A type system for information flow control

Mid-term exam, MPRI 2-4

2019/11/29 — Duration: 3h00

Answers are judged by their correctness, but also by their clarity, conciseness, and accuracy. You don't have to justify answers unless explicitly required. Although the questions are in English, it is permitted to answer in French.

In this problem, we study *information flow* in programs that manipulate various pieces of data at different security levels. For example, a program can manipulate both *secret data* (such as confidential clear text or secret keys) and *public data* (such as encrypted text or public keys). We wish to ensure that no secret input can influence a public output, because that would be represent a *leak* of confidential information. This absence of dependency of a public output on a secret input is known as *noninterference*. Our goal is to study a type system that is able to distinguish public and secret data and prove that this type system guarantees noninterference.

We consider programs expressed in a call-by-value λ-calculus equipped with Boolean constants c. We refer to this calculus as "λ". Its syntax is as follows:

$$\begin{array}{rcll} c & ::= & \mathsf{t} \mid \mathsf{f} & \textit{(constants)} \\ v, w & ::= & c \mid \lambda x.a & \textit{(values)} \\ a, b & ::= & x \mid c \mid \lambda x.a \mid a\ a & \textit{(expressions, a.k.a. terms)} \end{array}$$

Proving that noninterference holds requires reasoning about *two* programs that are initially related, in the sense that they have different secret inputs but are otherwise identical, and proving that they remain related throughout their execution. In order to make such reasoning easier, we invent a calculus that allows representing a *couple* of related programs as a *single* term. We refer to this calculus as "$\lambda\lambda$". Its syntax is as follows:

$$\begin{array}{rcll} c & ::= & \mathsf{t} \mid \mathsf{f} & \textit{(constants)} \\ v, w & ::= & c \mid \lambda x.a \mid \langle v \mid v \rangle & \textit{(values)} \\ a, b & ::= & x \mid c \mid \lambda x.a \mid a\ a \mid \langle a \mid a \rangle & \textit{(expressions, a.k.a. terms)} \end{array}$$

The only change with respect to λ is the addition of the value form $\langle v \mid v \rangle$ and of the expression form $\langle a \mid a \rangle$. We refer to these constructs as "brackets".

We do not allow nesting $\langle \cdot \mid \cdot \rangle$ constructs, because that would not make sense. So, in the construct $\langle v_1 \mid v_2 \rangle$, the values v_1 and v_2 must be λ-values, and in the construct $\langle a_1 \mid a_2 \rangle$, the terms a_1 and a_2 must be λ-terms. This is a global syntactic restriction.

The $\lambda\lambda$-term $\langle a_1 \mid a_2 \rangle$ is intended to represent the couple of λ-terms a_1 and a_2. These brackets can appear at an arbitrary depth within a $\lambda\lambda$-term. For instance, assuming that a_1, a_2, and a are λ-terms, the $\lambda\lambda$-terms $\langle a_1 \mid a_2 \rangle\ a$ and $\langle a_1\ a \mid a_2\ a \rangle$ both represent the

couple of λ-terms $a_1\ a$ and $a_2\ a$. The former representation explicitly records the fact that the application node and its argument a are shared, while the latter does not.

The correspondence between the calculi λ and $\lambda\lambda$ is made explicit by means of two *projection functions* $\lfloor\cdot\rfloor_i$, where i ranges over $\{1, 2\}$. These functions map a $\lambda\lambda$-term to a λ-term. They satisfy the equation $\lfloor\langle a_1 \mid a_2\rangle\rfloor_i = a_i$ and are homomorphisms elsewhere. We remark that λ-terms form a subset of $\lambda\lambda$-terms and that the projection $\lfloor a\rfloor_i$ of a λ-term a is a itself.

Question 1

For each $i \in \{1, 2\}$, compute $\lfloor\lambda x.((\lambda y.\langle x \mid y\rangle)\ \langle \mathsf{t} \mid \mathsf{f}\rangle)\rfloor_i$. □

Let us give a hint of how $\lambda\lambda$ allows keeping track of the differences between two λ-terms throughout execution. Consider the constant function $\lambda x.\mathsf{t}$. Clearly, its result does not reveal any information about its argument: applying it to two distinct arguments produces the same result. For this reason, this function will have type "secret Boolean to public Boolean", among other types. One way of seeing that the programs $(\lambda x.\mathsf{t})\ v_1$ and $(\lambda x.\mathsf{t})\ v_2$ produce the same value is to encode them as a single $\lambda\lambda$-term, namely $(\lambda x.\mathsf{t})\ \langle v_1 \mid v_2\rangle$. The two projections of this term are the two programs of interest. Note that the "secret" inputs v_1 and v_2 appear under brackets, while the structure that is common to both programs, namely the application of $\lambda x.\mathsf{t}$, is shared: it appears outside the brackets. According to the operational semantics of $\lambda\lambda$, which we will introduce later on, the term $(\lambda x.\mathsf{t})\ \langle v_1 \mid v_2\rangle$ reduces to the value t. The fact that this term does not contain any brackets implies that its projections coincide: in other words, the two programs that we started with compute the same result.

Reduction sequences in $\lambda\lambda$ can be more complex than the simple reduction sequence described above. Indeed, some of the reduction rules must "lift" the brackets when they block reduction. For instance, the application $\langle\lambda x.x \mid \lambda x.\mathsf{t}\rangle\ \mathsf{f}$ cannot be reduced by the standard β-reduction rule, Beta, so it must be taken care of by some other reduction rule. We introduce a new rule, Lift-Beta, which reduces it to $\langle(\lambda x.x)\ \mathsf{f} \mid (\lambda x.\mathsf{t})\ \mathsf{f}\rangle$. This step affects none of the two projections: it has no computational content. By moving the brackets, it keeps track of information flow. Each side of the new bracket is now a β-redex, allowing reduction to proceed under the brackets. In the end, we obtain $\langle\lambda x.x \mid \lambda x.\mathsf{t}\rangle\ \mathsf{f} \to^\star \langle\mathsf{f} \mid \mathsf{t}\rangle$.

The substitution of a (closed) value v for a variable x in an expression e is defined in the usual way, except at brackets, where an appropriate projection must be used: $\langle a_1 \mid a_2\rangle[x \leftarrow v]$ is $\langle a_1[x \leftarrow \lfloor v\rfloor_1] \mid a_2[x \leftarrow \lfloor v\rfloor_2]\rangle$.

The operational semantics of $\lambda\lambda$ is defined by the following reduction rules, where $\diamond$ denotes the hole in an evaluation context:

$$(\lambda x.a)\ v \quad\longrightarrow\quad a[x \leftarrow v] \qquad\qquad \textsc{Beta}$$

$$\langle v_1 \mid v_2\rangle\ v \quad\longrightarrow\quad \langle v_1\ \lfloor v\rfloor_1 \mid v_2\ \lfloor v\rfloor_2\rangle \qquad\qquad \textsc{Lift-Beta}$$

$$\textsc{Context}\quad \frac{a \longrightarrow a'}{E[a] \longrightarrow E[a']} \qquad\qquad E \quad ::= \quad \diamond\ a \mid v\ \diamond \mid \langle\diamond \mid a\rangle \mid \langle a \mid \diamond\rangle$$

Question 2

Is reduction in λ deterministic? Is reduction in $\lambda\lambda$ deterministic? □

Question 3

Let a stand for $(\lambda x.\lambda y.x)\ \langle \mathsf{t} \mid \mathsf{f} \rangle\ \mathsf{t}$. Let b stand for $(\lambda x.\lambda y.x)\ \mathsf{t}\ \langle \mathsf{t} \mid \mathsf{f} \rangle$. Give the reduction sequences of the terms a and b. □

Question 4

Our calculi λ and $\lambda\lambda$ include the Boolean constants t and f. Is there something odd about this? What could be done to remedy this issue? A brief answer is expected. □

Question 5

In this question only, we extend the syntax of expressions with a conditional construct $\mathsf{if}\ a\ \mathsf{then}\ a\ \mathsf{else}\ a$. Which new reduction rules and evaluation contexts do we need for this construct to have the expected semantics in the calculi λ and $\lambda\lambda$? Give a reduction sequence out of the term $(\lambda x.\mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ x)\ \langle \mathsf{t} \mid \mathsf{f} \rangle$. □

Question 6 (Simulation)

Let a be a closed $\lambda\lambda$-term. a can exhibit one of three behaviors: (1) a can take one reduction step, or (2) a is a value, or (3) a is stuck. In each scenario, what statement can one make about the behavior of the λ-terms $\lfloor a \rfloor_1$ and $\lfloor a \rfloor_2$? (No proof is required.) □

Question 7

Prove that an infinite reduction sequence in $\lambda\lambda$ cannot consist solely of "lifting" steps (that is, applications of rule Lift-Beta under an evaluation context). □

Question 8 (Reverse simulation)

Let a be a closed $\lambda\lambda$-term. Suppose that, for every i in $\{1, 2\}$, we have $\lfloor a \rfloor_i \longrightarrow^\star v_i$. Prove, then, that there must exist a value v such that $a \longrightarrow^\star v$. What can be said of the projections of v? □

The syntax of types is as follows:

$$\begin{array}{lcll} \ell & ::= & L \mid H & \textit{(security levels)} \\ \tau & ::= & \rho^\ell & \textit{(types)} \\ \rho & ::= & \mathsf{bool} \mid \tau \to \tau & \textit{(raw types)} \end{array}$$

A *security level* ℓ is one of L or H. These levels stand for "low" and "high": low-security data means public data, while high-security data means secret data. A type τ is composed of a raw type ρ and a security level ℓ. A raw type is either bool or a function type $\tau_1 \to \tau_2$.

Security levels, types, and raw types are equipped with a subtyping relation, which is defined as follows.

$$\ell \leq \ell \qquad L \leq H \qquad \frac{\rho_1 \leq \rho_2 \qquad \ell_1 \leq \ell_2}{{\rho_1}^{\ell_1} \leq {\rho_2}^{\ell_2}} \qquad \mathsf{bool} \leq \mathsf{bool} \qquad \frac{\tau_2 \leq \tau_1 \qquad {\tau_1}' \leq {\tau_2}'}{\tau_1 \to {\tau_1}' \leq \tau_2 \to {\tau_2}'}$$

We admit that subtyping is reflexive and transitive; it would be easily proved.

We write $\mathit{level}(\tau)$ for the security level that appears at the root of the type τ. (This function is defined by $\mathit{level}(\rho^\ell) = \ell$.) We write $\ell \lhd \tau$ as a short-hand for $\ell \leq \mathit{level}(\tau)$.

The typing judgment $\Gamma \vdash a : \tau$ is inductively defined as follows:

$$\textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Bool}\ \frac{c \in \mathsf{t}, \mathsf{f}}{\Gamma \vdash c : \mathsf{bool}^{\ell}} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash a : \tau_2}{\Gamma \vdash \lambda x.a : (\tau_1 \to \tau_2)^{\ell}} \qquad \textsc{App}\ \frac{\Gamma \vdash a_1 : (\tau_2 \to \tau_1)^{\ell} \quad \Gamma \vdash a_2 : \tau_2 \quad \ell \lhd \tau_1}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{Bracket}\ \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau \quad H \lhd \tau}{\Gamma \vdash \langle a_1 \mid a_2 \rangle : \tau} \qquad \textsc{Sub}\ \frac{\Gamma \vdash a : \tau \quad \tau \leq \tau'}{\Gamma \vdash a : \tau'}$$

In one sentence, the premise $\ell \lhd \tau_1$ in App means that a high-security function must produce a high-security result. The premise $H \lhd \tau$ in Bracket means that every bracket must have a high-security type.

Question 9

What are the possible types of the terms a and b of Question 3? Could this help you predict, informally, the result of their evaluation? (Give an informal argument; no proof is required.) □

Question 10

Assuming that $\Gamma \vdash a : \tau$ holds, what statement can be made about the type of $\lfloor a \rfloor_i$? (No proof is required.) □

Question 11

Prove that if a λ-term is well-typed in simply-typed λ-calculus ($STLC$), then it is also well-typed in $\lambda\lambda$. (We expect a clear high-level argument, not a detailed proof by induction where every case is explicitly treated.) □

Question 12 (Bracket inversion)

Prove that $\Gamma \vdash \langle a_1 \mid a_2 \rangle : \tau$ implies $\Gamma \vdash a_1 : \tau$ and $\Gamma \vdash a_2 : \tau$ and $H \lhd \tau$. □

Question 13 (Value substitution)

Prove that $\emptyset \vdash v : \tau$ and $x : \tau \vdash a : \tau'$ imply $\emptyset \vdash a[x \leftarrow v] : \tau'$. Give the general structure of the proof, and treat only the case of the typing rule Bracket in detail. □

Question 14 (Subject reduction)

Give the statement of the "subject reduction" property for closed terms in $\lambda\lambda$, and prove this statement. Give the general structure of the proof, and treat only the case of the reduction rule Lift-Beta in detail. □

Question 15 (Main lemma)

Prove that if $\emptyset \vdash a : \mathsf{bool}^{L}$ and $a \longrightarrow^{\star} v$, then $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$. □

Question 16 (Noninterference)

Let a be a λ-term such that $x : \tau \vdash a : \mathsf{bool}^{L}$ and $H \lhd \tau$ hold. Prove that if for every i in $\{1, 2\}$ we have $\emptyset \vdash v_i : \tau$ and $a[x \leftarrow v_i] \longrightarrow^{\star} w_i$ then $w_1 = w_2$. Explain informally, in a few sentences, why this result is the "noninterference" property that an information flow type system must satisfy. □

1 Solutions

Question 1

The left and right projections of this term are as follows:

$$\begin{array}{lcl} \lfloor \lambda x.((\lambda y.\langle x \mid y\rangle)\ \langle \mathsf{t} \mid \mathsf{f}\rangle)\rfloor_1 & = & \lambda x.((\lambda y.x)\ \mathsf{t}) \\ \lfloor \lambda x.((\lambda y.\langle x \mid y\rangle)\ \langle \mathsf{t} \mid \mathsf{f}\rangle)\rfloor_2 & = & \lambda x.((\lambda y.y)\ \mathsf{f}) \end{array}$$

Note that nothing was asked about the *reduction* of these terms. Describing their reduction was unnecessary. Mixing projection and reduction (which some students did) was particularly unfortunate, as this meant that our question (which was purely about projection) was not answered. □

Question 2

Reduction in λ is deterministic: it is a standard call-by-value λ-calculus with left-to-right evaluation at applications. Reduction in $\lambda\lambda$ is not deterministic: indeed, the two sides of a bracket $\langle a_1 \mid a_2\rangle$ can be reduced independently.

Question 3

$$a \longrightarrow (\lambda y.\langle \mathsf{t} \mid \mathsf{f}\rangle)\ \mathsf{t} \longrightarrow \langle \mathsf{t} \mid \mathsf{f}\rangle \qquad\qquad b \longrightarrow (\lambda y.\mathsf{t})\ \langle \mathsf{t} \mid \mathsf{f}\rangle \longrightarrow \mathsf{t}$$

Question 4

There are no destructors of Boolean values. Hence, these values can be passed around, but never inspected. As a result, there is no way of distinguishing t and f. To remedy this problem, one could extend the calculi with a conditional construct $\mathsf{if}\ a\ \mathsf{then}\ a\ \mathsf{else}\ a$.

Question 5

The following evaluation contexts and reduction rules are required:

$$\begin{array}{rcll} E & ::= & \ldots \mid \mathsf{if}\ \diamond\ \mathsf{then}\ e\ \mathsf{else}\ e & \\ \mathsf{if}\ \mathsf{t}\ \mathsf{then}\ a\ \mathsf{else}\ a' & \longrightarrow & a & \textsc{If-True} \\ \mathsf{if}\ \mathsf{f}\ \mathsf{then}\ a\ \mathsf{else}\ a' & \longrightarrow & a' & \textsc{If-False} \\ \mathsf{if}\ \langle v_1 \mid v_2\rangle\ \mathsf{then}\ a\ \mathsf{else}\ a' & \longrightarrow & \langle\ \mathsf{if}\ v_1\ \mathsf{then}\ \lfloor a\rfloor_1\ \mathsf{else}\ \lfloor a'\rfloor_1 & \textsc{Lift-If} \\ & & \quad \mid \mathsf{if}\ v_2\ \mathsf{then}\ \lfloor a\rfloor_2\ \mathsf{else}\ \lfloor a'\rfloor_2\rangle & \end{array}$$

A possible reduction sequence is as follows:

$$\begin{array}{rcl} (\lambda x.\mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ x)\ \langle \mathsf{t} \mid \mathsf{f}\rangle & \longrightarrow & \mathsf{if}\ \langle \mathsf{t} \mid \mathsf{f}\rangle\ \mathsf{then}\ \langle \mathsf{t} \mid \mathsf{f}\rangle\ \mathsf{else}\ \langle \mathsf{t} \mid \mathsf{f}\rangle \\ & \longrightarrow & \langle \mathsf{if}\ \mathsf{t}\ \mathsf{then}\ \mathsf{t}\ \mathsf{else}\ \mathsf{t} \mid \mathsf{if}\ \mathsf{f}\ \mathsf{then}\ \mathsf{f}\ \mathsf{else}\ \mathsf{f}\rangle \\ & \longrightarrow & \langle \mathsf{t} \mid \mathsf{if}\ \mathsf{f}\ \mathsf{then}\ \mathsf{f}\ \mathsf{else}\ \mathsf{f}\rangle \\ & \longrightarrow & \langle \mathsf{t} \mid \mathsf{f}\rangle \end{array}$$

Because evaluation is permitted within both components of a bracket, the operational semantics of $\lambda\lambda$ is nondeterministic. Thus, there exists another reduction sequence, where reduction takes place first in the right-hand side of the bracket, then in its left-hand side. The final result is the same. One could prove that the operational semantics is confluent.

Question 6

The following statements can be made:

1. If $a \longrightarrow a'$, then, for every i in $\{1, 2\}$, either $\lfloor a \rfloor_i \longrightarrow \lfloor a' \rfloor_i$ or $\lfloor a \rfloor_i = \lfloor a' \rfloor_i$.
2. If a is a value, then, for every i in $\{1, 2\}$, $\lfloor a \rfloor_i$ is a value.
3. If a is stuck, then, for some i in $\{1, 2\}$, $\lfloor a \rfloor_i$ is stuck.

Regarding item 1, many students made the statement that "either $\lfloor a \rfloor_1$ or $\lfloor a \rfloor_2$ can take a reduction step". This is false. If a can take one reduction step via Lift-Beta, then $\lfloor a \rfloor_1$ and $\lfloor a \rfloor_2$ must be applications, but nothing indicates that at least one of these terms can take a reduction step (and indeed one can construct examples were neither of them can).

Regarding item 3, many students made the statement that "for every i in $\{1, 2\}$, $\lfloor a \rfloor_i$ is stuck". This is false. For example, if v_1 is a λ-value and a_2 is a stuck λ-term, then the $\lambda\lambda$-term $(\lambda x.a)\ \langle v_1 \mid a_2 \rangle$ is stuck, yet its first projection $(\lambda x.\lfloor a \rfloor_1)\ v_1$ is not stuck.

Question 7

Brackets cannot be nested. Therefore, every syntax tree node exists either outside any brackets or inside a (single) bracket. The lifting reduction rule Lift-Beta moves a subterm from the outside of a bracket to the inside (where it is duplicated). Thus, every lifting step decreases the number of syntax tree nodes that exist outside brackets. Thus, an infinite reduction sequence cannot consist solely of lifting steps.

Question 8

Consider a maximal reduction sequence out of a. We first argue that this sequence cannot be infinite. Indeed, if it were infinite, then, because an infinite reduction sequence in $\lambda\lambda$ cannot consist solely of lifting steps (question 7), it would contain an infinite subsequence of β-reduction steps (either outside or inside brackets). Because every such step gives rise to a β-reduction step in *at least one* of the projections, this would entail that at least one of the terms $\lfloor a \rfloor_1$ and $\lfloor a \rfloor_2$ admits an infinite reduction sequence. This contradicts the hypothesis that these terms reduce to values.

Therefore, this maximal reduction sequence must be finite: it must be of the form $a \longrightarrow^\star a'$, where a' is irreducible. There remains to prove that a' cannot be stuck; this will guarantee that a' must be a value.

If a' were stuck, then, by item 3 of question 6, one of its projections would be stuck, say $\lfloor a' \rfloor_i$. However, by item 1 of question 6, $a \longrightarrow^\star a'$ implies $\lfloor a \rfloor_i \longrightarrow^\star \lfloor a' \rfloor_i$. Thus, $\lfloor a \rfloor_i$

would reduce to a stuck term. This contradicts the hypothesis that this term reduces to a value.

So, we have obtained $a \longrightarrow^\star v$. By item 1 of question 6, this implies $\lfloor a \rfloor_i \longrightarrow^\star \lfloor v \rfloor_i$. By item 2 of question 6, $\lfloor v \rfloor_i$ is a value. However, we also have $\lfloor a \rfloor_i \longrightarrow^\star v_i$. Because a λ-term cannot reduce to two distinct values (question 2), we have $\lfloor v \rfloor_i = v_i$.

To our surprise, almost no student gave a correct solution to this question. In particular, we saw many attempts to establish the main result (that is, "a reduces to some value") directly by structural induction on the expression a. However, these proof attempts are needlessly complicated and (fatally) involve applying the induction hypothesis to an expression that is *not* a subexpression of a.

Question 9

The following judgements hold:

$\emptyset \vdash a : \mathsf{bool}^H$

$\emptyset \vdash b : \mathsf{bool}^\ell$ (for any security level ℓ)

In particular, we have $\emptyset \vdash b : \mathsf{bool}^L$. Assuming that subject reduction holds, if b terminates, then its result must be a value v of type bool^L. This value cannot be a "bracket", because $H \lhd \mathsf{bool}^L$ does not hold. (The typing rule Bracket requires every "bracket" expression to have a high-security type.) Therefore, this value must a Boolean constant, that is, t or f. Type-checking alone does not allow us to predict which one of the two it might be.

Because a has high-security type, the type system does not help us predict the result of its evaluation: it could be either a Boolean constant or a bracket of Boolean constants. (The fact that a term does not admit a low-security type does *not* imply that this term must evaluate to a bracket!)

Question 10

For every i in $\{1, 2\}$, the judgement $\Gamma \vdash \lfloor a \rfloor_i : \tau$ holds.

Question 11

The types of $STLC$ and types of $\lambda\lambda$ are closely related: the only difference is that, in a $\lambda\lambda$ type, every node carries a security level. One could define an erasure function that erases all security levels and maps a $\lambda\lambda$ type to a λ type. Of greater interest, here, is a function that works in the other direction and maps a λ type to a $\lambda\lambda$ type, by decorating every node with a security level. It is good enough, for our purposes, to fix a security level ℓ and to decorate every node with this level. Thus, if T denotes a simple type ($T ::= \mathsf{bool} \mid T \to T$), let $\mathit{decorate}_\ell(T)$ be the $\lambda\lambda$ type obtained from T by decorating every node with the level ℓ. Now, we would like to prove that if the judgement $\Gamma \vdash a : T$ holds in $STLC$, then $\mathit{decorate}_\ell(\Gamma) \vdash a : \mathit{decorate}_\ell(T)$ holds in λ. This can be easily proved by induction. (We omit the details.) We note that the premise $\ell \lhd \tau_1$ in rule App is trivially satisfied because we use the same security level everywhere, namely ℓ. The existence of the typing rule Bracket is never exploited, because the term a is a λ-term and therefore does not contain any brackets. The existence of the typing rule

SUB is never exploited either.

Question 12

The proof is by induction on the typing derivation. Only two typing rules can conclude $\Gamma \vdash \langle a_1 \mid a_2 \rangle : \tau$, therefore there are only two proof cases:

Case BRACKET. The result is immediate.

Case SUB. The premises of the rule are $\Gamma \vdash \langle a_1 \mid a_2 \rangle : \tau'$ and $\tau' \leq \tau$. By the induction hypothesis, we have $\Gamma \vdash a_1 : \tau'$ and $\Gamma \vdash a_2 : \tau'$ and $H \leq \tau'$. By applying rule SUB, we get $\Gamma \vdash a_1 : \tau$ and $\Gamma \vdash a_2 : \tau$ By transivity of subtyping, we get $H \lhd \tau$.

Question 13

The statement must first be strengthened as follows: the typing judgements $\emptyset \vdash v : \tau$ and $x : \tau, \Gamma \vdash a : \tau'$ imply $\Gamma \vdash a[x \leftarrow v] : \tau'$. (Without this generalization, the case of rule LAM cannot go through.) Then, the proof of the strengthened statement is by induction on the derivation of the typing judgement $x : \tau, \Gamma \vdash a : \tau'$. Thus, there is one case per typing rule. We describe only the following case:

Case BRACKET. Then, a is $\langle a_1 \mid a_2 \rangle$, and the premises of the rule are $x : \tau, \Gamma \vdash a_1 : \tau'$ and $x : \tau, \Gamma \vdash a_2 : \tau'$ and $H \lhd \tau'$. By question 10 applied to the hypothesis $\emptyset \vdash v : \tau$, we have $\emptyset \vdash \lfloor v \rfloor_i : \tau$ for every $i \in \{1, 2\}$. By the induction hypothesis (applied twice), we get $\Gamma \vdash a_i[x \leftarrow \lfloor v \rfloor_i] : \tau'$ for every i in $\{1, 2\}$. The conclusion follows by applying rule BRACKET; indeed, recall that, by definition of substitution in $\lambda\lambda$, $\langle a_1 \mid a_2 \rangle[x \leftarrow v]$ is equal to $\langle a_1[x \leftarrow \lfloor v \rfloor_1] \mid a_2[x \leftarrow \lfloor v \rfloor_2] \rangle$.

Question 14

The statement is that $\emptyset \vdash a : \tau$ (**1**) and $a \longrightarrow a'$ imply $\emptyset \vdash a' : \tau$ (**2**). We prove this statement first in the restricted case where the derivation of the typing judgement (1) does not end with an instance of Rule SUB. We reason by cases on the reduction rule that is used to derive $a \longrightarrow a'$. We give only the following case:

Case LIFT-BETA. a is $\langle v_1 \mid v_2 \rangle\ v$ and a' is $\langle v_1\ v \mid v_2\ v \rangle$. The derivation of (1) must end with an instance of BRACKET whose premises are $\emptyset \vdash \langle v_1 \mid v_2 \rangle : (\tau_2 \to \tau)^\ell$ and $\emptyset \vdash v : \tau_2$ (**3**) and $\ell \lhd \tau$ (**4**). By bracket inversion (question 12), we have $\emptyset \vdash v_i : (\tau_2 \to \tau)^\ell$ (**5**) for every i in $\{1, 2\}$ and $H \lhd (\tau_2 \to \tau)^\ell$, which is equivalent to $H \leq \ell$ (**6**). By APP, we have $\emptyset \vdash v_i\ v : \tau$ (**7**) for every i in $\{1, 2\}$. By transitivity of subtyping, (6) and (4) imply $H \lhd \tau$. Therefore, we can apply BRACKET to the premises (7), which yields the desired conclusion (2).

We now return to the general case, where the derivation of the typing judgement (1) ends with a series of instances of SUB. We reason by induction on the number of instances of rule SUB. The base case has been dealt with above. In the inductive case, the derivation ends with an instance of SUB, so we have $\emptyset \vdash a : \tau'$ (**8**) and $\tau' \leq \tau$. By the induction hypothesis, applied to (8), we get $\emptyset \vdash a' : \tau'$. We conclude $\emptyset \vdash a' : \tau$ by applying SUB.

Question 15

By subject reduction (question 14), we have $\emptyset \vdash v : \mathsf{bool}^L$ (**9**). We reason by cases on v:

Case v is $\langle v_1 \mid v_2 \rangle$. By bracket inversion (question 12), we get $H \lhd \mathsf{bool}^L$, which is equivalent to $H \leq L$ and is a contradiction. In other words, this case is impossible.

Case v is `t` *or* `f`. Then the result is immediate.

Case v is $\lambda x.a'$. This case is impossible. Indeed, the derivation of the judgement (9) would then end with an instance of Lam, followed by a series of instances of Sub. By transitivity of subtyping, this would imply a subtyping judgement of the form $(\tau_1 \to \tau_2)^\ell \leq \mathsf{bool}^L$, which is not derivable.

Question 16

We begin with the requested informal explanation. According to the hypotheses $x : \tau \vdash a : \mathsf{bool}^L$ and $H \lhd \tau$, the term a has a low-security type, and the variable x plays the role of a named hole with a high-security type. The values v_1 and v_2 represent two (well-typed) "secret" inputs that can be plugged in this hole (by substitution). The equality $w_1 = w_2$ guarantees that the final result of the computation is independent of which value is plugged in the hole; in other words, a low-security output cannot depend on a high-security input.

We note that termination *is* allowed to depend on high-security inputs. Indeed, in the above statement, the equality $w_1 = w_2$ is guaranteed to hold *if* both $a[x \leftarrow v_1]$ and $a[x \leftarrow v_2]$ terminate, but nothing guarantees that they will both terminate; it could be the case that one terminates and the other diverges, thus leaking some information to an observer who is willing to wait long enough. This is known as a "weak" noninterference statement.

Let us now come to the proof. Let a' stand for $(\lambda x.a)\ \langle v_1 \mid v_2 \rangle$. By Lam, we have $\emptyset \vdash \lambda x.a : (\tau \to \mathsf{bool}^L)^L$. By Bracket, we have $\emptyset \vdash \langle v_1 \mid v_2 \rangle : \tau$. By App, we have $\emptyset \vdash a' : \mathsf{bool}^L$. Besides,

$$\lfloor a' \rfloor_i \ = \ (\lambda x.a)\ v_i \ \longrightarrow \ a[x \leftarrow v_i] \ \longrightarrow^\star \ w_i$$

By reverse simulation (question 8), there must exist a value w such that $a' \longrightarrow^\star w$ holds and the projections of w are w_1 and w_2. Therefore, by the previous lemma (question 15), we have $w_1 = w_2$.