A Language for Automatically Enforcing Privacy Policies

Jean Yang Kuat Yessenov Armando Solar-Lezama

MIT CSAIL

{jeanyang, kuat, asolar} @csail.mit.edu

Abstract

It is becoming increasingly important for applications to protect sensitive data. With current techniques, the programmer bears the burden of ensuring that the application's behavior adheres to policies about where sensitive values may 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: 1) core programs describing functionality independent of privacy concerns and 2) declarative, decentralized policies controlling how sensitive values are disclosed. Each sensitive value encapsulates multiple views; policies describe which views are allowed based on the output context. The system is responsible for automatically ensuring that 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.

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

General Terms Languages, security

Keywords Language design, run-time system, privacy, security

1. Introduction

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

Ensuring compliance with privacy policies requires reasoning 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 statically [4, 19] and dynamically [27]. While these checking tools can help avoid data leaks, the programmer is still responsible for implementing applications that display enough information to satisfy the user's needs without violating privacy policies. The programming model that we propose goes beyond checking to simplify the process of writing the code that preserves confidentiality.

The main contribution of this paper is a new programming model that makes the system responsible for automatically producing outputs consistent with programmer-specified policies. This automation makes it easier for programmers to enforce policies specifying how each sensitive value should be displayed in a given context. The programming model has two components: a core program representing policy-agnostic functionality and privacy policies controlling the disclosure of sensitive values. This separation of policies from core functionality allows the programmer to express policies explicitly in association with sensitive data rather than implicitly across the code base. The declarative nature of policies allows the system to ensure compliance even when these policies interact in non-trivial ways.

We have implemented this programming model in a new functional constraint language named Jeeves. Jeeves introduces three main concepts: *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 a level 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 level variables. The language of policies is a decidable logic of quantifier-free arithmetic constraints, boolean constraints, and equality constraints over records and record fields. A policy may refer to a *context* value characterizing the output channel and containing relevant information about how the data is viewed.

For example, in a small social networking application we implemented as a case study, we included a policy that allows users to restrict the disclosure of their location to users in their geographic vicinity. Because the location is a sensitive value, a function such as **print** that tries to output a value derived from a location will need to be passed a context containing the location of the user to whom this value is about to be displayed. Using this context, the runtime system can then derive an output that is consistent with the policy.

We formally specify Jeeves to show that the high-confidentiality component of a sensitive value can only affect program output if the policies allow it. We define Jeeves in terms of λ_J, a constraint functional language that describes the propagation and enforcement of policies in Jeeves. λ_J is different from existing constraint functional languages [9, 10, 15, 18, 25] in the restrictions it places on the logical model and its use of default logic to provide determinism in program behavior. These restrictions make it possible for λ_J to have an efficient execution model without sacrificing too much expressiveness. There is a straightforward translation from Jeeves

This work was funded in part by the U.S. Government under the DARPA UHPC and NSF Graduate Research 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.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

$$
\begin{array}{lcll}
\mathit{Level} & ::= & \bot \mid \top & \text{levels} \\
\mathit{Exp} & ::= & \upsilon \mid \mathit{Exp}_1 \ (\mathrm{op}) \ \mathit{Exp}_2 & \text{expressions} \\
 & \mid & \textbf{if } \mathit{Exp}_1 \textbf{ then } \mathit{Exp}_t \textbf{ else } \mathit{Exp}_f & \\
 & \mid & \mathit{Exp}_1 \ \mathit{Exp}_2 & \\
 & \mid & \langle \mathit{Exp}_\bot \mid \mathit{Exp}_\top \rangle(\ell) & \\
 & \mid & \textbf{level } \ell \textbf{ in } \mathit{Exp} & \\
 & \mid & \textbf{policy } \ell\colon \mathit{Exp}_p \textbf{ then } \mathit{Level} \textbf{ in } \mathit{Exp} & \\
\mathit{Stmt} & ::= & \textbf{let } x : \tau = \mathit{Exp} & \\
 & \mid & \textbf{print } \{\mathit{Exp}_c\} \ \mathit{Exp} &
\end{array}
$$

Figure 1: Jeeves syntax.

to λ_J: Jeeves level 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 allow Jeeves to automatically enforce policies about information flow.

We implemented Jeeves as a domain-specific language embedded in Scala [20] using the Z3 SMT solver [17] to resolve constraints as a way to demonstrate the feasibility of the Jeeves programming model. To evaluate the expressiveness of Jeeves, we have used our Scala embedding to implement a small conference management system and a small social network. The case studies show that Jeeves allows the separate implementation of functionality and policies. For example, in the conference management example the code makes no attempt to differentiate between users, or even the general public, yet the policies ensure that the system displays the right level of information to each user through every phase of the review process.

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

- We present a programming model and language, Jeeves, that allows programmers to separate privacy concerns from core program functionality.
- We formalize the dynamic and static semantics of Jeeves in terms of a λ_J, a new constraint functional language. 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 an embedded domain-specific language in Scala using the Z3 SMT solver.
- We describe small case studies that show that Jeeves supports the desired policies and allows the programmer to separately develop core functionality and policies.

2. Delegating Privacy to Jeeves

Jeeves allows the programmer to specify policies explicitly and upon data creation rather than implicitly across the code base. The Jeeves system trusts the programmer to correctly specify policies describing high- and low-confidentiality views of sensitive values and to correctly provide context values characterizing output channels. The runtime system is responsible for producing outputs consistent with the policies given the contexts. Jeeves guarantees that the system will not leak information about a high-confidentiality value unless the policies allow this value to be shown.

In this section, we introduce a simple conference management example to explain the main ideas in Jeeves: introducing sensitive values, writing policies, providing contexts, and implementing the core program logic. Conference management systems have simple information flow policies that are difficult to implement given the interaction of features. Being able to separately specify the policies allows the core functionality to be quite concise. In fact, we can write a single program that allows all viewers to access the list of papers directly for searching and viewing. The program relies on the Jeeves runtime system to display the appropriately anonymized information for reviewers vs. the general public.

For the sake of brevity, we present Jeeves using an ML-like concrete syntax, shown in Figure 1.

2.1 Introduction to Jeeves

We first describe how to introduce sensitive values, use them to compute result values, and display the results in different output contexts. Suppose we have the policy that a sensitive value name should be seen as "Alice" by users with a high confidentiality level and as "Anonymous" by anybody else. A Jeeves program can use the name value as follows:

```
let msg = "Author is " + name
print { alice } msg  (* Output: "Author is Alice" *)
print { bob } msg    (* Output: "Author is Anonymous" *)
```

To achieve the different outputs for alice and bob, we associate a policy with name by declaring it through the following Jeeves code:

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

This code introduces a level variable a and associates with it a policy that if the **context** value is not alice, then the value is $\bot$. The context value represents a viewer in the output channel. The code then attaches this policy to the sensitive value <"Anonymous" | "Alice">(a), which defines the low-confidentiality view as the string "Anonymous" and the high-confidentiality view as "Alice". When this code is executed, 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 "Author is Anonymous".

Each sensitive value defines a low-confidentiality and high-confidentiality view for a value. The Jeeves programmer defines sensitive values by introducing a tuple $\langle v_\bot | v_\top \rangle_\ell$ where $v_\bot$ is the low-confidentiality value, $v_\top$ is the high confidentiality value, and ℓ is a level variable associated with a set of policies determining which of the two values to show. An expression containing n sensitive values can evaluate to one of 2^n possible views. 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). It is possible to encode more than two privacy levels, but for the sake of simplicity the paper assumes only two.

Policies, introduced through **policy** expressions, provide declarative rules describing when to set a level variable to $\top$ or $\bot$. Notice that the policy above forces a to be $\bot$ when the user is not alice; other policies could further restrict the level variable to be $\bot$ even for alice, but no amount of policy interactions can allow a different user to see the $v_\top$ value in contradiction with the policy. Policies may mention variables in scope and also the **context** variable, which corresponds to an implicit parameter characterizing the output channel.

The **context** construct relieves the programmer of the burden of structuring code to propagate values from the output context to the policies. Statements such as **print** that release information to the viewer require a context parameter. The Jeeves runtime system propagates policies associated with sensitive values so that when a value is about to be displayed though an output channel, the right context can be inserted into the policy and the appropriate result can be produced. In addition to **print** shown above, other output channels include sending e-mail and writing to file.

2.2 Declarative and Decentralized Policies

We now describe how to write policies in Jeeves using fragments of our conference management example. The paper record is defined below; it assumes single author papers to simplify the presentation.

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

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

```
let mkPaper
  ( title : string) (author: string)
  (reviews: review list ) (accepted: bool option): paper =
  level tp, authp, rp, accp in
  let p = {  title  = < "" |  title >(tp)
           ; author = < "Anonymized" | author>(authp)
           ; reviews = < [] |  reviews>(rp)
           ; accepted = < none | some accepted >(accp) } in
  addTitlePolicy p tp; addAuthorPolicy p authp;
  addReviewsPolicy p rp; addAcceptedPolicy p accp;
    p
```

This function introduces level variables for each of the fields, creates sensitive values for each of the fields, attaches policies to the level variables, and returns the resulting paper record.

The Jeeves programmer associates policies with sensitive values by introducing level variables, attaching policies to them, and using them to create sensitive values. Consider the policy that the title of a paper should be visible to the authors of the paper, reviewers, and PC members and only visible to the general public after it is public that the paper has been accepted. We can define addTitlePolicy as follows:

```
let addTitlePolicy  (p: paper) (a: level ): unit =
  policy a: ! (context.viewer = p.author
    || context.viewer. role  = Reviewer
    || context.viewer. role  = PC
    || (context.stage = Public && isAccepted p)) then ⊥
```

This function attaches to level variable a a policy that sets the level to ⊥ unless the viewer has a right to see the paper title.

Policies may refer to values corresponding to the output channel through the **context** variable. The condition for the policy in function addTitlePolicy uses the viewer and stage fields of the **context** variable, which have types confView and confStage types, respectively:

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

A context value of type confView must be produced in order for an output channel to access a sensitive value produced by the addTitlePolicy function. With type inference, it is not necessary for the programmer to provide the context type annotation. In the Scala implementation, the programmer does not need to provide context annotations.

2.2.1 Policy Interactions

The Jeeves model helps prevent the programmer from inadvertently leaking information about one value through the enforcement of a policy for another value. We show how Jeeves can prevent the programmer from writing code where the policy for paper titles leaks information about a paper record's accepted field.

Consider the following situation in which the policy for one sensitive field (paper titles) depends on another sensitive field (whether the paper has been accepted). In the addTitlePolicy function, the predicate isAccepted p depends on the accepted field of the paper p, which is **some** accepted if a decision has been made (and the decision is known) or **none** otherwise. The accepted field needs its own policy to prevent its status from being leaked early. The following function adds the appropriate policy:

```
let addAcceptedPolicy (p: paper) (a: level ): unit =
  policy a: ! (context.viewer. role  = Reviewer
    || context.viewer. role  = PC
    || context.stage = Public) then ⊥
```

This policy allows reviewers and program committee members to always see whether a paper has been accepted and for others to see this field only if the stage is Public. With this policy in place, the title field cannot leak information about the accepted tag. Even if the policy for paper titles were to drop the **context**.stage = Public requirement, the policy for accepted would prevent the titles of accepted papers from being leaked before the Public stage.

2.2.2 Circular Dependencies and Defaults

The Jeeves system can also enforce policies when there are circular dependencies between sensitive values, as could happen when a context value depends on a sensitive value. Consider the following function that associates a policy with the authors of a paper:

```
let addAuthorPolicy (p: paper)(n: level ) : unit =
  policy n:
    !( isAuthor  p context.user  ||
      (context.stage = Public && isAccepted p)) then ⊥
```

This policy says that to see the author of a paper, the user must be an author or the paper must be a publicly accepted paper. 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. Here, **context**.user is a sensitive value, as the value of the author variable depends on the viewer.

This leads to a circular dependency that makes the solution underconstrained: the value of the message recipient on the context value, which contains the message recipient. Either sending mail to the empty user or sending mail to the author is correct under the policy. The latter behavior is preferred, as it ensures that user *a* can communicate with user *b* without knowing private information about user *b*. The Jeeves runtime ensures this maximally functional behavior by setting level variables to ⊤ by default: if the policies allow a level variable to be ⊤ or ⊥, the value will be ⊤.

3. The λ_J Language and Semantics

To more formally describe the guarantees, we define the semantics of Jeeves in two steps; first, we introduce λ_J, a simple constraint functional language based on the λ-calculus, and then we show how to translate Jeeves to λ_J. λ_J differs from existing constraint functional languages [9, 10, 15, 25] 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 allow 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 **defer** and **assert** for introducing and constraining logic variables, as well as a **concretize** construct to produce concrete values from them. The dynamic semantics describe the lazy evaluation of expressions with logic variables and the interaction with the constraint environment. The static semantics describe how the system guarantees evaluation progress and enforces restrictions on symbolic values and recursion. The translation from Jeeves to λ_J illustrates how Jeeves uses λ_J's lazy evaluation and constraint propagation, combined with Jeeves restrictions on how logic variables are used, to provide privacy guarantees.

$$
\begin{array}{lcll}
c & ::= & n \mid b \mid \lambda x:\tau.e \mid \ \mathsf{record}\ x \vec{:} \upsilon & \\
 & \mid & \textbf{error} \mid () & \text{concrete primitives} \\
\sigma & ::= & x \mid \textbf{context}\ \tau & \text{symbolic values} \\
 & \mid & c_1\ (\mathrm{op})\ \sigma_2 \mid \sigma_1\ (\mathrm{op})\ c_2 & \\
 & \mid & \sigma_1\ (\mathrm{op})\ \sigma_2 & \\
 & \mid & \textbf{if}\ \sigma\ \textbf{then}\ \upsilon_t\ \textbf{else}\ \upsilon_f & \\
\upsilon & ::= & c \mid \sigma & \text{values} \\
e & ::= & \upsilon \mid e_1\ (\mathrm{op})\ e_2 & \text{expressions} \\
 & \mid & \textbf{if}\ e_1\ \textbf{then}\ e_t\ \textbf{else}\ e_f \mid e_1\ e_2 & \\
 & \mid & \textbf{let}\ x:\tau = e_1\ \textbf{in}\ e_2 & \\
 & \mid & \textbf{let rec}\ f:\tau = e_1\ \textbf{in}\ e_2 & \\
 & \mid & \textbf{defer}\ x:\tau\ \{e\}\ \textbf{default}\ \upsilon_d & \\
 & \mid & \textbf{assert}\ e & \\
 & \mid & \textbf{concretize}\ e\ \textbf{with}\ \upsilon_c &
\end{array}
$$

Figure 2: The λ_J abstract syntax.

3.1 The λ_J Language

λ_J is the λ-calculus extended with logic variables. Figure 2 shows the abstract syntax of λ_J. Expressions (e) 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. λ_J evaluation produces irreducible values (υ), which are either concrete (c) or symbolic (σ). Concrete values are what one would expect from λ-calculus, while symbolic values are values that cannot be reduced further due to the presence of logic variables. Symbolic values also include the **context** construct which allows constraints to refer to a value supplied at concretization time. The **context** variable is an implicit parameter [14] provided in the **concretize** expression. In the semantics we model the behavior of the **context** variable as a symbolic value that is constrained during evaluation of **concretize**. λ_J contains a **let rec** construct that handles recursive functions in the standard way using **fix**.

A novel feature of λ_J is that logic variables are also associated with a *default value* that serves as a default assumption: this is the assigned value for the logic variable unless it is inconsistent with the constraints. The purpose of default values is to provide some determinism when logic variables are underconstrained.

3.2 Dynamic Semantics

The λ_J evaluation rules extend λ-calculus evaluation with constraint propagation and symbolic evaluation of logic variables. Evaluation involves keeping track of constraints which are required 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) path condition. Evaluation happens in the context of a path condition $\mathcal{G}$, an environment $\Sigma = \emptyset \mid \{\sigma\} \mid \Sigma \cup \Sigma'$ storing the current set of constraints, and an environment $\Delta = \emptyset \mid \{\sigma\} \mid \Delta \cup \Delta'$ storing the set of constraints on default values for logic variables. Evaluation rules take the form

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

Evaluation produces a tuple $\langle \Sigma', \Delta', e' \rangle$ of a resulting expression e' along with modified constraint and default environments. In Figure 3 we show the small-step dynamic λ_J semantics.

3.2.1 Evaluation with Logic Variables

λ_J has the expected semantics for function application 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 whose conditions evaluate to concrete values evaluate according to the E-CONDTRUE, and E-CONDFALSE rules as one would expect. When the condition evaluates to a symbolic value, the whole conditional evaluates to a symbolic **if-then-else** value by evaluating both branches as described by the E-CONDSYMT and E-CONDSYMF rules. Note that λ_J expressions are pure (effects are only in Jeeves statements) so side effects cannot occur in conditional branches.

Evaluating under symbolic conditions is potentially dangerous because evaluation of such conditionals with a recursive function application in a branch could lead to infinite recursion when the condition is symbolic. Our system prevents this anomalous behavior by using the type system to enforce that recursive calls are not made under symbolic conditions (Section 3.3).

3.2.2 Introduction and Elimination of Logic Variables

In λ_J, logic variables are introduced through the **defer** keyword. To illustrate the semantics of **defer** consider the example below.

```
let x: int = defer x': int { x' > 0 } default 42
```

As we show in the E-DEFER evaluation rule, the right-hand side of the assignment evaluates to an α-renamed version of the logic variable x'. Evaluation adds the constraint $\mathcal{G} \Rightarrow$ x' > 0 to the constraint environment and the constraint $\mathcal{G} \Rightarrow$ x' $= 42$ to the default constraint environment. The constraint $\mathcal{G} \Rightarrow$ x' > 0 is a hard constraint that must hold for all derived outputs, while $\mathcal{G} \Rightarrow$ x' $= 42$ is a constraint that is only used if it is consistent with the resulting logical environment. Hard constraints are introduced within the braces ($\{\}$) of **defer** expressions and through **assert** expressions; soft constraints are introduced through the **default** clause of **defer** expressions.

In addition to the constraints in **defer**, the program can introduce 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:

```
if (x > 0) then assert (x = 42) else assert (x = −42).
```

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

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 context:

```
let result : int = concretize x with 42.
```

As we describe in the E-CONCRETIZESAT rule, concretization adds the constraint **context** $= 42$ to the constraint environment and finds an assignment to x consistent with the constraint and default environments. An important observation is that the context itself may be symbolic, in which case the system will also be finding a concrete context consistent with the hard constraints. The MODEL function takes the constraint and default environments, computes a satisfying assignment to free variables, and produces a substitution $\mathcal{M} : \upsilon \to c$ that is used to produce a concrete value, as shown in the E-CONCRETIZESAT rule.

The CONCRETIZE-UNSAT rule 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.

3.2.3 Interaction with the Constraint Environment

Valid constraint expressions consist of 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

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

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

$$\frac{}{\mathcal{G} \vdash \langle \Sigma, \Delta, \lambda x.e\ \upsilon \rangle \to \langle \Sigma, \Delta, e[x \mapsto \upsilon] \rangle}\ \textsc{E-AppLambda} \qquad \frac{c' = c_1\ (\text{op})\ c_2}{\mathcal{G} \vdash \langle \Sigma, \Delta, c_1\ (\text{op})\ c_2 \rangle \to \langle \Sigma, \Delta, c' \rangle}\ \textsc{E-Op}$$

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

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

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

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

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

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

$$\frac{\text{fresh}\ x'}{\mathcal{G} \vdash \langle \Sigma, \Delta, \textbf{defer}\ x : \tau\ \{\upsilon_c\}\ \textbf{default}\ \upsilon_d \rangle \to \langle \Sigma \cup \{\mathcal{G} \Rightarrow \upsilon_c[x \mapsto x']\}, \Delta \cup \{\mathcal{G} \Rightarrow x' = \upsilon_d\}, x' \rangle}\ \textsc{E-Defer}$$

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

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

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

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

Figure 3: Dynamic semantics for λ_J.

MODEL procedure in the E-CONCRETIZE rule is 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

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

The 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

The λ_J static semantics ensures that evaluation produces either a concrete expression or a well-formed symbolic expression. Recall that symbolic expressions must be valid constraints, which include arithmetic, boolean, and conditional expressions but not functions. (In the λ_J semantics we do not explicitly address data structures such as lists. Data structures are also required to be concrete but may have symbolic elements.) Thus the static semantics guarantee that 1) concrete values are supplied when concrete values are expected, 2) symbolic values are well-formed, 3) evaluation under symbolic conditions does not cause unexpected infinite recursion, and 4) context values have the appropriate types. The type system therefore ensures that the logical state will always be well formed, although it cannot guarantee that it will be logically consistent.

δ	$::=$	**concrete** $\mid$ **sym**	determinism tag
β	$::=$	$\textbf{int}_c \mid \textbf{bool}_c \mid \textbf{unit}$	base type
	$\mid$	**int** $\mid$ **bool**	
τ	$::=$	$\beta \mid \tau_1 \overset{nr}{\to} \tau_2 \mid \tau_1 \to \tau_2$	type

Figure 4: λ_J types.

To guarantee properties 1-3, the λ_J type system tracks the flow of symbolic values, ruling out symbolic functions and *reentrant applications* under symbolic conditions. Reentrant applications are function applications that may recursively call the current function; we prevent such applications under symbolic conditions to prevent non-termination that may arise from evaluating both sides of a conditional branch with a symbolic condition. Property 4, on the other hand, is enforced by ensuring that the type of the context used at concretization is an upper bound for the types of contexts required by all the relevant policies.

We show the λ_J types in Figure 4 and the subtyping, type well-formedness, and typing rules in Figure 5. Base types β are the standard λ-calculus types extended with the $\textbf{int}_c$ and $\textbf{bool}_c$ types to indicate values that are necessarily concrete. (Expressions of function type are not permitted to be symbolic.) There are two function types: $\overset{nr}{\to}$ for functions whose application cannot be reentrant and $\to$ for functions whose application can be reentrant. We will use the term *reentrant function* to refer to a function whose application can be reentrant.

Typing judgments have the form $\Gamma;\gamma \vdash e : \tau$. A judgment says that in the type environment Γ under a path of type γ (**sym** or **concrete**), the expression e has type τ. Γ is defined $\Gamma ::= \cdot \mid x : \tau \mid \Gamma, \Gamma'$. The typing rules keep track of whether a value may be symbolic (**int** or **bool** type) or must be concrete ($\mathbf{int}_c$, $\mathbf{bool}_c$, and functions). This information is used to determine the value of the γ tag in the T-CONDC and T-CONDSYM rules. Information about whether the condition is symbolic is used in 1) ruling out symbolic functions and 2) ruling out self-calls under symbolic branches.

Symbolic functions are prevented by the T-DEFER and T-CONDSYM rules, which restrict the production of symbolic functions. The T-DEFER rule restricts the explicit introduction of symbolic values to have base type β, while the T-CONDSYM rule restricts the implicit introduction of symbolic values to base type β.

The T-LETREC rule shows that recursive functions must be considered reentrant (have $\to$ type) within their own definitions, since applying the function will cause a recursive call to the current function. Outside their declaration, on the other hand, they can have $\overset{nr}{\to}$ type. A second restriction on reentrant functions is imposed by the type well-formedness predicate **rep**, which requires that functions taking arguments that may be reentrant ($\to$) be themselves labeled as reentrant. This prevents high-order functions from being used to circumvent the restrictions on reentrant calls.

According to the rules, a reentrant call cannot occur under a symbolic condition. The T-CONDSYM rule sets $\gamma = \textbf{sym}$ when the condition is symbolic. The T-APPCURREC rule allows applications of recursive functions only under concrete paths. This implies that canonical recursive functions such as factorial can only type-check if they require a concrete argument. This restriction does not prevent recursive sort or other recursive structure-traversing functions because data structures are necessarily concrete, so conditions involving their structure are also concrete.

The subtyping relationship <: allows values that are necessarily concrete to be used as potentially symbolic values. This way, functions that require concrete values can only be applied when concrete arguments are supplied, but a concrete value can be used as a symbolic value (for instance, $\mathbf{int}_c$ as **int**). The subtyping rules allow non-reentrant functions ($\overset{nr}{\to}$) to be used as reentrant functions ($\to$).

3.3.1 Contexts

We also have typing rules (not shown in Figure 5) ensuring that contexts of the appropriate type are provided in **concretize** expressions. In the T-CONCRETIZE rule, the context typing judgment $\vdash^c$ enforces that the context type supplied is the context type expected. The context typing judgement is $\Gamma \vdash^c x : \tau_c$, where τ_c is the *context* type of an expression. The rules propagate the context type, enforce that matching contexts are provided for sub-expressions, and enforce that the correct context type is supplied at concretization.

We define a lattice describing when different context types may be combined. The bottom of the lattice is $\perp$ and for all types τ, we have the relationship $\perp <:_c \tau$. Contexts support width subtyping on record types:

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

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

3.4 Translation from Jeeves

There is a straightforward translation from Jeeves to λ_J. 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. We have the $Exp \hookrightarrow e$ rule to describe how a Jeeves expression translates to a λ_J expression e. The translation has the following properties: 1) level variables are the only logic variables, 2) expressions containing sensitive values yield symbolic results, 3) only Jeeves policies introduce assertions, and 4) the **concretize** construct can only appear at the outermost level 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 sensitive values are symbolic, all expressions computed from this sensitive value are subject to policies depending on the value of level variable a.

3.4.2 Level Variables

Jeeves level variables are translated to λ_J expressions binding a new logic variable of **level** type equal to either $\perp$ or $\top$. The default value of level variables is $\top$: the constraint solving oracle first resolves the constraint environment with the assumption that each level is $\top$ and only adjusts this belief if the variable must be equal to $\perp$. This provides the programmer with some guarantees about program behavior when level variables are underconstrained. Underconstraint can arise, for instance, if values in the context depend on sensitive values.

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)
```

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

$$\begin{array}{ll} \textbf{true} & \\ \therefore (1)\ \mathsf{b} = \top \text{ (line 3)} & \therefore (2)\ \mathsf{x} \neq 1 \text{ (line 4)} \\ \therefore (3)\ \mathsf{x} = 0 \text{ (line 1)} & \therefore (4)\ \mathsf{a} = \perp \text{ (line 1)} \end{array}$$

Making underconstrained level variables $\top$ by default forces programmers to explicitly introduce policies setting level variables to $\perp$. For this reason, underspecification will only cause level variables to be set to $\perp$ instead of $\top$.

3.4.3 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 $\vdash \langle \emptyset, \emptyset, e \rangle \to^* \langle \Sigma', \Delta', \upsilon \rangle$. This policy contains constraints determining whether level variables can be $\perp$ or $\top$.

$$\boxed{\tau_1 <: \tau_2}$$

$$\frac{}{\tau <: \tau}\ \textsc{S-Reflexive} \qquad \frac{}{\mathbf{int}_c <: \mathbf{int}}\ \textsc{S-Int} \qquad \frac{}{\mathbf{bool}_c <: \mathbf{bool}}\ \textsc{S-Bool} \qquad \frac{}{\tau_1 \xrightarrow{nr} \tau_2 <: \tau_1 \to \tau_2}\ \textsc{S-RecFun} \qquad \frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ \textsc{S-Fun}$$

$$\boxed{\mathbf{rep}\ \tau}$$

$$\frac{\mathbf{rep}\ \tau' \quad \tau' <: \tau}{\mathbf{rep}\ \tau}\ \textsc{Ok-SubType} \qquad \frac{}{\mathbf{rep}\ \beta}\ \textsc{Ok-BaseType} \qquad \frac{\mathbf{rep}\ \tau_2}{\mathbf{rep}\ \beta_1 \to \tau_2}\ \textsc{Ok-BaseFunction}$$

$$\frac{\mathbf{rep}\ (\tau_1 \xrightarrow{nr} \tau') \quad \mathbf{rep}\ \tau_2}{\mathbf{rep}\ (\tau_1 \xrightarrow{nr} \tau') \xrightarrow{nr} \tau_2}\ \textsc{Ok-HOFunction} \qquad \frac{\mathbf{rep}\ \tau_1 \to \tau_2 \quad \mathbf{rep}\ \tau'}{\mathbf{rep}\ (\tau_1 \to \tau_2) \to \tau'}\ \textsc{Ok-RecFunction}$$

$$\boxed{\Gamma;\gamma \vdash e : \langle \tau, \delta \rangle}$$

$$\frac{x \in \Gamma}{\Gamma;\gamma \vdash x : \Gamma(x)}\ \textsc{T-Var} \qquad \frac{}{\Gamma;\gamma \vdash n : \mathbf{int}_c}\ \textsc{T-Int} \qquad \frac{}{\Gamma;\gamma \vdash b : \mathbf{bool}_c}\ \textsc{T-Bool} \qquad \frac{}{\Gamma;\gamma \vdash () : \mathbf{unit}}\ \textsc{T-Unit} \qquad \frac{\mathbf{rep}\ \tau}{\Gamma;\gamma \vdash \mathbf{context}\ \tau : \tau}\ \textsc{T-Context}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \tau_1 \quad \Gamma;\gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 <: \tau \quad \mathbf{rep}\ \tau}{\Gamma;\gamma \vdash e_1\ (\mathrm{op})\ e_2 : \tau}\ \textsc{T-Op} \qquad \frac{\Gamma;\gamma \vdash e : \mathbf{bool}_c \quad \Gamma;\gamma \vdash e_t : \tau_1 \quad \Gamma;\gamma \vdash e_f : \tau_2 \quad \tau_1, \tau_2 <: \tau \quad \mathbf{rep}\ \tau}{\Gamma;\gamma \vdash\ \mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \tau}\ \textsc{T-CondC}$$

$$\frac{\Gamma;\gamma \vdash e : \mathbf{bool} \quad \Gamma;\mathbf{sym} \vdash e_t : \beta_1 \quad \Gamma;\mathbf{sym} \vdash e_f : \beta_2 \quad \beta_1, \beta_2 <: \beta_c \quad \mathbf{rep}\ \beta_c}{\Gamma;\gamma \vdash\ \mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f : \beta_c}\ \textsc{T-CondSym}$$

$$\frac{\Gamma, x : \tau_d;\gamma \vdash e : \tau' \quad \mathbf{rep}\ \tau \quad \mathbf{rep}\ \tau'}{\Gamma;\gamma \vdash (\lambda x : \tau_d . e) : \tau_d \to \tau'}\ \textsc{T-Lambda} \qquad \frac{\Gamma;\gamma \vdash e_1 : \tau_1 \xrightarrow{nr} \tau_2 \quad \Gamma;\gamma \vdash e_2 : \tau_1' \quad \tau_1' <: \tau_1 \quad \mathbf{rep}\ \tau_1 \quad \mathbf{rep}\ \tau_2}{\Gamma;\gamma \vdash (e_1\ e_2) : \tau_2}\ \textsc{T-App}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2;\gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma, f : \tau_1 \xrightarrow{nr} \tau_2;\gamma \vdash e_2 : \tau_2 \quad \mathbf{rep}\ \tau \quad \mathbf{rep}\ \tau_2}{\Gamma;\gamma \vdash\ \mathbf{let\ rec}\ f : \tau_1 \xrightarrow{nr} \tau_2 = e_1\ \mathbf{in}\ e_2 : \tau_2}\ \textsc{T-LetRec}$$

$$\frac{\gamma = \mathbf{concrete} \quad \Gamma;\gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma;\gamma \vdash e_2 : \tau_1' \quad \tau_1' <: \tau_1 \quad \mathbf{rep}\ \tau_1 \quad \mathbf{rep}\ \tau_2}{\Gamma;\gamma \vdash (e_1\ e_2) : \tau_2}\ \textsc{T-AppCurRec}$$

$$\frac{\Gamma, x : \beta;\gamma \vdash e_c : \mathbf{bool} \quad \Gamma;\gamma \vdash \upsilon : \beta}{\Gamma;\gamma \vdash (\mathbf{defer}\ x : \beta\{\ e_c\}\ \mathbf{default}\ \upsilon) : \beta}\ \textsc{T-Defer} \qquad \frac{\Gamma;\gamma \vdash e_c : \mathbf{bool}}{\Gamma;\gamma \vdash (\mathbf{assert}\ e_c) : \mathbf{unit}}\ \textsc{T-Assert}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \beta \quad \Gamma;\gamma \vdash^c e_1 : \beta' \quad \Gamma;\gamma \vdash \upsilon : \beta'}{\Gamma;\gamma \vdash (\mathbf{concretize}\ e_1\ \mathbf{with}\ \upsilon) : \beta_c}\ \textsc{T-Concretize}$$

Figure 5: Static semantics for λ_J describing simple type-checking and enforcing restrictions on scope of nondeterminism and recursion. Recall that β refers to base (non-function) types.

$$\frac{}{\bot \hookrightarrow \mathbf{false}} \qquad \frac{}{\top \hookrightarrow \mathbf{true}} \qquad \frac{Exp_l \hookrightarrow e_l \quad Exp_h \hookrightarrow e_h}{\langle Exp_l \mid Exp_h \rangle(\ell) \hookrightarrow\ \mathbf{if}\ \ell\ \mathbf{then}\ e_h\ \mathbf{else}\ e_l}\ \textsc{Tr-SValue}$$

$$\frac{Exp \hookrightarrow e}{\mathbf{level}\ \ell\ \mathbf{in}\ Exp \hookrightarrow \mathbf{let}\ \ell = \mathbf{defer}\ \ell' : \mathbf{bool\ default\ true\ in}\ e}\ \textsc{Tr-Level}$$

$$\frac{Exp_p \hookrightarrow e_p \quad Exp \hookrightarrow e \quad Lvl \hookrightarrow b}{\mathbf{policy}\ \ell :\ Exp_p\ \mathbf{then}\ Lvl\ \mathbf{in}\ Exp \hookrightarrow \mathbf{assert}\ (e_p \Rightarrow (\ell = b))\ \mathbf{in}\ e}\ \textsc{Tr-Policy}$$

$$\frac{Exp_c \hookrightarrow e_c \quad Exp \hookrightarrow e}{\mathbf{print}\ \{Exp_c\}\ Exp \hookrightarrow \mathbf{print}\ (\mathbf{concretize}\ e\ \mathbf{with}\ e_c)}\ \textsc{Tr-Print}$$

.

Figure 6: Translation from Jeeves to λ_J

3.4.4 Contextual Enforcement at Output Channels

Effectful computations such as **print** in Jeeves require contexts corresponding to the viewer to whom the result is displayed. As we show by 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.

4. Properties

We describe more formally the guarantees that Jeeves provides. We prove progress and preservation properties for λ_J. We show that the only way the value for the high component of a sensitive value to affect the output of the computation is if the policies permit it.

4.1 Progress and Preservation

We first 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 υ 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 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 υ is a value of type $\tau_1 \to \tau_2$, then $\upsilon = \lambda x : \tau_1.e$, where e has type τ_2.*

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

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 $\Gamma \vdash e : \tau\delta$ and $e \to e'$, then $\Gamma \vdash e' : \tau\delta$.*

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 Confidentiality Theorem

We show that level variables enforce the confidentiality of values: once the policy sets a level variable $\ell = \bot$, where we have some $\langle Exp_l | Exp_h\rangle_\ell$, the output will be derived as if Exp_h was not involved in evaluation at all. Because we have $\ell = \bot$ if and only if we have policies that require $\ell = \bot$, Jeeves programmers can rely on policies to enforce confidentiality.

We first prove that the high-confidentiality views of the sensitive values are protected by level variables. We can show that for a sensitive value $v = \langle Exp_l | Exp_h\rangle_\ell$, the only way the value for the high component Exp_h may affect the output of the computation is when $\ell = \top$ is consistent with the policies. It is impossible for an observer to distinguish between $v = \langle Exp_l | Exp_h\rangle$ and $v' = \langle Exp_l | Exp'_h\rangle$ if the policy requires $\ell = \bot$.

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

$$E[H \mapsto E_h] \hookrightarrow e \qquad \vdash \langle \emptyset, \emptyset, e\rangle \to^* \langle \Sigma, \Delta, \sigma\rangle$$
$$E[H \mapsto E'_h] \hookrightarrow e' \qquad \vdash \langle \emptyset, \emptyset, e'\rangle \to^* \langle \Sigma', \Delta', \sigma'\rangle$$

For any context value v, if

$$\Sigma \cup \{\textbf{\textit{context}} = v\} \vdash \ell = \bot$$
$$\Sigma' \cup \{\textbf{\textit{context}} = v\} \vdash \ell = \bot$$

then

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

Proof. From the rules of Jeeves translation, V maps to an irreducible symbolic expression **if** ℓ **then** e_h**else** e_l in e where $E_l \hookrightarrow e_l$ and $E_h \hookrightarrow e_h$. Thus, we can say that e' is e with the expression e_h replaced by e'_h where $E'_h \hookrightarrow e'_h$. In addition, we also know that both e and e' are λ_J expressions with no **concretize** sub-expressions. This makes evaluation of e and e' deterministic and allows us to put their derivation trees in correspondence. There are two places where evaluation differs: (1) reduction of e_h and e'_h (rule E-CONDSYMT) and (2) substitution of the reduced sensitive value (rule E-APPLAMBDA.) Let us understand how they affect the logical environment and the resulting symbolic value.

The values σ and σ' may differ only in the subexpressions e_h and e'_h reduce to. These subexpressions are guarded by the level variable ℓ. Since the logical environments entail that $\ell = \bot$ under context v, any model chosen at concretization sets ℓ to $\bot$. Therefore, under such models σ and σ' evaluate to the same value. Then to show that the set of concrete values is the same, it suffices to show the models of (Δ, Σ) are models of (Δ', Σ') and vice versa.

The dynamic semantics populates Σ and Σ' with the same constraints (modulo substitution of the sensitive variable) except during reduction of e_h and e'_h. The constraints added at rule E-CONDSYMT are all guarded by the level variable ℓ. Since Σ and Σ' both entail $\neg\ell$, and these guarded constraints are implied by $\neg\ell$, we can safely eliminate them from both Σ and Σ'. That leaves us with the same set of hard constraints introduced through **defer** and **assert** expressions.

The default judgements in Jeeves all have the form $\ell_0 =$ **true**. Therefore, for the shared level variables Δ and Δ' use the same default value **true**. The remaining level variables do not affect evaluation of σ and σ'. □

Our non-interference theorem allows programmers rely on policies to enforce confidentiality. In Jeeves, policies have the form $\phi \Rightarrow (\ell = \top)$ or $\phi \Rightarrow (\ell = \bot)$. By the theorem, once the policy setting ℓ to $\bot$ is guaranteed to be added to the constraint environment, the output is going to be the same as if the high view component of the sensitive value was not involved in evaluation at all. If policies permit both $\bot$ and $\top$ levels, then the default logic model finder will guide evaluation to a model maximizing levels set to $\top$.

Note that if policies are contradictory and the set of constraints is unsatisfiable, the evaluation halts with an error and no value is exposed. The theorem still holds and this behavior is safe.

5. Scala Embedding

We have implemented Jeeves as an embedded domain-specific language in Scala programming language [20]. 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 to implement 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.[1]

5.1 ScalaSMT: Scala Embedding of λ_J

Every kind of symbolic expression in λ_J has 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 ones but requires type annotations where a symbolic expression is expected to be used.

The three core language extensions **defer**, **assert**, and **concretize** are implemented as library calls. We implement the library as a Scala trait that maintains the logical and default constraint environments as lists of symbolic boolean expressions. Calls to **concretize** invoke an off-the-shelf SMT solver [17] for the satisfiability query Model. We translate λ_J constraints to the 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, the Scala embedding supports symbolic expressions for objects corresponding to λ_J records with equality theory. Objects are modeled as a finite algebraic data type in Z3 [17]. The set of available objects is maintained by ScalaSMT using registration of instances of a special trait Atom. Object fields are modeled as total functions interpreted at the time of concretization. Fields are (sort-)typed with values that are arbitrary ScalaSMT expressions and constants. ScalaSMT does not check types of symbolic object expressions: we rely on Scala's support for dynamic invocaton to resolve field dereferences. We use special zero values (null , 0, or false) to represent undefined fields in SMT.

ScalaSMT does not support symbolic collections. Instead, we use implicit type conversions to extend the 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. The boolean condition is passed by name to delay its evaluation until concretization. This way policies that refer to mutable parts of the heap will produce correct constraints for the snapshot of the system at concretization.

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

  def mkLevel(): LevelVar
  def policy ( lvar : LevelVar, f: ⇒ Formula, l: Level)

  def mkSensitiveInt ( lvar : LevelVar,
    high: IntExpr, low: IntExpr): IntExpr
  def mkSensitive( lvar : LevelVar,
    high: Symbolic, low: Symbolic): Symbolic

  def concretize [T](ctx: Symbolic, e: Expr[T]): T
}
```

Figure 7: Jeeves library in Scala

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.

6. Experience

We have implemented a conference management system and a social network. Our experience suggests that Jeeves allows the programmer to separate the "core," non-privacy-related functionality from the privacy policies, allowing the programmer to separately test policies and functionality.

6.1 Conference Management System

We have implemented a simple conference management system backend, JConf, to demonstrate how a well-known system with privacy concerns looks in Jeeves. This system is similar to the example we described in Section 2. Our implementation demonstrates that Jeeves allows us to implement all JConf functionality, including search and display over final paper versions, with a core functionality that is separate from the policies.

JConf supports the following subset of the functionality mentioned on the website for the HotCRP conference management system [12]: smart paper search (by ID, by reviewer, etc.), paper tagging (for instance, "Accepted" and "Reviewed by: . . . ") and search by tags, managing reviews (assigning, collecting responses, and displaying), and managing 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:

- *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.

[1] The code is publicly available at http://code.google.com/p/scalasmt/.

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:

- *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 *p* before they can see *p*'s reviews.

Our JConf implementation allows us to separate the declaration of policies and code: we show the breakdown of code and policies in Table 1. The policies are concentrated in the data classes PaperRecord.scala and PaperReview.scala, which describe the attributes and policies associated with maintaining data associate with papers and paper reviews. The other files, including JConfBackend.scala, do not contain policies. This allows the core functionality to be concise: the implementation of our back-end functionality as specified 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 implement policies specified 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 policy updates. 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. The programmer does not have to make coordinated changes across the code base to update policies.

6.2 Social Network

For social networks it is important to rapidly develop code that implements information flow policies. Privacy issues have put the social network website Facebook under the scrutiny of the American Federal Trade Commission [26], making it crucial that they do not leak sensitive data. On the other side, one of Facebook's key values is to "move fast:" rapidly develop innovative features [28]. Separation of policies and core program functionality can help developers rapidly develop privacy-aware features.

To investigate this hypothesis, we have implemented Jeeves Social Net, a toy social network that uses Jeeves policies to control confidentiality of user-shared data. Jeeves Social Net core functionality involves storing and allowing queries over 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 which versions of these attributes 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 are stateful: for instance, a policy on the visibility of user *u*'s location refers 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. 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 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:

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

We use this function 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. 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. The location mash-up website PleaseRobMe [3] demonstrates that if disclosure of geographic location 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.

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 $\top$ 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 [27] that do not use symbolic constraints.

6.3 Jeeves Limitations

Jeeves currently provides only a limited amount of static checking. The implementation of Jeeves as a domain-specific embedded library in Scala relies on Scala type-checking to enforce static properties. At present, Jeeves does not provide static feedback about more complex program properties. For instance, neither the Jeeves design nor the implementation provide support for statically determining whether policies are consistent or total. We anticipate being able to detect properties such as underspecification and inconsistency using enhanced static analysis that we can implement as a Scala compiler extension.

There are many open questions regarding the usability of Jeeves. Symbolic evaluation and SMT are technologies that have been improving in performance, but it is not clear they can handle the demands of real-world applications. One direction for future exploration includes scalability of Jeeves programs, how to efficiently handle data persistence, and development of lighter-weight execution models. Another direction for exploration involves the ease of programming and testability of Jeeves programs.

7. Related Work

Jeeves privacy policies yield comparable expressiveness to state-of-the-art languages for verifying system security such as Jif [19], Fine [4], and Ur/Web [5]. These are static approaches that have no dynamic overhead. Rather than providing support for verifying properties, the Jeeves execution model handles policy enforcement, guaranteeing that programs adhere to the desired properties by construction, but with dynamic overhead.

The Jeeves runtime is similar to the system-level data flow framework Resin [27], which allows the programmer to insert checking code to be executed at output channels. Jeeves's declarative policies allow the programmer to specify policies at a higher level and allow automatic handling of dependencies between policies.

There are also parallels with dynamic approaches to security. Devriese and Piessens's secure multi-execution approach executes multiple copies of the program, providing defaults to copies that should not get access to secret inputs [7]. Jeeves's symbolic evaluation obviates the need to execute multiple program copies and Jeeves allows more complex policies, for instance ones that may depend on sensitive values. In this space is also Kashyap *et al.*'s scheduling-based dynamic approach, which partitions a program into sub-programs for each security level for ensuring timing and termination non-interference. The focus of this is different from our work, which does not address timing or termination.

Jeeves can also be compared to aspect-oriented programming (AOP) [11]. 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 [24] and method for generating aspect code for behavior such as error logging automatically [23]. 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 allows policies to affect control flow decisions.

The way Jeeves handles privacy is inspired by angelic nondeterminism [8]. Jeeves most directly borrows from CFLP-L, a constraint functional programming calculus presented by Mück *et al.* [18]; similar functional logic models have also been implemented in languages such as Mercury [25], Escher [15], and Curry [9, 10]. Our system differs in the restrictions we place on nondeterminism and the execution model. λ_J leaves functions and the theory of lists out of the logical 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 [16, 21] in our goal of propagating nondeterminism alongside the core program rather than executing isolated nondeterministic sub-procedures. Program repair approaches such as Demsky's data structure repair [6], the Plan B [22] system for dynamic contract checking, and Kuncak *et al.*'s synthesis approach [13] also target local program expressions.

8. Conclusions

Our main contribution is a programming model that allows programmers to separate privacy concerns from core program functionality. We present the Jeeves programming language, formally define the underlying constraint functional language λ_J, and prove that Jeeves executions satisfy a non-interference property between low and high components of sensitive values. We describe our implementation as an embedded domain-specific language in Scala. We also describe our case studies, which illustrate how the programmer can separately develop core functionality and privacy policies while relying on the system to produce outputs consistent with the policies.

Acknowledgments

We would like to thank Saman Amarasinghe, Arvind, Michael Carbin, Gregory Malecha, Sasa Misailovic, Andrew Myers, Joseph Near, Martin Rinard, and Joe Zimmerman for their input and feedback.

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] D. Devriese and F. Piessens. Noninterference through secure multi-execution. *Security and Privacy, IEEE Symposium on*, 0:109–124, 2010. ISSN 1081-6011. doi: http://doi.ieeecomputersociety.org/10.1109/SP.2010.15.

[8] 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`.

[9] 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.

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

[11] 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.

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

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

[14] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 108–118, New York, NY, USA, 2000. ACM. ISBN 1-58113-125-9. doi: http://doi.acm.org/10.1145/325694.325708. URL `http://doi.acm.org/10.1145/325694.325708`.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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