ITP20005 LOO

Type Checking

JC

Semantics Exercise

- Recursion

```
[aRecSub (name symbol?)
         (value-box (box/c RCFAE-Value?))
         (ds DefrdSub?)])

[rec (bound-id named-expr first-call)
                (local [(define value-holder (box (numV 198)))
                        (define new-ds (aRecSub bound-id
                                                value-holder
                                                ds))]
                  (begin
                    (set-box! value-holder (interp named-expr new-ds))
                    (interp first-call new-ds)))]
```

$$\frac{\{\text{fun } \{i\}\ b\},\varepsilon \Rightarrow \langle i,b,\varepsilon'\rangle \quad a,\varepsilon \Rightarrow a_v \quad b,\varepsilon''[f \leftarrow (\langle i,b,\varepsilon'\rangle), i \leftarrow a_v] \Rightarrow b_v}{\{\text{rec } \{f\ \{\text{fun } \{i\}\ b\}\}\ \{f\ a\}\},\varepsilon \Rightarrow b_v}$$

Semantics Exercise

- Recursion

```
[aRecSub (name symbol?)
         (value-box (box/c RCFAE-Value?))
         (ds DefrdSub?)])

[rec (bound-id named-expr first-call)
                (local [(define value-holder (box (numV 198)))
                        (define new-ds (aRecSub bound-id
                                                value-holder
                                                ds))]
                  (begin
                   (set-box! value-holder (interp named-expr new-ds))
                   (interp first-call new-ds)))]
```

$$\frac{f,\varepsilon \Rightarrow \langle i,b,\varepsilon' \rangle \quad a,\varepsilon \Rightarrow a_v \quad b,\varepsilon''[f \leftarrow (\langle i,b,\varepsilon' \rangle), i \leftarrow a_v] \Rightarrow b_v}{\{\text{rec } \{f \{\text{fun } \{i\} \ b\}\} \{f \ a\}\},\varepsilon \Rightarrow b_v}$$

Semantics Exercise

- Recursion

```
[aRecSub (name symbol?)
         (value-box (box/c RCFAE-Value?))
         (ds DefrdSub?)])

[rec (bound-id named-expr first-call)
                (local [(define value-holder (box (numV 198)))
                        (define new-ds (aRecSub bound-id
                                                value-holder
                                                ds))]
                  (begin
                   (set-box! value-holder (interp named-expr new-ds))
                   (interp first-call new-ds)))]
```

$$\frac{f \Rightarrow \langle i,b,\varepsilon' \rangle \quad a,\varepsilon \Rightarrow a_v \quad b,\varepsilon''[f \leftarrow (\langle i,b,\varepsilon' \rangle), i \leftarrow a_v] \Rightarrow b_v}{\{f\ a\},\varepsilon \Rightarrow b_v}$$

Type Checking

No type cases

{+ {fun {x : num} x} 3}: no type

{7 5} : no type

code → (Parser) → code in AST → (Interpreter) -> Result

code → (Parser) → code in AST → (type checker) → (Interpreter) → Result

TFAE Grammar

e ::= n
| {+ e e}
| {- e e}
| x
| {fun {x:τ} e}
| {e e}

τ :: = num
| bool
| (τ -> τ)

TFAE Expressions

lang plai-typed

← How to install plai-typed: https://lists.racket-lang.org/users/archive/2013-August/059187.html

```
; abstract syntax tree (AST) for TFAE
(define-type TFAE
   [num (n : number)]
   [add  (lhs : TFAE) (rhs : TFAE)]
   [sub  (lhs : TFAE) (rhs : TFAE)]
   [id     (name : symbol)]
   [fun   (param : symbol) (type : TE) (body : TFAE)]
   [app  (fun-expr : TFAE) (arg-expr : TFAE)])
```

TFAE Expressions and Types

```
(define-type TE
  [numTE]
  [boolTE]
  [arrowTE (arg : TE) (result : TE)])

(define-type Type
  [numT]
  [boolT]
  [arrowT (arg : Type) (result : Type)])

(define-type TypeEnv
  [mtEnv]
  [aBind (name : symbol) (type : Type) (rest : TypeEnv)])
```

Type Check and Interpret

- typecheck and interp

```
(define eval : (TFAE -> TFAE-Value)
  (lambda (tfae)
    (begin
      ;; before calling interp, we can do type checking
      ;; based on our rules.
      (try (typecheck tfae (mtEnv))
           (lambda () (error 'type-error "typecheck")))
      (interp tfae (mtsub)))))
```

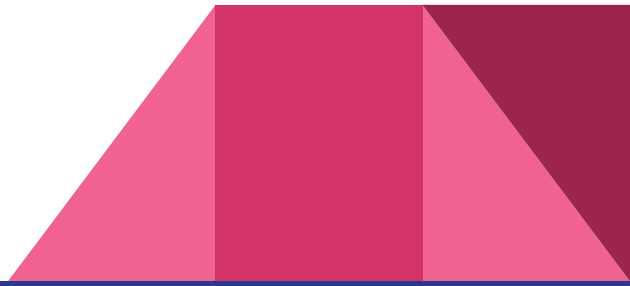
Type Check and Interpret

; typecheck : TFAE TypeEnv -> Type
(typecheck tfae env) = t
$\Gamma \vdash e : \tau$

; interp : TFAE DeferdSub -> TFAE-Value
(interp tfae ds) = v
$\sigma \vdash e \Rightarrow v$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      …)))
```



TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [num (n) …]
      …)))
```

$\Gamma \vdash n : num$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [num (n) (numT)]
      …)))
```

$\Gamma \vdash n : num$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [add (l r)
           … (typecheck l env) …
           … (typecheck r env) … ]
      …)))
```

$$\frac{\Gamma \vdash e_1 : \text{num} \qquad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \{+\ e_1\ e_2\} : \text{num}}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [add (l r)
           (type-case Type (typecheck l env)
             [numT ()
                   (type-case Type (typecheck r env)
                     [numT () (numT)]
                     [else    (type-error r "num")])]
             [else   (type-error l "num")])]
      ...)))
```

$$\frac{\Gamma \vdash e_1 : \text{num} \qquad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash \{+\ e_1\ e_2\} : \text{num}}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [id  (name) … ]
      …)))
```

$$[\ldots\ x : \tau\ \ldots\ ] \vdash x : \tau$$

or

$$\frac{x \in \text{Domain}\ (\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [id  (name)   (get-type name env)]
      …)))
```

$$[\ldots\ x : \tau\ \ldots\ ] \vdash x : \tau$$

or

$$\frac{x \in \text{Domain}\ (\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [fun  (name te body)
            ...]
      ...)))
```

$$\frac{\Gamma[x : \tau] \vdash e : \tau_2}{\Gamma \vdash \{\text{fun } \{x : \tau_1\}\ e\} : (\tau_1 \rightarrow \tau_2)}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [fun  (name te body)
            … (parse-type te) …
            … (typecheck body …) … ]
      …)))
```

$$\frac{\Gamma[x : \tau] \vdash e : \tau_2}{\Gamma \vdash \{\text{fun } \{x : \tau_1\}\ e\} : (\tau_1 \rightarrow \tau_2)}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [fun  (name te body)
            (local [(define param-type (parse-type te))]
             … (typecheck body (aBind name
                                      param-type
                                      env))
                                      … )]
      …)))
```

$$\frac{\Gamma[x : \tau] \vdash e : \tau_2}{\Gamma \vdash \{\text{fun } \{x : \tau_1\}\ e\} : (\tau_1 \rightarrow \tau_2)}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [fun  (name te body)
            (local [(define param-type (parse-type te))]
              (arrowT param-type (typecheck body (aBind name
                                                        param-type
                                                        env))))]
      ...)))
```

$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \{\text{fun } \{x : \tau_1\}\ e\} : (\tau_1 \rightarrow \tau_2)}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [app  (fn arg)
                          … ]
      …)))
```

$$\frac{\Gamma \vdash e_1 : (\tau_2 \rightarrow \tau_3) \qquad \Gamma \vdash e_1 : \tau_2}{\Gamma \vdash \{e_1\ e_2\} : \tau_3}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [app  (fn arg)
                    … (typecheck fn env) …
                    … (typecheck arg env) … ]
      …)))
```

$$\frac{\Gamma \vdash e_1 : (\tau_2 \rightarrow \tau_3) \qquad \Gamma \vdash e_1 : \tau_2}{\Gamma \vdash \{e_1\ e_2\} : \tau_3}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [app  (fn arg)
                  (type-case Type (typecheck fn env)
                    [arrowT (param-type result-type)
                                ... (typecheck arg env) ... ]
                    [else (type-error fn "function")])]
      ...)))
```

$$\frac{\Gamma \vdash e_1 : (\tau_2 \rightarrow \tau_3) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \{e_1\ e_2\} : \tau_3}$$

TFAE Type Checker

```
(define typecheck : (TFAE TypeEnv -> Type)
  (lambda (tfae env)
    (type-case TFAE tfae
      [app  (fn arg)
                (type-case Type (typecheck fn env)
                  [arrowT (param-type result-type)
                          (if (equal? param-type
                                      (typecheck arg env))
                              result-type
                              (type-error arg
                                          (to-string param-type)))]
                  [else (type-error fn "function")])]
      ...)))
```

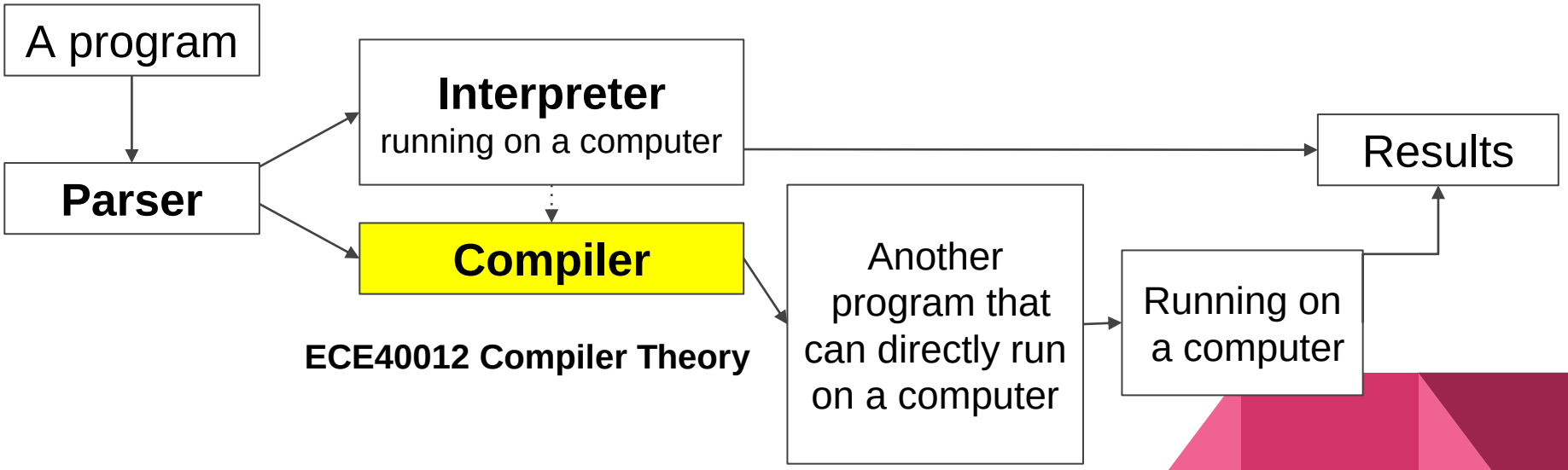
$$\frac{\Gamma \vdash e_1 : (\tau_2 \rightarrow \tau_3) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \{e_1\ e_2\} : \tau_3}$$

Topics we covered

- Racket tutorials (L2,3, HW)
- Modeling languages (L4,5, HW
- Interpreting arithmetic (L5)
- Language principles
  - **Substitution** (L6, HW)
  - **Function** (L7)
  - **Deferring Substitution** (L8,L9)
  - **First-class Functions** (L10-12)
  - **Laziness** (L13, L14)
  - **Recursion** (L15, L16)
  - **Mutable data structures** (L17,18,19)
  - **Variables** (L20, L21)
  - **Continuations** (L22,23,24)
  - Garbage collection
  - **Semantics** (L25)
  - **Type** (L26,L27)
- Guest Video Lecture (L28)

Big Picture (modeling languages)

- Just **write an interpreter** to explain a language.
- By writing an interpreter, we can understand the language!
- Interpreter can be converted into a compiler!!!



Creation and Languages

In the beginning was the Word, and the Word was with God, and the Word was God. He was with God in the beginning. Through him all things were made; without him nothing was made that has been made. (John 1:1-3)