

Types and Programming Language Project Report*

Jiajun Jiang
jiajun.jiang@pku.edu.cn

Mengwei Xu
xumw@pku.edu.cn

Introduction

Here is the introduction...

1 Language Design

1.1 Jeeves

Here we describe Jeeves syntax in Figure 1. Totally, it contains three types

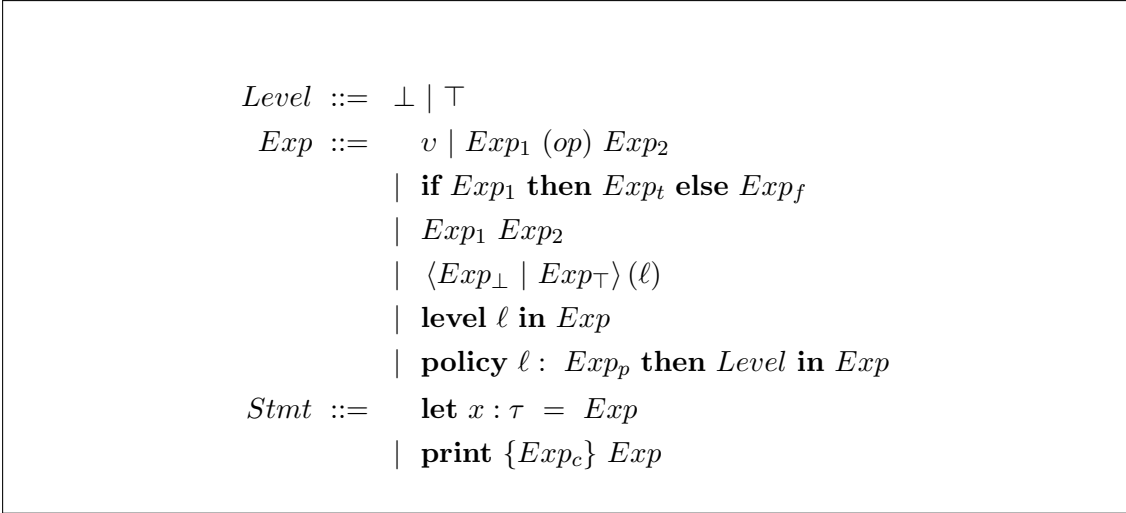

$$\begin{aligned} \textit{Level} &::= \perp \mid \top \\ \textit{Exp} &::= v \mid \textit{Exp}_1 \textit{ (op) Exp}_2 \\ &\quad \mid \textbf{if } \textit{Exp}_1 \textbf{ then } \textit{Exp}_t \textbf{ else } \textit{Exp}_f \\ &\quad \mid \textit{Exp}_1 \textit{ Exp}_2 \\ &\quad \mid \langle \textit{Exp}_\perp \mid \textit{Exp}_\top \rangle (\ell) \\ &\quad \mid \textbf{level } \ell \textbf{ in } \textit{Exp} \\ &\quad \mid \textbf{policy } \ell : \textit{Exp}_p \textbf{ then } \textit{Level} \textbf{ in } \textit{Exp} \\ \textit{Stmt} &::= \textbf{let } x : \tau = \textit{Exp} \\ &\quad \mid \textbf{print } \{\textit{Exp}_c\} \textit{Exp} \end{aligned}$$

Figure 1: Jeeves syntax

1.2 Lambda J

Here we describe the λ_J language shown in Figure 2.

2 Semantics

Here is the semantics...

*done at June 5th, 2016

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

Figure 2: λ_J syntax

3 Properties

Lemma 1. (ConcreteFunction). *if v is a value of type $\tau_1 \rightarrow \tau_2$, 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. \square

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

Proof. we can ... \square

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

Proof. we can ... \square

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

```

$\boxed{\mathcal{G} \vdash \langle \Sigma, \Delta, e \rangle \rightarrow \langle \Sigma', \Delta', e' \rangle}$	
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \rightarrow \langle \Sigma', \Delta', e'_1 \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \ e_2 \rangle \rightarrow \langle \Sigma', \Delta', e'_1 \ e'_2 \rangle}$	E-APP1
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_2 \rangle \rightarrow \langle \Sigma', \Delta', e'_2 \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, v \ e_2 \rangle \rightarrow \langle \Sigma', \Delta', v \ e'_2 \rangle}$	E-APP2
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, \lambda x.e \ v \rangle \rightarrow \langle \Sigma', \Delta', e[x \mapsto v] \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, c_1 \ (op) \ c_2 \rangle \rightarrow \langle \Sigma', \Delta', c'_1 \ \langle \Sigma', \Delta', c'_2 \rangle \rangle}$	E-APPLAMBDA
$\frac{c' = c_1 \ (op) \ c_2}{\mathcal{G} \vdash \langle \Sigma, \Delta, c_1 \ (op) \ c_2 \rangle \rightarrow \langle \Sigma', \Delta', c' \rangle}$	E-OP
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \rightarrow \langle \Sigma', \Delta', e'_1 \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \ (op) \ e_2 \rangle \rightarrow \langle \Sigma', \Delta', e'_1 \ (op) \ e'_2 \rangle}$	E-OP1
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_2 \rangle \rightarrow \langle \Sigma', \Delta', e'_2 \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, v \ (op) \ e_2 \rangle \rightarrow \langle \Sigma', \Delta', v \ (op) \ e'_2 \rangle}$	E-OP2
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_c \rangle \rightarrow \langle \Sigma', \Delta', e'_c \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{if } e_c \text{ then } e_t \text{ else } e_f \rangle \rightarrow \langle \Sigma', \Delta', \text{if } e'_c \text{ then } e_t \text{ else } e_f \rangle}$	E-COND
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_t \rangle \rightarrow \langle \Sigma', \Delta', e'_t \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{if true then } e_t \text{ else } e_f \rangle \rightarrow \langle \Sigma', \Delta', e'_t \rangle}$	E-CONDTRUE
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_f \rangle \rightarrow \langle \Sigma', \Delta', e'_f \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{if false then } e_t \text{ else } e_f \rangle \rightarrow \langle \Sigma', \Delta', e'_f \rangle}$	E-CONDFALSE
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_t \rangle \rightarrow \langle \Sigma', \Delta', e'_t \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{if } \sigma \text{ then } e_t \text{ else } e_f \rangle \rightarrow \langle \Sigma', \Delta', \text{if } \sigma \text{ then } e'_t \text{ else } e_f \rangle}$	E-CONDSYMT
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e_f \rangle \rightarrow \langle \Sigma', \Delta', e'_f \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{if } \sigma \text{ then } v_t \text{ else } e_f \rangle \rightarrow \langle \Sigma', \Delta', \text{if } \sigma \text{ then } v_t \text{ else } e'_f \rangle}$	E-CONDSYMF
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e \rangle \rightarrow \langle \Sigma', \Delta', e' \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{defer } x:\tau\{e\} \text{ default } v_d \rangle \rightarrow \langle \Sigma', \Delta', \text{defer } x:\tau\{e'\} \text{ default } v_d \rangle}$	E-DEFERCONSTRAINT
$\frac{\text{fresh } x'}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{defer } x:\tau\{e\} \text{ default } v_d \rangle \rightarrow \langle \Sigma \cup \{\mathcal{G} \Rightarrow v_c[x \mapsto x']\}, \Delta \cup \{\mathcal{G} \Rightarrow x' = v_d\}, x' \rangle}$	E-DEFER
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e \rangle \rightarrow \langle \Sigma', \Delta', e' \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{assert } e \rangle \rightarrow \langle \Sigma', \Delta', \text{assert } e' \rangle}$	E-ASSERTCONSTRAINT
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{assert } v \rangle \rightarrow \langle \Sigma \cup \{\mathcal{G} \Rightarrow v\}, \Delta, () \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{assert } v \rangle \rightarrow \langle \Sigma \cup \{\mathcal{G} \Rightarrow v\}, \Delta, () \rangle}$	E-ASSERT
$\frac{\mathcal{G} \vdash \langle \Sigma, \Delta, e \rangle \rightarrow \langle \Sigma', \Delta', e' \rangle}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{concretize } e \text{ with } v_c \rangle \rightarrow \langle \Sigma', \Delta', \text{concretize } e' \text{ with } v_c \rangle}$	E-CONCRETIZEEXP
$\frac{\text{MODLE}(\Delta, \Sigma \cup \{\mathcal{G} \cap \text{context} = v_c\}) = \mathcal{M} \quad c = \mathcal{M}[[v_\nu]]}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{concretize } v_\nu \text{ with } v_c \rangle \rightarrow \langle \Sigma, \Delta, c \rangle}$	E-CONCRETIZESAT
$\frac{\text{MODLE}(\Delta, \Sigma \cup \{\mathcal{G} \cap \text{context} = v_c\}) = \text{UNSAT}}{\mathcal{G} \vdash \langle \Sigma, \Delta, \text{concretize } v_\nu \text{ with } v_c \rangle \rightarrow \langle \Sigma, \Delta, \text{error} \rangle}$	E-CONCRETIZEUNSAT

Figure 3: Dynamic semantics for λ_J .

$\delta ::=$	concretize sym	determinism tag
$\beta ::=$	int _c bool _c unit int bool	base type
$\tau ::=$	β $\tau_1 \xrightarrow{nr} \tau_2$ $\tau_1 \rightarrow \tau_2$	type

Figure 4: λ_J types

$\tau_1 <: \tau_2$							
$\overline{\tau <: \tau}$	S-REFLEXIVE	$\overline{\text{int}_c <: \text{int}}$	S-INT	$\overline{\text{bool}_c <: \text{bool}}$	S-BOOL		
$\frac{}{\tau_1 \xrightarrow{nr} \tau_2 <: \tau_1 \rightarrow \tau_2}$		S-RECFUN	$\frac{\tau'_1 <: \tau_1 \quad \tau_2 <: \tau'_2}{\tau_1 \rightarrow \tau_2 <: \tau'_1 \rightarrow \tau'_2}$		S-FUN		
$\text{rep } \tau$							
$\frac{\text{rep } \tau \quad \tau' <: \tau}{\text{rep } \tau}$		OK-SUBTYPE	$\overline{\text{rep } \beta}$		OK-BASETYPE		
$\frac{\text{rep } \tau_2}{\text{rep } \beta_1 \rightarrow \tau_2}$		OK-BASEFUNCTION	$\frac{\text{rep } (\tau_1 \xrightarrow{nr} \tau') \quad \text{rep } \tau_2}{\text{rep } (\tau_1 \xrightarrow{nr} \tau') \xrightarrow{nr} \tau_2}$		OK-HOFUNCTION		
$\frac{\text{rep } \tau_1 \rightarrow \tau_2}{\text{rep } (\tau_1 \rightarrow \tau_2) \rightarrow \beta}$		OK-RECFUNCTIONBASE	$\frac{\text{rep } \tau_1 \rightarrow \tau_2 \quad \text{rep } \tau'_1 \rightarrow \tau'_2}{\text{rep } (\tau_1 \rightarrow \tau_2) \rightarrow (\tau'_1 \rightarrow \tau'_2)}$		OK-RECFUNCTION		
$\Gamma; \gamma \vdash e : \langle \tau, \delta \rangle$							
$\frac{x \in \Gamma}{\Gamma; \gamma \vdash x : \Gamma(x)}$	T-VAR	$\overline{\Gamma; \gamma \vdash n : \text{int}_c}$	T-INT	$\overline{\Gamma; \gamma \vdash b : \text{bool}_c}$	T-BOOL	$\overline{\Gamma; \gamma \vdash () : \text{unit}}$	T-UNIT
$\frac{\text{rep } \tau}{\Gamma; \gamma \vdash \text{context } \tau : \tau}$		T-CONTEXT	$\frac{\Gamma; \gamma \vdash e_1 : \tau_1 \quad \Gamma; \gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 <: \tau \quad \text{rep } \tau}{\Gamma; \gamma \vdash e_1 (op) e_2 : \tau}$		T-OP		
$\frac{\Gamma; \gamma \vdash e : \text{bool}_c \quad \Gamma; \gamma \vdash e_t : \tau_1 \quad \Gamma; \gamma \vdash e_f : \tau_2 \quad \tau_1, \tau_2 <: \tau \quad \text{rep } \tau}{\Gamma; \gamma \vdash \text{if } e \text{ then } e_t \text{ else } e_f : \tau}$							
$\frac{\Gamma; \gamma \vdash e : \text{bool} \quad \Gamma; \text{sym} \vdash e_t : \beta_1 \quad \Gamma; \text{sym} \vdash e_f : \beta_2 \quad \beta_1, \beta_2 <: \beta_c \quad \text{rep } \beta_c}{\Gamma; \gamma \vdash \text{if } e \text{ then } e_t \text{ else } e_f : \beta_c}$							
$\frac{\Gamma, x : \tau_d; \gamma \vdash e : \tau' \quad \text{rep } \tau_d \quad \text{rep } \tau'}{\Gamma; \gamma \vdash (\lambda x : \tau_d. e) : \tau_d \rightarrow \tau'}$		T-LAMBDA	$\frac{\Gamma; \gamma \vdash e_1 : \tau_1 \xrightarrow{nr} \tau_2 \quad \Gamma, x : \tau_d; \gamma \vdash e_2 : \tau'_1 \quad \tau'_1 <: \tau_1 \quad \text{rep } \tau_1 \quad \text{rep } \tau_2}{\Gamma; \gamma \vdash (e_1 e_2) : \tau_2}$		T-APP		
$\frac{\Gamma, f : \tau_1 \rightarrow \tau_2; \gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma, f : \tau_1 \xrightarrow{nr} \tau_2; \gamma \vdash e_2 : \tau_2 \quad \text{rep } \tau \quad \text{rep } \tau_2}{\Gamma; \gamma \vdash \text{let rec } f : \tau_1 \xrightarrow{nr} \tau_2 = e_1 \text{ in } e_2 : \tau_2}$							
$\frac{\gamma = \text{concrete} \quad \Gamma; \gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma; \gamma \vdash e_2 : \tau'_1 \quad \tau'_1 <: \tau_1 \quad \text{rep } \tau_1 \quad \text{rep } \tau_2}{\Gamma; \gamma \vdash (e_1 e_2) : \tau_2}$							
$\frac{\Gamma, x : \beta; \gamma \vdash e_c : \text{bool} \quad \Gamma; \gamma \vdash v : \beta}{\Gamma; \gamma \vdash (\text{defer } x : \beta \{ e_c \} \text{ default } v) : \beta}$		T-DEFER	$\frac{\Gamma; \gamma \vdash e_c : \text{bool}}{\Gamma; \gamma \vdash (\text{assert } e_c) : \text{unit}}$		T-ASSERT		
$\frac{\Gamma; \gamma \vdash e_1 : \beta \quad \Gamma; \gamma \vdash e_1 : \beta' \quad \Gamma; \gamma \vdash v : \beta'}{\Gamma; \gamma \vdash (\text{concretize } e_1 \text{ with } v) : \beta_c}$							
T-CONCRETIZE							

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.