Bidirectional Type Rules and a **`#lang`** for Writing Them

HYOL Feb 27 2018

Implementing type checking

- $\mathtt{G} \vdash \mathtt{e} : \tau$
- `_ ⊢ _ : _`
  - What is the signature?

- `_ ⊢ _ : _`
  - `TypeEnv Expr -> Type`
  - Or???
  - `TypeEnv Expr Type -> Bool`

Let's look at its usage in type rules

- Quiz: What are inputs and outputs?

$$\frac{G \vdash e_1 : \tau_1 \to \tau_2 \qquad G \vdash e_2 : \tau_2}{G \vdash e_1\, e_2 : \tau_2} \quad \text{[T-App]}$$

Usage suggests both signatures needed

- Quiz: What are inputs and outputs?

INPUT INPUT OUTPUT INPUT INPUT INPUT

$$\frac{G \vdash e_1 : \tau_1 \to \tau_2 \qquad G \vdash e_2 : \tau_2}{G \vdash e_1\, e_2 : \tau_2}$$

INPUT INPUT OUTPUT?

Give separate names to the 2 functions

- **`compute-type`**
  - `TypeEnv Expr -> Type`
- **`typecheck?`**
  - `TypeEnv Expr Type -> Bool`

Rewrite type rule(s) with these 2 fns

```
compute-type G e1 = (-> t1 t2)
typecheck? G e2 t1
---------------------
compute-type G (e1 e2) = t2
```

[T-App-Compute]

```
compute-type G e2 = t1
typecheck? G e1 (-> t1 t2)
---------------------
typecheck? G (e1 e2) t2
```

[T-App-Check]

Type theorists use ⇒ and ⇐ instead

- ~~`compute-type`~~ ⇒
  - `TypeEnv Expr -> Type`
- ~~`typecheck?`~~ ⇐
  - `TypeEnv Expr Type -> Bool`

Bidirectional-style type rules

$$\frac{G \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad G \vdash e_2 \Leftarrow \tau_2}{G \vdash e_1\, e_2 \Rightarrow \tau_2} \quad \text{[T-App-}\Rightarrow\text{]}$$

$$\frac{G \vdash e_1 \Leftarrow \tau_1 \to \tau_2 \qquad G \vdash e_2 \Rightarrow \tau_1}{G \vdash e_1\, e_2 \Leftarrow \tau_2} \quad \text{[T-App-}\Leftarrow\text{]}$$

(For more info, see Pierce and Turner 2000, “Local Type Inference”)

Wouldn’t it be nice to write bidirectional rules to implement a type checker?

It turns out that bidir rules are already similar to our type checking macros.

- To show this, we convert the type rules into a form that looks more like our type checking macros.
- Conversion steps:
  - Move inputs to the top, outputs to the bottom, like normal code
  - Use syntax-parse-like forms
  - Interleave rewriting and type checking

Start with these rules from before

```
compute-type G e1 = (-> t1 t2)
typecheck? G e2 t1
---------------------             [T-App-Compute]
compute-type G (e1 e2) = t2

compute-type G e2 = t1
typecheck? G e1 (-> t1 t2)
---------------------             [T-App-Check]
typecheck? G (e1 e2) t2
```

Move inputs to top, outputs to bottom

```
(define-? (compute-type G (e1 e2))
 (compute-type G e1) = (-> t1 t2)
 (typecheck? G e2 t1) = true
 t2)

(define-? (typecheck? G (e1 e2) t2)
 (compute-type G e2) = t1
 (typecheck? G e1 (-> t1 t2)))
```

Use some syntax-parse forms

Still equivalent to bidirectional rules but starting to look like our type checking macros?

```
(define-? (compute-type G (e1 e2))
 #:with (~-> t1 t2) (compute-type G e1)
 #:fail-unless (typecheck? G e2 t1)
 t2)

(define-? (typecheck? G (e1 e2) t2)
 #:with t1 (compute-type G e2)
 (typecheck? G e1 (-> t1 t2)))
```

Interleave checking and term rewriting

```
(define-? (compute-type+rewrite G (e1 e2))
 #:with (e1’ (~-> t1 t2)) (compute+rewrite G e1)
 #:with e2’ (typecheck?+rewrite G e2 t1)
 (e1’ e2’) t2)

(define-? (typecheck?+rewrite G (e1 e2) t2)
 #:with (e2’ t1) (compute+rewrite G e2)
 #:with e1’ (typecheck?+rewrite G e1 (-> t1 t2))
 (e1’ e2’))
```

Interleave checking and term rewriting

We can do the same thing with the ⇒ and ⇐ versions of the rules.

$$\frac{G \vdash e_1 \gg e_{1'} \Rightarrow \tau_1 \to \tau_2 \qquad G \vdash e_2 \gg e_{2'} \Leftarrow \tau_2}{G \vdash e_1\, e_2 \gg e_{1'} e_{2'} \Rightarrow \tau_2}$$

$$\frac{G \vdash e_1 \gg e_{1'} \Leftarrow \tau_1 \to \tau_2 \qquad G \vdash e_2 \gg e_{2'} \Rightarrow \tau_1}{G \vdash e_1\, e_2 \gg e_{1'} e_{2'} \Leftarrow \tau_2}$$

Now use bidirectional syntax instead

But keeping inputs at top, outputs at bottom.

```
(define-? G (e1 e2) ≫
 G ⊢ e1 ≫ e1’ ⇒ (~-> t1 t2)
 G ⊢ e2 ≫ e2’ ⇐ t1
 -----------------
 ⊢ (e1’ e2’) ⇒  t)

(define-? G (e1 e2) ⇐ t2 ≫
 G ⊢ e2 ≫ e2’ ⇒  t1
 G ⊢ e1 ≫ e1’ ⇐ (-> t1 t2)
 --------------------
 ⊢ (e1’ e2’))
```

Drop explicit G (handled by expander)

```
(define-? (e1 e2) ≫
 ⊢ e1 ≫ e1’ ⇒ (~-> t1 t2)
 ⊢ e2 ≫ e2’ ⇐ t1
 -----------------
 ⊢ (e1’ e2’) ⇒ t)

(define-? (e1 e2) ⇐ t2 ≫
 ⊢ e2 ≫ e2’ ⇒ t1
 ⊢ e1 ≫ e1’ ⇐ (-> t1 t2)
 --------------------
 ⊢ (e1’ e2’))
```

Combine clauses. Macro dispatch is based on the kind of language construct

(See “Expressivity” slide at the end)

```
(define-? typed-app
 [(_ e1 e2) ≫        ; compute-type
  ⊢ e1 ≫ e1’ ⇒ (~-> t1 t2)
  ⊢ e2 ≫ e2’ ⇐ t1
  ------------------
  ⊢ (e1’ e2’) ⇒ t]
 [(_ e1 e2) ⇐ t2 ≫ ; typecheck?
  ⊢ e2 ≫ e2’ ⇒ t1
  ⊢ e1 ≫ e1’ ⇐ (-> t1 t2)
  ---------------------
  ⊢ (e1’ e2’)])
```

Add some more parens … #lang turnstile

```
#lang turnstile

(define-syntax/typecheck typed-app
 [(_ e1 e2) ≫       ; compute-type
  [⊢ e1 ≫ e1’ ⇒ (~-> t1 t2)]
  [⊢ e2 ≫ e2’ ⇐ t1]
  -----------------
  [⊢ (e1’ e2’) ⇒ t]]
 [(_ e1 e2) ⇐ t2 ≫ ; typecheck?
  [⊢ e2 ≫ e2’ ⇒ t1]
  [⊢ e1 ≫ e1’ ⇐ (-> t1 t2)]
  --------------------
  [⊢ (e1’ e2’)]])
```

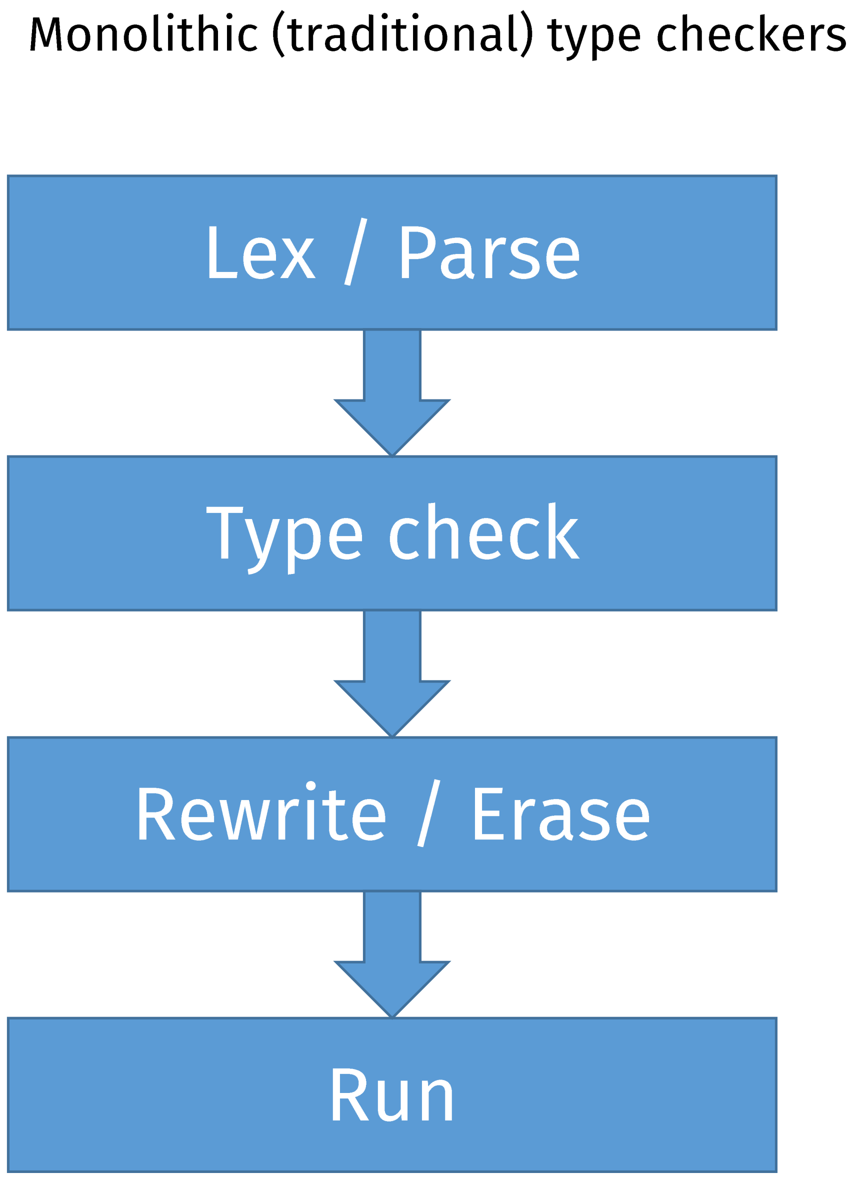
“Expressivity” Problem

- “monolithic” style
  - One “definition” to process the entire grammar (ie, data def)
    - E.g., `compute-type` function
  - Add operation: <u>EASY</u>
    - E.g., `compute-type`, `typecheck?`
    - Just add another definition
  - Extend grammar: <u>HARDER</u>
    - E.g., `#%app`, `lambda`
    - Must extend each definition

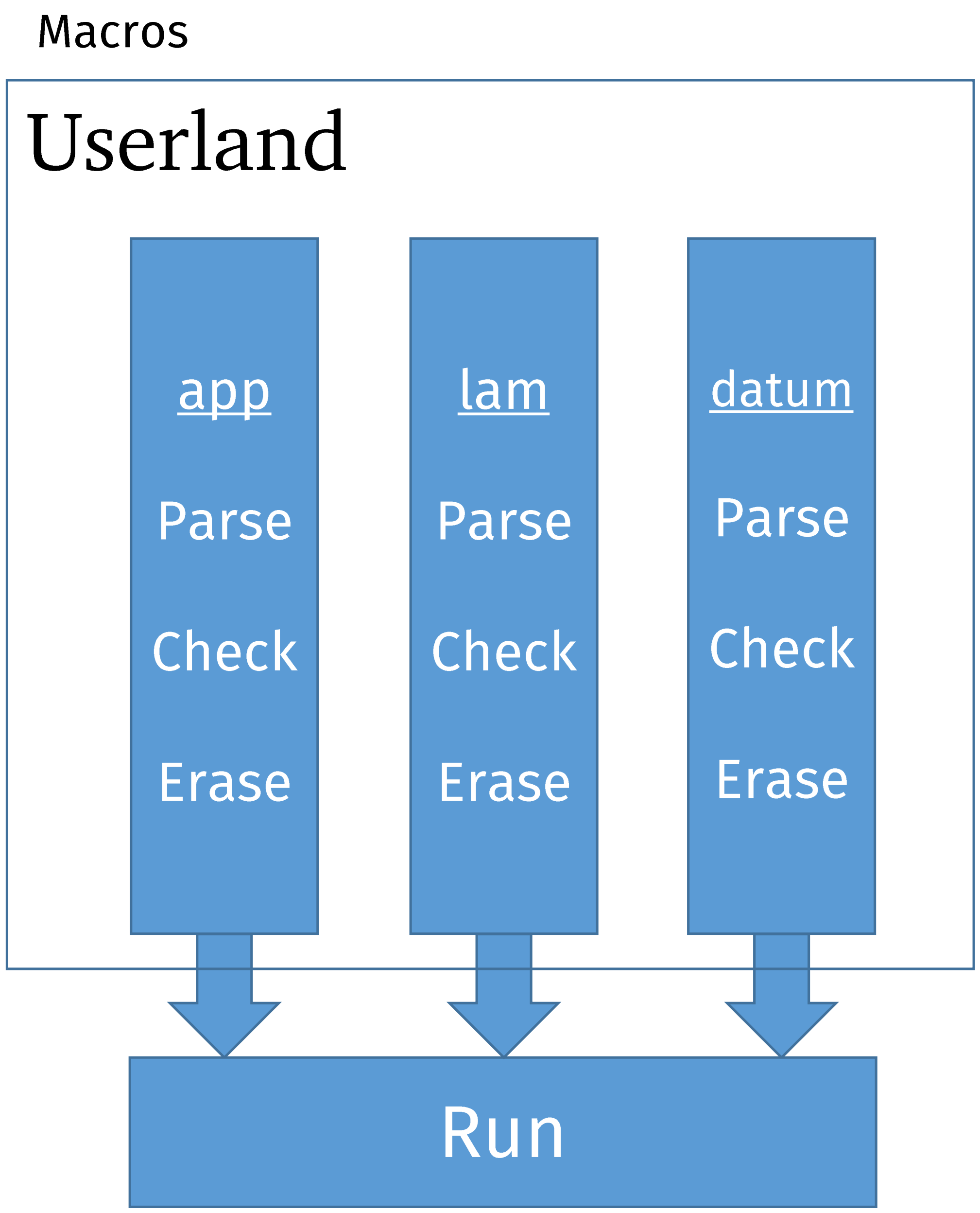
VS

- “macros” style
  - One “definition” for each grammar (ie, data def) clause
    - E.g., `#%app` macro
  - Add operation: <u>HARDER</u>
    - E.g., `compute-type`, `typecheck?`
    - Must extend each definition
  - Extend grammar: <u>EASY</u>
    - E.g., `#%app`, `lambda`
    - Just add another definition

See also: FP vs OOP


Monolithic (traditional) type checkers
Lex / Parse
Type check
Rewrite / Erase
Run

vs


Macros
Userland
app
Parse
Check
Erase
lam
Parse
Check
Erase
datum
Parse
Check
Erase
Run