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

The semantics are divided into two parts: dynamic and static. Dynamic semantic is the same as the evaluation rules in our class, while static semantic includes but not limited

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

Figure 2: λ_J syntax

to the typing rules. The static semantic also includes some rules to rule out the infinite recursive calling that may be caused by the logical variables introduced by λ_J.

As shown in Figure 3, 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 ($\mathscr{G}$). Here, Σ stores the current set of constraints, and an environment Δ stores the set of constraints on default values for logic variables. The rules are no surprising, quite similar to the rules learned in our class. One interesting thing is when evaluating under symbolic conditions, because evaluation of such conditionals with a recursive function application in a branch could lead to infinite recursion when the condition is symbolic. λ_J prevents this anomalous behavior by using the type system.

Static semantic, as shown in Figure 5, describes simple type-checking and enforce restriction on scope of nondeterminism and recursion. The types are defined in Figure 4. The largest difference of typing rules between the normal λ-calculus and λ_J is the symbolic values. For int, bool or if expression, it can have two options: concrete or symbolic. And therefore the issue comes when applying reentrant functions under symbolic conditionals. To rule out such kind of cases, λ_J explicitly marks values as concrete or symbolic. For example, a function that might be reentrant should be marked as ($\longrightarrow$). Furthermore, it introduces **rep**, defined as a predicate that requires that functions taking arguments that may be reentrant be themselves labeled as reentrant. This can help prevent high-order functions from being used to circumvent the restrictions on reentrant calls. Overall, the static semantics can guarantee that

- concrete values are supplied when concrete values are expected,
- symbolic values are well-formed,

- evaluation under symbolic conditions does not cause unexpected infinite recursion,
- context values have the appropriate types.

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 $\vdash \langle \Sigma, \Delta, e \rangle \to \langle \Sigma', \Delta', e' \rangle$.*

Proof. According to the dynamic demantics 3 and static semantics 5 of λ_J, we will proof that any λ_J program holds the *progress* property by structral induction over the syntax of λ_J. According to Figure 2, there are nine kinds of expressions including the value expression v. Next we discuss each of them respectively.

- For the value expression v, according to the definition we can conclude that it holds the property immediately.
- For the expression e_1 (op) e_2, according to the hyposthesis, we know that sub-expression e_1 and e_2 are both closed and well-typed expressions. Thus if e_1 is not a value, according to E-OP1, there must be a expression e_1' such that $e_1 \to e_1'$. Therefore the expression e_1 (op) e_2 can be reduced to e_1' (op) e_2, and the same as if the e_1 is already a value but e_2 is not according to E-OP2. While if both e_1 and e_2 are values, we know that e_1 (op) e_2 is a value as well. Let $c = e_1$ (op) e_2, according to E-OP, $\vdash \langle \Sigma, \Delta, e_1 \ (op) \ e_2 \rangle \to \langle \Sigma', \Delta', c \rangle$. Hence, we can conclude that expression e_1 (op) e_2 holds the *progress* property.
- For the expression **if** e_1 **then** e_t **else** e_f, the conditional expression e_1 is a either a **concrete** expression or a **symbolic** expression. If expression e_1 is **concrete** and not a value, according to E-COND, there must be an expression e_1' such that $\vdash \langle \Sigma, \Delta, \textbf{if } e_1 \textbf{ then } e_t \textbf{ else } e_f \rangle \to \langle \Sigma', \Delta', \textbf{if } e_1' \textbf{ then } e_t \textbf{ else } e_f \rangle$. Similarly, if expression e_1 is a **concrete** value, it must be **true** or **false**. According to E-CONDTRUE and E-CONDFALSE, **if** e_1 **then** e_t **else** e_f can be reduced as well. On the other hand, if the expression e_1 is **symbolic**, according to E-CONDSYMT and E-CONDSYMF, it holds the *progress* property.
- For the expression e_1 e_2, if the sub-expression e_1 or e_2 is not a value, similar as the expression e_1 (op) e_2, according to E-APP1 and E-APP2, it can be at least reduce one step and evetually both e_1 and e_2 are values. Then according to the previous **Concrete Function Lemma**, e_1 e_2 must be the form as $\lambda x : e.v$, according to E-APPLAMBDA, it holds the property immediately.
- For the expression **defer** $x : \tau\{e\}$ **default** v_d, if sub-expression e is not a value, by the induction hypothesis, there must be an expression e' such that $\vdash \langle \Sigma, \Delta, e \rangle \to \langle \Sigma', \Delta', e' \rangle$, then by E-DEFERCONSTRAINT, the expression can perform a reduction $\vdash \langle \Sigma, \Delta, \textbf{defer } x : \tau\{e\} \textbf{ default } v_d \rangle \to \langle \Sigma', \Delta', \textbf{defer } x : \tau\{e'\} \textbf{ default } v_d \rangle$. In addition, if sub-expression e is a value, according to static semantics, it must be some concrete value v_c, then according to E-DEFER, a fresh variable named x' will be generated and a new defaut condition will be added to the Δ environment, which indicates that the expression **defer** is progressive.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

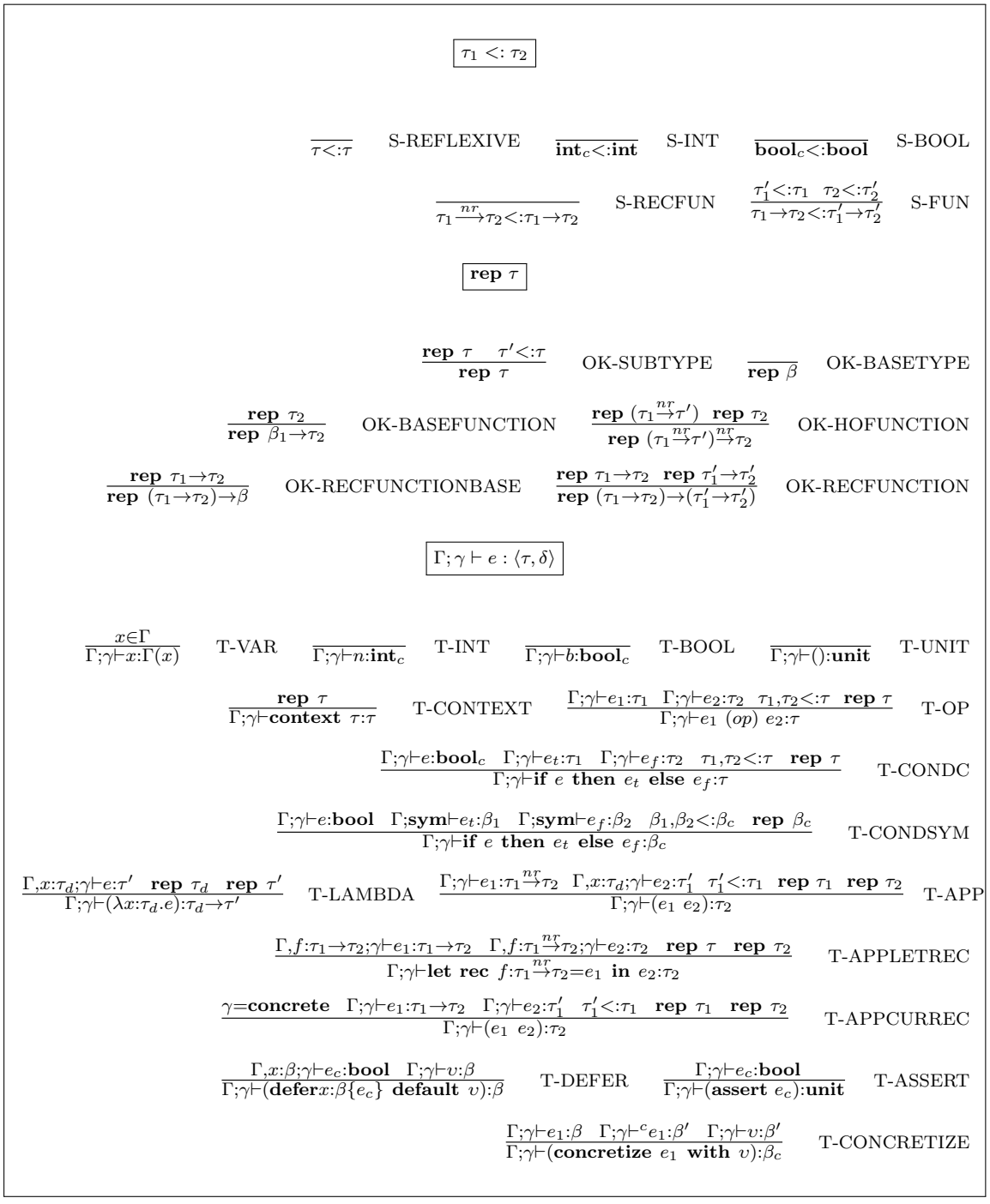

Figure 5: Static semantics for λ_J describing simple type-checking and enforce restriction on scope of nondeterminism and recursion. Recall that β refers to base (non-function) types.

- For the expression **assert** e, similar as expression of **defer** $x : \tau\{e\}$ **default** v_d, according to E-ASSERTCONSTRAINT and E-ASSERT, it holds the progress property as well.

- For the expression **concretize** e **with** v_c, if sub-expression e is not a value, according to the hypothesis, there exists an expression e' such that $\vdash \langle \Sigma, \Delta, e \rangle \rightarrow \langle \Sigma', \Delta', e' \rangle$. According to E-CONCRETIZEEXP, it holds the **progress** property. On the contrary, if sub-expression e is a value, then a MODEL will be built to model the constraints, and it can be reduced to a concrete value c that satisfies the model or an **error** will be generated while the model cannot be satisfied according to E-CONCRETIZESAT and E-CONCRETIZEUNSAT. Therefore, we can conclude the expression **concretize** e **with** v_c holds the property as well.

- For the expression **let** $x : \tau = e_1$ **in** e_2 and **let rec** $f : \tau = e_1$ **in** e_2, expressions e_1 and e_2 are composed by one ore more above expressions. Thus, it holds the property accordingly.

□

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

Proof. According to the static semantics 5, we can proof that λ_J program is type-preserved by structral induction over the syntax of λ_J. According to T-VAR, T-INT, T-BOOL, T-UNIT, T-CONTEXT, we know the simple expressions are all type-preserved. Next we will discuss the *operation*, *condition*, λ *abstraction*, *application*, *let*, *assert*, $defer$ and *concretize* expressions respectively. At the beginning, by the typing rules from Figure 5, we conclude that all the δ value is the same for both sides except for the T-CONCRETIZE in which, however, the type is also preserved for it only cast the symbolic type to the corresponding concrete type. Therefore, we will not discuss the δ type in the following proof. In addition, by the hypothesis, all the sub-expressions are well-typed while proof the specific expression.

- For the operation expression e_1 (op) e_2, assume that expression e_1 is with the type τ_1 and expression e_2 is with the type τ_2, according to T-OP there must be a type τ that is the common super type of e_1 and e_2, such that the expression e_1 (op) e_2 is with the type τ. According to E-OP, E-OP1 and E-OP2, after one step evaluation, the expression e_1 will be some e_1' with the type τ_1', such that $\tau_1' <: \tau_1$, or the expression e_2 will be reduced to some e_2' with the type τ_2' and $\tau_2' <: \tau_2$, assume that the evaluated expression is with the type τ', according to subtyping, we know $\tau' <: \tau$, which indicates that the type is preserved.

- For the conditional expression **if** e_1 **then** e_t **else** e_f, assume expressions e_t and e_f are with the type τ_1 and τ_2 respectively. According to T-CONDC and T-CONDSYM, if the expression e_1 is with the symbolic type, **if** e_1 **then** e_t **else** e_f must be with some type τ which is the common super type of τ_1 and τ_2. Then according to E-CONDSYMT, after one step evaluation e_t will be reduced to some e_t' with the type τ_1'. According to the hypothesis, $\tau_1' <: \tau_1$. Therefore the type τ is the common super type of τ_1' and τ_2 as well. And the E-CONDSYMF is the same. In addition, if the expression e_1 is with the concrete type, there will be only one branch being taken, either τ_1 or τ_2, either of which holds the subtyping relation naturally. Consequently, the conditional expression is type-preserved.

- For the λ abstraction expression $\lambda x : \tau.e$, according to E-APPLAMBDA, the type of the expression will not change and it is type-preserved naturally.

- For the application expression $e_1\ e_2$, assume the type of e_1 and e_2 are $\tau_1 \to \tau_2$ and τ_1' respectively. In addition, $\tau_1' <: \tau_1$, and according to T-APP, the type of application expression is τ_2. Then according to E-APP1 and E-APP2, the evaluated expression will preserve the expression type due to the hypothesis that the sub-expressions e_1 and e_2 preserve the corresponding type respectively.
- Similarly for the expression **let rec** $f : \tau = e_1$ **in** e_2, **assert** e, **defer** $x : \tau\{e\}$ **default** v_d and **concretize** e **with** v_c.

□

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
                     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

The hardest part for us in this course project is trying to understand how the λ_J works. Since the limited page number, the authors only list the necessary semantics and a small

part of property proof, eliminating most detailed explanation. The only resource we can find is a presentation video, a github project with thousands of lines of source codes besides this page. We went through several days' heated discussion before we finally get to understand how λ_J rule out the cases of infinite recursion and what **rep** is designed for. Another non-trivial task is to run the existing codes successfully and know how it works roughly inside.

The most important thing we have learned from this project, is how to design a new programming language for a specific goal, and what we should deal with as it may introduce extra problems. For example, λ_J syntax leads to infinite recursive calls via symbolic expression, so it provides complicated static semantic to constrain the type checking. Also, since all is based on λ-calculus, it makes us understand the power of this simple language deeper. It's also a good chance for us to look back to the teaching materials in previous courses. All these help us to understand types and programming language better.

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.