_1$ (2) - hyp
$\Delta \vdash S_2$ (3) - hyp
$S_1 =_{\mathsf{s}} S_2$ (4) - hyp
$(S_2; e) \longrightarrow (S_2'; e')$ (5) - hyp

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$
(6) - inv. (E-UNREFINERECORD) of (1)

$\mathcal{S}_1\{x \doteq v\} \models x.m_j \doteq \mathsf{v}$ and $\mathcal{S}_2\{x \doteq e\} \models x.m_j \doteq \mathsf{v}$
(7) - inv. (E-UNREFINERECORD) of (1)

$\mathcal{S}_2\{x \doteq e'\} \models x.m_j \doteq \mathsf{v}$ (8) - by (7,5) since reduction preserves $\doteq$, so $e \doteq e'$

$\Delta \subseteq \Delta'$ by I.H. with (2,3,4,5,6)

$\Delta' \vdash S_1$ by I.H. with (2,3,4,5,6)

$\Delta' \vdash S_2'$ by I.H. with (2,3,4,5,6)

$S_1 =_{\mathsf{s}} S_2'$ by I.H. with (2,3,4,5,6)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(m_j)} \times \ldots]^{\mathsf{t}}$ (9) - by I.H. with (5,7,4,2,3)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j{:}\tau_j^{\mathsf{s}_j} \times \ldots \times m_i{:}\tau_i^{\ell_i(\mathsf{v})} \times \ldots]^{\mathsf{t}}$
by rule (E-UNREFINERECORD) with (7,8,9)

Case (E-SUB):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \tau^{\mathsf{s}'}$ (1) - hyp

$\Delta \vdash S_1$ (2) - hyp

$\Delta \vdash S_2$ (3) - hyp

$S_1 =_{\mathsf{s}} S_2$ (4) - hyp

$(S_2; e) \longrightarrow (S_2'; e')$ (5) - hyp

$\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \tau^{\mathsf{s}''}$ (6) - inv. (E-SUB) of (1)

$\tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'}$ and $\mathsf{r} \leq \mathsf{r}'$ (7) - inv. (E-SUB) of (1)

$\Delta \subseteq \Delta'$ by I.H. with (2,3,4,5,6)

$\Delta' \vdash S_1$ by I.H. with (2,3,4,5,6)

$\Delta' \vdash S_2'$ by I.H. with (2,3,4,5,6)

$S_1 =_{\mathsf{s}} S_2'$ by I.H. with (2,3,4,5,6)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''}$ (8) - by I.H. with (2,3,4,5,6)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'}$ by rule (E-SUB) with (7,8)

Case (E-COLLECTION):

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \{\ldots, v, \ldots\} \cong_{\mathsf{s}} \{\ldots, e, \ldots\} : \tau'^{*\mathsf{s}'}$ (1) - hyp

$\Delta \vdash S_1$ (2) - hyp

$\Delta \vdash S_2$ (3) - hyp

$S_1 =_{\mathsf{s}} S_2$ (4) - hyp

$(S_2; \{\ldots, e, \ldots\}) \longrightarrow (S_2'; \{\ldots, e', \ldots\})$ (5) - hyp

$(S_2; e) \longrightarrow (S_2'; e')$ (6) - inv. (COLLECTION) of (5)

$\forall_i\, \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_i \cong_{\mathsf{s}} e_i : \tau'^{\mathsf{s}'}$ (7) - inv. (E-COLLECTION) of (1)

$\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e : \tau'^{\mathsf{s}'}$ (8) - by (7)

$\Delta \subseteq \Delta'$ by I.H. with (2,3,4,6,8)

$\Delta' \vdash S_1$ by I.H. with (2,3,4,6,8)

$\Delta' \vdash S_2'$ by I.H. with (2,3,4,6,8)

$S_1 =_{\mathsf{s}} S_2'$ by I.H. with (2,3,4,6,8)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''}$ (9) - by I.H. with (2,3,4,6,8)

$\forall_i\ \Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_i \cong_{\mathsf{s}} e_i{:}\tau'^{\mathsf{s}'}$ (10) - by Weakining Lemma for $\cong_{\mathsf{s}}$ with (7)

$\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} \{\ldots, v, \ldots\} \cong_{\mathsf{s}} \{\ldots, e', \ldots\} : \tau'^{*\mathsf{s}'}$ by rule (E-COLLECTION) with (9,10)

□

2015

A Type System for Value-dependent Information Flow Analysis

Luísa Lourenço