A Language for Automatically Enforcing Privacy Policies

Abstract

It is becoming increasingly important for applications to not leak sensitive data. With current techniques, the programmer bears the burden of ensuring that the application's behavior adheres to policies about information flow. Unfortunately, privacy policies are difficult to manage because their global nature requires coordinated reasoning and enforcement.

To address this problem, we describe a programming model that makes the system responsible for ensuring adherence to privacy policies. The programming model has two components: core programs describing functionality independent of privacy concerns and declarative, decentralized policies controlling how sensitive values are disclosed depending on the context. Each sensitive value encapsulates multiple views; policies describe which views are allowed based on the output context. The system is responsible for automatically ensuring outputs are consistent with the policies.

We have implemented this programming model in a new functional constraint language named Jeeves. In Jeeves, sensitive values are introduced as symbolic variables and policies correspond to constraints that are resolved at output channels. We have implemented Jeeves as a Scala library using an SMT solver as a model finder. In this paper we describe the dynamic and static semantics of Jeeves and the properties about policy enforcement that the semantics guarantees. We also describe our experience implementing a conference management system and a social network.

1. Introduction

As users share more personal data online, it becomes increasingly important for applications to protect its confidentiality. This places the burden on programmers to ensure compliance with privacy policies when both the application and the policies themselves may evolve rapidly.

Ensuring compliance with privacy policies is difficult because of the need to reason globally about both the flow of information and the interaction of different policies affecting this information. A number of tools have been developed to check code against privacy policies either statically [4, 17] or dynamically [26]. Such checking tools can help avoid data leaks, but the programmer is still responsible for implementing applications that display enough information to satisfy the user's needs without violating any privacy policies. This opens an opportunity for a new class of programming technologies that go beyond checking and actually simplify the process of writing code that preserves confidentiality.

The main contribution of this paper is a new programming model that makes the system responsible for automatically producing program outputs consistent with programmer-provided confidentiality policies. The programming model has two components: a core program representing program functionality that is policy-agnostic and privacy policies controlling how sensitive values are disclosed. This separation of policies from core functionality allows the programmer to explicitly associate policies with sensitive data rather than expressing them implicitly across the code base. The declarative nature of policies allows the system to ensure compliance with privacy policies even when these policies interact in non-trivial ways.

We have implemented this programming model in a new functional constraint language named Jeeves. There are three main concepts in Jeeves: *sensitive values*, *policies*, and *contexts*. *Sensitive values* are introduced as pairs $\langle v_\bot | v_\top \rangle_\ell$, where $v_\bot$ is the low-confidentiality value, $v_\top$ is the high-confidentiality value, and ℓ is the guard variable that can take on the values $\{ \bot, \top \}$ and determines which view of the value should be shown. *Policies* correspond to constraints on the values of guard variables. A policy may refer to the *context*, which characterizes the output channel and contains information identifying the viewer of the data as well as other relevant information. For example, a social network we implemented as a case study allows users to share location data with others in their vicinity; in that case, the context must contain information about the location of the viewer in addition to her identity. The execution model propagates policies and lazily evaluates computations involving sensitive values. At output channels, which correspond to effectful computations such as **print**, the programmer must provide the context parameter so that a concrete value can be produced.

We formally define Jeeves in terms of λ_J, a constraint functional language we have defined to describe the propagation and enforcement of policies in Jeeves. λ_J is different from existing constraint functional language in 1) the restrictions it places on the logical model and 2) its use of default logic to help the programmer reason about nondeterminism. There is a straightforward translation from Jeeves to λ_J: Jeeves guard variables for sensitive values are logic variables, policies are assertions, and all values depending on logic variables are evaluated symbolically. The symbolic evaluation and constraint propagation in λ_J allows Jeeves to automatically enforce policies about information flow.

We have implemented Jeeves as a domain-specific language embedded in Scala [18] using the Z3 SMT solver [15] to resolve constraints. The language of policies is a decidable logic of quantifier-free arithmetic and boolean constraints and also equality constraints over records and record fields. To evaluate the expressiveness of Jeeves, we have used our Scala embedding to implement a conference management system and a social network. On both case studies, Jeeves allowed the separate implementation of functionality and policies. For example, the conference management system separates the implementation of the core functionality for displaying and searching over papers from the policies determining who, and at what point in the conference process, can see information such as paper titles, paper authors, and reviewer information. The result is

that all users, including the general public, can use the same code to search over the same database for information about conference papers without compromising confidentiality.

In summary, we make the following contributions in this paper.

- We present a programming model that allows programmers to separate privacy concerns from core program functionality.
- We formalize the dynamic and static semantics of a novel programming language Jeeves as a set of constraint-based extensions to the λ-calculus. We prove that Jeeves executions satisfy a non-interference property between low and high components of sensitive values.
- We describe the implementation of Jeeves as a Scala library using the Z3 SMT solver.
- We describe our experience using Jeeves to preserve confidentiality in a conference management system and a social network.

2. Delegating Privacy to Jeeves

Jeeves allows the programmer to specify policies explicitly and upon data creation rather than implicitly across the code base. We describe in the context of a simple conference management example how to use the main features of Jeeves: *sensitive values*, *policies*, and *contexts*. We show how to introduce sensitive values, how to write policies with varying levels of complexity and interaction, how these policies are enforced, and how to write concise core programs that are agnostic to the policies.

To concisely convey the main ideas of Jeeves, we present an ML-like concrete syntax for Jeeves, inventing our own syntax for sensitive values, level variables, policies, and contextual enforcement of policies.

2.1 Introduction to Jeeves

We first describe how to introduce sensitive values, use them to compute result values, and display these results in different output context. Below we show how language features are used in creating a sensitive value representing an author's name:

```
let name = <"Anonymous" | "Alice">(context = alice)
let msg = "Author is " + name
print { alice } msg
print { bob } msg
```

The sensitive value name should be seen as "Alice" by users with a high confidentiality level and as "Anonymous" by anybody else. The policy defines the privacy level as a function of the program state and of the *context*, which represents a viewer for the data. The policy context = alice is shorthand for saying that if the output context is not user alice, the confidentiality level is low. Thus only user alice can see her name appearing as the author in the string msg. When the program prints output to user bob, the system will display the string "Author is Anonymous".

The Jeeves programmer defines sensitive values by introducing a tuple $\langle v_\bot | v_\top \rangle_\ell$ where $v_\bot$ is the low-confidentiality, $v_\top$ is the high confidentiality value, and ℓ corresponds to a set of policies on the sensitive value. Each sensitive value defines a low-confidentiality and high-confidentiality view for a value. An expression containing n sensitive values will evaluate to a result value containing up to 2^n views. We can encode more than two privacy level as well, but for the sake of simplicity we present two in this paper.

Which view is displayed for each sensitive value depends on the value of its *level variable*. In the above example, the level variable is introduced implicitly together with the policy. We show below what the right-hand side of the name would look like with explicit **level** and **policy** expressions:

```
level a in
policy a: ! (context = alice) then ⊥ in
<"Anonymous" | "Alice">(a)
```

In a sensitive value $\langle v_\bot | v_\top \rangle_\ell$ variable ℓ is a level variable where $\ell = \bot$ corresponds to low-confidentiality and $\ell = \top$ corresponds to high-confidentiality. Level variables provide the means of abstraction to specify policies incrementally and independently of the sensitive value declaration. Level variables can be constrained directly (by explicitly passing around a level variable) or indirectly (by constraining another level variable when there is a dependency) by subsequent evaluation.

Policies, which are introduced through **policy** expressions, introduce declarative rules describing when to set a level variable to $\top$ or $\bot$. These rules may mention variables in scope and also the depending the **context** parameter, an implicit parameter supplied at the output channel that represents information about the viewer. The **context** construct relieves the programmer of the burden of structuring the code to propagate values from the output context. By default, level variables take value $\top$.

Statements that release information such as **print** require a context parameter to produce outputs according to the policies. The Jeeves runtime system propagates policies associated with sensitive values in order to display the appropriate results at output channels. Throughout this example we use **print** as the canonical way of displaying output; other output channels include sending e-mail and writing to file. Because the policies are declarative, Jeeves automatically handles dependencies between policies. Jeeves provides the guarantee that the system will not output a value computed from the high-confidentiality view of a sensitive value (such as name) if policies prohibit it from being shown.

2.2 Declarative and Decentralized Policies

We now describe how to write policies in Jeeves. To move along our conference management example, we define a paper record:

```
type paper { title : string
           ; author: user
           ; reviews: review list
           ; accepted: bool option; ... }
```

Let us first consider the title field of paper records. There is a simple policy: the title of a paper should be visible to the authors of the paper, to reviewers and PC members, and also to the general public after the public announcement if the paper has been accepted. We write a function that, given variable p : paper, takes a title and adds the policy that the viewer sees title only if permitted:

```
let addTitlePolicy(p: paper, title : string) =
  level a in
  policy a: ! (context.viewer = p.author
    || context.viewer.role = Reviewer
    || context.viewer.role = PC
    || (context.stage = Public && isAccepted p)) then ⊥ in
  <"" | title >(a)
```

In the function we introduce a new sensitive value for the title. We declare a policy that sets the level to $\bot$ unless the viewer has access to see the paper title. Once this policy is attached to the paper title, any output produced that includes the title string, will show the empty string if the output viewer does not have access. We apply this function to the title before storing the paper record to ensure that policies are attached.

The condition for the policy in function addTitlePolicy mentions the viewer and stage fields of the **context** variable. As previously mentioned, Jeeves policies can use the **context** keyword to access values provided at the output channel. For this example we define the following context and confstage types:

```
type confView { viewer: user; stage: confstage }
type confstage = Submission | Review | Decision | Public
```

The programmer must define this context type in order to write policies that reference it. A sensitive value produced by the addTitlePolicy function is evaluated lazily in the context of the output channel, where a value of type confView is provided.

2.2.1 Policy Interactions

One difficulty of reasoning about security policies is in making sure one policy does not leak information from another policy. For instance, the policy for paper titles could leak information about the accept tag: for instance, if the policy were weakened to drop the **context**.stage = Public, then anyone can see the title as soon as the accept tag is set to **true**. Being able to attach policies to sensitive values helps prevent the programmer from inadvertently leaking information in this way. We show how to set up Jeeves policies to prevent this information leak.

In the addTitlePolicy function, we define the predicate isAccepted p depends on the accepted field of the paper p, which is going to be **some** accepted if a decision has been made (and the decision is known) or **none** otherwise. We can associate this field with its own policy to prevent the situation where a change in the policy for paper titles can accidentally leak information about the accepted field. For instance, we could write the following function for adding a policy:

```
let addAcceptedPolicy(accepted: bool) =
  level a in
  policy a: ! (context.viewer.role = Reviewer
    || context.viewer.role = PC
    || context.stage = Public) then ⊥ in
  <none | some accepted>(a)
```

This policy allows reviewers and program committee members to always see whether a paper has been accepted, but for others to see this field only during the Public stage. If the accepted tag has this policy associated, then the title field cannot leak information about the accepted tag.

2.2.2 Gradually Strengthened Policies

Level variables allow us to strengthen policies gradually. For example, we can guard reviews field with a level reviewsLevel and allow subsequent evaluations to strengthen the policy on reviewsLevel to further restrict access. Declaring a conflict should exclude the viewer from seeing the reviews. We can implement that in Jeeves as follows:

```
let addConflict (reviewsLevel: level, conflict: user) =
  policy reviewsLevel: context.viewer = conflict then ⊥
```

This function strengthens the policy on the reviewsLevel level variable to exclude the new conflicted user.

2.2.3 Policy Dependencies

The Jeeves system can also automatically resolve dependencies. Consider the following function that associates a policy with the authors of a paper:

```
let addAuthorPolicy (author: user): user =
  level n in
  policy n: context.user = author then ⊤ in
  policy n: ! (context.stage = Review && context.user = x)
    then ⊥ in
  <anonAuthor | author>(n)
```

Now consider functionality that sends messages to authors of papers:

```
let sendMsg (author: user) =
  let msg = "Dear " + author.name + ... in
  sendmail { user = author; stage = Review } msg
```

The policy for level variable n depends on **context**.user, which in this case is itself a sensitive value. This circular dependency makes the solution underconstrained: either sending mail to the empty user or sending mail to the author is correct under the policy. The desired behavior is the latter, as it is permitted by the policies and ensures that user a can communicate with user b without knowing private information about user b.

Jeeves sets level variables to $\top$ by default to define program behavior when these dependencies arise. Unless there are explicit policies requiring a level value to be $\bot$, its value is $\top$. The default value of $\top$ for level variables causes the system to send the e-mail with the actual name to the actual author. Such circular dependencies may arise whenever the context is made up of values that may be sensitive.

Even with circular dependencies, the system guarantees that values will only be shown to users if the policies allow it. A level variable ℓ can only become underconstrained if there are no policies requiring ℓ to be $\bot$. In this case, it is always correct with respect to the policies to allow ℓ to be $\top$.

2.3 Core Functionality

Being able to separately specify the policies allows the core functionality to be quite simple. In fact, the program could simply allow all viewers to operate over the list of papers directly and rely on the Jeeves system to display the appropriately anonymized information:

```
papers: paper list
```

As we describe in the conference management case study in Section 6, the functions implementing the core program are quite concise because they do not contain policy code.

3. The λ_J Language and Semantics

To describe Jeeves execution we present λ_J, a simple constraint functional language based on the λ-calculus. There is a straightforward translation from Jeeves to λ_J. λ_J differs from existing constraint functional languages [8, 9, 13, 24] in two key ways: 1) λ_J restricts its constraint language to quantifier-free constraints involving boolean and arithmetic expressions over primitive values and records and 2) λ_J supports *default values* for logic variables to facilitate reasoning about nondeterminism. λ_J's restrictions on the constraint language allows execution to rely on an off-the-shelf SMT solver.

In this section, we introduce the λ_J language, the dynamic semantics, the static semantics, and the translation from Jeeves. The λ_J language extends the λ-calculus with the **defer**, **assert**, and construct for introducing, constraining, and producing concrete values from logic variables. The dynamic semantics describe the lazy evaluation of expressions involving logic variables and the interaction with the constraint environment. The static semantics describe how the system guarantees evaluation progress and enforces programmer restrictions on concrete function arguments. The translation from Jeeves to λ_J illustrates how Jeeves uses the lazy evaluation and constraint propagation, combined with Jeeves restrictions on how logic variables are used, to provide privacy guarantees.

3.1 The λ_J Language

We show the abstract syntax of λ_J in Figure 1: values (v) and expressions (e). Expressions include the standard λ expressions extended with the **defer** construct for introducing logic variables, the **assert** construct for introducing constraints, and the **concretize** construct for producing concrete values consistent with the constraints. A novel feature of λ_J is that logic variables are also associated with a *default value* that serves as a default assumption; this provides some determinism when logic variables are underconstrained.

λ_J evaluation produces either values, which are either concrete values c or symbolic value σ. Concrete values include records, which have value fields. Symbolic values are irreducible arithmetic,

$$
\begin{array}{lcll}
c & ::= & n \mid b \mid \lambda x : \tau . e \mid \textbf{record}\ x \vec{:} v & \\
 & \mid & \textbf{error} \mid () & \text{concrete primitives} \\
\sigma & ::= & x \mid \textbf{context}\ \tau & \text{symbolic values} \\
 & \mid & c_1\ (\text{op})\ \sigma_2 \mid \sigma_1\ (\text{op})\ c_2 & \\
 & \mid & \textbf{if}\ \sigma\ \textbf{then}\ v_t\ \textbf{else}\ v_f & \\
v & ::= & c \mid \sigma & \text{values} \\
e & ::= & v \mid e_1\ (\text{op})\ e_2 & \text{expressions} \\
 & \mid & \textbf{if}\ e_1\ \textbf{then}\ e_t\ \textbf{else}\ e_f \mid e_1\ e_2 & \\
 & \mid & \textbf{defer}\ x : \tau\ \{e\}\ \textbf{default}\ v_d & \\
 & \mid & \textbf{assert}\ e & \\
 & \mid & \textbf{concretize}\ e\ \textbf{with}\ v_c &
\end{array}
$$

Figure 1: λ_J abstract syntax.

boolean expressions, conditional expressions involving primitive values, or the **error** value. Evaluation treats symbolic values as fully evaluated expressions and handles them by creating symbolic expressions.

λ_J includes the **context** construct to allow constraints to refer to the value supplied at the output context. The **context** variable behaves like an implicit parameter provided in the **concretize** expression. This implicit parameter. In the semantics we model the behavior of the **context** variable as a symbolic value that is constrained during evaluation of **concretize**.

We describe in more detail in Section 3.4 the translation from Jeeves to λ_J, which places restrictions on how logic variables are introduced and constrained. Sensitive values and level variables in Jeeves correspond to logic variables, level policies correspond to assertions, and contextual enforcement corresponds to producing concrete values consistent with the logical environment. Default values provide determinism in handling policy dependencies.

3.2 Dynamic Semantics

In Figure 2 we show the small-step dynamic λ_J semantics. λ_J execution involves keeping track of the set of constraints assumed to be true (hard constraints) and the set of constraints we use for guidance if consistent with our hard constraints (default assumptions). To correctly evaluate conditionals with symbolic conditions, we also need to keep track of the (possibly symbolic) guard condition. Evaluation happens in the context of a logical environment in three parts:

- an environment $\Sigma = \varnothing \mid \{\sigma\} \mid \Sigma \cup \Sigma'$ storing the current set of constraints,
- an environment $\Delta = \varnothing \mid \{\sigma\} \mid \Delta \cup \Delta'$ storing the set of constraints on default values for logic variables, and
- a path condition $\mathcal{G}$.

Evaluation rules take the form

$$\mathcal{G}, \Sigma, \Delta \vdash e \rightarrow \langle \Sigma', \Delta', e' \rangle.$$

Evaluation produces a tuple $\langle \Sigma', \Delta', e' \rangle$ of resulting constraint and default environments and an expression e'.

The semantics require all logic variables must have unique names; we can ensure this by generating new names for each logic variable and α-renaming within the constraint. The λ_J evaluation rules extend λ-calculus evaluation with constraint propagation and symbolic evaluation of expressions involving logic variables, including evaluation of symbolic conditionals.

3.2.1 Evaluation with Logic Variables

λ_J has the expected semantics for function applications and arithmetic and boolean operations. The E-APP1, E-APP2, and E-APPLAMBDA rules describe a call-by-value semantics. The E-OP1 and E-OP2 rules for operations show that the arguments are evaluated to irreducible expressions and then, if both arguments become concrete, the E-OP rule can be applied to produce a concrete result.

Conditionals with symbolic conditions are evaluated by creating an **if-then-else** symbolic value created from evaluating both branches. The E-COND, E-CONDTRUE, and E-CONDFALSE rules describe the evaluation of different parts of a conditional expression. The E-CONDSYMT and E-CONDSYMF rules describe the evaluation of each conditional branch if the condition is a symbolic expression.

Evaluation of a conditional with a recursive function application in a branch will not terminate if the condition is symbolic. To help the programmer avoid applications of recursive functions guarded by symbolic conditions, we allow the programmer to annotate function arguments required to be concrete. The programmer can rely on the type system to enforce these requirements in order to make sure recursive calls are not made under symbolic conditions 3.3.

3.2.2 Introduction and Elimination of Logic Variables

In λ_J, logic variables are introduced through the **defer** keyword.

$$\textbf{let}\ \mathsf{x}\colon\ \textbf{int}\ =\ \textbf{defer}\ \textbf{int}\ \mathsf{x'}\ \{\ \mathsf{x'} > 0\ \}\ \textbf{with}\ \textbf{default}\ 42$$

As we show in the E-DEFER evaluation rule, the right-hand side of the assignment would evaluate to an α-renamed version of the logic variable $\mathsf{x'}$. Evaluation adds the constraint $\mathcal{G} \Rightarrow \mathsf{x'} > 0$ to the constraint environment and the constraint $\mathcal{G} \Rightarrow \mathsf{x'} = 42$ to the default constraint environment. The constraint $\mathcal{G} \Rightarrow \mathsf{x'} = 0$ is a hard constraint that must hold for all derived outputs, while $\mathcal{G} \Rightarrow \mathsf{x'} = 42$ is a constraint that is only used if it is consistent with the resulting logical environment. The E-DEFERCONSTRAINT rule describes the evaluation of the constraint expression to a symbolic value.

The program introduces constraints on logic variables through **assert** expressions. The E-ASSERT rule describes how the constraint is added to the constraint environment, taking into account the path condition $\mathcal{G}$. For instance, consider the following code:

$$\textbf{if}\ \ (\mathsf{x} > 0)\ \textbf{then}\ \textbf{assert}\ \ (\mathsf{x} = 42)\ \textbf{else}\ \ \textbf{assert}\ \ (\mathsf{x} = -42)$$

Evaluation adds the constraints $\mathsf{x} > 0 \Rightarrow \mathsf{x} == 42$ and $\neg(\mathsf{x} > 0) \Rightarrow \mathsf{x} == -42$ to the constraint environment.

Symbolic expressions can be made concrete through the **concretize** construct. Evaluation of **concretize** expressions either produces a concrete value or an error. A **concretize** expression includes the expression to concretize and a default expression:

$$\textbf{let}\ \ \mathsf{result}:\ \ \textbf{int}\ =\ \textbf{concretize}\ \mathsf{x}\ \textbf{with}\ 42$$

As we describe in the E-CONCRETIZESAT rule, concretization would add the constraint $\textbf{context} = 42$ to the constraint environment and find an assignment for x consistent with the constraint and default environments. The MODEL function takes the constraint and default environments, computes a satisfiable assignment to free variables, and produces a substitution $\mathcal{M} : v \rightarrow c$ that is used to produce a concrete value. When concretization produces a value, evaluation adds constraint guarded by the path condition $\mathcal{G}$ and the context value to the constraint environment to ensure that subsequent concretizations are consistent.

The CONCRETIZE-UNSAT describes what happens if there is no satisfiable expression consistent with the constraint environment. In this case, evaluation of the **concretize** expression produces the **error** value. The guard constraint is added to the environment to reflect the fact that this is an infeasible computation path.

3.2.3 Interaction with the Constraint Environment

Valid constraint expressions correspond to λ_J expressions that do not contain λ-expressions. This constraint language corresponds to constraints that can be solved by off-the-shelf SMT solvers. The MODEL procedure we show in the E-CONCRETIZE rules is

$$\boxed{\mathcal{G}, \Sigma, \Delta \vdash e \to \langle \Sigma', \Delta', e' \rangle}$$

$$\frac{\mathcal{G}, \Sigma, \Delta \vdash e_1 \to \langle \Sigma', \Delta', e_1' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash e_1\ e_2 \to \langle \Sigma', \Delta', e_1'\ e_2 \rangle}\ \text{E-App1} \qquad \frac{\mathcal{G}, \Sigma, \Delta \vdash e_2 \to \langle \Sigma', \Delta', e_2' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash v\ e_2 \to \langle \Sigma', \Delta', v\ e_2' \rangle}\ \text{E-App2}$$

$$\frac{}{\mathcal{G}, \Sigma, \Delta \vdash \lambda x.e\ v \to \langle \Sigma', \Delta', e[x \mapsto v] \rangle}\ \text{E-AppLambda}$$

$$\frac{\mathcal{G}, \Sigma, \Delta \vdash e_1 \to \langle \Sigma', \Delta', e_1' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash e_1\ (\text{op})\ e_2 \to \langle \Sigma', \Delta', e_1'\ (\text{op})\ e_2 \rangle}\ \text{E-Op1} \qquad \frac{\mathcal{G}, \Sigma, \Delta \vdash e_2 \to \langle \Sigma', \Delta', e_2' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash v\ (\text{op})\ e_2 \to \langle \Sigma', \Delta', v\ (\text{op})\ e_2' \rangle}\ \text{E-Op2}$$

$$\frac{c' = c_1\ (\text{op})\ c_2}{\mathcal{G}, \Sigma, \Delta \vdash c_1\ (\text{op})\ c_2 \to \langle \Sigma, \Delta, c' \rangle}\ \text{E-Op}$$

$$\frac{\mathcal{G}, \Sigma, \Delta \vdash e_c \to \langle \Sigma', \Delta', e_c' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{if}\ e_c\ \textbf{then}\ e_t\ \textbf{else}\ e_f \to \langle \Sigma', \Delta', \textbf{if}\ e_c'\ \textbf{then}\ e_t\ \textbf{else}\ e_f \rangle}\ \text{E-Cond}$$

$$\frac{\mathcal{G}, \Sigma, \Delta \vdash e_t \to \langle \Sigma', \Delta', e_t' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{if}\ \textbf{true}\ \textbf{then}\ e_t\ \textbf{else}\ e_f \to \langle \Sigma', \Delta', e_t' \rangle}\ \text{E-CondTrue} \qquad \frac{\mathcal{G}, \Sigma, \Delta \vdash e_f \to \langle \Sigma', \Delta', e_f' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{if}\ \textbf{false}\ \textbf{then}\ e_t\ \textbf{else}\ e_f \to \langle \Sigma', \Delta', e_f' \rangle}\ \text{E-CondFalse}$$

$$\frac{\sigma \wedge \mathcal{G}, \Sigma, \Delta \vdash e_t \to \langle \Sigma', \Delta', e_t' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{if}\ \sigma\ \textbf{then}\ e_t\ \textbf{else}\ e_f \to \langle \Sigma', \Delta', \textbf{if}\ \sigma\ \textbf{then}\ e_t'\ \textbf{else}\ e_f \rangle}\ \text{E-CondSymT}$$

$$\frac{\neg\sigma \wedge \mathcal{G}, \Sigma, \Delta \vdash e_f \to \langle \Sigma', \Delta', e_f' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{if}\ \sigma\ \textbf{then}\ v_t\ \textbf{else}\ e_f \to \langle \Sigma', \Delta', \textbf{if}\ \sigma\ \textbf{then}\ v_t\ \textbf{else}\ e_f' \rangle}\ \text{E-CondSymF}$$

$$\frac{\mathcal{G}, \Sigma, \Delta \vdash e \to \langle \Sigma', \Delta', e' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{defer}\ x : \tau\ \{e\}\ \textbf{default}\ v_d \to \langle \Sigma', \Delta', \textbf{defer}\ x : \tau\ \{e'\}\ \textbf{default}\ v_d \rangle}\ \text{E-DeferConstraint}$$

$$\frac{}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{defer}\ x : \tau\ \{v_c\}\ \textbf{default}\ v_d \to \langle \Sigma' \cup \{\mathcal{G} \Rightarrow v_c\}, \Delta' \cup \{x = v_d\}, x \rangle}\ \text{E-Defer}$$

$$\frac{\mathcal{G}, \Sigma, \Delta \vdash e \to \langle \Sigma', \Delta', e' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{assert}\ e \to \langle \Sigma', \Delta', \textbf{assert}\ e' \rangle}\ \text{E-AssertConstraint} \qquad \frac{}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{assert}\ v \to \langle \Sigma \cup \{\mathcal{G} \Rightarrow v\}, \Delta, () \rangle}\ \text{E-Assert}$$

$$\frac{\mathcal{G}, \Sigma, \Delta \vdash e \to \langle \Sigma', \Delta', e' \rangle}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{concretize}\ e\ \textbf{with}\ v_c \to \langle \Sigma', \Delta', \textbf{concretize}\ e'\ \textbf{with}\ v_c \rangle}\ \text{E-ConcretizeExp}$$

$$\frac{\textsc{Model}(\Delta, \Sigma \cup \{\mathcal{G} \wedge \textbf{context} = v_c\}) = \mathcal{M} \quad c = \mathcal{M}[\![v_v]\!]}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{concretize}\ v_v\ \textbf{with}\ v_c \to \langle \Sigma, \Delta, c \rangle}\ \text{E-ConcretizeSat}$$

$$\frac{\textsc{Model}(\Delta, \Sigma \cup \{\mathcal{G} \wedge \textbf{context} = v_c\}) = \textsc{Unsat}}{\mathcal{G}, \Sigma, \Delta \vdash \textbf{concretize}\ v_v\ \textbf{with}\ v_c \to \langle \Sigma \cup \{(\mathcal{G} \wedge (\textbf{context} = v_c)) \Rightarrow \textbf{false}\}, \Delta, \textbf{error} \rangle}\ \text{E-ConcretizeUnsat}$$

Figure 2: Dynamic semantics for Jeeves.

the model finding procedure for default logic [1]. The default environment Δ and constraint environment Σ specify a supernormal default theory (Δ, Σ) where each default judgement $\sigma \in \Delta$ has the form:

$$\frac{\textbf{true} : \sigma}{\sigma}$$

MODEL procedure produces either a model $\mathcal{M}$ for the theory if it is consistent, or UNSAT. We use a fixed-point algorithm for MODEL that uses classical SMT model-generating decision procedures and iteratively saturates the logical context with default judgements in a non-deterministic order.

3.3 λ_J Static Semantics

We have defined typing rules for λ_J that perform the following three functions: 1) evaluation of well-typed expressions does not get stuck, 2) enforcing that applications of functions requiring concrete arguments are given concrete values, and 3) ensuring that contexts of the appropriate type are provided in **concretize** expressions. This guarantees that a program either produces a value or halts in an error. There are two sources of errors: programmer-introduced **error** values and inconsistent logical states when evaluating **concretize**. The type system does not describe the consistency of the logical state. λ_J types only matter statically and make no difference to the dynamic semantics.

δ	::=	**concrete** | **sym**	determinism tag
β	::=	**int** | **bool** | **string** | **unit** | **exn**	base type
	|	**record** $x\ \vec{:}\ \tau$	
τ	::=	b | $\tau_1 \to \tau_2$ | β_δ	type

Figure 3: λ_J types.

We show the λ_J types in Figure 3. The λ_J type system extends the standard λ-calculus types: base types and the function type $\tau_1 \to \tau_2$. λ_J also has δ annotations to help the programmer enforce and program with static restrictions on nondeterminism. Function

arguments that are a base type are required to be annotated with a type β_δ where δ = { **concrete**, **sym** } describing whether the argument is expected to be symbolic. Expressions of function type are not permitted to be symbolic.

3.3.1 Restricting Symbolic Expressions

We show the rules in Figure 4. The typing judgment is

$$\Gamma \vdash e : \langle \tau, \delta \rangle$$

and says that in the type environment Γ, the expression e has *surface type* τ with determinism tag δ. Γ is defined $\Gamma ::= \cdot \mid x : \tau \mid \Gamma, \Gamma'$ We define the operations $\vee$ and $\wedge$ on δ-tags with $\textbf{sym} \vee \delta = \textbf{sym}$, $\textbf{sym} \wedge \delta = \textbf{sym}$, and $\textbf{concrete} \wedge \textbf{concrete} = \textbf{concrete}$. We have simplified the rules for the case when all logic variables have integer type; to support logic variables of different types we would need to carry an additional environment for those types. The static semantics propagate δ tags to infer whether an expression must be concrete or may be symbolic.

Information about whether an expression may be symbolic comes from the leaf expressions: concrete constants, logic variables, and concretizations of expressions. The T-DEFER rule shows that delegated variables must be symbolic. The T-INT, T-BOOL, T-UNIT, and T-EXN rules shows that primitive values and program constants are concrete. The T-CONCRETIZE rule shows that a concretized value is concrete.

The propagation of the δ tag is different for functions and function application. The T-LAMBDA rule uses the δ tag to propagate whether the *result* of a function is concrete. We can use the δ tag for this purpose because the δ tag is checked only for conditions, which must have type **bool** and so cannot be a function.

One reason the static semantic rules propagate the δ tag is to restrict symbolic values to non-functions. This restriction is enforced with the T-DEFER and T-CONDS rules. The T-DEFER rule restricts delegated expressions to simple base types (*e.g.* **int**, **bool**). The T-CONDS rule restricts branches of conditionals to have non-function types if the condition may be nondeterministic.

The other reason the rules propagate the δ tag is to enforce that functions with type $\tau_{\textbf{concrete}} \rightarrow \tau'$ are given concrete arguments. We show in the T-APPC rule how concrete arguments are checked at function applications.

3.3.2 Contexts

The λ_J static semantics also ensure that contexts of the appropriate type are provided in **concretize** expressions. In the T-CONCRETIZE rule, we also refer to a context typing judgment to enforce that the context type supplied is the context type expected. We have also defined a set of typing judgments (not shown) of the form $\Gamma \vdash_{\mathcal{C}} x : \tau_c$, where τ_c is the *context* type of an expression. The rules propagate the context type, enforce that matching contexts are provided when there is an expressions involving multiple subexpressions, and enforce that the correct context type is supplied at concretization.

To allow flexibility in the context type allowed, we define a lattice on context types to define when two types requiring different context types may be combined. The bottom of the lattice is $\bot$ and for all types τ, we have the relationship $\bot <:_c \tau$. An expression for which the context type does not matter (context type $\bot$) can be used whenever a context of type τ is supplied. We also have the more useful subtyping relationship on record types:

$$\textbf{record}\ \vec{m} <:_c \textbf{record}\ \vec{n}, \forall n_i.(\exists m_i | m_i = n_i).$$

A record with fields $\vec{m}$ can be used as a context whenever a record with field $\vec{n}$ expected as long as the labelled fields of m_i are a superset of the labelled fields of $\vec{n}$.

$Level$	::=	$\bot \mid \top$	levels
Exp	::=	$\upsilon \mid Exp_1$ (op) Exp_2	expressions
		if Exp_1 **then** Exp_t **else** Exp_f	
		$Exp_1\ Exp_2$	
		$\langle Exp_1 \mid Exp_2 \rangle(\ell)$	
		level ℓ **in** Exp	
		policy ℓ: Exp_p **then** $Level$ **in** Exp	
$Stmt$	::=	**let** $x : \tau = Exp$	
		print $\{Exp_c\}$ Exp	

Figure 5: Jeeves syntax.

3.4 Translation from Jeeves

We now describe the translation from Jeeves to λ_J. As a refresher, we show the Jeeves syntax in Figure 5. Sensitive values and level variables in Jeeves correspond to λ_J logic variables, level policies correspond to λ_J assertions, and contextual enforcement corresponds to producing concrete values consistent with the logical environment. Default values provide determinism in handling policy dependencies.

We show the translation of levels and sensitive values from Jeeves to λ_J in Figure 6. To convey Jeeves expression Exp translates to λ_J expression e, we write $Exp \hookrightarrow e$. The translation has the following properties:

- Level variables are the only logic variables introduced by a Jeeves programs.
- Sensitive values are symbolic values defined in terms of level variables, and thus all expressions containing sensitive values produce symbolic results.
- Only level policies in Jeeves programs introduce assertions.
- The **concretize** construct can only appear at the outermost level of an expression and is associated with an effectful computation.

3.4.1 Sensitive Values

A Jeeves sensitive value <v1 | v2>(a) is translated to a symbolic value equal to either v1 or v2 depending on the value of level variable a. Because this sensitive values is symbolic, all expressions computed from the sensitive value are subject to policies affecting the value of level variable a.

3.4.2 Level variables

Jeeves level variables are translated to λ_J expression binding a new logic variable of **level** type equal to either $\bot$ or $\top$. The default value of level variables is $\top$, meaning that the constraint solving oracle first resolves the constraint environment with the assumption that each level is $\top$ and only adjusts this belief if the level variable must be equal to $\bot$. This provides the programmer with some guarantees about program behavior when level variables are underconstrained. The default is $\top$ to require the programmer to be explicit about what restrictions on what users may not see. This way, underconstraint should only arise when there are circular dependencies.

Besides being useful in handling circular dependencies, having the default value of level variables as $\top$ prevents the programmer from leaking a value as a result of an underspecified value. If a level variable is underconstrained, policies on a subsequent variable can affect the value it can take:

```
1 let x = level a in <0 | 1>(a)
2 let y = level b in
3   policy b: true then ⊤ in
4   policy b: x = 1 then ⊥ in
5   <0 | 1>(b)
```

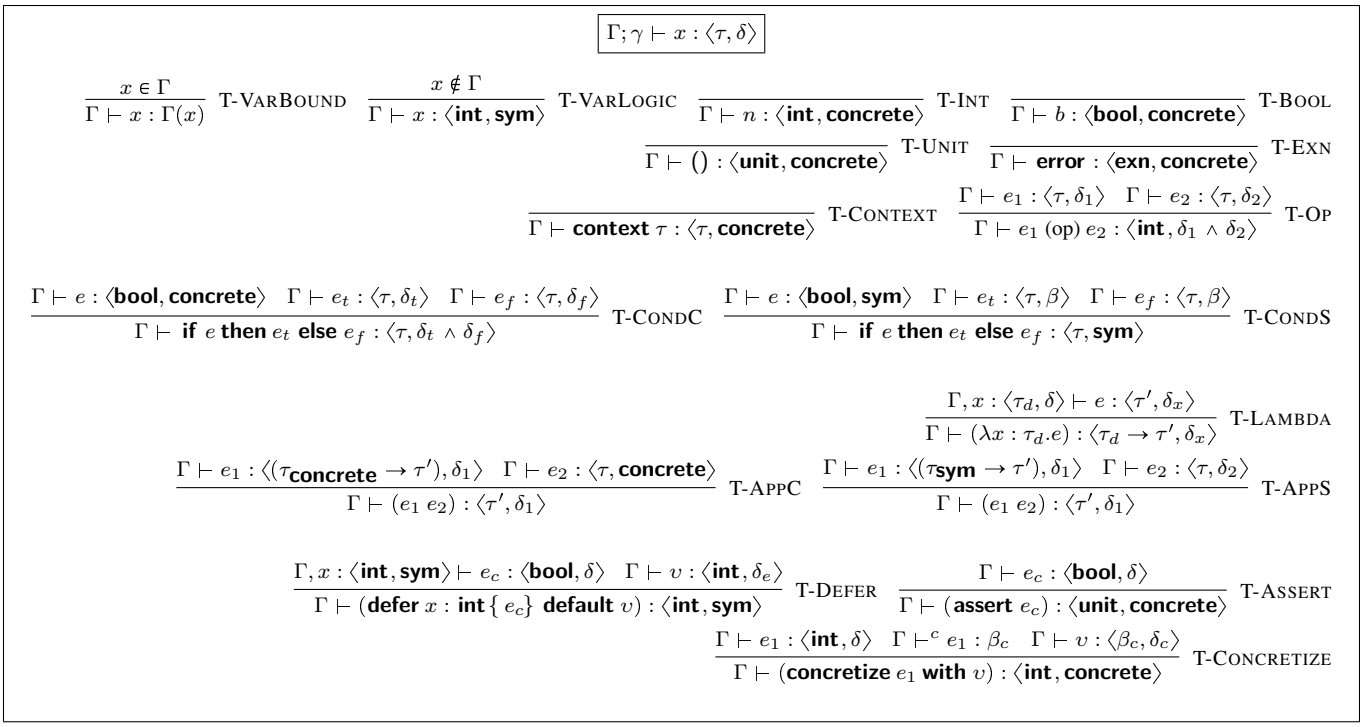
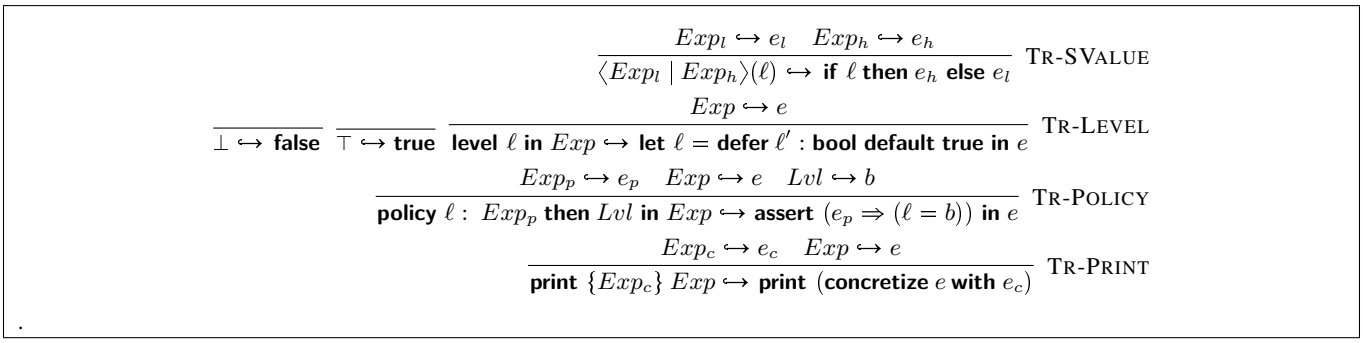


Figure 4: Static semantics for Jeeves describing simple type-checking and enforcing restrictions on scope of nondeterminism and recursion.



.

Figure 6: Translation from Jeeves to λ_J

If the value of x were fixed, this would yield a contradiction, but instead these policies indirectly fix the value of x and a:

true

$\therefore$ (1) b $= \top$ (line 3) $\quad\quad$ $\therefore$ (2) x $\neq 1$ (line 4)

$\therefore$ (3) x $= 0$ (line 1) $\quad\quad$ $\therefore$ (4) a $= \bot$ (line 1)

Making underconstrained level variables $\top$ by default forces programmers to explicitly introduce policies setting level variables to $\bot$. This way, errors from underspecification will only cause level variables to be set to $\bot$ instead of $\top$ rather than the other way around.

3.4.3 Specifying Declarative Constraint Policies

As we show in Table 6, level policies are translated to λ_J **assert** expressions. Level policies can be introduced on any logic variables in scope and are added to the environment based on possible path assumptions made up to that point. The policy that a Jeeves expression Exp enforces consists of the constraint environment produced when evaluating Exp as a λ_J expression. More specifically, we are talking about Σ', Δ' where $Exp \hookrightarrow e$ and $\vdash e \to^* \langle \Sigma', \Delta', v \rangle$. This policy contains constraints constraining whether level variables can be $\bot$ or $\top$.

3.4.4 Contextual Enforcement at Output Channels

Effectful computations such as **print** in Jeeves are statements that require contexts corresponding to the viewer to whom the result is displayed. As we show in the Tr-Print rule, **concretize** is inserted in the translation. Because sensitive values can only produce concrete values consistent with the policies, this ensures enforcement of policies at output channels. The only **concretize** expressions coming from Jeeves programs are at the outermost level of resulting λ_J expressions and are only associated with outputs.

4. Properties

We describe more formally the guarantees that Jeeves provides. Jeeves has the following two main properties:

- **Enforcement Correctness**. If we have a λ_J logic variable x introduced through a **defer** expression, any expression containing

e will have x as a sub-expression and can only be made concrete in the context of all constraints that may affect the value of x.

- **Privacy Non-interference**. We can show that if a Jeeves expression Exp contains a sensitive value $\langle Exp_l \mid Exp_h \rangle(\ell)$, if at an output channel ℓ is resolved to have level $\bot$, then we should output the same value if we had executed with Exp_l.

Like static information flow systems like Jif [17] or Pottier *et al.*'s information flow system for ML [19], Jeeves guarantees that high-confidentiality values do not flow to low-confidentiality output contexts. Unlike these systems, Jeeves policies are defined dynamically with respect to the dynamic constraint environment.

In this section, we show Enforcement Correctness property by proving progress and preservation and showing that the symbolic evaluation and constraint propagation happens correctly. We show the Privacy Non-interference property by showing properties about the determinism of evaluating expressions without **concretize** sub-expressions and arguing about the equivalence of the lazy symbolic evaluation to eager evaluation under the model found when evaluating **concretize**.

4.1 Progress and Preservation

To show the correctness of enforcement, we first need to show show the correctness of evaluation. We can prove progress and preservation properties for λ_J: evaluation of an expression e always results in a value v and preserves the type of e, including the internal nondeterminism tag δ.

There are two interesting parts to the proof: showing that all function applications can be reduced and the second involves showing that all **defer** and **assert** expressions can be evaluated to produce appropriate constraint expressions. We can first show that the λ_J type system guarantees that all functions are concrete.

Lemma 1 (Concrete Functions)**.** *If v is a value of type $\tau_1 \to \tau_2$, then $v = \lambda x : \tau_1.e$, where e has type τ_2.*

Theorem 4.1 (Progress)**.** *Suppose e is a closed, well-typed expression. Then e is either a value v or there is some e' such that $\vdash e \to \langle \Sigma', \Delta', e' \rangle$.*

Proof. The proof mostly involves induction on the typing derivations. One interesting case is ensuring that MODEL will either return a valid model $\mathcal{M}$ or UNSAT for the E-CONCRETIZESAT and E-CONCRETIZEUNSAT rules. Since the λ_J type system rules out symbolic functions, only well-formed constraints can be added. The other interesting case is function applications $e = e_1\ e_2$, where e_1 and e_2 are well-typed with types $\tau_1 \to \tau_2$ and τ_1. We can rule out the cases when e_1 and e_2 are not values by applying the induction hypothesis. For the case when e_1 and e_2 are both values, we can apply the Concrete Functions Lemma to deduce that e_1 must have the form $\lambda x : \tau_1 : e$, where $e : \tau_1$. In this case, we can apply the E-APPABS rule. □

We can also prove a preservation theorem that evaluation does not change the type of a λ_J expression.

Theorem 4.2 (Preservation)**.** *If $\Gamma \vdash e : \langle \tau, \delta \rangle$ and $e \to e'$, then $\Gamma \vdash e' : \langle \tau, \delta \rangle$.*

Proof. We can show the preservation of both τ and δ by induction on the typing derivation. The δ value for all evaluation rules except for the E-CONCRETIZE rules is the same for both sides. □

4.2 Level Variables Enforce Confidentiality

Theorem 4.3 (View Non-interference)**.** *Consider a sensitive value $V = \langle E_l | E_h \rangle_\ell$ in Jeeves expression E. Assume:*

$$E \hookrightarrow e \qquad \vdash e \to^* \langle \Sigma, \Delta, \sigma \rangle$$
$$E[V \to E_l] \hookrightarrow e' \qquad \vdash e' \to^* \langle \Sigma', \Delta', \sigma' \rangle$$

If $\ell = \top$ is inconsistent with theory (Δ, Σ), then

1. *$\Sigma' = \Sigma$ and $\Delta' = \Delta$.*
2. *For any context v:*

$$\{c \mid \Sigma, \Delta \vdash \textbf{concretize}\ \sigma\ \textbf{with}\ v \to \langle \Sigma_0, \Delta_0, c \rangle\} =$$
$$\{c \mid \Sigma', \Delta' \vdash \textbf{concretize}\ \sigma'\ \textbf{with}\ v \to \langle \Sigma'_0, \Delta'_0, c \rangle\}$$

Proof. If σ is output, then the concretization of σ must have produced a model assigning values to levels consistent with the logical environment.

The high view of the sensitive value is selected only if the level variable is set to $\top$. □

4.3 Privacy Non-Interference

We can now show Jeeves executions satisfy a non-interference property. If a policy determines that a sensitive value is low-confidence, the value output is the same as if the high-confidence value was not involved in evaluation at all, and vice versa for high-confidence values.

Theorem 4.4 (Privacy Non-Interference)**.** *Consider a sensitive value $v = \langle Exp_{low} | Exp_{high} \rangle_\ell$. Let Exp be an expression containing v, $Exp \hookrightarrow e$, $Exp_{low} \hookrightarrow e_l$, $\vdash e \to^* \langle \Sigma', \Delta', v \rangle$, and $\Sigma', \Delta' \vdash$ **concretize** $v \to \langle \Sigma'', \Delta'', c \rangle$ If $\ell = \bot$ is consistent with Σ'', then $\vdash$ **concretize** $e[v \mapsto e_l] \to^* \langle \Sigma', \Delta', c \rangle$.*

Proof. We can show by induction on the evaluation rules that evaluation of any λ_J expression e with no **concretize** sub-expressions is deterministic, producing a unique value v and logic environment Σ, Δ. We can then show that if we have $\vdash e \to^* \langle \Sigma, \Delta, \sigma \rangle$ and evaluating **concretize** σ produces model $\mathcal{M}$, applying these substitutions when the logic variables are created in e equivalent to applying these substitution to σ. Then for each sensitive value $v = \langle Exp_{low} | Exp_{high} \rangle_\ell$, the result is equivalent to that if $\mathcal{M}$ were applied to determine the value of ℓ and thus the value of v. □

5. Scala Embedding

We have implemented Jeeves as an embedded domain-specific language in Scala programming language [18]. Scala's overloading capabilities offer us the necessary flexibility in designing a domain specific language for λ_J with the benefit of interoperability with existing Java technology.

In this section we discuss our Scala embedding of λ_J and our implementation of the Jeeves library on top of that. We describe how we used features of Scala language to interpret λ_J's lazy evaluation of symbolic expressions, how we collect constraints, and how we interact with the Z3 SMT solver. On top of the functional model we have presented, we also handle objects and mutation.

The code is publicly available at:
`http://code.google.com/p/scalasmt/`.

5.1 ScalaSMT: Scala Embedding of λ_J

We defined the λ_J abstract syntax in Scala to support the evaluation of symbolic expressions. For every kind of symbolic expression of λ_J, we have declared a corresponding Scala case class, for instance IntExpr corresponding to symbolic integer expressions. Arithmetic and boolean operators are defined as methods constructing new expressions. We use *implicit type conversions* to lift concrete Scala

values to symbolic constants. Scala's type inference resolves x+1 to x.+(Constant(1)) which in turn evaluates to Plus(x, Constant(1)), where x is a symbolic integer variable. Implicit type conversion allows us to use concrete expressions in place of symbolic but require type annotations where a symbolic expression is expected to be used as an argument to a function. In the case of integer expressions, the type Int is used for integers that must be concrete and the type IntExpr is used otherwise.

The three core language extensions **defer**, **assert**, and **concretize** are implemented as library calls. The library is represented as a trait in Scala that maintains the logical and default constraint environments as lists of symbolic boolean expressions. Calls to **concretize** invoke an off-the-shelf SMT solver [15] for the satisfiability query MODEL. We translate λ_J constraints to QF_LIA logic of SMT-LIB2 [2] and use incremental scripting to implement the default logic decision procedure. Concretization in ScalaSMT differs from λ_J in two ways. First, **concretize** accepts an arbitrary boolean expression rather than a **context** equality predicate. Second, **concretize** is not allowed to be a part of a symbolic expression in ScalaSMT. Since concretization generally happens as part of **print** routine, this restriction does not affect our case studies.

In addition to boolean and linear integer constraints, Scala embedding has symbolic expressions for objects with equality theory corresponding to records in λ_J. Objects are modeled as a finite algebraic datatype in Z3 [15]. The set of available objects is maintained by ScalaSMT using registration of instances of a special trait Atom. Fields of objects are modeled as total functions interpreted at the time of concretization. Fields are (sort-)typed. Field values are arbitrary ScalaSMT expressions and constants. ScalaSMT does not check types of symbolic object expressions as we rely on Scala's dynamic invocaton to resolve field dereferences. Whenever a field is undefined for an object, we use special zero values (null , 0, or false) for the corresponding function value in SMT.

ScalaSMT does not have symbolic collections of expressions. Instead, we use implicits to extend standard Scala collection library with filter and has methods that take symbolic arguments. The argument to filter is a function f from an element to a symbolic boolean. It maps every element o to conditional expression IF (f(o)) o ELSE NULL. Method has takes a symbolic object o and produces a disjunction of equalities between elements of the collection and o.

5.2 Jeeves as a Library in Scala

We have implemented Jeeves as a library on top of ScalaSMT. Our library has function calls corresponding to Jeeves's sensitive values, **level** construct, **policy** construct, and contextual output functions (see Figure 7.)

Levels are introduced using mkLevel method that returns a logical level variable which can be either $\top$ or $\bot$. Sensitive values are created with mkSensitive methods that take a level variable together with high and low values. Context is a logical object variable CONTEXT. To introduce a level policy, the programmer calls policy method and supplies a level variable, the desired level, and a boolean condition. Boolean condition is passed by name to delay its evaluation till concretization. This way policies that refer to mutable parts of the heap will produce correct constraints for the snapshot of the system at the concretization time.

The Jeeves library supports mutation in variables and object fields by treating the mutable state as part of the context in **concretize** call to ScalaSMT. Mutable fields are interpreted at the time of **concretize**. Policies that depend on mutable state are evaluated to boolean conditions during concretization. The set of allocated JeevesRecords is supplied at concretization. These conditions together with the equality predicate CONTEXT = ctx are used to concretize expressions in ScalaSMT.

```
trait JeevesLib extends ScalaSMT {
  trait JeevesRecord extends Atom {register(this)}
  val CONTEXT: Symbolic // Context variable.

  // Level variables and policies.
  def mkLevel(): LevelVar
  def policy(lvar: LevelVar, f: ⇒ Formula, l: Level)

  // Creating sensitive values.
  def mkSensitiveInt(lvar: LevelVar,
    high: IntExpr, low: IntExpr): IntExpr
  def mkSensitive(lvar: LevelVar,
    high: Symbolic, low: Symbolic): Symbolic

  // Concretizing for output.
  def concretize[T](ctx: Symbolic, e: Expr[T]): T
}
```

Figure 7: Jeeves library in Scala

6. Experience

To evaluate the expressiveness of Jeeves and the feasibility of the execution model we have implemented case studies. We have implemented a conference management system and a social netowrk. Our case studies demonstrate the following:

- Jeeves allows the programmer to separate the "core," non-privacy-related functionality from the privacy policies. This allows each portion of the code to be concise and allows the programmer to separately test policies and functionality.
- Jeeves policies are sufficiently expressive to capture the policies for a conference management system and social network.
- [*Jeeves programs can run without too much solver overhead.*]

6.1 Conference Management System

Conference management systems have information flow policies that are simple, yet important and also difficult to implement given the interaction of features the systems support. We have implemented a conference management system, JConf, to demonstrate how a well-known system with privacy concerns looks in Jeeves. Our implementation demonstrates that Jeeves allows us to implement all JConf functionality, including search and display over final paper versions, with a simple core functionality. Jeeves facilitates the implementation of common policies for a conference management system without requiring changes to code implementing the core functionality.

JConf supports the following subset of the functionality mentioned on the website for the HotCRP conference management system [11]:

- Smart paper search (by ID, by reviewer, etc.).
- Paper tagging (for instance, "Accepted" and "Reviewed by: ...") and search by tags.
- Assigning reviews, collecting responses to reviews, and displaying review content.
- Management of final paper versions.

JConf does not implement functionality for which confidentiality is less key: for instance, the process of bidding for papers.

All JConf core functionality adheres to the privacy policies. JConf implements the following information flow policies:

File	Total LOC	Policy LOC
ConfUser.scala	11	0
PaperRecord.scala	103	37
PaperReview.scala	21	6
ConfContext.scala	6	0
JConfBackend.scala	56	0
Total	195	42

Table 1. Breakdown of lines of code across the JConf source.

```
class PaperReview(id: Int, reviewerV: ConfUser, var body:
      String, var score: Int) extends JeevesRecord {
  val reviewer = {
    val level = mkLevel();
    val vrole = CONTEXT.viewer.role;
    val isInternal = (vrole == ReviewerStatus) ||
      (vrole == PCStatus)
    policy(level, isInternal, ⊤);
    policy(level, ! isInternal, ⊥);
    mkSensitive[ConfUser](level, reviewerV, NULL)
  }
}
```

Figure 8:

- *Paper titles* are visible to the authors of the paper, reviewers, and PC members during all stages. Paper titles are visible to everyone during the public stage.
- *Author names* are visible to the authors on the paper during all stages, to reviewers and PC members during and after the rebuttal stage, and to everyone during the public stage if the paper has been accepted.
- *Reviewer identities* are revealed only to PC members.
- *Reviews* and *scores* are revealed to authors of the paper, reviewers, and PC members after the review phase. During the review phase, reviewers must have submitted a review for a paper before they are allowed to see other reviews for the paper.

Our JConf implementation allows us to separate the declaration of policies and code. In Table 1 we show the breakdown of code and policies across the source. The policies are concentrated in the data classes PaperRecord.scala and PaperReview.scala, which are class definitions describing the attributes and policies associated with maintaining data associated with paper records and paper reviews. The ConfContext file contains the definition for the output context. The other files, including the implementation of the core functionality in JConfBackend.scala, do not contain policies. This allows the core functionality to be concise: the implementation of our back-end functionality as specified above is only 56 lines.

The implementation of the core functionality of JConf is agnostic to the policies. The JConf back end stores a list of PaperRecord objects and supports adding papers, updating components of papers, and searching over papers by ID, name, and tags. We show the function to search papers by tag below:

```
def searchByTag(tag: PaperTag) =
  papers.filter(_.getTags().has(tag))
```

This function produces a list of symbolic PaperRecord objects which are equal to objects containing paper data if the paper tag tag is present and null otherwise. The core program can be concise because it does not have to be concerned with policies.

We use Jeeves policies to implement the policies specified above in terms of program variables such as a paper's list of tags and values from the output context. To provide an example of a data class definition, we show the definition of the PaperReview class in Figure 8. A PaperReview object has the fields reviewer, body, and score. The PaperReview class defines a policy that the identity of the reviewer as stored in the reviewer field is visible only to other reviewers and PC members. The code introduces a new level variable level, adds a policy that the context viewer must be a reviewer or PC member to see the object. The policies on allowed contexts for seeing the entire PaperReview object are defined in the PaperRecord class representing data associated with papers.

Localizing the policies with respect to data facilitates updating policies. To change at what stage of the conference when reviewers are allowed to see names of authors, we can simply change the few lines of code corresponding to the author list policy. Because the programmer is not responsible for handling policy enforcement, the programmer does not have to make coordinated changes across the code base to update policies.

6.2 Social Network

Being able to rapidly develop code that implements information flow policies is important for social networking websites. Privacy issues have put the social network website Facebook under the scrutiny of the American Federal Trade Commission [25], making it important that they do not continue to leak sensitive data. On the other hand, one of Facebook's key values is to "move fast," rapidly developing innovative features to keep users engaged [27]. The separation of policies and core program in Jeeves can help developers achieve this rapid development of features that may process sensitive data.

We show how to use Jeeves to make it easier for the programmer to automatically enforce user-defined settings about information flow. We have implemented Jeeves Social Net, a social network that demonstrates how we can add policies to control confidentiality of user-shared data with few changes to the query code. As with the conference management system, Jeeves allows for a separation of core policies from code.

Jeeves Social Net core functionality involves storing and allowing queries over the following data:

- user attributes such as names, e-mails, and networks,
- a friendship relation between users, and
- dynamically changing properties such as user location.

Jeeves Social Net allows a user u to define policies about who can see attributes such as name, e-mail, and current location based on the relationship of the viewer to the u. The system allows the user to define different versions of their information to be shown to viewers given which level they satisfy. These policies can be stateful: for instance, a policy on the visibility of user u's location can refer to the location of u and the location of output viewer v.

Jeeves allows the programmer to develop policies and core functionality separately. In our source, all policies reside in UserRecord class representing a user, while the query code in SocialNetBackend is left intact. This facilitates rapid development of core functionality, as the programmer can extend the SocialNetBackend arbitrarily and rely on the Jeeves system to enforce information policies. The programmer can also easily change the policies enforced across the entire program by changing the policy code in UserRecord.

In the rest of this section, we walk through how we implement interesting policies in Jeeves Social Net: support for user-defined policies that may depend on the friendship relation, stateful location-data policies, and policies that have mutual dependencies as a result of a symbolic context.

Defining Viewer Levels. Each sensitive field in a UserRecord object is defined in terms of the level of the output viewer. We use Jeeves level variables to define three levels: Anyone is most permissive and allows public access, Friends allows access only to

friends, and Self is most restrictive and disallows access to everyone except the user herself. The following function creates level variables associated with user-defined viewer levels (Anyone, Self, or Friends) using mkLevel and policy:

```
def level(ul: UserLevel) = {
  val a = mkLevel();
  val me = CONTEXT == this;
  ul match {
    case Anyone ⇒
    case Self ⇒ policy(l, ! me, ⊥)
    case Friends ⇒
      policy(l, ! (me || friends.has(CONTEXT)), ⊥);
  };
 a
}
```

The CONTEXT refers to the user at the other end of the output channel. The set of friends, which is mutable, is encapsulated in a private field friends of UserRecord.

We can use this level variable to create sensitive values for user fields based on user-specified viewer levels. The constructor for the UserRecord class takes parameters nameL: UserLevel and friendL: UserLevel to specify who can see the name and friends fields. To create a sensitive property for the name of a user, passed to the constructor as nameV: string, we declare an observer field:

```
val name = mkSensitive(level(nameL), nameV, NULL)
```

We can create a friends list that is visible based on the friends level friendsL as follows:

```
def getFriends() = {
  val l = level(friendsL);
  friends.map(mkSensitive(l, _))
}
```

When these fields are accessed, the results will only be displayed to viewers who have an appropriate level of access.

Policies become implicitly combined when different sensitive values interact. For example, to get names of friends of a user, we simply call:

```
user.getFriends().map(_.name)
```

Although the code looks the same as if without Jeeves, the context user here must simultaneously be able to access the list of friends and the name property to see the name of a friend.

Location Policy. More than ever, users are socially sharing data about their current locations online. The location mash-up website PleaseRobMe [3] demonstrates that if disclosure of this information is not carefully managed, people can easily use this information for harm, for instance in determining candidates for apartment robberies. Jeeves allows programmers to easily express policies protecting location data based on not just "friend" relationships, but also on policies involving dynamically-changing user locations.

In Jeeves Social Net, a user may choose to share her location with friends, with users nearby, or only to friends who are nearby. To write the policy that only a nearby user can see the location, we create sensitive values for coordinates in the setter method guarded by DISTANCE policy:

```
1  var X: IntExpr = 1000
2  var Y: IntExpr = 1000
3
4  def setLocation(x: BigInt, y: BigInt) {
5    val l = mkLevel();
6    policy(l, DISTANCE(CONTEXT, this) ⩾ 10, ⊥);
7    this.X = mkSensitiveInt(l, x, 1000);
8    this.Y = mkSensitiveInt(l, y, 1000);
9  }
10
11 def DISTANCE(a: Symbolic, b: Symbolic) =
12   ABS(a.X − b.X) + ABS(a.Y − b.Y)
```

The policy uses sensitive values for X and Y to guard the values themselves. We can do this because whenever there are such circular dependencies, the Jeeves runtime will choose a safe but locally-maximal assignment to levels. For example, if all users in the network are nearby, it is safe to return low values for everyone. However, Jeeves would output the actual values, since that maximizes the number of ⊤ levels without sacrificing safety.

Since policies and query code are separated, to change the location policy, we only need to modify the setter. A stronger policy that permits only *friends* nearby to see the location requires one change to line 5 to replace mkLevel() with level(Friends).

Symbolic Context. Jeeves also allows the context to contain sensitive values. As an example, consider the following function, which sends a user's name to her friends:

```
def announceName(u: UserRecord) =
  for (f ← u.getFriends())
    yield email(f, u.name)
```

The email function sends to f a concretizated version of u.name with CONTEXT = f. Since the friends list is symbolic, f is symbolic as well. This means that f will take high value only if the corresponding friend of u is allowed to see the list of friends of u. The name of u is revealed only if its policies permit f to see. Because Jeeves handles circular dependencies by finding a safe but locally-maximal assignment, the Jeeves runtime system will send the name to each friend if the friend is permitted to see the name. Such reasoning about symbolic contexts is hard to simulate in runtime systems such as Resin [26] that do not use symbolic constraints.

6.3 Jeeves Limitations

At present, Jeeves does not provide support for determining whether policies are consistent or total. The programer can, however, easily discover bugs with underspecification or inconsistency through testing: underspecified level variables default to ⊤ and inconsistent policies will cause the system never to output a value. We anticipate that both underspecification and inconsistency are issues for which we can provide more support using enhanced static analysis.

7. Related work

Jeeves privacy policies yield comparable expressiveness to state-of-the-art languages for verifying system security such as Jif [17], Fine [4], and Ur/Web [5]. Rather than providing support for verifying properties, however, the Jeeves execution model handles policy enforcement, guaranteeing that all programs adhere to the desired properties.

The Jeeves runtime is similar to the system-level data flow framework Resin [26], which allows the programmer to insert checking code to be executed at output channels. Jeeves's declarative policies allows the programmer to specify policies at a higher level and capture dependencies between policies that can be automatically resolved.

Jeeves can also be compared to aspect-oriented programming (AOP) [10]. Existing frameworks for AOP provide hooks for explicit annotations at join points. Jeeves differs from AOP because Jeeves's constraint-based execution model supports more a more powerful interaction with the core program. The most similar work in AOP is Smith's logical invariants [23] and method for generating aspect code for behavior such as error logging automatically [22]. Smith's method is static and involves reconstructing values such as the runtime call stack in order to insert the correct code at fixed control

flow points. Jeeves's symbolic evaluation allows privacy policies to affect control flow decisions.

The way Jeeves handles privacy is inspired by angelic nondeterminism [7]. The mechanism for angelic nondeterminism in Jeeves borrows from CFLP-L, a constraint functional programming calculus presented by Mück *et al.* [16]; similar functional logic models have also been implemented in languages such as Mercury [24], Escher [13], and Curry [8, 9]. Our system differs from these systems in terms of the restrictions we place on nondeterminism and the execution model. λ_J leaves functions and the theory of lists out of the logical model, which circumvents the need to have a backtracking and/or unification-based execution model. λ_J execution also supports default logic [1] to facilitate reasoning when programming with constraints.

Our work is also related to work in executing specifications and dynamic synthesis. Jeeves differs from existing work in executing specifications [14, 20] because our goal is to propagate nondeterminism alongside the core program rather than to execute isolated nondeterministic sub-procedures. Program repair approaches such as Demsky's data structure repair [6], the Plan B [21] system for dynamic contract checking, and Kuncak *et al.*'s synthesis approach [12] also target local program expressions.

References

[1] G. Antoniou. A tutorial on default logics. *ACM Computing Surveys (CSUR)*, 31(4):337–359, 1999.

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

[3] B. Borsboom, B. v. Amstel, and F. Groeneveld. PleaseRobMe. `http://pleaserobme.com`, July 2011.

[4] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. *SIGPLAN Not.*, 45 (6):412–423, 2010. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1809028.1806643.

[5] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1924943.1924951`.

[6] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 176–185, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062499.

[7] R. W. Floyd. Nondeterministic algorithms. *J. ACM*, 14:636–644, October 1967. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321420.321422. URL `http://doi.acm.org/10.1145/321420.321422`.

[8] M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 1–23. Springer LNCS 631, 1992.

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

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[11] E. Kohler. HotCRP. `http://www.cs.ucla.edu/~kohler/hotcrp/`.

[12] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

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

[14] C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/44501.44503.

[15] L. D. Moura and N. Björner. Z3: An efficient SMT solver. In *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[16] A. Mück and T. Streicher. A tiny constraint functional logic language and its continuation semantics. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 439–453, London, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3.

[17] A. C. Myers. JFlow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[18] 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.

[19] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[20] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *OOPSLA Companion*, pages 999–1006, 2009.

[21] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[22] D. R. Smith. A generative approach to aspect-oriented programming. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2004. ISBN 3-540-23580-9.

[23] D. R. Smith. Aspects as invariants. In O. Danvy, H. Mairson, F. Henglein, and A. Pettorossi, editors, *Automatic Program Development: A Tribute to Robert Paige*, pages 270–286, 2008.

[24] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an efficient purely declarative logic programming language. In *In Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.

[25] J. E. Vascellaro. Facebook grapples with privacy issues. In *The Wall Street Journal*. May 19 2010.

[26] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22th ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[27] H. Zhao. Hiphop for PHP: Move fast. `http://developers.facebook.com/blog/post/358/`, February 2010.