Dependent Information Flow Types

Luísa Lourenço Luís Caires

CITI and NOVA Laboratory for Computer Science and Informatics
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Abstract

In this paper, we develop a novel notion of dependent information flow types. 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, 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, still considered a challenge to security enforcement in software systems such as data-centric web-based applications.

We base our development on the very general setting of a minimal λ-calculus with references and collections. We illustrate its expressiveness, showing how secure operations on relevant scenarios can be modelled and analysed using our dependent information flow type system, which is also shown to be amenable to algorithmic type checking. Our main results include type-safety and non-interference theorems ensuring that well-typed programs do not violate prescribed security policies.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection – Information Flow Controls; F.3.3 [*Logics and Meanings of Programs*]: Type Structure

Keywords Information Flow; Dependent Type Systems

1. Introduction

Among the main challenges currently posed to software development one inevitably finds security enforcement, particularly relevant for the construction of data-centric applications to be made available on the web, or on mobile computing environments. Many reported security leaks in globally used information systems, including email management or social network systems, turn out in many cases to result from programming errors introducing insecure information flows in complex and heterogeneous software layers. A relevant research issue consists on finding improved techniques to ensure trustworthy confidentiality in the presence of multi-tenancy and container sharing, as e.g., when logically different security compartments are not statically mapped to different physical or data structure compartments, but are instead dynamic and dependent on runtime data, including on configuration parameters.

In this paper, we develop the notion of *dependent information flow types*. Dependent information flow types provide a direct, natural and elegant way to express and statically enforce fine grained security policies on programs, 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, still considered a challenge to security enforcement in software systems such as data-centric web-based applications. In standard information flow type systems [1, 8, 10, 12, 28], 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 $(\mathtt{int}^{\top} \rightarrow \mathtt{int}^{\top})^{\bot}$ 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 [17], instead of a plain security label, as described above. To achieve this goal, we present in this paper a novel theory of dependent information flow types within the framework of dependent type theory, including sum and function dependent types, aiming to capture the general essence of value dependent security classification. A simple example of a dependent (function) information flow type is

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

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 $\mathtt{usr}$("pat"), where $\mathtt{usr}(x)$ is a value dependent security label. For another simple example, consider the following dependent (labeled product) information flow type:

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

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 $\mathtt{usr}(x)$, 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 $\mathtt{usr}$("joe") and $\mathtt{usr}$("pat") to be incomparable, thus avoiding insecure infor-

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676994

mation 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 type. The type assigned to such a collection would be:

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

(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 (the **`foreach`** iterator is a familiar functional collection fold combinator [3]; `x` is the current item/cursor and `acum` denotes the value accumulated from previous iteration, with initial value $\{\}$). 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:

$$\texttt{getPasswords} : \Pi u{:}\texttt{string}^{\perp}.\texttt{string}^{*\texttt{usr}(u)}$$

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.

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, 12]), 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 `r` (the "program counter"). The additional component $\mathcal{S}$ represents a set of the equational constraints, used to refine label indices, and establish type equality.

We summarise the contributions of this paper. First, we motivate, introduce, and develop a type-based information flow analysis for a λ-calculus with references and collections. We build on a new notion of dependent information flow types, developed along the lines of a standard dependent type theory but where the security level (not just the structure) of data types may depend on values (Section 2). Second, we illustrate the expressiveness of our language and analysis using several examples (Section 3). Third, we present our main technical results: type safety (preservation and progress) (Theorems 5.4 and 5.5) and a non-interference (Theorem 5.11), which together imply that well-typed programs do not break data confidentiality according to the security policies prescribed by dependent information flow types (Section 5). After discussing algorithmic type checking (Section 6), we overview key related work (Section 7), and offer some concluding remarks (Section 8).

2. Dependent Information Flow Types

In this section, we motivate dependent (function and sum) information flow types by means of several examples, providing an informal overview of the approach developed in the paper.

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 (which concerns access control). 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 levels 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), as shown in Figure 1d for the types declared in Figure 1a: `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 2.1 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 2.2 Operation `viewAuthorPapers` iterates the `Submissions` collection to build a list of all records with a given author id

```
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
```

Example 2.3 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
```

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

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

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

(a) Purely structural types (not expressing security levels)

$\tau_b \stackrel{\text{def}}{=} [\texttt{uid} : \bot \times \texttt{name} : \texttt{U} \times \texttt{univ} : \texttt{U} \times \texttt{email} : \texttt{U}]$

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

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

(b) Standard security types for information flow (e.g., [1, 12]) - we omit base types

$\tau_c \stackrel{\text{def}}{=} \Sigma[\texttt{uid} : \bot \times \texttt{name} : \texttt{U(uid)} \times \texttt{univ} : \texttt{U(uid)} \times \texttt{email} : \texttt{U(uid)}]$

$\sigma_c \stackrel{\text{def}}{=} \Sigma[\texttt{uid} : \bot \times \texttt{sid} : \bot \times \texttt{title} : \texttt{A(uid,sid)} \times \texttt{abs} : \texttt{A(uid,sid)} \times \texttt{paper} : \texttt{A(uid,sid)}]$

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

(c) Dependent information flow types

$\texttt{let Users} = \texttt{ref}_{\texttt{ref}(\tau_{(\cdot)})^*\bot}\ (\texttt{ref}_{\tau_{(\cdot)}}\ [\,]\ )::\{\}\ \texttt{in}$

$\texttt{let Submissions} = \texttt{ref}_{\texttt{ref}(\sigma_{(\cdot)})^*\bot}\ (\texttt{ref}_{\sigma_{(\cdot)}}\ [\,]\ )::\{\}\ \texttt{in}$

$\texttt{let Reviews} = \texttt{ref}_{\texttt{ref}(\delta_{(\cdot)})^*\bot}\ (\texttt{ref}_{\delta_{(\cdot)}}\ [\,]\ )::\{\}$

(d) Declaration of (mutable) collections of mutable records (replace $(\cdot)$ with a, b or c for according type)

Figure 1: Expressing security policies

The **foreach** primitive computes the accumulated value of a list's elements. 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, 12]), both a type τ and a security label s are assigned to expressions by our typing judgment $\Delta \vdash e{:}\tau^{\texttt{s}}$, expressing the fact that the value of e will only be affected by computations interfering at security levels $\leq$ s. 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 for the data that can be disclosed to any registered user; A for data observable to authors; and PC for data that only programme committee members can see. In such simple case, we may let $\bot$<U<A<PC<$\top$ and specify the according security policy for each conference manager entity as shown in Figure 1d for the types declared in Figure 1b. Such policy states that a registered user's information is observable from security level U, meaning any registered user (including authors and PC members) can see it; that 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 and 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 2.2) using a different id than his own, which clearly violates confidentiality. Thus, the security policy that we need is the following: 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 that comments to the PC can *only* be observable to the other members that are *also reviewers of the submission*, and that comments and grade of the submission can be seen by *its author 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 [17]. Using indexed security labels we may partition 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 (with id 70).

Going back to our example, we assume the following defined security levels: U(uid), for registered users with id uid; A(uid,sid), for author of submission with id sid and whose user id is uid; and PC(uid,sid), 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 $\ell(\overline{v}, u, \overline{w}) \leq \ell(\overline{v}, \top, \overline{w})$ and $\ell(\overline{v}, \bot, \overline{w}) \leq \ell(\overline{v}, u, \overline{w})$, for all u, $\overline{v}$ and $\overline{w}$. So, e.g, for all uid we have U($\bot$) $\leq$ U(uid) $\leq$ U($\top$); we can see U($\top$) as the approximation (from above) of any U(uid), e.g, standing for the standard label U (Figure 1b). Levels A($\bot$,$\bot$), and PC($\bot$,$\bot$) stand for the security compartment accessible to any author and any PC member, respectively; but A($\top$,$\top$), and PC($\top$,$\top$) respectively represent the compartments containing the information of all authors and all PC members; level A(uid,$\top$) stands for registered users with id uid that are authors; A($\top$,sid) represents the security compartment of authors of submission with id sid; A(uid,$\bot$) means a registered author with no authority over submitted papers; and so on. We define the declarations in Figure 1d for the types declared in Figure 1c, and the partial order defining the sample security lattice by the following axioms (quantifiers ranging over natural numbers):

$\forall$uid,sid. U(uid)$\leq$ A(uid,sid)

$\forall$uid1,uid2,sid. A(uid1,sid)$\leq$ PC(uid2,sid)

We proceed our motivation of dependent information flow types with a series of code snippets relevant for our conference manager scenario.

Consider then the following code

Example 2.4

```
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 (Figure 1c). 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 already explained in the Introduction). For example, notice that the security level of the `title` field 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 the record initialising the reference `new_rec`, we need to obtain type `[uid:`$\bot\times$`sid:`$\bot\times$`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 illegal flow of information.

For brevity, as in the example above, 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` as `uid=t_sub.uid`, e.g, `sid=t_sub.sid`, etc. Consider now the following code fragment

```
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
```

The result of evaluating this program is 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 `[uid:`$\bot$ $\times$ `sid:`$\bot$ $\times$ `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 Σ`[uid:`$\bot$ $\times$ `sid:`$\bot$ $\times$ `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 `A(42,sid)`$\leq$ `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 (uid_a).
  foreach(x in !Users) with y = {} do
    let usr = !x in
      if usr.uid = uid_a 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 Π`(uid_a:`$\bot$`).` Σ`[uid:`$\bot\times$`name:U(uid_a)`$\times$`univ:U(uid_a)`$\times$`email:U(uid_a)]`*. Notice that the return type depends on the value of the function argument, so the type of `viewUserProfile` is a functional dependent type. Then the expression **first**`(viewUserProfile(42)).email` has type `U(42)`.

Example 2.5 The `addCommentSubmission` operation is used by the PC members to add comments to submissions. The types ensure that only PC members assigned to the particular paper will see such comments.

```
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 (`Π(uid_r:`$\bot$`).C`, where type `C` is Σ`[uid:`$\bot\times$`sid:`$\bot\times$`title:A(uid,sid)`$\times$`abs:A(uid,sid)` $\times$ `paper:A(uid,sid)]`* (Example 2.3). Since there is no dependency (`uid_r` not free in `C`), we may abbreviate the functional type by `Int`$^{\bot}\!\!\rightarrow$ `C`, thus identifier `p` has type `C`. Function *comment* returns a given paper's PC comments, and has type Π `u:`$\bot$`.`Π`s:`$\bot$`.`Π`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 the type of the collection's references (type δ_c of Figure 1c). 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 `A(`$\top$`,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 `A(`$\top$`,p.sid)` and thus, due to `A(`$\top$`,p.sid)` $\leq$ `PC(`$\bot$`, p.sid)`, we can up-classify *comment*`(p.uid, p.sid,p)` with `PC(t_rev.uid, p.sid)`. Notice that this up-classification is only possible to PC members assigned to the paper whose `sid` is `p.sid`, so only those PC members will be able to see the added comment. So we can, finally, type the record `up_rec` with the dependent sum type Σ `[uid:`$\bot$ $\times$`sid:`$\bot$ $\times$`PC_only:PC(uid,sid)` $\times$`review:A(`$\top$`,sid)` $\times$`grade:A(`$\top$`,sid)]`. We refer back to this example in Example 4.3, 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 `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.

After introducing our core language and dependent information flow types, we now formally introduce its syntax and semantics.

$e ::=$		(*expression*)					$c ::=$		(*conditions*)
	$\mid e.m$	(field access)						$\mid \neg c$	(negation)
	$\mid$ **let** $x = e_1$ **in** e_2	(let)						$\mid c_1 \vee c_2$	(disjunction)
	$\mid e_1(e_2)$	(application)	$v ::=$			(*values*)		$\mid V = V$	(equality)
	$\mid$ **if** c **then** e_1 **else** e_2	(conditional)		$\overline{v}$		(collection)		$\mid V$	(term)
	$\mid [\overline{m = e}]$	(record)		$\mid [\overline{m = v}]$		(record)	$V ::=$		(*terms*)
	$\mid \overline{e}$	(collection)		$\mid \lambda(x{:}\tau^{\mathsf{s}}).e$		(abstraction)		$\overline{V}$	(collection)
	$\mid e_1 :: e_2$	(cons)		$\mid$ **true**		(true)		$\mid [\overline{m = V}]$	(record)
	$\mid$ **foreach**$(e_1, e_2, x.y.e_3)$	(iteration)		$\mid$ **false**		(false)		$\mid \lambda(x{:}\tau^{\mathsf{s}}).e$	(abstraction)
	$\mid$ **ref**$_{\tau^s}$ e	(reference)		$\mid ()$		(unit)		$\mid$ **true**	(true)
	$\mid e_1 := e_2$	(assign)		$\mid l$		(locations)		$\mid$ **false**	(false)
	$\mid\ !e$	(deref)						$\mid x$	(identifier)
	$\mid x$	(identifier)						$\mid ()$	(unit)
	$\mid v$	(value)						$\mid V.m$	(field access)

Figure 2: Syntax: expressions, values, conditions, terms

3. Core Language

Our core language (Figure 2) is a higher-order functional language with references and primitive collections. Primitive values (v) include abstractions, booleans, records, collections (lists of values, also present in languages such as DMinor [3]), and locations. We consider two values v, u equal, $v = u$, when they are the same up to reordering of record labels, and assume intensional equality on lambda abstractions. Expressions (e) include let-expression, conditional, application, field access, cons operator to add an element to a collection, collection iterator, values, variables, locations, references, dereference and assignment. These primitives are to be understood as usual, namely the collection iterator **`foreach`** computes the accumulated value of a list's elements, as showed in the examples of Section 2. For readability purposes, we sometimes use the more natural concrete syntax adopted in the previous section. We assume other basic data types (integers, strings) and associated operations, such as **`first`**`(-)` and **`rest`**`(-)` for collections (but omit standard details). To illustrate primitive types, we will usually include just booleans in our formal presentation. As for abbreviations, we use an overbar to represented indexed sets of syntactic elements. Concretely, $[\overline{m = e}]$ stands for $[m_1{=}e_1, \ldots, m_n{=}e_n]$, and $\overline{e}$ stands for $\{e_1, \ldots, e_n\}$. Logical conditions c are used in conditionals, which we require to be pure (side-effect free). We also consider terms V, used in conditions, e.g., terms are values that may be checked for equality. Pure expressions are those side-effect free, in concrete, all expressions except assignment, reference expressions and deference.

Programs in our language are closed expressions. The operational semantics is defined using a reduction relation, defined by the rules in Figure 3, which basically extend the standard ones for a call-by-value λ-calculus. Reduction is defined between configurations of the form $(S; e)$, where S is a store, and e is an expression. A reduction step of the form $(S; e) \longrightarrow (S'; e')$ states that expression e under state S evolves in one computational step to expression e' under state S'. A store S is a mapping from *locations* to values. Reduction rules follow predictable lines, we briefly comment on the less standard ones. Rules (*foreach-left*) and (*foreach-right*) reduce the first and second expressions of the list iterator operator, respectively. These rules, together with rule (*foreach*), imply an evaluation order from left to right. Evaluation of (store independent) logical conditions is given by the map $\mathcal{C} : c \to \{\mathbf{true}, \mathbf{false}\}$, using an auxiliary evaluation map for closed terms $\mathcal{T} : V \to v$, defined as expected (see [34] for more details).

4. Type System

As already discussed above, our type system for information flow builds on fairly traditional concepts from information flow type systems [1, 12], 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 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. As in typical type-based information flow analyses, types are annotated with a security label. In our system, we consider value dependent security labels $\ell(\overline{\mathsf{v}})$, so our types take the form τ^{s} where τ is a data type and s is a (indexed) security label $\ell(\overline{\mathsf{v}})$ - non-indexed labels may be represented by $\ell(\top)$.

If an expression e is assigned type τ^{s} then the system must ensure that only users with enough permissions to read information at security level s 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 an user. As for the attacker model, we assume an attacker can observe information, including stored data, that has security level $\bot$ (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 s can only observe information classified up to s.

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

$\mathsf{v} ::=$		`(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 call *concrete* a security label if all of its indexes is $\bot$,$\top$, or a value v, not a (record) field identifier m or a variable x. As made clear 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.

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 suggested above, 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

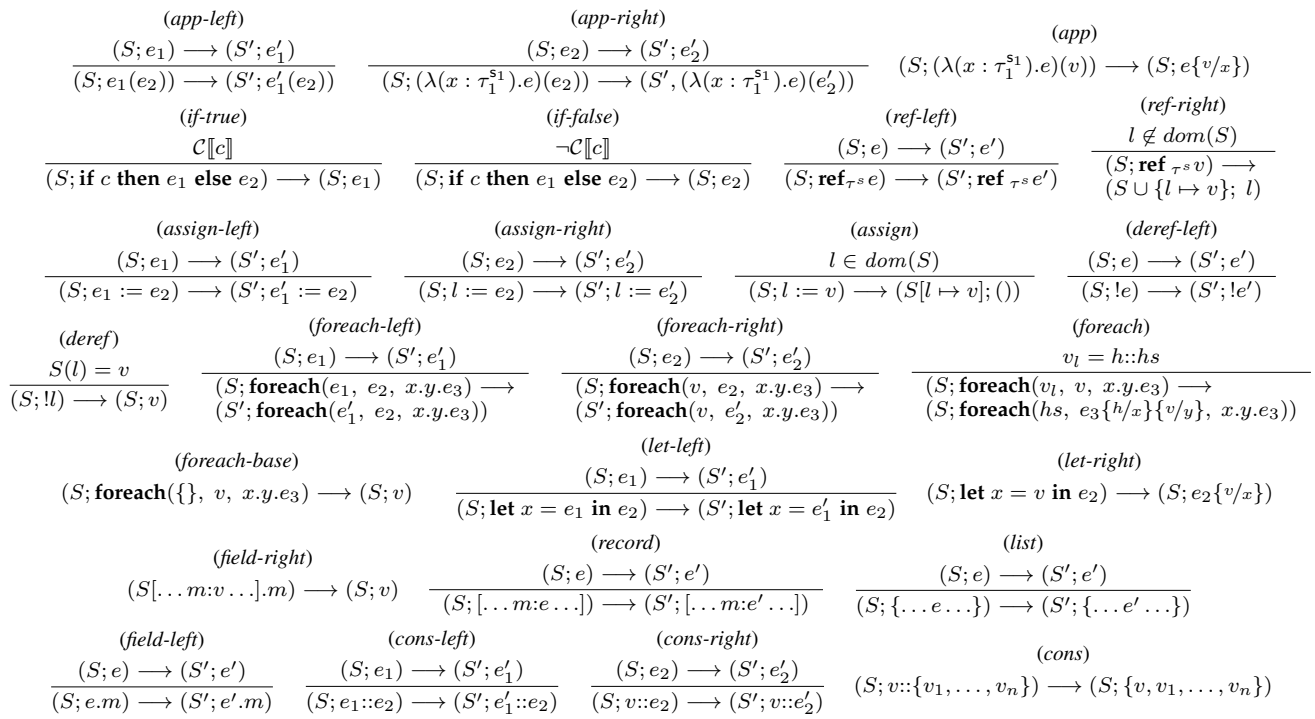

Figure 3: Reduction

$\mathsf{s},\mathsf{r},\mathsf{t},\mathsf{q} ::=$	$\ell(\overline{\mathsf{v}})$	security label
$\tau^{\mathsf{s}},\sigma^{\mathsf{s}} ::=$	$\mid \mathtt{Bool}^{\mathsf{s}}$	bool type
	$\mid (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r};\sigma^{\mathsf{q}})^{\mathsf{t}}$	dependent function type
	$\mid \Sigma[m:\tau^{\mathsf{s}}]^{\mathsf{r}}$	dependent sum type
	$\mid \tau^{*\mathsf{s}}$	collection type
	$\mid \mathtt{ref}(\tau^{\mathsf{s}})^{\mathsf{t}}$	reference type
	$\mid \mathtt{cmd}^{\mathsf{s}}$	command type

Figure 4: Types

$\mathcal{L}$, and that the ordering between labels is well defined and satisfies the lattice property (i.e., well defined meets and joins, etc). We assume that the intended lattice (required for each particular security analysis) is specified by a list 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 $\overline{u},\overline{v}$. We already gave examples of lattice assertions in Section 2.

Types. The type structure (Figure 4) includes boolean, unit (or command), reference, dependent sum, dependent function, and collection types. In collection type $\tau^{*\mathsf{s}}$ each collection element has type τ^{s}. We assume other basic types, such as integers and strings with their associated operations, not formalised for the sake of simplicity, but used in examples. Dependent sum types and dependent functional types assume a key role in our type system as they allows us to express (runtime) value dependency on security labels, as already highlighted. A dependent sum type has the form

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

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

$$\Sigma[\mathtt{uid}{:}\bot \times \mathtt{name}{:}\mathtt{user}(\mathtt{uid})]^{\bot}$$

is a dependent sum type where field `name` has the value dependent security level, `user(uid)`, which is indexed by the (runtime) value in field `uid`. A dependent functional type has the general form

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

where the (security level of) the codomain type σ^{q} may depend on the value of the argument (denoted by the bound variable x). Level r specifies a lower bound on the security level of the function effects (writes), if omitted it is assumed to be $\bot$. 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 $\bot$. 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.

Dependencies in Security Labels. As stated earlier, and for natural reasons, 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 example, should we project field `name` of a record typed with $\Sigma[\mathtt{uid} : \bot \times \mathtt{name} : \mathtt{user}(\mathtt{uid})]^{\bot}$, 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 $\Sigma[\mathtt{uid} : \bot \times \mathtt{name} : \mathtt{user}(\mathtt{uid})]^{\bot}$ and then initialise with a record with type $\Sigma[\mathtt{uid} : \bot \times \mathtt{name} : \mathtt{user}(0)]^{\bot}$, then we would need to introduce the field dependency in `user(0)`. We achieve such introduction and elimination of dependencies in security labels by: (1) tracking knowledge regarding dependencies in a constraint set $\mathcal{S}$ carried along in typing judgements; (2) using a equational theory to entail runtime values or dependencies, depending whether we are eliminating or capturing dependencies in security labels.

A constraint set $\mathcal{S}$ is a finite set of equational 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 $S \models e \doteq e'$

$$\frac{\ell(\top) \leq \mathsf{s}}{\tau^{\ell(\mathsf{v})} <: \tau^{\mathsf{s}}}\ \textit{(s-indexLeft)} \qquad \frac{\mathsf{s} \leq \ell(\bot)}{\tau^{\mathsf{s}} <: \tau^{\ell(\mathsf{v})}}\ \textit{(s-indexRight)} \qquad \frac{\forall_i\ \tau_i^{\mathsf{s}_i} <: \tau_i^{\mathsf{s}_i'} \quad \mathsf{t}' \leq \mathsf{t}}{\Sigma[\overline{m : \tau^{\mathsf{s}}}]^{\mathsf{t}} <: \Sigma[\overline{m : \tau^{\mathsf{s}'}}]^{\mathsf{t}'}}\ \textit{(s-record)}$$

$$\frac{\mathsf{s} \leq \mathsf{s}'}{\tau^{\mathsf{s}} <: \tau^{\mathsf{s}'}}\ \textit{(s-expr)} \qquad \frac{\tau^{\mathsf{s}'} <: \tau^{\mathsf{s}} \quad \sigma^{\mathsf{q}} <: \sigma^{\mathsf{q}'} \quad \mathsf{r}' \leq \mathsf{r}}{(\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}; \sigma^{\mathsf{q}})^{\mathsf{t}} <: (\Pi x{:}\tau^{\mathsf{s}'}.\mathsf{r}'; \sigma^{\mathsf{q}'})^{\mathsf{t}}}\ \textit{s-arrow}$$

Figure 5: (Key) Subtyping rules

for the entailment of $e \doteq e'$ given the constraints in $\mathcal{S}$. We also naturally require $\doteq$ to be compatible with reduction in the sense that for any e, e' pure if $(S; e) \longrightarrow (S; e')$ then $\models e \doteq e'$. As for any equational theory, we assume that $S \models E$ and $\mathcal{S} \cup \{E\} \models E'$ implies $S \models E'$ (deduction closure). 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 instance, $\{x.\texttt{uid} \doteq \texttt{uid\_r}, \texttt{uid\_r} \doteq 42\} \models x.\texttt{uid} \doteq 42$. 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 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).

Typing Judgements and Rules. A typing judgment has 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" [18, 22]), a familiar technique to prevent implicit flows. 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. In Figure 6 we show the typing rules. The system also relies on a simple subtyping relation, denoted $<:$, which allows up-classification of security labels, defined in Figure 5. Rules *s-indexLeft/Right/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. We also define well-formed types and well-formedness definition of typing contexts, denoted as as $\Delta \vdash \diamond$. 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 $\mathcal{N}$. The full set of rules defining well-formed types can be found in [34].

We avoid commenting on typing rules standard for any typed λ-calculus and focus on key typing rules specific to our system. For simplicity and w.l.o.g we consider in our presentation that security labels are indexed by a single label index, assuming the obvious extension of type rules to deal with labels with multiple indexes, when necessary, e.g, in examples. In rule (*t-record*) we require 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. Since a field's security label s may have dependencies, we approximate the values they denote with $\bot$, via $|\mathsf{s}|^{\downarrow}$. 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 will get back to this discussion below. Rule (*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 $\mathsf{v} = \top$. In rule (*t-foreach*), we require the security level of all sub expressions to be the same. This is required 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.

Example 4.1 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. On the other hand, assume `boxed` to be a collection of records typed as $\texttt{boxed} : (\Sigma[\texttt{secret} : \texttt{string}^{\top}])^{*\bot}$. All 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, it is possible to 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 spec, it would be allowed to a low observer to observe the collection size, but not the contents of the `secret` fields, preserving non-interference.

Rule (*t-if*) is as expected: 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. 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*) 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*), 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 4.2 Recall the `viewAuthorPapers` from Example 2.2,

```
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
```

While typing expression `tuple::y`, while typing the **then** branch, we obtain type Σ`[uid:`$\bot$ $\times$ `sid:`$\bot$ $\times$ `title:A(uid,sid)` $\times$ `abs:A(uid,sid)` $\times$ `paper:A(uid,sid)]`$^{*\bot}$. However, at this point, we know that `tuple.uid = uid_a`, which was added to the constraint set $\mathcal{S}$ according to rule (*t-if*). So, to type `tuple`, we can apply rule (*t-unrefineRecord*), adding a new constraint $\{x \doteq \texttt{tuple}\}$ for a fresh identifier x, and entail
$\mathcal{S} \cup \{\texttt{tuple.uid=uid\_a} \doteq \texttt{true},\ x \doteq \texttt{tuple}\} \models x.\texttt{uid} \doteq \texttt{uid\_a}$
to eliminate the field dependency `uid` in the security label, obtaining the type Σ`[uid:`$\bot$ $\times$ `sid:`$\bot$ $\times$ `title:A(uid_a,sid)` $\times$ `abs:A(uid_a,sid)` $\times$ `paper:A(uid_a,sid)]`$^{\bot}$.

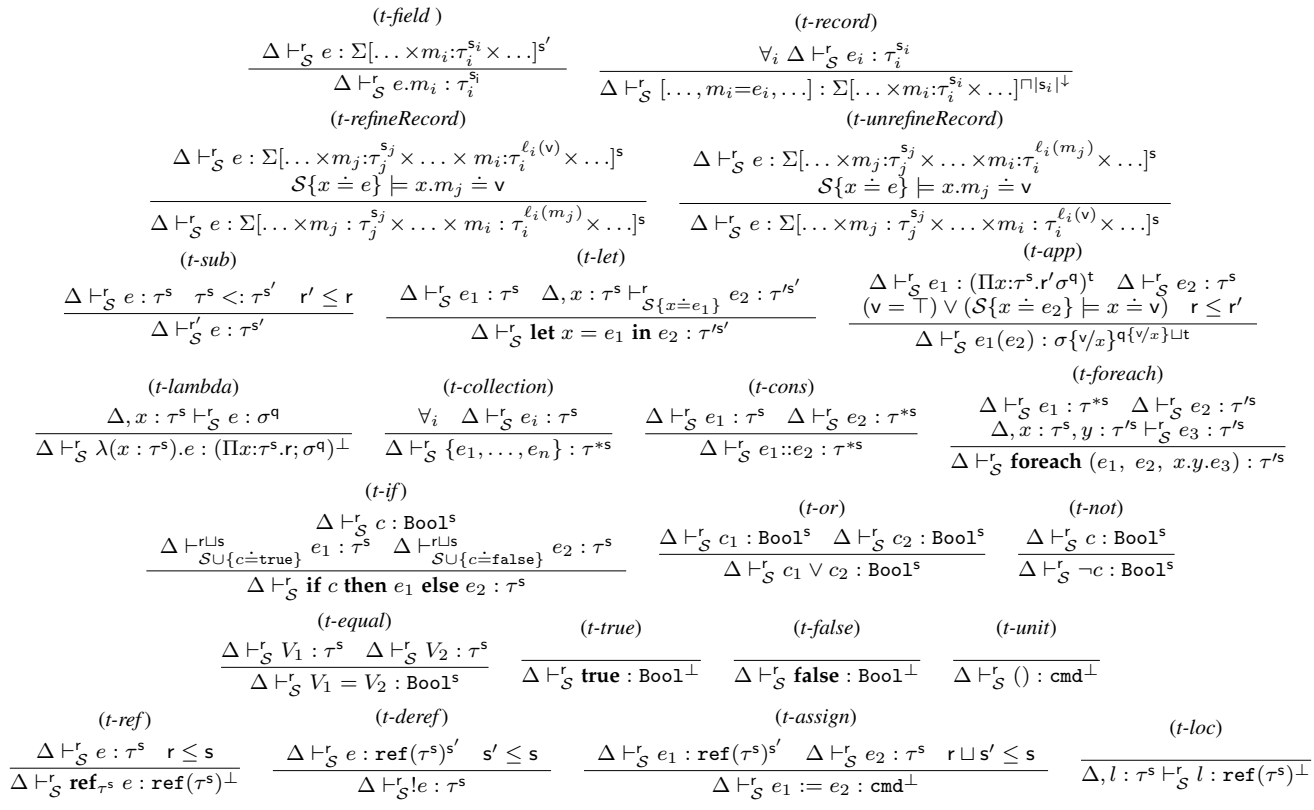

Figure 6: Typing Rules

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 Π `(uid_a:⊥).Σ[uid:⊥× sid:⊥× title:A(uid_a,sid)× ...]*⊥`.

Example 4.3 We now refer back to Example 2.5. 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 `y:= up_rec` we need to type `up_rec` with the declared type for the elements of collection `Reviews`, which is type δ_c in Figure 1c. As shown in Example 2.5, identifier `p` has type

`Σ[uid:⊥ × sid:⊥ × ... × title:A(uid,sid)]`

The type for *comment*`(p.uid,p.sid,p)` has level `A(p.uid,p.sid)`. By the lattice order `A(p.uid,p.sid)≤ PC(⊥,p.sid)` and by (*t-indexRight*) we get `PC(⊥,p.sid) ≤ PC(t_rev.uid,p.sid)`, hence `A(p.uid,p.sid)≤ PC(t_rev.uid,p.sid))`. We may then type the `PC_only` record field with `PC(t_rev.uid,p.sid)`. At this point, we tracked the following knowledge (via conditionals and let-expressions) into the following constraint set:

```
{ p.sid ≐ sid_r, t_rev.sid ≐ p.sid,
  up_rec ≐ [uid = t_rev.uid, sid = t_rev.sid,
            PC_only = comment(p.uid,p.sid,p), ...] }
```

Let us typecheck `up_rec` in the context of the assignment `y:= up_rec`. After typing the record value, we eliminate field dependencies, adding to $\mathcal{S}$ the constraint $\{x \doteq$ `up_rec`$\}$, for a fresh x. Then, we can derive the entailments

$\mathcal{S}, x \doteq$ `up_rec` $\models x.$`sid` $\doteq$ `p.sid`

$\mathcal{S}, x \doteq$ `up_rec` $\models x.$`uid` $\doteq$ `t_rev.uid`.

So we refine the type of the `PC_only` field to `PC(uid,sid)` by (*t-refineRecord*) and type the record bound to `up_rec` with the dependent type `Σ[uid:⊥,sid:⊥, PC_only:PC(uid,sid), ...]`, thus matching the expected type of reference `y`.

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 4.3, we retrieve essential precision in our analysis, required to obtain the correct typing for `PC_only`, `PC(uid,sid)`, and to typecheck function `addCommentSubmission`.

We briefly overview the remaining typing rules. Rule (*t-ref*) imposes a lower bound on the security level of the expression initializing the reference allocation to the computational context security level. Otherwise, illegal implicit flows could occur. For instance, in **`let`** `x = (`**`if`** `high` **`then`** $\mathbf{ref}_{\tau^{\perp}}$ `low)` **`in`** `!x`, we would be able to observe that a new reference was allocated. Rule (*t-deref*) sets the reference's security level as the lower bound for the deference's security level, to prevent implicit flows since references are typed initially at security level $\perp$ but may raise to a different security level, given the computational context. For instance,

`let` `x =` $\mathbf{ref}_{\tau^{\perp}}$ `low` **`in`** `(let y = (if high then x else ref`$^{\top}_{\tau}$ `0) in !y)`, would leak the value of `high` at low level, but does not typecheck . In (*t-assign*), we require that the least upper bound of the computational context security label and the reference's security level to be the lower bound of the content's security level. This way

one can safely type basic values and commands at the $\bot$ level, even at higher computational contexts.

5. Type Preservation and Non-Interference

In this section we present our main technical results: Theorem 5.4 (Type Preservation) - types are preserved by the reduction relation; Theorem 5.5 (Progress) - well-typed expressions are either a value or have a reduction step; and Theorem 5.11 (Non-interference) - well-typed expressions preserve non-interference. Our results establish that our system ensures that 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. For detailed proofs see [34].

Type Safety. We start by introducing some preliminary definitions.

Definition 5.1 (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 $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} S(l) : \Delta(l)$.

Definition 5.2 (Well-typed Configuration) A configuration $(S; e)$ is well-typed in typing environment Δ if $\Delta \vdash S$ and $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$.

Theorem 5.4 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.

Lemma 5.3 (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 $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e\{^{v}/_{x}\}{:}(\tau^{\mathsf{s}})\{^{v}/_{x}\}$.

Theorem 5.4 (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 $\Delta \subseteq \Delta'$.

Theorem 5.5, states that well-typed programs never get stuck.

Theorem 5.5 (Progress)
Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}} e{:}\tau^{\mathsf{s}}$ and $\Delta \vdash S$. If e is not a value then $(S; e) \longrightarrow (S'; e')$.

Noninterference Theorem. To develop our main noninterference result, we now introduce some relevant concepts. The first is the relation of store equivalence up to a security level s. Auxiliary function $redact(\Delta, S, \mathsf{s})$ returns the store obtained by "redacting" (replacing with a dummy value $\star$) all stored values in S with security level higher than s, or incomparable with s (detailed definitions in [34]). 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 5.6 (Store Equivalence) Let $\Delta \vdash S_1$ and $\Delta \vdash S_2$. Then S_1 is equivalent to S_2 up to level s (written $S_1 =_{\mathsf{s}} S_2$) if and only if $redact(\Delta, S_1, \mathsf{s}) = redact(\Delta, S_2, \mathsf{s})$.

Example 5.7 Assume `user(42)#user(666)`, and let S_1 and S_2 be stores containing location `private_file`. Also, let `private_file` have type Σ`[uid:`$\bot$ $\times$ `content:user(uid)]` under typing environment Δ, which also types both stores.
Then if S_1`(private_file) = {(42, "xpto"), (666, "fire")}`, and S_2`(private_file) = {(42, "puppies"), (666, "fire")}`. We have $S_1 =_{\texttt{user(666)}} S_2$ since the values `"xpto"` and `"puppies"`, classified as `user(42)`, are not visible at level `user(666)`, because $redact(\Delta, S_1$`,user(666)) = {(42,`$\star$`),(666,"fire")} =` $redact(\Delta, S_2$`,user(666))`.
However, S_1 and S_2 are not equivalent at security level `user(42)`, $S_1 =_{\texttt{user(42)}} S_2$, since the values `"xpto"` and `"puppies"` are visible at level `user(42)`.
In fact, $redact(\Delta, S_1$`,user(42)) = {(42,"xpto"), (666,`$\star$`)}` $\neq$ `{(42,"puppies"),(666,`$\star$`) }=`$redact(\Delta, S_2$`,user(42))`.

(*e-val*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} v : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} v : \tau^{\mathsf{s}'}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} v : \tau^{\mathsf{s}'}}$$

(*e-exprOpaque*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1} e_1 : \tau^{\mathsf{s}'} \quad \Delta \vdash^{\mathsf{r}}_{\mathcal{S}_2} e_2 : \tau^{\mathsf{s}'} \quad \mathsf{s} < \mathsf{s}' \sqcap \mathsf{r}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2 : \tau^{\mathsf{s}'}}$$

(*e-refineRecord*)

$$\frac{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(\mathsf{v})}, \ldots]^{\mathsf{s}'} \quad \mathcal{S}_1\{x \doteq e\} \models x.m_j \doteq \mathsf{v} \quad \mathcal{S}_2\{x \doteq e'\} \models x.m_j \doteq \mathsf{v}}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \Sigma[\ldots \times m_j : \tau_j^{\mathsf{s}_j} \times \ldots \times m_i : \tau_i^{\ell_i(m_j)}, \ldots]^{\mathsf{s}'}}$$

(*e-sub*)

$$\frac{\Delta \vdash^{\mathsf{r}'}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}''} \quad \tau^{\mathsf{s}''} <: \tau^{\mathsf{s}'} \quad \mathsf{r} \leq \mathsf{r}'}{\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e \cong_{\mathsf{s}} e' : \tau^{\mathsf{s}'}}$$

Figure 7: Equivalence of expressions up to level s (sample rules)

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.

Definition 5.8 (Expression Equivalence) Expression equivalence of e_1 and e_2 up to s is asserted by $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2{:}\tau^{\mathsf{s}'}$.

Notice that two expressions may be equivalent up to level s even if they are typed at a different level s'. We show key rules for expression equivalence in Figure 7 (see [34] for the complete definition). We may now present our non-interference theorems.

Lemma 5.9 (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 $(S_1, e_1) \rightarrow (S_1', e_1')$ and $(S_2, e_2) \rightarrow (S_2', e_2')$, 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 5.9 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.

Lemma 5.10 (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 $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v \cong_{\mathsf{s}} e'{:}\tau^{\mathsf{s}'}$ (and symmetrically).

We can then prove our non-interference theorem:

Theorem 5.11 (Non-interference) *Let $\Delta \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} e_1 \cong_{\mathsf{s}} e_2{:}\tau^{\mathsf{s}'}$, with $vars(\Delta) = \emptyset$, $\Delta \vdash S_1$, $\Delta \vdash S_2$ and $S_1 =_{\mathsf{s}} S_2$. 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 $\Delta \subseteq \Delta'$, $\Delta' \vdash S_1'$, $\Delta' \vdash S_2'$, $S_1' =_{\mathsf{s}} S_2'$ and $\Delta' \vdash^{\mathsf{r}}_{\mathcal{S}_1,\mathcal{S}_2} v_1 \cong_{\mathsf{s}} v_2{:}\tau^{\mathsf{s}'}$.*

Proof: By induction on $m + n$, using Lemma 5.9 in the case $m > 0$ and $n > 0$. For $m = 0$ and $n > 0$ we rely on Lemma 5.10.

Suppose we apply theorem Theorem 5.11 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 *e-exprOpaque* is not applicable to derive $\Delta' \vdash^{\mathsf{r}}_{S_1,S_2} v_1 \cong_{\mathsf{s}} v_2{:}\tau^{\mathsf{s}'}$). 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 $\mathsf{l} \not\leq s$.) We now illustrate our noninterference results.

Example 5.12 Recall our conference manager from Section 2, 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 := [uid = 42, sid = p.sid, name = p.name,
            unit = p.univ, email = p.email] :: !Users
```

Since the new record value is associating information of security level `U(42)` (value `p`) with user id `42`, this program should be deemed secure and the noninterference property checked.

We 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.
Assume `U(42)#U(666)`, and let S_1 and S_2 be stores such that $S_1(\texttt{Users}) = \{(\texttt{42,70}, A_1, A_2, A_3), (\texttt{666,9}, B_1, B_2, B_3)\}$ and $S_2(\texttt{Users}) = \{(\texttt{42,70}, C_1, C_2, C_3), (\texttt{666,9}, B_1, B_2, B_3)\}$.
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 $\Delta \vdash^{\mathsf{r}}_{S_1,S_2} e \cong_{\texttt{U(666)}} e{:}\texttt{cmd}^{\bot}$.
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

$S_1'(\texttt{Users}) = \{(\texttt{42,70}, A_1, A_2, A_3), (\texttt{666,9}, B_1, B_2, B_3), (\texttt{42,70}, A_1, A_2, A_3)\}$ and
$S_2'(\texttt{Users}) = \{(\texttt{42,70}, C_1, C_2, C_3), (\texttt{666,9}, B_1, B_2, B_3), (\texttt{42,70}, C_1, C_2, C_3)\}$

which means noninterference is satisfied, because we have
$\mathit{redact}(\Delta, S_1', \texttt{U(666)}) = \{(\texttt{42,70,}\star\texttt{,}\star\texttt{,}\star), (\texttt{666,9}, B_1, B_2, B_3), (\texttt{42,70,}\star\texttt{,}\star\texttt{,}\star)\} = \mathit{redact}(\Delta, S_2', \texttt{U(666)})$.
That is, $S_1' =_{\texttt{U(666)}} S_2'$. So, the effects of expression e are not visible at security level `U(666)`, as expected.

Now consider a slight modification to the presented code, where we replace expression e with another assignment operation (which we shall refer as e'):

```
Users := [uid = 666, sid = p.sid, name = p.name,
            unit = p.univ, email = p.email] :: !Users
```

Using subexpression e' instead of e in the main program will now associate the contents of the profile of author with id `42` to a profile of author with id `666`. This clearly violates confidentiality, among other things, and is disallowed by the security lattice since `U(42)#U(666)`, so the program using this subexpression should be considered insecure. Let us look this in detail.
Again, we have $\Delta \vdash^{\mathsf{r}}_{S_1,S_2} e' \cong_{\texttt{U(666)}} e'{:}\texttt{cmd}^{\bot}$. After the reduction steps $(S_1; e') \longrightarrow (S_1'; ())$ and $(S_2; e') \longrightarrow (S_2'; ())$, we have

$S_1'(\texttt{Users}) = \{(\texttt{42,70}, A_1, A_2, A_3), (\texttt{666,9}, B_1, B_2, B_3), (\texttt{666,70}, A_1, A_2, A_3)\}$ and
$S_2'(\texttt{Users}) = \{(\texttt{42,70}, C_1, C_2, C_3), (\texttt{666,9}, B_1, B_2, B_3), (\texttt{666,70}, C_1, C_2, C_3)\}$

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

$\mathit{redact}(\Delta, S_1', \texttt{U(666)}) = \{(\texttt{42,70,}\star\texttt{,}\star\texttt{,}\star), (\texttt{666,9}, B_1, B_2, B_3), (\texttt{666,70}, A_1, A_2, A_3)\}$
and
$\mathit{redact}(\Delta, S_2', \texttt{U(666)}) = \{(\texttt{42,70,}\star\texttt{,}\star\texttt{,}\star), (\texttt{666,9}, B_1, B_2, B_3), (\texttt{666,70}, C_1, C_2, C_3)\}$

As expected, the thesis of non-interference theorem is not satisfied.

Of course, insecure programs like Example 5.12 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`.

6. Algorithmic Typechecking

We briefly discuss a type-checking algorithm for a suitable annotated version of our core language (Figure 8, remaining cases in [34]). The algorithm already allows us to verify many interesting examples, including those in paper, and support a prototype implementation [35]. For pragmatic reasons, we require type annotations on reference creation, record fields, and bound variables, 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, and on a constraint solving procedure, which we represent by $\mathcal{S} \models V \doteq U$ tests. 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. 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 [7]. The completeness of the algorithm is relative to the completeness of the required constraint solving problem.

7. Related Work

Language-based information flow analysis has attracted substantial research effort for a long time (see e.g., [22]). In early works the focus was on imperative languages [23, 28], λ-calculus [1, 12, 21], object-oriented languages [18], and concurrent languages [13, 30]. More recently, there has been a growing interest in studying secure information flows in the context of data-centric applications, namely in web scripting languages like Javascript (JS) as well as more general-purpose data manipulation languages (DML). While the work in this paper is not focused on reasoning about information flow analysis for data-centric applications, our core language can easily encode common DML high-level operation and thus our analysis is general enough to ensure noninterference on such applications involving expressive security policies, depending on runtime values as often required in realistic applications.

Static approaches have also been employed for secure information flows in data-centric applications, to cite a few of the relevant work: [3–5, 15], these works do not provide any kind of value-dependent information flow analysis, as we do here.

$$tc(\Delta,\mathcal{S},\mathsf{r},\mathbf{true}) \stackrel{\text{def}}{=} \mathtt{Bool}^{\perp}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},1) \stackrel{\text{def}}{=} \mathtt{Int}^{\perp}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},x) \stackrel{\text{def}}{=} \Delta(x)$$

$$tc(\Delta,\mathcal{S},\mathsf{r},[\tau^{\mathsf{s}}]e) \stackrel{\text{def}}{=} \mathbf{if}\ tc(\Delta,\mathcal{S},\mathsf{r},e) <: \tau^{\mathsf{s}}\ \mathbf{then}\ \tau^{\mathsf{s}}\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},]\mathsf{s}[e) \stackrel{\text{def}}{=} \mathbf{if}\ \mathsf{r}\leq\mathsf{s}\ \mathbf{then}\ tc(\Delta,\mathcal{S},\mathsf{s},e)\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},\lambda(x{:}\tau^{\mathsf{s}}).e) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta\cup\{x{:}\tau^{\mathsf{s}}\},\mathcal{S},\mathsf{r},e)\ \mathbf{in}\ (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r};\sigma^{\mathsf{t}})^{\perp}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},e_1(e_2)) \stackrel{\text{def}}{=} \mathbf{if}\ tc(\Delta,\mathcal{S},\mathsf{r},e_1) = (\Pi x{:}\tau^{\mathsf{s}}.\mathsf{r}';\sigma^{\mathsf{t}})^{\mathsf{q}}\ \mathbf{and}\ tc(\Delta,\mathcal{S},\mathsf{r},e_2) <: \tau^{\mathsf{s}}\ \mathbf{and}\ \mathsf{r}\leq\mathsf{r}'\ \mathbf{then}\ \mathbf{if}\ \mathcal{S}\models e_2\doteq\mathsf{v}\ \mathbf{then}\ \sigma^{\mathsf{t}}\{\mathsf{v}/x\}\ \mathbf{else}\ \sigma^{\mathsf{t}}\{\top/x\}\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},[\ldots,m_i{:}\tau_i^{\mathsf{s}_i}=e_i,\ldots]) \stackrel{\text{def}}{=} \mathbf{let}\ \Sigma[\ldots\times m_i{:}\tau_i'^{\mathsf{s}_i}\times\ldots]^{\mathsf{s}} = \mathit{unref}(\mathcal{S},[\ldots,m_i{:}\tau_i^{\mathsf{s}_i}=e_i,\ldots])\ \mathbf{in}\ \mathbf{if}\ (\mathtt{forall}\ e_i.\ \sigma_i^{\mathsf{t}} = tc(\Delta,\mathcal{S},\mathsf{r},e_i)\ \mathbf{and}\ \sigma_i^{\mathsf{t}} <: \tau_i'^{\mathsf{s}_i})\ \mathbf{then}\ \Sigma[\ldots\times m_i{:}\tau_i^{\mathsf{s}_i}\times\ldots]^{\mathsf{s}}\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},e.m) \stackrel{\text{def}}{=} \mathbf{let}\ \tau^{\mathsf{s}} = tc(\Delta,\mathcal{S},\mathsf{r},e)\ \mathbf{in}\ \mathbf{if}\ \tau=\Sigma[\ldots\times m{:}\sigma^{\ell(n)}\times\ldots]\ \mathbf{then}\ (\ \mathbf{if}\ \mathcal{S}\{f\doteq e\}\models f.n\doteq\mathsf{v}\ \mathbf{then}\ \sigma^{\ell(\mathsf{v})}\ \mathbf{else}\ \sigma^{\ell(\top)}\ )\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},\ \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \stackrel{\text{def}}{=} \mathbf{let}\ \tau^{\mathsf{s}} = tc(\Delta,\mathcal{S},\mathsf{r},e_1)\ \mathbf{in}\ \mathbf{if}\ \tau=\mathtt{Bool}\ \mathbf{then}\ (\ \mathbf{let}\ \tau_2{}^{\mathsf{s}_2} = tc(\Delta,\mathcal{S}\cup\{e_1\doteq\mathtt{true}\},\mathsf{r}\sqcup\mathsf{s},e_2)\ \mathbf{in}\ \mathbf{let}\ \tau_3{}^{\mathsf{s}_3} = tc(\Delta,\mathcal{S}\cup\{e_1\doteq\mathtt{false}\},\mathsf{r}\sqcup\mathsf{s},e_3)\ \mathbf{in}\ \mathbf{if}\ \tau_2=\tau_3\ \mathbf{then}\ \tau_2^{\mathsf{s}\sqcup\mathsf{s}_2\sqcup\mathsf{s}_3}\ \mathbf{else}\ \mathtt{typerror}\ )\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},\ \mathbf{ref}_{\tau^{\mathsf{s}}}\ e) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta,\mathcal{S},\mathsf{r},e)\ \mathbf{in}\ \mathbf{if}\ \sigma^{\mathsf{t}} <: \tau^{\mathsf{s}}\ \mathbf{and}\ \mathsf{r}\leq\mathsf{s}\ \mathbf{then}\ (\mathtt{ref}_{\tau^{\mathsf{s}}})^{\perp}\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},!e) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta,\mathcal{S},\mathsf{r},e)\ \mathbf{in}\ \mathbf{if}\ \sigma^{\mathsf{t}} <: (\mathtt{ref}_{\tau^{\mathsf{s}}})^{\mathsf{t}}\ \mathbf{and}\ \mathsf{t}\leq\mathsf{s}\ \mathbf{then}\ \tau^{\mathsf{s}}\ \mathbf{else}\ \mathtt{typerror}$$

$$tc(\Delta,\mathcal{S},\mathsf{r},e_1:=e_2) \stackrel{\text{def}}{=} \mathbf{let}\ \sigma^{\mathsf{t}} = tc(\Delta,\mathcal{S},\mathsf{r},e_1)\ \mathbf{in}\ \mathbf{let}\ \tau^{\mathsf{s}} = tc(\Delta,\mathcal{S},\mathsf{r},e_2)\ \mathbf{in}\ \mathbf{if}\ \sigma^{\mathsf{t}} <: (\mathtt{ref}_{\tau^{\mathsf{s}}})^{\mathsf{t}}\ \mathbf{and}\ \mathsf{r}\sqcup\mathsf{t}\leq\mathsf{s}\ \mathbf{then}\ \mathtt{Cmd}^{\perp}\ \mathbf{else}\ \mathtt{typerror}$$

Figure 8: Typechecking algorithm

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 [27], and security labels that can dynamically change, by Zheng and Myers in [32]. The former is based on the (seminal) decentralised label model (DLM) introduced by Myers and Liskov in [19], and presents a typed λ-calculus where principals are values and thus can be used during program execution, 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 [2]) 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 our 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. With our label model, we are able to specify security labels that depend on the actual program's (stored) data.

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.

Several recent works explore applications of dependent types [20, 25, 26] to language-based security in the context of stateful static information flow. In [25] Swamy *et. al.* present FINE, a general-purpose and very expressive dependently typed language based on Fable [24], 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. In [25] 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). Moreover, 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 [20], 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 [26] and [20], is RF* [33], 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 [25], 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.

A concept of indexed security label was introduced in [17], 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 here. The developments in this paper 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.

Several proposals for dynamic information flow analysis on web languages have been put forward such as [6, 9, 11, 14, 16, 31]. In this work we focus on static certification techniques, developing the new notion of dependent information flow types. In future work it would be interesting to study combinations of static and dynamic typing in the context of our dependent type system.

8. Concluding Remarks

In this paper, we have motivated, 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. 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. Our development is carried out on top of a minimalistic λ-calculus with general references and collections, thus adding generality and application scope to the approach. Our main technical results are expressed by type safety and non-interference theorems, 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.

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. We have also briefly discussed some algorithmic aspects of our approach, that has led to a prototype implementation [35], which can already be used to check examples, including those in this paper.

Adding variant types to our language is a trivial exercise, and would be important also for practical reasons. It would be interesting to investigate formulations of our type system integrating notions of type refinement (e.g, [29]), and type inference. As another follow up topic, as 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.

Acknowledgments. We thank the anonymous reviewers for their insightful comments and suggestions. We also thank Jorge A. Perez and João C. Seco for related discussions. This work is supported by CITI PEst-OE/EEI/UI0527/2014, FCT/MEC grant SFRH/BD/68801/2010, and FLEX-Agile grant by OutSystems SA.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke:. A Core Calculus of Dependency. In *POPL* 1999.

[2] 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 SSP 2012*.

[3] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic Subtyping with an SMT Solver. *J. Funct. Program.*, 2012.

[4] A. Chlipala. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *USENIX OSDI* 2010.

[5] B. J. Corcoran, N. Swamy, and M. W. Hicks. Cross-tier, Label-based Security Enforcement for Web Applications. In *ACM SIGMOD Int. Conf. on Management of Data*, 2009.

[6] B. Davis and H. Chen. DBTaint: Cross-Application Information Flow Tracking via Databases. In *USENIX WebApps* 2010.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS 2008*.

[8] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 1977.

[9] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. M., and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX OSDI* 2010.

[10] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE SSP* 1982.

[11] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *IEEE CSF 2012*.

[12] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL* 1998.

[13] K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP 2000*, LNCS.

[14] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *IEEE SSP 2013*.

[15] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *IEEE CSFW 2005*.

[16] 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 *ACM SOSP 2009*.

[17] L. Lourenço and L. Caires. Information Flow Analysis for Valued-Indexed Data Security Compartments. In *TGC 2013*.

[18] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL* 1999.

[19] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *ACM SOSP* 1997.

[20] A. Nanevski, A. Banerjee, and D. Garg. Verification of Information Flow and Access Control Policies with Dependent Types. In *IEEE SSP* 2011.

[21] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL 2002*.

[22] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE JSAC*, 21(1):5-19, Jan. 2003.

[23] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 2001.

[24] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE SSP 2008*.

[25] N. Swamy, J. Chen, and R. Chugh. Enforcing Stateful Authorization and Information Flow Policies in Fine. In *ESOP 2010*.

[26] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure Distributed Programming with Value-dependent Types. In *ICFP 2011*.

[27] S. Tse and S. Zdancewic. Run-time Principals in Information-flow Type Systems. *ACM Trans. Program. Lang. Syst.*, 2007.

[28] D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 1996.

[29] H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *POPL* 1999.

[30] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *IEEE CSFW 2003*.

[31] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing Distributed Systems with Information Flow Control. In *USENIX NSDI 2008*.

[32] L. Zheng and A. C. Myers. Dynamic Security Labels and Static Information Flow Control. *Int. J. Inf. Sec.*, 2007.

[33] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy and S. Z. Béguelin. Probabilistic relational verification for cryptographic implementations. In *POPL 2014*.

[34] L. Lourenço and L. Caires. Dependent Information Flow Types. Technical report, UNL, 2014.

[35] DIFT Prototype. `http://ctp.di.fct.unl.pt/DIFTprototype`.