Faceted Execution of Policy-Agnostic Programs

Thomas H. Austin
UC Santa Cruz
taustin@ucsc.edu

Jean Yang
MIT CSAIL
jeanyang@csail.mit.edu

Cormac Flanagan
UC Santa Cruz
cormac@ucsc.edu

Armando Solar-Lezama
MIT CSAIL
asolar@csail.mit.edu

Abstract

It is important for applications to protect sensitive data. Even for simple confidentiality and integrity policies, it is often difficult for programmers to reason about how the policies should interact and how to enforce policies across the program. A promising approach is *policy-agnostic programming*, a model that allows the programmer to implement policies separately from core functionality. Yang et al. describe Jeeves [48], a programming language that supports information flow policies describing how to reveal sensitive values in different output channels. Jeeves uses symbolic evaluation and constraint-solving to produce outputs adhering to the policies. This strategy provides strong confidentiality guarantees but limits expressiveness and implementation feasibility.

We extend Jeeves with *faceted values* [6], which exploit the structure of sensitive values to yield both greater expressiveness and to facilitate reasoning about runtime behavior. We present a faceted semantics for Jeeves and describe a model for propagating multiple views of sensitive information through a program. We provide a proof of termination-insensitive non-interference and describe how the semantics facilitate reasoning about program behavior.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Languages, Security
Language design, run-time system, privacy, security

1. Introduction

It is increasingly important for applications to protect user privacy. Even for simple confidentiality and integrity policies, it is often difficult for programmers to reason about how the policies should interact and how to enforce policies across the program.

Policy-agnostic programming has the goal of allowing the programmer to implement core functionality separately from privacy policies. The programmer specifies policies as declarative rules and relies on the system to produce outputs adhering to the policies. Yang et al. describe Jeeves [48], a language that supports confidentiality policies describing how to reveal views of sensitive values based on the output channel. Sensitive values are pairs $\langle \ell \ ? \ v_H : v_L \rangle$, where v_H is the high-confidentiality value, v_L is the low-confidentiality value, and guard ℓ is a *label*. The initial implementation of Jeeves relies on symbolic evaluation and constraint-solving to produce outputs adhering to the policies. This strategy provides strong confidentiality guarantees, but at the cost of expressiveness and implementation feasibility. For instance, this implementation restricts recursion under symbolic conditionals and requires the cumulative constraint environment to persist.

In this paper, we present a faceted semantics for Jeeves that exploits the structure of sensitive values in order to increase expressiveness, facilitate reasoning about runtime behavior, and automatically enforce confidentiality policies. We base the Jeeves evaluation strategy on Austin et al.'s *faceted execution* [6], which manipulates explicit representations of sensitive values. With this strategy, labels variables are the only symbolic variables, allowing Jeeves to lift restrictions on the flow of sensitive values. To further improve ease of reasoning, Jeeves allows policies to only constrain labels to **low**. This guarantees that the constraint environment is always consistent, a property that allows for policy garbage-collection.

In this paper we make the following contributions:

- We present a faceted evaluation semantics for Jeeves, a language for automatically enforcing confidentiality policies. The execution model exploits the structure of sensitive values in order to increase expressiveness and to facilitate reasoning about runtime behavior.
- We present a dynamic semantics for faceted execution of Jeeves in terms of the λ^{jeeves} core language. We prove termination-insensitive non-interference, and policy compliance. We show that it is possible to reason about termination, policy consistency, and policy independence: properties that were not possible to reason about with the original semantics of Jeeves [48].
- We describe our implementation of Jeeves as an embedded domain-specific language in Scala and our experience using Jeeves to implement a conference management system that interacts with a web-based frontend and a persistent database.

2. Jeeves and Faceted Evaluation

We introduce faceted values into Jeeves in order to provide confidentiality guarantees, and compare its design with systems that rely on a declassification primitive and with the symbolic execution strategy used in an earlier implementation of Jeeves.

In this section, we present Jeeves using an ML-like concrete syntax, shown in Figure 1. Jeeves extends the λ-calculus with refer-

PLAS'13, June 20, 2013, Seattle, WA, USA.

ences, facets ($\langle \ell \; ? \; Exp_H : Exp_L \rangle$), a label construct for introducing labels that guard access to facets, and a **restrict** construct for introducing policies on labels. Jeeves statements include let-bound expressions and the effectful **print** statement.

2.1 Jeeves for Confidentiality

Jeeves allows the programmer to introduce a variable name that can be either "Alice" or "Anonymous" depending on the output channel:

```
let name: string = label a in
  <a ? "Alice" : "Anonymous" >
in ...
```

The above code introduces a label a that determines whether the private (high-confidentiality) value "Alice" or the public (low-confidentiality) "Anonymous" should be revealed. Labels take on the values { **low**, **high** }.

A simple policy on a sensitive value name is that the user must be the user alice to have high-confidentiality status:

```
let name: string = label a in
  restrict a: λ(c: User).(c == alice) in
    <a ? "Alice" : "Anonymous">
in ...
```

The **restrict** statement introduces a rule that strengthens the policy relating the output channel to the high-confidentiality value. To produce an assignment to label, the Jeeves system translates this rule to the declarative constraint $!(c == alice) \Rightarrow (a == \textbf{low})$. This rule is not used until evaluation of **print**, so other policies could further restrict the label to be **low**.

In Jeeves programs, sensitive values can be used as regular program values and effectful statements such as **print** require a context parameter:

```
let msg: string = "Sender is " + name in
print { alice } msg /* Output: "Sender is Alice" */
```

During program evaluation, the Jeeves runtime ensures that only the user alice can see her name appearing as the author in the string msg. User bob sees the string "Sender is Anonymous":

```
let msg = "Sender is " + name in
print { bob } msg     /* Output: "Sender is Anonymous" */
```

Unlike the previous implementation of Jeeves [48], which performs symbolic evaluation, Jeeves evaluation propagates *faceted values*, such as the following faceted value for msg:

```
<a ? "Sender is Alice" : "Sender is Anonymous">
```

Producing concrete outputs involves finding assignments to labels that satisfy the policies. The Jeeves system tries to assign labels to **high**, setting labels to **low** only if the policies require it. Assigning all labels to **low** always yields a consistent solution.

Jeeves allows the output channel to be sensitive:

```
let u: user = label b in
  restrict b: λ(c: User).(c == alice) in <b ? alice : nobody>
in print {u} u.name
```

There is a circular dependency: the context u is a sensitive value <b ? alice : nobody> guarded by a policy depending on the context. Such a policy allows two outcomes: b is **high** and we display alice.name to user alice and b is **low** and we display nobody.name to user nobody. The Jeeves runtime ensures maximal functionality: if the policies allow a labels to be **high** or **low**, the value will be **high**.

2.2 A Health Database in Jeeves

To show how to use Jeeves for real-world applications, let us build a simple health database with records of the following form:

```
type Patient { identity : User ref
             ; doctor: User ref
             ; meds: ( Medication list ) ref }
```

In these records, each of the fields identity, doctor, and meds could be sensitive values that show different values of the correct type to low-confidentiality output channels.

In this example, the output context has type HealthCtxt, which we define as follows:

```
type HealthCtxt { viewer: User, time: Date }
```

This context contains information not just for the viewer but also for the current date, allowing policies to define activation and expiration times for visibility.

The idiomatic way of attaching policies to a value is to create sensitive values for each field and then attach policies:

```
let mkPatient (identity : User) (doctor: User)
      ( meds: Medication list ) : Patient =
  label np, dp, mp in
  let p = { identity = <np ? identity : nobody>
          ; doctor   = <dp ? doctor : nobody>
          ; meds     = <mp ? meds : []> in
  addNamePolicy        p np;
  addDoctorPolicy      p dp;
  addMedicationsPolicy p mp;
in p
```

This function introduces labels, creates sensitive values, attaches policies to the labels, and returns the resulting Patient record. The function makes use of the add... Policy functions for attaching policies to the labels. The add... Policy functions take a Patient record and a labels and uses the record fields to attach a policy to the label. We define addMedicationsPolicy as:

```
let addMedicationsPolicy (p: Patient) (mp: label): unit =
  restrict mp: λ(c: HealthCtxt).
      (c.viewer == p.identity || c.viewer == p.doctor)
in ...
```

This policy sets the label to **low** unless the viewer is the patient or the patient's doctor. Jeeves automatically handles dependencies between policies and sensitive values: to have access to the medication list, the viewer needs to be able to see that their identity is equal to either p.identity or p.doctor.

2.3 Comparison to Declassification

Declassification primitives are used in many systems that make information flow guarantees. For instance, in an auction system the last bid might be considered private information until the auction has been completed, at which point the final bid should be made public. In a system with a declassification primitive instead of support for policy-agnostic programming, the relevant code to allow the release of this data might look something like the following:

```
let finalBid : (int ref) = ref label a in <a ? 42 : 0>
in let ...
  if currentTime < closeOfBid
    then finalBid := declassify ( finalBid )
in print { bidder } { ! finalBid }
```

At each print statement involving the final bid, the above code would need to be repeated. These declassification statements refine the core policy. The original paper on faceted values [6] shows how a declassification primitive may be designed for faceted evaluation.

The downside with this approach is that the effective policy for the system is littered throughout the code, leading to obvious

x			variables
ℓ			labels
p, r			primitives, records
$Label$	::=	**low** | **high**	labels
τ	::=	**int** | **bool** | **string** | **record** $\overrightarrow{x:\tau}$	types
	|	$\tau_2 \rightarrow \tau_2$ | τ **ref** | $Label$	
Exp	::=	x | p | r	expressions
	|	$\lambda x : \tau . Exp$	
	|	Exp_1 (op) Exp_2	
	|	**if** Exp_1 **then** Exp_t **else** Exp_f	
	|	$Exp_1\ Exp_2$	
	|	Exp_1 ; Exp_2	
	|	! Exp	
	|	$x := Exp$	
	|	$\langle \ell\ ?\ Exp_{high}\ :\ Exp_{low} \rangle$	
	|	**let** $x = Exp$ **in** Exp	
	|	label ℓ **in** Exp	
	|	**restrict** ℓ : Exp_p **in** Exp	
$Stmt$	::=	**let** $x : \tau = Exp$ **in** Exp_b	statements
	|	**print** $\{Exp_c\} Exp$	

Figure 1: Jeeves syntax.

Identity	Doctor
$\langle a$? alice : default$\rangle$	$\langle e$? erica : default$\rangle$
$\langle b$? bob : default$\rangle$	$\langle f$? fred : default$\rangle$
$\langle c$? claire : default$\rangle$	$\langle f'$? fred : default$\rangle$

Table 1. Sample patient records.

problems with the readability and maintainability of the policy-related code. In aspect-oriented [25] terminology, this approach suffers from a *tangling of aspects*.

While declassification can provide the flexibility needed in real-world systems, we argue that policy-agnostic programming is a more elegant solution. Since the policy code is kept separately, it is easier to get a holistic picture of the policy for data in the system, resulting in improved readability and maintainability. Also, since policy code can be kept separate, it might potentially be easier to protect policy mechanisms from abuse by malicious third parties than it would be to protect the use of a declassification primitive.

Continuing with the auction example, the policy code for the final bid is shown below. No matter how many channels we write to with the print statement, we do not need to repeat the policy code that determines if the value of finalBid can be released.

```
let finalBid : (int ref) = ref label a in
  restrict a: λ (x: bool). currentTime < closeOfBid in
    <a ? 42 : 0> /* The starting high bid is 42 */
in ...
```

As an additional benefit, we note that policy-agnostic programming offers a good solution for approaches such as secure multi-execution [17] that rely on separate processes. Since policy code is only applied when data is released, it eliminates the need for coordinating between processes (assuming that the policy is consistent).

2.4 Advantages of Faceted Execution over Symbolic Execution

Explicit representation of facets allows the runtime to prune branches of execution. Consider the following function, which takes a list of patients and a doctor and calls fold to count of the number of patients with a doctor field matching the doctor argument, on the records in Table 1 with doctor = erica.

```
let countPatients (patients: Patient list) (doctor: User): int
  = fold (λ (p: Patient) . λ (accum: int) .
           (if (p.doctor == doctor)
               then (accum + 1)
               else accum)
         0 patients)
in ...
```

Consider the behavior of this function on the records in Table 1 with a call to countPatients with doctor = erica. Evaluation of <e ? erica : default> == erica yields the expression <e ? erica == erica : default == erica>, which can be simplified to <e ? **true** : **false**>. Evaluation of faceted function applications creates a new faceted value resulting from applying the function to each facet. If e is in the set of path condition assumptions, then only the high facet is used. Evaluation of the conditional produces the expression

<e ? **if** (**true**) **then** (accum + 1) **else** accum : **if** (**false**) ... >,

which simplifies to <e ? accum + 1 : accum>. Depending on whether the output user is allowed to see that p.doctor is equal to erica, the resulting sum is either accum or accum + 1.

Storing an explicit representation for facets allows the runtime to prune branches. For instance, if the doctor is not equal to erica on either facet, then the faceted evaluation only needs to store a single value. The system may also prune facets based on path assumptions: if evaluation is occurring under the assumption that guard k is true, then subsequent evaluation can assume guard k. This is particularly advantageous when there are a small number of labels corresponding to a fixed set of principals.

3. Core Semantics

We model the semantics of Jeeves with λ^{jeeves}, a simple core language that extends the faceted execution semantics of Austin and Flanagan [6] with a declarative policy language for confidentiality. The λ^{jeeves} semantics describes how to evaluate faceted values, store policies, and use the policy environment to provide assignments to labels for producing concrete outputs. We use these semantics to prove non-interference and policy compliance guarantees.

We show the source syntax in Figure 2. The language λ^{jeeves} extends the λ-calculus with expressions for allocating references (ref e), dereferencing (!e), assignment (e_1:=e_2), creating faceted

Syntax:

$e ::=$		*Term*
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e_1\ e_2$	application
	ref e	reference allocation
	$!e$	dereference
	$e{:=}e$	assignment
	$\langle k\ ?\ e_1 : e_2 \rangle$	faceted expression
	label k in e	label declaration
	restrict(k, e)	policy specification
$S ::=$		*Statement*
	let $x = e$ in S	let statement
	print $\{e\}\ e$	print statement
$c ::=$		*Constant*
	f	file handle
	b	boolean
	i	integer
x, y, z		*Variable*
k, l		*Label*

Standard encodings:

$$
\begin{aligned}
\mathit{true} &\stackrel{\text{def}}{=} \lambda x.\lambda y.x \\
\mathit{false} &\stackrel{\text{def}}{=} \lambda x.\lambda y.y \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 &\stackrel{\text{def}}{=} (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
\text{if } e_1 \text{ then } e_2 &\stackrel{\text{def}}{=} \text{if } e_1 \text{ then } e_2 \text{ else } 0 \\
\text{let } x = e_1 \text{ in } e_2 &\stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1 \\
e_1\ \wedge_f\ e_2 &\stackrel{\text{def}}{=} \lambda x.e_1\ x\ \wedge\ e_2\ x \\
e_1\ \wedge\ e_2 &\stackrel{\text{def}}{=} \text{if } e_1 \text{ then } e_2 \text{ else } \mathit{false}
\end{aligned}
$$

Figure 2: The source language λ^{jeeves}

expressions ($\langle k\ ?\ e_1 : e_2 \rangle$), specifying policy (restrict(k, e)), and declaring labels (label k in e). Additional statements exist for let-statements (let $x = e$ in S) and printing output (print $\{e_1\}\ e_2$). Conditionals are encoded in terms of function application.

In λ^{jeeves}, values V contain *faceted values* of the form

$$\langle k\ ?\ V_{\text{H}} : V_{\text{L}} \rangle$$

A viewer authorized to see k-sensitive data will observe the private facet V_{H}. Other viewers will instead see V_{L}. For example, the value $\langle k\ ?\ 42 : 0 \rangle$ specifies a value of 42 that should only be viewed when k is **high** according to the policy associated with k. When the policy specifies **low**, the observed value should instead be 0.

A *program counter label* pc records when execution is influenced by public or private facets. For instance, in the conditional test

$$\text{if } (\langle k\ ?\ \mathit{true} : \mathit{false} \rangle) \text{ then } e_1 \text{ else } e_2$$

our semantics needs to evaluate both e_1 and e_2. The label k is added to pc during the evaluation of e_1. By doing so, our semantics records the influence of k on this computation. Similarly, $\overline{k}$ is added to pc during the evaluation of e_2 to record that the execution should have no effects observable to k. A *branch* h is either a label k or its negation $\overline{k}$. Therefore pc is a set of branches that never contains both k and $\overline{k}$, since that would reflect influences from both the private and public facet of a value.

The operation $\langle\!\langle\ pc\ ?\ V_1\ :\ V_2\ \rangle\!\rangle$ creates a faceted value. The value V_1 is visible when the specified policies correspond with *all* branches in pc. Otherwise, V_2 is visible instead.

$$
\begin{aligned}
\langle\!\langle\ \emptyset\ ?\ V_n : V_o\ \rangle\!\rangle &\stackrel{\text{def}}{=} V_n \\
\langle\!\langle\ \{k\} \cup \mathit{rest}\ ?\ V_n : V_o\ \rangle\!\rangle &\stackrel{\text{def}}{=} \langle k\ ?\ \langle\!\langle\ \mathit{rest}\ ?\ V_n : V_o\ \rangle\!\rangle : V_o \rangle \\
\langle\!\langle\ \{\overline{k}\} \cup \mathit{rest}\ ?\ V_n : V_o\ \rangle\!\rangle &\stackrel{\text{def}}{=} \langle k\ ?\ V_o : \langle\!\langle\ \mathit{rest}\ ?\ V_n : V_o\ \rangle\!\rangle \rangle
\end{aligned}
$$

For example, $\langle\!\langle\ \{k, l\}\ ?\ V_{\text{H}} : V_{\text{L}}\ \rangle\!\rangle$ returns $\langle k\ ?\ \langle l\ ?\ V_{\text{H}} : V_{\text{L}} \rangle : V_{\text{L}} \rangle$. We occasionally abbreviate $\langle\!\langle\ \{k\}\ ?\ V_{\text{H}} : V_{\text{L}}\ \rangle\!\rangle$ as $\langle\!\langle\ k\ ?\ V_{\text{H}} : V_{\text{L}}\ \rangle\!\rangle$.

The semantics are defined via the big-step evaluation relation:

$$\Sigma, e \Downarrow_{pc} \Sigma', V$$

This relation evaluates an expression e in the context of a store Σ and program counter label pc. It returns a modified store Σ' reflecting updates and a value V. In Figure 3 we show the evaluation rules, which uses additional runtime syntax (also shown in Figure 3).

Our language includes support for reference cells, which introduce additional complexities in handling implicit flows. The rule [F-REF] handles reference allocation (ref e). It evaluates an expression e, encoding any influences from the program counter pc to the value V, and adds it to the store Σ' at a fresh address a. Facets in V inconsistent with pc are set to 0. (Critically, to maintain non-interference, $\Sigma(a) = 0$ for all a not in the domain of Σ.)

The rule [F-DEREF] for dereferencing ($!e$) evaluates the expression e to a value V, which should either be an address or a faceted values where all of the "leaves" are addresses. The rule uses a helper function $\mathit{deref}(\Sigma', V, pc)$ (defined in Figure 3), which takes the addresses from V, retrieves the appropriate values from the store Σ', and combines them in the return value V'. As an optimization, addresses that are not compatible with pc are ignored.

The rule [F-ASSIGN] for assignment ($e_1{:=}e_2$) is similar to [F-DEREF]. It evaluates e_1 to a possibly faceted value V_1 corresponding to an address and e_2 to a value V'. The helper function $\mathit{assignOp}(\Sigma_2, pc, V_1, V')$ defined in Figure 3 decomposes V_1 into separate addresses, storing the appropriate facets of V' into the returned store Σ'. The changes to the store may come from both V_1 and pc.

The rule [F-LABEL] dynamically allocates a label (label k in e), adding a fresh label to the store with the default policy of $\lambda x.\mathit{true}$. Any occurrences of k in e are α-renamed to k' and the expression is evaluated with the updated store. Policies may be further refined (restrict(k, e)) by the rule [F-RESTRICT], which evaluates e to a policy V that should be either a lambda or a faceted value comprised of lambdas. The additional policy check is restricted by pc, so that policy checks cannot themselves leak data. It is then joined with the existing policy for k, ensuring that policies can only become more restrictive.

When a faceted expression $\langle k\ ?\ e_1\ :\ e_2 \rangle$ is evaluated, both sub-expressions must be evaluated in sequence, as per the rule [F-SPLIT]. The influence of k is added to the program counter for the evaluation of e_1 to V_1 and $\overline{k}$ for the evaluation of e_2 to V_2, tracking the branch of code being taken. The results of both evaluations are joined together in the operation $\langle\!\langle\ k\ ?\ V_1 : V_2\ \rangle\!\rangle$. As an optimization, only one expression is evaluated if the program counter already contains either k or $\overline{k}$, as indicated by the rules [F-LEFT] and [F-RIGHT].

Function application ($e_1\ e_2$) is somewhat complex in the presence of faceted values. The rule [F-APP] evaluates e_1 to V_1, which should either be a lambda or a faceted value containing lambdas, and evaluates e_2 to the function argument V_2. It then delegates the application ($V_1\ V_2$) to an auxiliary relation defined in Figure 4:

$$\Sigma, (V_1\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V'$$

Runtime Syntax

$$
\begin{array}{rcll}
e & \in & \mathit{Expr} & ::= \quad ... \mid a \\
\Sigma & \in & \mathit{Store} & = \quad (\mathit{Address} \rightarrow_p \mathit{Value}) \cup (\mathit{Label} \rightarrow \mathit{Value}) \\
R & \in & \mathit{RawValue} & ::= \quad c \mid a \mid (\lambda x.e) \\
a & \in & \mathit{Address} & \\
V & \in & \mathit{Val} & ::= \quad R \mid \langle k \ ?\ V_1 : V_2 \rangle \\
h & \in & \mathit{Branch} & ::= \quad k \mid \overline{k} \\
pc & \in & \mathit{PC} & = \quad 2^{\mathit{Branch}}
\end{array}
$$

Expression Evaluation Rules $\boxed{\Sigma, e \Downarrow_{pc} \Sigma', V}$

$$\frac{}{\Sigma, R \Downarrow_{pc} \Sigma, R} \quad \text{[F-VAL]}$$

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1 \quad \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V_2 \quad \Sigma_2, (V_1\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V'}{\Sigma, (e_1\ e_2) \Downarrow_{pc} \Sigma', V'} \quad \text{[F-APP]}$$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma', V' \quad a \notin dom(\Sigma') \quad V = \langle\!\langle\, pc \ ?\ V' : 0 \,\rangle\!\rangle}{\Sigma, (\text{ref}\ e) \Downarrow_{pc} \Sigma'[a := V], a} \quad \text{[F-REF]}$$

$$\frac{k \notin pc \ \text{ and } \overline{k} \notin pc \quad \Sigma, e_1 \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1 \quad \Sigma_1, e_2 \Downarrow_{pc \cup \{\overline{k}\}} \Sigma', V_2 \quad V' = \langle\!\langle\, k \ ?\ V_1 : V_2 \,\rangle\!\rangle}{\Sigma, \langle k \ ?\ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V'} \quad \text{[F-SPLIT]}$$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma', V \quad V' = \mathit{deref}(\Sigma', V, pc)}{\Sigma, !e \Downarrow_{pc} \Sigma', V'} \quad \text{[F-DEREF]}$$

$$\frac{k \in pc \qquad \Sigma, e_1 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k \ ?\ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad \text{[F-LEFT]}$$

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1 \quad \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V' \quad \Sigma' = \mathit{assign}(\Sigma_2, pc, V_1, V')}{\Sigma, e_1 := e_2 \Downarrow_{pc} \Sigma', V'} \quad \text{[F-ASSIGN]}$$

$$\frac{\overline{k} \in pc \qquad \Sigma, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k \ ?\ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad \text{[F-RIGHT]}$$

$$\frac{k' fresh \quad \Sigma[k' := \lambda x.true], e[k := k'] \Downarrow_{pc} \Sigma', V}{\Sigma, \text{label}\ k\ \text{in}\ e \Downarrow_{pc} \Sigma', V'} \quad \text{[F-LABEL]}$$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma_1, V \quad \Sigma' = \Sigma_1[k := \Sigma_1(k) \wedge_f \langle\!\langle\, pc \cup \{k\} \ ?\ V : \lambda x.true \,\rangle\!\rangle]}{\Sigma, \text{restrict}(k, e) \Downarrow_{pc} \Sigma', V} \quad \text{[F-RESTRICT]}$$

Auxiliary Functions

$$
\begin{array}{lcl}
\mathit{deref} : \mathit{Store} \times \mathit{Val} \times \mathit{PC} & \rightarrow & \mathit{Val} \\
\mathit{deref}(\Sigma, a, pc) & = & \Sigma(a) \\
\mathit{deref}(\Sigma, \langle k \ ?\ V_{\text{H}} : V_{\text{L}} \rangle, pc) & = & \begin{cases} \mathit{deref}(\Sigma, V_{\text{H}}, pc) & \text{if } k \in pc \\ \mathit{deref}(\Sigma, V_{\text{L}}, pc) & \text{if } \overline{k} \in pc \\ \langle\!\langle\, k \ ?\ \mathit{deref}(\Sigma, V_{\text{H}}, pc) : \mathit{deref}(\Sigma, V_{\text{L}}, pc) \,\rangle\!\rangle & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcll}
\mathit{assign} : \mathit{Store} \times \mathit{PC} \times \mathit{Val} \times \mathit{Val} & \rightarrow & \mathit{Store} & \\
\mathit{assign}(\Sigma, pc, a, V) & = & \Sigma[a := \langle\!\langle\, pc \ ?\ V : \Sigma(a) \,\rangle\!\rangle] & \\
\mathit{assign}(\Sigma, pc, \langle k \ ?\ V_{\text{H}} : V_{\text{L}} \rangle, V) & = & \Sigma' & \text{where } \Sigma_1 = \mathit{assign}(\Sigma,\ pc \cup \{k\}, V_{\text{H}}, V) \\
 & & & \text{and } \ \Sigma' = \mathit{assign}(\Sigma_1, pc \cup \{\overline{k}\}, V_{\text{L}}, V)
\end{array}
$$

Figure 3: Faceted Evaluation Semantics

Application Rules $\boxed{\Sigma, (V_1\ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V'}$

[FA-FUN]
$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', V'}{\Sigma, ((\lambda x.e)\ V) \Downarrow_{pc}^{\text{app}} \Sigma', V'}$$

[FA-SPLIT]
$$\frac{\begin{array}{c} k \notin pc \qquad \overline{k} \notin pc \\ \Sigma, (V_{\text{H}}\ V_2) \Downarrow_{pc \cup \{k\}}^{\text{app}} \Sigma_1, V'_{\text{H}} \\ \Sigma_1, (V_{\text{L}}\ V_2) \Downarrow_{pc \cup \{\overline{k}\}}^{\text{app}} \Sigma', V'_{\text{L}} \\ V' = \langle\!\langle\, k\ ?\ V'_{\text{H}} : V'_{\text{L}} \,\rangle\!\rangle \end{array}}{\Sigma, (\langle k\ ?\ V_{\text{H}} : V_{\text{L}}\rangle\ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V'}$$

[FA-LEFT]
$$\frac{\begin{array}{c} k \in pc \\ \Sigma, (V_{\text{H}}\ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V \end{array}}{\Sigma, (\langle k\ ?\ V_{\text{H}} : V_{\text{L}}\rangle\ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V}$$

[FA-RIGHT]
$$\frac{\begin{array}{c} \overline{k} \in pc \\ \Sigma, (V_{\text{L}}\ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V \end{array}}{\Sigma, (\langle k\ ?\ V_{\text{H}} : V_{\text{L}}\rangle\ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V}$$

Statement Evaluation Rules $\boxed{\Sigma, S \Downarrow V_p, f : R}$

[F-LET]
$$\frac{\Sigma, e \Downarrow_{\emptyset} \Sigma', V \qquad \Sigma, S[x := V] \Downarrow V_p, f : R}{\Sigma, \text{let } x = e \text{ in } S \Downarrow V_p, f : R}$$

[F-PRINT]
$$\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{\emptyset} \Sigma_1, V_f \\ \Sigma_1, e_2 \Downarrow_{\emptyset} \Sigma_2, V_c \\ e_p = \lambda x.true\ \wedge_f\ \Sigma_2(k_1)\ \wedge_f\ \ldots\ \wedge_f\ \Sigma_2(k_n) \\ \Sigma_2, e_p\ V_f \Downarrow_{\emptyset} \Sigma_3, V_p \\ \{\ k_1\ \ldots\ k_n\ \} \text{ includes all labels in } V_f,\ V_c,\ V_p \end{array}}{\text{pick } pc \text{ such that } pc(V_f) = f, pc(V_c) = R,\ pc(V_p) = true \quad \Sigma, \text{print } \{e_1\}\ e_2 \Downarrow V_p, f : R}$$

Semantics for Derived Encodings

[F-IF-TRUE]
$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, true \qquad \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$$

[F-IF-FALSE]
$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, false \qquad \Sigma_1, e_3 \Downarrow_{pc} \Sigma', V}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$$

[F-IF-SPLIT]
$$\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k\ ?\ V_H : V_L\rangle \\ e_H = \text{if } V_H \text{ then } e_2 \text{ else } e_3 \\ e_L = \text{if } V_L \text{ then } e_2 \text{ else } e_3 \\ \Sigma_1, \langle k\ ?\ e_H : e_L\rangle \Downarrow_{pc} \Sigma', V \end{array}}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$$

Figure 4: Faceted Evaluation Semantics for Application and Statements

This relation breaks apart faceted values and tracks the influences of the labels through the rules [FA-SPLIT], [FA-LEFT], and [FA-RIGHT] in a similar manner to the rules [F-SPLIT], [F-LEFT], and [F-RIGHT] discussed previously. The actual application is handled by the [FA-FUN] rule. The body of the lambda $(\lambda x.e)$ is evaluated with the variable x replaced by the argument V.

Conditional branches (if e_1 then e_2 else e_3) are Church-encoded as function calls for the sake of simplicity. However, Figure 4 shows direct rules for evaluating conditionals in the presence of faceted values. Under the rule [F-IF-SPLIT], If the condition e_1 evaluates to a faceted value $\langle k\ ?\ V_H : V_L\rangle$, the if statement is evaluated twice with V_H and V_L as the conditional tests.

While expressions handle most of the complexity of faceted values, statements in λ^{jeeves} illustrate how faceted values may be concretized when exporting data to an external party. The semantics for statements are defined via the big-step evaluation relation:

$$\Sigma, S \Downarrow V_p, f : R$$

The rules for statements are specified in Figure 4. The rule [F-LET] handles let expressions (let x = e in S), evaluating an expression e to a value V, performing the proper substitution in statement S. The rule [F-PRINT] handles print statements (print $\{e_1\}\ e_2$), where the result of evaluating e_2 is printed to the channel resulting from the evaluation of e_1. Both the channel V_f and the value to print V_c may be faceted values, and furthermore, we must select the facets that correspond with our specified policies. The expression e_p contains all relevant policies included in the store Σ_2. It is evaluated and applied to V_f, returning the policy check V_p that is a faceted value containing booleans. A program counter pc is chosen such that the policies are satisfied, which determines the channel f and the value to print R. Note that there exists a $pc' \in PC$ where all branches are set to **low**, which may always be displayed, thereby ensuring that there is always at least one valid choice for pc.

This property allows garbage collection of policies and facets. Because the constraints are always consistent, the only set of policies relevant to an expression e to output are associated with the transitive closure of labels L_e appearing in e and the policies associated with L_e. Thus any policy associated with an out-of-scope variable may be garbaged-collected. In addition, once a policy has been set to the equivalent of $\lambda x.false$ for a label k, k-sensitive facets and policies cannot be used in a print statement. These properties are advantages over the previous symbolic-execution strategy used by an earlier implementation of Jeeves [48], since the earlier approach could introduce inconsistent policies.

4. Properties

We prove that a single execution with faceted values is equivalent to multiple different executions without faceted values. From this we know that if execution terminates on each facet of a sensitive value, then faceted execution terminates. Jeeves does not have this property because execution keeps sensitive values as symbolic; thus Jeeves restricts applications of recursive functions.

We also prove that the system cannot leak sensitive information either via the output or by the choice of output channel.

4.1 Projection Theorem

A key property of faceted evaluation is that it simulates multiple executions. In other words, a single execution with faceted values *projects* to multiple different executions without faceted values.

$$
\begin{array}{l}
pc : \mathit{Expr}\ (\text{with facets}) \to \mathit{Expr}\ (\text{with fewer facets}) \\
pc(\langle k\ ?\ e_1 : e_2\rangle) = \begin{cases} pc(e_1) & \text{if } k \in pc \\ pc(e_2) & \text{if } \overline{k} \in pc \\ \langle k\ ?\ pc(e_1) : pc(e_2)\rangle & \text{otherwise} \end{cases} \\
pc(\langle k\ ?\ V_1 : V_2\rangle) = \begin{cases} pc(V_1) & \text{if } k \in pc \\ pc(V_2) & \text{if } \overline{k} \in pc \\ pc(V_1) & \text{if } pc(V_1) = pc(V_2) \\ \langle k\ ?\ pc(V_1) : pc(V_2)\rangle & \text{otherwise} \end{cases} \\
pc(\ldots) = \text{compatible closure}
\end{array}
$$

We extend pc to project faceted stores $\Sigma \in \mathit{Store}$ into stores with fewer facets.

$$
\begin{array}{rcl}
pc : \mathit{Value} & \to & \mathit{Value} \\
pc(\Sigma) & = & \lambda a.\ pc(\Sigma(a)) \ \cup\ \lambda k.\ pc(\Sigma(k))
\end{array}
$$

Thus pc projection does not remove policies, it only removes some labels on expressions or values. We say that pc_1 and pc_2 are *consistent* if

$$\neg\exists k.\ (k \in pc_1 \wedge \overline{k} \in pc_2)\ \vee\ (\overline{k} \in pc_1 \wedge k \in pc_2)$$

We note some key lemmas regarding projection.

Lemma 1. *If* $V = \langle\!\langle\, pc\ ?\ V_1 : V_2 \,\rangle\!\rangle$ *then* $\forall q \in PC$

$$q(V) \ =\ \begin{cases} \langle\!\langle\, pc \setminus q\ ?\ q(V_1) : q(V_2) \,\rangle\!\rangle & \text{if } q \text{ is consistent with } pc \\ q(V_2) & \text{otherwise} \end{cases}$$

Lemma 2. *If* $V' = \mathit{deref}(\Sigma, V, pc)$ *then* $\forall q \in PC$ *where* q *is consistent with* pc, $q(V') = \mathit{deref}(q(\Sigma), q(V), pc \setminus q)$.

Lemma 3. *If* $\Sigma' = \mathit{assign}(\Sigma, pc, V_1, V_2)$ *then* $\forall q \in PC$

$$q(\Sigma') \ =\ \begin{cases} \mathit{assign}(q(\Sigma), pc \setminus q, q(V_1), q(V_2)) & \text{if } q \text{ consistent with } pc \\ q(\Sigma) & \text{otherwise} \end{cases}$$

Lemma 4. *Suppose* pc *and* q *are not consistent and that either*

$$\begin{array}{rl} & \Sigma, e \Downarrow_{pc} \Sigma', V \\ \mathit{or} & \Sigma, (V_1 V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V \end{array}$$

Then $q(\Sigma) = q(\Sigma')$.

The following projection theorem shows how a single faceted evaluation simulates (or projects) to multiple executions, each with fewer facets, or possibly with no facets at all (if for each label k in the program, either k or $\overline{k}$ is in q).

Theorem 1 (Projection Theorem)**.** *Suppose*

$$\Sigma, e \Downarrow_{pc} \Sigma', V$$

Then for any $q \in PC$ *where* pc *and* q *are consistent*

$$q(\Sigma), q(e) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$$

This theorem significantly extends the projection property of Austin and Flanagan [6], in that it supports dynamic label allocation and flexible, dynamically specified policies, and is also more general in that it can either remove none, some, or all top-level labels in a program, depending on the choice of the projection $PC\ q$. A full proof of the projection theorem is available in the appendix.

4.2 Termination-Insensitive Non-Interference

The projection property captures that data from one collection of executions, represented by the corresponding set of branches pc, does not leak into any incompatible views, thus enabling a straightforward proof of non-interference.

Two faceted values are *pc-equivalent* if they have identical values for the set of branches pc. This notion of pc-equivalence naturally extends to stores ($\Sigma_1 \sim_{pc} \Sigma_2$) and expressions ($e_1 \sim_{pc} e_2$):

$$
\begin{array}{rcl}
(V_1 \sim_{pc} V_2) & \text{iff} & pc(V_1) = pc(V_2) \\
(\Sigma_1 \sim_{pc} \Sigma_2) & \text{iff} & pc(\Sigma_1) = pc(\Sigma_2) \\
(e_1 \sim_{pc} e_2) & \text{iff} & pc(e_1) = pc(e_2)
\end{array}
$$

The notion of pc-equivalence and the projection theorem enable a concise statement and proof of termination-insensitive non-interference.

Theorem 2 (Termination-Insensitive Non-Interference)**.**
Let pc *be any set of branches. Suppose* $\Sigma_1 \sim_{pc} \Sigma_2$ *and* $e_1 \sim_{pc} e_2$, *and that:*

$$\Sigma_1, e_1 \Downarrow_\emptyset \Sigma_1', V_1 \qquad \Sigma_2, e_2 \Downarrow_\emptyset \Sigma_2', V_2$$

Then $\Sigma_1' \sim_{pc} \Sigma_2'$ *and* $V_1 \sim_{pc} V_2$.

Proof. By the Projection Theorem:

$$
\begin{array}{l}
pc(\Sigma_1), pc(e_1) \Downarrow_\emptyset pc(\Sigma_1'), pc(V_1) \\
pc(\Sigma_2), pc(e_2) \Downarrow_\emptyset pc(\Sigma_2'), pc(V_2)
\end{array}
$$

The pc-equivalence assumptions imply that $pc(\Sigma_1) = pc(\Sigma_2)$ and $pc(e_1) = pc(e_2)$. Hence $pc(\Sigma_1') = pc(\Sigma_2')$ and $pc(V_1) = pc(V_2)$ since the semantics is deterministic. □

4.3 Termination-Insensitive Policy Compliance

While we have shown non-interference for a set of labels, the labels do not directly correspond to the output revealed to a given observer. In this section we show how we can prove termination-insensitive *policy compliance*; data is revealed to an external observer only if it is allowed by the policy specified in the program. Thus if S_1 and S_2 are terminating programs that differ only in k-labeled components and the computed policy V_i for each program does not permit revealing k-sensitive data to the output channel, then the set of possible outputs from each program is identical. Here, an output $f : v$ combines both the output channel f and the value v, to ensure that sensitive information is not leaked either via the output value or by the choice of output channel.

Theorem 3. *Suppose for* $i \in 1, 2$*:*

$$
\begin{array}{l}
S_i = \mathit{print}\ \{e\}\ C[\langle k\ ?\ e_i : e_l\rangle] \\
\emptyset, S_1 \Downarrow V_{p1}, f_1 : R_1 \\
\emptyset, S_2 \Downarrow V_{p2}, f_2 : R_2 \\
\forall pc'\ \text{with}\ k \in pc',\ pc'(V_{p1}) \neq \mathit{true}\ \mathit{and}\ pc'(V_{p2}) \neq \mathit{true}
\end{array}
$$

Then $\{\ f : R \mid \emptyset, S_1 \Downarrow V_p, f : R\ \} = \{\ f : R \mid \emptyset, S_2 \Downarrow V_p, f : R\ \}$.

Proof. We show left-to-right containment as follows. (The converse containment holds by a similar argument.) Suppose

$$
\begin{array}{l}
\emptyset, S_1 \Downarrow V_{p1}, f_1 : R_1 \\
\emptyset, S_2 \Downarrow V_{p2}, f_2 : R_2
\end{array}
$$

Then by the [F-PRINT] rule

$$\begin{array}{l}\emptyset, e \Downarrow_{\emptyset} \Sigma_{11}, V_{f1}\\ \Sigma_{11}, C[\langle k \ ? \ e_1 : e_l\rangle] \Downarrow_{\emptyset} \Sigma_{12}, V_{c1}\\ e_{p1} = \Sigma_{12}(k_1) \ \wedge_f \ \ldots \ \wedge_f \ \Sigma_{12}(k_n) \text{ where } \{ k_1 \ldots k_n \}\\ \quad \text{includes all labels in } V_{f1},\ V_{c1},\ V_{p1}\\ \Sigma_{12}, e_{p1}\ V_{f1} \Downarrow_{\emptyset} \Sigma_{13}, V_{p1}\\ pc_1(V_{f1}) = f_1,\ pc_1(V_{c1}) = R_1,\ pc_1(V_{p1}) = true, \text{ so } \overline{k} \in pc_1\end{array}$$

Also by the [F-PRINT] rule for the second execution

$$\begin{array}{l}\emptyset, e \Downarrow_{\emptyset} \Sigma_{21}, V_{f2}\\ \Sigma_{21}, C[\langle k \ ? \ e_2 : e_l\rangle] \Downarrow_{\emptyset} \Sigma_{12}, V_{c2}\\ e_{p2} = \Sigma_{22}(k_1) \ \wedge_f \ \ldots \ \wedge_f \ \Sigma_{22}(k_n) \text{ where } \{ k_1 \ldots k_n \}\\ \quad \text{includes all labels in } V_{f2},\ V_{c2},\ V_{p2}\\ \Sigma_{22}, e_{p2}\ V_{f2} \Downarrow_{\emptyset} \Sigma_{23}, V_{p2}\\ pc_2(V_{f2}) = f_2,\ pc_2(V_{c2}) = R_2,\ pc_2(V_{p2}) = true, \text{ so } \overline{k} \in pc_2\end{array}$$

By determinism, $\Sigma_{11} = \Sigma_{21}$ $V_{f1} = V_{f2}$.
Also, $C[\langle k \ ? \ e_1 : e_l\rangle] \sim_{\{\overline{k}\}} C[\langle k \ ? \ e_2 : e_l\rangle]$.
Hence by the projection theorem

$$\begin{array}{lll}\Sigma_{12} \sim_{\{\overline{k}\}} \Sigma_{22} & V_{c1} \sim_{\{\overline{k}\}} V_{c2} & e_{p1} \sim_{\{\overline{k}\}} e_{p2}\\ \Sigma_{13} \sim_{\{\overline{k}\}} \Sigma_{23} & V_{p1} \sim_{\{\overline{k}\}} V_{p2} & \end{array}$$

Pick $pc_2 = pc_1$. Then $R_2 = R_1$ and $f_2 = f_1$ as required. □

5. Scala Implementation

We have implemented Jeeves as an embedded domain-specific language in the Scala programming language [37]. We use Scala's overloading capabilities to implement faceted execution, constraint collection, and interaction with the Z3 SMT solver [33].[1] The implementation defines Scala classes for integers, booleans, objects, and functions that support operations over expressions e or faceted expressions $\langle k \ ? \ e_H : e_L\rangle$. The implementation overloads operators on these types so that faceted values can be used interchangeably with concrete values. For instance, the Expr[Int] class represents the type of concrete and faceted integer expressions. We use Scala's implicit type conversions to lift concrete Scala values.

We have implemented a Scala trait that stores a runtime environment to support methods creating labels, declaring policies, and concretizing expressions. The trait maintains the logical and default constraint environments as lists of functions of type Expr[T] ⇒ Formula, where Formula is a boolean expression that may contain facets. We have a partial evaluation procedure that simplifies expressions based on the value of each facet and the current path assumptions.

To assign values to labels, the implementation evaluates policies according to the context and heap state and invokes Z3 for resolving constraints. Our implementation translates constraints to the QF_LIA logic of SMT-LIB2 [7]. There are only quantifier-free boolean constraints. Labels are the only free variables. We use incremental scripting to implement default values according to default logic [2]. The implementation relies on Scala's support for dynamic invocation to resolve field dereferences. We use zero values (null, 0, or false) to represent undefined fields in SMT.

Our Jeeves library interface supports the introduction of labels, declaration of policies, creation of sensitive variables, and concretization of sensitive expressions. It also has functions for assignment, conditionals, and function evaluation according to the Jeeves semantics.

The library has the following API methods for introducing sensitive values and policies:

```
def mkLabel: Label
def restrict (lvar: Label, f: Expr[T] ⇒ Formula)
def mkSensitive(lvar: Label, high: Expr[T], low: Expr[T]):
    Expr[T]
```

[1] The code is available at `http://code.google.com/p/jeeveslib/`.

The programmer introduces labels, which are boolean logic variables mapped to HIGH and LOW, into scope by calling the mkLabel method. The restrict method for introducing policies takes a labels and a function that takes a context expression and returns a formula. The library stores policy functions and applies them with respect to the output context and output heap state to produce concrete outputs adhering to the policies. The programmer introduces sensitive values through the mkSensitive method, which takes a labels along with high-confidentiality and low-confidentiality views. To support evaluation with sensitive expressions, programs should accommodate values of type Expr[T] (*e.g.* IntExpr rather than BigInt). The library has methods for producing concrete state:

```
def concretize [T](ctxt: Expr[T], e: Expr[T]): T
def jprint [T](ctxt: Expr[T], e:Expr[T]): Unit
```

These functions take a context and an expression, both of which may be sensitive, and provides assignments to the labels to produce concrete views that adhere to the policies. The implementation treats the mutable state as part of the context in the concretize call to ScalaSMT. All classes that are used in constraint must extend the JeevesRecord class. The set of allocated JeevesRecords is supplied at concretization. This way, policies that refer to mutable parts of the heap will produce correct constraints for the snapshot of the system at concretization. The library provides support for evaluating conditionals and function applications:

```
def jif [T] (c: Formula, t : Unit ⇒ T, f: Unit ⇒ T): Unit
def jfun [A, B] (f : FunctionExpr[A, B], arg : A): B
```

The library stores the path condition as a set of labels and their negations. The jif method evaluates the condition and manages the path condition for each branch appropriately in order to produce a potentially faceted result. The jfun method behaves similarly. Both of these methods check against the path condition to avoid performing unnecessary computations.

6. Case Study: Conference Management

We have implemented JConf, a conference management system that uses Jeeves for confidentiality guarantees. The JConf backend interacts with a web-based frontend and a persistent database store. The original JConf implementation, written using an earlier implementation of Jeeves that used symbolic evaluation rather than faceted execution, was up for several hours at a time and a cumulative total of several days, processing submissions for the Student Research Competition for the Programming Language Design and Implementation Conference 2012. Our experience with this system motivated some of the design decisions in Jeeves, including the decision to use faceted execution.

The implementation of JConf has a backend written in Jeeves that defines Scala objects corresponding to data types (for instance, for representing users and papers) and associates policies with fields of these objects; object constructors add the policies. The backend contains functionality that supports the creation of, lookup of, updates to, and search over these objects. The frontend web code, written using the Scalatra web framework [1], makes calls to the backend functionality and to accessors of the objects. The JConf backend contains a layer that interacts with a MySQL database for persistent storage. The frontend web code and database-interaction code remain agnostic to the policies: the same code is used, for instance, to render a page (for instance, displaying appropriately anonymized information about a paper review) for an author, a reviewer, and a program committee member. Interaction with the Jeeves backend takes on the order of seconds; solving in the Z3

File	Total LOC	Policy LOC
ConfUser.scala	212	21
PaperRecord.scala	304	75
PaperReview.scala	116	32
ConfContext.scala	6	0
Backend + Squeryl	800	0
Frontend (Scalatra)	629	0
Frontend (SSP)	798	0
Total	**2865**	**128**

Table 2. Lines of code vs. policy in JConf.

SMT solver takes well under one second. The bulk of execution is involved in propagating sensitive values.

The JConf conference management system provides support for creating new users and updating profiles, creating papers and updating information, submitting papers, assigning reviews, and reviewing papers. We show the breakdown of the system in Table 2: classes describing the data (users, papers, paper reviews, and the context), backend code for accessing the data (including the interface to the database), the Scalatra code for the frontend web request handlers, and the Scalatra Server Page (SSP) code defining the browser pages themselves.

Policy code (calls to `mkLabel`, `mkSensitive`, `restrict`, and `concretize`) is concentrated in the data classes, enabling modular updates to the policy and core functionality. For instance, we can change the review process from double-blind to single-blind simply by tweaking the policies associated with paper and review fields. The policy code makes up less than 5% of the total lines of code.

The programmer defines a *getter*, a *setter*, and a *show function* for each sensitive field. The getter returns the sensitive value, the setter creates a new sensitive value based on the views, and the show function calls `concretize` to return a concrete value of the appropriate type. The programmer creates the sensitive value with a label in scope to which policies can be attached. It may make sense to share labels between field for some applications. The frontend calls the show functions to access concrete versions of values. We use the database only for persistent storage; all queries use Jeeves to ensure policy compliance.

7. Related Work

Jeeves follows a line of research in language-based information flow that began with the work of Denning [15, 16]. Sabelfeld and Myers [41] survey much of the literature in the field in subsequent years. Volpano et al. [47] develop a type system that guarantees non-interference for the language that Denning outlines. Heintze and Riecke [22] design a type system guaranteeing non-interference for a functional language, extended with constructs for reference cells, concurrency, and integrity guarantees. Smith [43] discusses some of the core concepts in information flow analysis.

Languages for verifying information flow security include Jif [34], Fine [12], F* [45], and Ur/Web [13]. Nanevski et al. [36] verify information flow policies through the use of dependent types. Hunt and Sands [23] describe a flow-sensitive type system. Zhang et al. [51] describe a type-based approach to mitigating timing channels. These static approaches have no dynamic overhead. Myers [34] discusses JFlow, a variant of Java with security types to provide strong information flow guarantees. Le Guernic et al. [20] examine code from branches not taken, increasing precision at the expense of run-time performance overhead. Shroff et al. [42] use a purely-dynamic analysis to track variable dependencies and reject more insecure programs over time. Jeeves mitigates programmer burden by guaranteeing that programs adhere to the desired properties by construction, but with dynamic overhead. Systems like Fabric [28] combine static and dynamic techniques, but the focus of the dynamic analyses is on checking rather than on helping the programmer produce correct outputs. Russo and Sabelfeld [40] discuss trade-offs between static and dynamic analyses.

Jeeves is also related to systems that provide support for inserting information flow checks. Broberg and Sands [9] describe flow locks for dynamic information flow policies. Birgisson et al. [8] show how capabilities can guarantee information flow policies. The system-level data flow framework Resin [49] allows the programmer to insert checking code to be executed at output channels. Privacy Integrated Queries (PINQ) [30] is a capability-based system that enforces differential privacy policies in declarative database queries. SEAL [35] specifies policies for label-based access control systems.

There are parallels with dynamic approaches that run multiple executions for security guarantees. Capizzi et al.'s *shadow executions* [10] maintain confidentiality by running both a public and private copy of the application. The public copy can communicate with the outside world but cannot access private data; the private copy has access to private information but lacks network access. Devriese and Piessens' *secure multi-execution* strategy [17] applies this approach to JavaScript code. Kashyap et al. [24] discuss properties of timing and termination for secure multi-execution.

Austin and Flanagan [6] simulate secure multi-execution with a single execution through the use of faceted values, avoiding overhead when code does not depend on confidential data, noticeably improving performance. The same paper also show how declassification may be performed with facets, though with Jeeves's policies, declassification is largely unnecessary. Rozzle [27] uses symbolic execution to detect malware, treating environment-specific data as symbolic and exploring both paths whenever a value branches on a symbolic value in a manner similar to faceted evaluation. Jeeves allows more complex policies, for instance ones that may depend on sensitive values. Faceted values are related to the non-interference work by Pottier and Simonet for *Core ML* [38]. Their proof approach involves a Core ML^2 language that has expression pairs and value pairs, similar to faceted expressions and faceted values respectively. While their approach is not intended as a dynamic enforcement mechanism, their work does include evaluation rules for Core ML^2 that may supplement understanding of faceted values.

The automatic policy enforcement is related to work in constraint functional programming and executing specifications. Like constraint functional languages, Jeeves integrates declarative constraints into a non-declarative programming model. Jeeves differs from languages such as Mercury [44], Escher [29], Curry [21], and Kaplan [26], which support rich operations over logic variables at the cost of potentially expensive runtime search and undecidability. In Jeeves, the logical environment is always consistent and the runtime only performs decidable search routines. Jeeves differs from the Squander system [31] for unified execution of imperative and declarative code in that Jeeves propagates constraints alongside the core program rather than executing isolated constraint-based subprocedures. As with relaxed approximate programs [11], Jeeves nondeterministically provides an acceptable output for a specific class of acceptability properties.

Jeeves is related to declarative domain-specific languages. Frenetic [19] provides a query language programming distributed collections of network switches. Engage [18] uses constraints to mitigate programmer burden in configuring, installing, and managing applications. Jeeves differs in that its target domain of privacy is cross-cutting with respect to other functionality.

Declassification is an important area of research for information flow analysis and overlaps a great deal with the applications of com-

putable policies. Zdancewic [50] uses integrity labels to provide *robust declassification*. permitting only high-integrity declassification decisions. Askarov and Myers [3] consider a similar approach for endorsement, *checked endorsements*. arguing that *checked endorsements* are needed to prevent an attacker from endorsing an unauthorized declassification. Chong and Myers [14] use a framework for application-specific declassification policies. Askarov and Sabelfeld [4] study a declassification framework specifying what and where data is released. Vaughan and Chong [46] infer declassification policies for Java programs.

The termination channel is another area of particular concern for information flow analysis. Askarov et al. [5] highlight complications of *intermediary output channels*, which allow an attacker to observe the output of a program during its execution, and discuss *progress-sensitive noninterference*. Moore et al. [32] include the concept of a budget for possible information loss through the termination channel, terminating the program when the budget has been exceeded. Rafnnson et al. [39] buffer output to reduce data lost from intermediary output channels and termination behavior.

8. Conclusions

Jeeves allows the programmer to implement core functionality separately from confidentiality policies. Our execution strategy exploits the structure of sensitive values to facilitate reasoning about runtime behavior. We present a semantics for faceted execution of Jeeves in terms of the λ^{jeeves} core language, and prove non-interference and policy compliance for confidentiality. We describe how Jeeves enables reasoning about termination, policy consistency, and policy independence. Finally, we describe our implementation of Jeeves in Scala and our experience using Jeeves to implement an end-to-end conference management system.

Acknowledgments

This work was supported by NSF grant CNS-0905650, by the U.S. Government under the DARPA UHPC, and by Facebook Fellowship programs. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government or of Facebook Inc.

References

[1] Scalatra: A tiny Sinatra-like web framework for Scala. `http://www.scalatra.org/`.

[2] G. Antoniou. A tutorial on default logics. *ACM Computing Surveys (CSUR)*, 31(4), 1999.

[3] A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In *European Symposium on Programming (ESOP)*, 2010.

[4] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2009.

[5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08*. Springer-Verlag, 2008.

[6] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Symposium on Principles of Programming Languages (POPL)*, 2012.

[7] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In *SMT Workshop*, 2010.

[8] A. Birgisson, A. Russo, and A. Sabelfeld. Capabilities for information flow. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2011.

[9] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *European Symposium on Programming (ESOP)*, 2006.

[10] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. Sistla. Preventing information leaks through shadow executions. In *Annual Computer Security Applications Conference (ACSAC)*, dec 2008.

[11] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[12] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[13] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[14] S. Chong and A. C. Myers. Security policies for downgrading. In *Conference on Computer and Communications Security (CCS)*, 2004.

[15] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976.

[16] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[17] D. Devriese and F. Piessens. Noninterference through secure multi-execution. *IEEE Symposium on Security and Privacy*, 2010.

[18] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[19] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *International Conference on Functional Programming (ICFP)*, 2011.

[20] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *ASIAN*, 2006.

[21] M. Hanus, H. Kuchen, J. J. Moreno-Navarro, R. Aachen, and I. Ii. Curry: A truly functional logic language, 1995.

[22] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages (POPL)*, 1998.

[23] S. Hunt and D. Sands. On flow-sensitive security types. In *Symposium on Principles of Programming Languages (POPL)*, 2006.

[24] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *IEEE Symposium on Security and Privacy*, 2011.

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, 1997.

[26] A. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *Symposium on Principles of Programming Languages (POPL)*, 2012.

[27] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. Technical Report MSR-TR-2011-94, Microsoft Research Technical Report, 2011.

[28] 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 *Symposium on Operating System Principles (SOSP)*, 2009.

[29] J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.

[30] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *International Conference on Management of Data (SIGMOD)*, 2009.

[31] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *International Conference on Software Engineering (ICSE)*, 2011.

[32] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *ACM Conference on Computer and Communications Security*, pages 881–893, 2012.

[33] L. D. Moura and N. Björner. Z3: An efficient SMT solver. In *Tools and algorithms for the construction and analysis of systems (TACAS)*, 2008.

[34] A. C. Myers. JFlow: Practical mostly-static information flow control. 1999.

[35] P. Naldurg and R. K. R. Seal: a logic programming framework for specifying and verifying access control models. In *ACM Symposium on Access Control Models and Technologies*, pages 83–92, 2011.

[36] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, 2011.

[37] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

[38] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25 (1), 2003.

[39] W. Rafnsson and A. Sabelfeld. Limiting information leakage in event-based communication. In *Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 2011.

[40] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2010.

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

[42] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, 2007.

[43] G. Smith. Principles of secure information flow analysis. In M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007. ISBN 978-0-387-32720-4.

[44] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an efficient purely declarative logic programming language. In *Australian Computer Science Conference*, 1995.

[45] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP)*, 2011.

[46] J. Vaughan and S. Chong. Inference of expressive declassification policies. In *IEEE Security and Privacy*, 2011.

[47] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3), 1996.

[48] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Symposium on Principles of Programming Languages (POPL)*, 2012.

[49] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Symposium on Operating System Principles (SOSP)*, 2009.

[50] S. Zdancewic. A type system for robust declassification. In *19th Mathematical Foundations of Programming Semantics Conference*, 2003.

[51] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Conference on Computer and Communications Security (CCS)*, 2011.

A. Proof of Projection

Theorem 1. *Suppose*

$$\Sigma, e \Downarrow_{pc} \Sigma', V$$

Then for any $q \in PC$ where pc and q are consistent

$$q(\Sigma), q(e) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$$

Proof. We prove a stronger inductive hypothesis, namely that for any $q \in PC$ where $\neg\exists k.(k \in pc \wedge \overline{k} \in q) \vee (\overline{k} \in pc \wedge k \in q)$

1. If $\Sigma, e \Downarrow_{pc} \Sigma', V$ then $q(\Sigma), q(e) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.
2. If $\Sigma, (V_1\ V_2) \Downarrow^{\mathtt{app}}_{pc} \Sigma', V$ then $q(\Sigma), (q(V_1)\ q(V_2)) \Downarrow^{\mathtt{app}}_{pc\setminus q} q(\Sigma'), q(V)$.

The proof is by induction on the derivation of $\Sigma, e \Downarrow_{pc} \Sigma', V$ and the derivation of $\Sigma, (V_1\ V_2) \Downarrow^{\mathtt{app}}_{pc} \Sigma', V$, and by case analysis on the final rule used in that derivation.

- For case [F-LABEL], $e = \text{label } k \text{ in } e'$.
  By the antecedents of this rule:
  $$\frac{k' \text{ fresh} \qquad \Sigma[k' := \lambda x.true], e'[k := k'] \Downarrow_{pc} \Sigma', V}{}$$
  By induction
  $$q(\Sigma[k' := \lambda x.true]), q(e'[k := k']) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$$
  Since $k' \notin \Sigma$, we know that $k' \notin q(\Sigma)$.
  Therefore, $q(\Sigma)[k' := \lambda x.true] = q(\Sigma[k' := \lambda x.true])$.
  By α-renaming, we assume $k \notin q$, $\overline{k} \notin q$, $k' \notin q$, and $\overline{k'} \notin q$.
  Therefore $q(e')[k := k'] = q(e'[k := k'])$.
- For case [F-RESTRICT], $e = \text{restrict}(k, e')$. By the antecedents of this rule:
  $$\frac{\Sigma, e' \Downarrow_{pc} \Sigma_1, V \qquad \Sigma' = \Sigma_1[k := \Sigma_1(k) \wedge_f \langle\!\langle\ pc \cup \{k\}\ ?\ V : \lambda x.true\ \rangle\!\rangle]}{}$$
  By induction, $q(\Sigma), q(e') \Downarrow_{pc\setminus q} q(\Sigma_1), q(V)$.
  $$\begin{aligned} q(\Sigma') &= q(\Sigma_1[k := \Sigma_1(k) \wedge_f \langle\!\langle\ pc \cup \{k\}\ ?\ V : \lambda x.true\ \rangle\!\rangle]) \\ &= q(\Sigma_1)[k := q(\Sigma_1(k)) \wedge_f \\ &\qquad q(\langle\!\langle\ pc \cup \{k\}\ ?\ V : \lambda x.true\ \rangle\!\rangle)] \\ &= q(\Sigma_1)[k := q(\Sigma_1(k)) \wedge_f \\ &\qquad \langle\!\langle\ pc \cup \{k\} \setminus q\ ?\ q(V) : \lambda x.true\ \rangle\!\rangle)] \end{aligned}$$
  by Lemma 1
- For case [F-VAL], $e = V$.
  Since $\Sigma, V \Downarrow_{pc} \Sigma, V$ and $q(\Sigma), q(V) \Downarrow_{pc\setminus q} q(\Sigma), q(V)$, this case holds.
- For case [F-REF], $e = \text{ref } e'$. Then by the antecedents of the [F-REF] rule:
  $$\frac{\Sigma, e' \Downarrow_{pc} \Sigma'', V' \qquad a \notin dom(\Sigma'') \qquad V'' = \langle\!\langle\ pc\ ?\ V' : 0\ \rangle\!\rangle \qquad \Sigma' = \Sigma''[a := V''] \qquad V = a}{}$$
  By induction, $q(\Sigma), q(e') \Downarrow_{pc\setminus q} q(\Sigma''), q(V')$.
  Since $a \notin dom(\Sigma'')$, $a \notin dom(q(\Sigma''))$.
  By Lemma 1, $q(V'') = \langle\!\langle\ pc \setminus q\ ?\ q(V') : q(0)\ \rangle\!\rangle$.
  Since $\Sigma' = \Sigma''[a := V'']$, $q(\Sigma') = q(\Sigma'')[a := q(V'')]$.
  Therefore $q(\Sigma), \text{ref } q(e') \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.
- For case [F-DEREF], $e = !e'$. Then by the antecedents of the [F-DEREF] rule:
  $$\frac{\Sigma, e' \Downarrow_{pc} \Sigma', V' \qquad V = deref(\Sigma', V', pc)}{}$$
  By induction, $q(\Sigma), q(e') \Downarrow_{pc\setminus q} q(\Sigma'), q(V')$.
  By Lemma 2, $q(V) = deref(q(\Sigma'), q(V'), pc \setminus q)$.
  Therefore $q(\Sigma), q(!e') \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.
- For case [F-ASSIGN], $e = (e_a := e_b)$.
  By the antecedents of the [F-ASSIGN] rule:
  $$\frac{\Sigma, e_a \Downarrow_{pc} \Sigma_1, V_1 \qquad \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V \qquad \Sigma' = assign(\Sigma_2, pc, V_1, V)}{}$$

By induction

$$q(\Sigma), q(e_a) \Downarrow_{pc\setminus q} q(\Sigma_1), q(V_1)$$
$$q(\Sigma_1), q(e_b) \Downarrow_{pc\setminus q} q(\Sigma_2), q(V)$$

By Lemma 3, $q(\Sigma') = assign(q(\Sigma_2), pc \setminus q, q(V_1), q(V))$. Therefore $q(\Sigma), q(e_a := e_b) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.

- For case [F-APP], $e = (e_a\ e_b)$. By the antecedents of the [F-APP] rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, V_1$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V_2$$
$$\Sigma_2, (V_1\ V_2) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V$$

By induction

$$q(\Sigma), q(e_a) \Downarrow_{pc\setminus q} q(\Sigma_1), q(V_1)$$
$$q(\Sigma_1), q(e_b) \Downarrow_{pc\setminus q} q(\Sigma_2), q(V_2)$$
$$q(\Sigma_2), (q(V_1)\ q(V_2)) \Downarrow^{\mathsf{app}}_{pc\setminus q} q(\Sigma'), q(V)$$

Therefore $q(\Sigma), q(e_a\ e_b) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.

- For case [F-LEFT], $e = \langle k\ ?\ e_a : e_b \rangle$. By the antecedents of this rule

$$k \in pc$$
$$\Sigma, e_a \Downarrow_{pc} \Sigma', V$$

  - If $k \in q$, then $q(\langle k\ ?\ e_a : e_b \rangle) = q(e_a)$.
    By induction $q(\Sigma), q(e_a) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.

  - Otherwise $k \notin q$ and $\overline{k} \notin q$.
    Therefore $q(\langle k\ ?\ e_a : e_b \rangle) = \langle k\ ?\ q(e_a) : q(e_b) \rangle$.
    Since $k \in pc \setminus q$, it holds by induction that

$$q(\Sigma), \langle k\ ?\ q(e_a) : q(e_b) \rangle \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$$

- Case [F-RIGHT] holds by a similar argument as [F-LEFT].
- For case [F-SPLIT], $e = \langle k\ ?\ e_a : e_b \rangle$. By the antecedents of the [F-SPLIT] rule:

$$\Sigma, e_a \Downarrow_{pc\cup\{k\}} \Sigma_1, V_1$$
$$\Sigma_1, e_b \Downarrow_{pc\cup\{\overline{k}\}} \Sigma', V_2$$
$$V = \langle k\ ?\ V_1 : V_2 \rangle$$

  - Suppose $k \in q$. Then $q(e) = q(e_a)$ and $q(V_1) = q(V)$.
    By induction, $q(\Sigma), q(e_a) \Downarrow_{pc\cup\{k\}\setminus q} q(\Sigma_1), q(V_1)$.
    Lemma 4 implies $q(\Sigma_1) = q(\Sigma')$, so this case holds.
  - If $\overline{k} \in q$, Then $q(e) = q(e_b)$ and $q(V_2) = q(V)$.
    By Lemma 4 we know that $q(\Sigma) = q(\Sigma_1)$.
    By induction, $q(\Sigma_1), q(e_b) \Downarrow_{pc\cup\{k\}\setminus q} q(\Sigma'), q(V_2)$.
  - If $k \notin q$ and $\overline{k} \notin q$, then by induction

$$q(\Sigma), q(e_a) \Downarrow_{pc\cup\{k\}\setminus q} q(\Sigma_1), q(V_1)$$
$$q(\Sigma_1), q(e_b) \Downarrow_{pc\cup\{\overline{k}\}\setminus q} q(\Sigma'), q(V_2)$$

    By Lemma 1, $q(V) = \langle\!\langle\ pc \setminus q\ ?\ q(V_1) : q(V_2)\ \rangle\!\rangle$.
- For case [FA-FUN], $V_1 = \lambda x.e'$. By the antecedent of this rule

$$\Sigma, e'[x := V_2] \Downarrow_{pc} \Sigma', V$$

  We know that $q(\lambda x.e'\ V_2) = q(e'[x := V_2])$.
  By induction $q(\Sigma), q(e'[x := V_2]) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.
- Both cases [FA-LEFT] and [FA-RIGHT] hold by a similar argument as [F-LEFT].
- Case [FA-SPLIT] holds by a similar argument as [F-SPLIT].

□