Types and Programming Language Project Report*

Jiajun Jiang
jiajun.jiang@pku.edu.cn

Mengwei Xu
xumw@pku.edu.cn

Introduction

In current Internet environment, users tend to share more personal data online. This phenomenon makes it increasingly important for applications to protect confidentiality. For existing approaches to achieve privacy control, programmers are forced to ensure compliance by their own efforts, even when both the application and the policies may be evolving rapidly. This can cause considerable burdens to application developers.

This academic paper, called *A Language for Automatically Enforcing Privacy Policies.*[1], proposes 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, therefore solves the problem mentioned above. Furthermore, they have implemented this programming model in a new functional constraint language named **Jeeves**.

We carried out our course project based on this paper. More specifically, we first read and comprehended this paper, including the language design, the semantics, the evaluation and typing rules, the partial property proof and the implementation details as we did in our class. Then we proved the progress and preservation parts which is left out in the paper by ourselves. Finally, we successfully run the implementation codes provided by the authors and wrote our own use cases to understand the implementation details.

1 Language Design

1.1 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. Figure 1 shows the jeeves syntax, which looks like a high-level language since it's based on and translated from λ_J described in next subsection.

Several key words in this syntax tell what Jeeves wants to do and what it is capable of doing.

- **Level** provides variables 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).

*done at June 5th, 2016

[1]http://www.cs.cmu.edu/ jyang2/papers/popl088-yang.pdf

- Policies, introduced through **policy** expressions, provide declarative rules describing when to set a level variable to top or bottom.
- **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.

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

Figure 1: Jeeves syntax

1.2 Lambda J

One interesting thing is the authors don't formally implement Jeeves from the scratch. Instead, they introduce λ_J, a simple constraint functional language based on the λ-calculus, and then they show how to translate Jeeves from λ_J. Here we describe the λ_J language show in Figure 2.

Basically, The λ_J language extends the λ-calculus with logical variables. 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 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.

2 Semantics

Here is the semantics...

$$
\begin{array}{rcl}
c & ::= & n \mid b \mid \lambda x : \tau.e \mid \mathit{record}\ \bar{x{:}v} \\
& \mid & \textbf{error} \mid () \\
\sigma & ::= & x \mid \textbf{contex}\ \tau \\
& \mid & c_1\ (op)\ \sigma_2 \mid \sigma_1\ (op)\ c_2 \\
& \mid & \sigma_1\ (op)\ \sigma_2 \\
& \mid & \textbf{if}\ \sigma\ \textbf{then}\ v_t\ \textbf{else}\ v_f \\
v & ::= & c \mid \sigma \\
e & ::= & v \mid e_1\ (op)\ e_2 \\
& \mid & \textbf{if}\ e_1\ \textbf{then}\ e_t\ \textbf{else}\ e_f \mid e_1\ e_2 \\
& \mid & \textbf{let}\ x : \tau = e_1\ \textbf{in}\ e_2 \\
& \mid & \textbf{let rec}\ f : \tau = e_1\ \textbf{in}\ e_2 \\
& \mid & \textbf{defer}\ x : \tau\{e\}\ \textbf{defaut}\ v_d \\
& \mid & \textbf{assert}\ e \\
& \mid & \textbf{concretize}\ e\ \textbf{with}\ v_c
\end{array}
$$

Figure 2: λ_J syntax

3 Properties

Lemma 1. (ConcreteFunction). *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.*

Proof. According to the λ_J syntax, we can get Lemma 1 immediately. □

Theorem 1. (Progress). *Suppose e is a closed, well-typed expression. Then e is either a value v or there is some e' such that $\vdash \langle \phi, \phi, e \rangle \to \langle \Sigma', \Delta', e' \rangle$.*

Proof. we can ... □

Theorem 2. (Preservision). *If $\Gamma \vdash e : \tau\delta$ and $e \to e'$, then $\Gamma \vdash e' : \tau\delta$.*

Proof. we can ... □

4 Evaluation

Here is the example area.

```
data Exp = E_BOOL Bool | E_NAT Int
         | E_STR String | E_CONST String
         | E_VAR Var    | E_CONTEXT
         | E_LAMBDA Var Exp | E_THUNK Exp
         | E_OP Op Exp Exp  | E_UOP UOp Exp
         | E_IF Exp Exp Exp | E_APP Exp Exp
         | E_DEFER Var Exp | E_ASSERT Exp Exp
         | E_LET Var Exp Exp
         | E_RECORD [(FieldName,Exp)]
         | E_FIELD Exp FieldName
```

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

$$\frac{\mathscr{G}\vdash\langle\Sigma,\Delta,e_1\rangle\to\langle\Sigma',\Delta',e_1'\rangle}{\mathscr{G}\vdash\langle\Sigma,\Delta,e_1\ e_2\rangle\to\langle\Sigma',\Delta',e_1'\ e_2\rangle}\ \text{E-APP1} \qquad \frac{\mathscr{G}\vdash\langle\Sigma,\Delta,e_2\rangle\to\langle\Sigma',\Delta',e_2'\rangle}{\mathscr{G}\vdash\langle\Sigma,\Delta,v\ e_2\rangle\to\langle\Sigma',\Delta',v\ e_2\rangle}\ \text{E-APP2}$$

$$\frac{}{\mathscr{G}\vdash\langle\Sigma,\Delta,\lambda x.e\ v\rangle\to\langle\Sigma',\Delta',e[x\mapsto v]\rangle}\ \text{E-APPLAMBDA} \qquad \frac{c'=c_1\ (op)\ c_2}{\mathscr{G}\vdash\langle\Sigma,\Delta,c_1\ (op)\ c_2\rangle\to\langle\Sigma',\Delta',c'\rangle}\ \text{E-OP}$$

$$\frac{\mathscr{G}\vdash\langle\Sigma,\Delta,e_1\rangle\to\langle\Sigma',\Delta',e_1'\rangle}{\mathscr{G}\vdash\langle\Sigma,\Delta,e_1\ (op)\ e_2\rangle\to\langle\Sigma',\Delta',e_1'\ (op)\ e_2\rangle}\ \text{E-OP1} \qquad \frac{\mathscr{G}\vdash\langle\Sigma,\Delta,e_2\rangle\to\langle\Sigma',\Delta',e_2'\rangle}{\mathscr{G}\vdash\langle\Sigma,\Delta,v\ (op)\ e_2\rangle\to\langle\Sigma',\Delta',v\ (op)\ e_2'\rangle}\ \text{E-OP2}$$

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

$$\frac{\mathscr{G}\vdash\langle\Sigma,\Delta,e_t\rangle\to\langle\Sigma',\Delta',e_t'\rangle}{\mathscr{G}\vdash\langle\Sigma,\Delta,\textbf{if true then}\ e_t\ \textbf{else}\ e_f\rangle\to\langle\Sigma',\Delta',e_t'\rangle}\ \text{E-CONDTRUE}$$

$$\frac{\mathscr{G}\vdash\langle\Sigma,\Delta,e_f\rangle\to\langle\Sigma',\Delta',e_f'\rangle}{\mathscr{G}\vdash\langle\Sigma,\Delta,\textbf{if false then}\ e_t\ \textbf{else}\ e_f\rangle\to\langle\Sigma',\Delta',e_f'\rangle}\ \text{E-CONDFALSE}$$

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

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

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

$$\frac{fresh\ x'}{\mathscr{G}\vdash\langle\Sigma,\Delta,\textbf{defer}\ x{:}\tau\{e\}\ \textbf{default}\ v_d\rangle\to\langle\Sigma\cup\{\mathscr{G}\Rightarrow v_c[x\mapsto x']\},\Delta\cup\{\mathscr{G}\Rightarrow x'=v_d\},x'\rangle}\ \text{E-DEFER}$$

$$\frac{\mathscr{G}\vdash\langle\Sigma,\Delta,e\rangle\to\langle\Sigma',\Delta',e'\rangle}{\mathscr{G}\vdash\langle\Sigma,\Delta,\textbf{assert}\ e\rangle\to\langle\Sigma',\Delta',\textbf{assert}\ e'\rangle}\ \text{E-ASSERTCONSTRAINT}$$

$$\frac{}{\mathscr{G}\vdash\langle\Sigma,\Delta,\textbf{assert}\ v\rangle\to\langle\Sigma\cup\{\mathscr{G}\Rightarrow v\},\Delta,()\rangle}\ \text{E-ASSERT}$$

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

$$\frac{\text{MODLE}(\Delta,\Sigma\cup\{\mathscr{G}\cap\textbf{context}{=}v_c\})=\mathscr{M}\quad c=\mathscr{M}[[v_\nu]]}{\mathscr{G}\vdash\langle\Sigma,\Delta,\textbf{concretize}\ v_\nu\ \textbf{with}\ v_c\rangle\to\langle\Sigma,\Delta,c\rangle}\ \text{E-CONCRETIZESAT}$$

$$\frac{\text{MODLE}(\Delta,\Sigma\cup\{\mathscr{G}\cap\textbf{context}{=}v_c\})=\text{UNSAT}}{\mathscr{G}\vdash\langle\Sigma,\Delta,\textbf{concretize}\ v_\nu\ \textbf{with}\ v_c\rangle\to\langle\Sigma,\Delta,\textbf{error}\rangle}\ \text{E-CONCRETIZEUNSAT}$$

Figure 3: Dynamic semantics for λ_{J}.

$$\begin{array}{rcll} \delta & ::= & \textbf{concretize} \mid \textbf{sym} & \text{determinism tag} \\ \beta & ::= & \textbf{int}_c \mid \textbf{bool}_c \mid \textbf{unit} \mid \textbf{int} \mid \textbf{bool} & \text{base type} \\ \tau & ::= & \beta \mid \tau_1 \xrightarrow{nr} \tau_2 \mid \tau_1 \to \tau_2 & \text{type} \end{array}$$

Figure 4: λ_{J} types

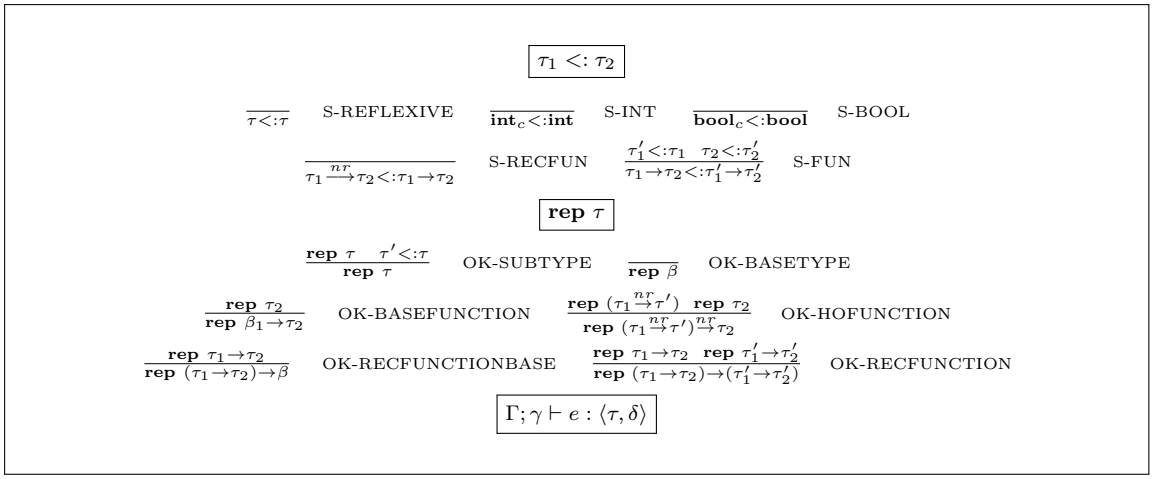

Figure 5: Static semantics for λ_J describing simple type-checking and enforce restriction on scope of nondeterminism and recursion. Recall that β refers to base (non-function) types.

```
                    deriving (Ord,Eq)

    data Op = OP_PLUS | OP_MINUS
                    | OP_LESS | OP_GREATER
                    | OP_EQ | OP_AND | OP_OR | OP_IMPLY
                    deriving (Ord,Eq)

    data UOp = OP_NOT   deriving (Ord,Eq)

    data FieldName = FIELD_NAME String  deriving(Ord,Eq)
    data Var = VAR String deriving (Ord,Eq)
```

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

let msg = "Author is " + name

print {alice} msg
print {bob} msg
------
```

5 Conclusion

Here is the conclusion section...

References

[1] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In Proceedings of the 39th annual ACM SIGPLAN-SIGACT

symposium on Principles of programming languages (POPL '12). ACM, New York, NY, USA, 85-96.

[2] B. Pierce. Types and Programming Languages, *MIT Press*, 2002.