


# Building Typed Languages with Turnstile

Stephen Chang  
Racket Summer School 2018  
(Thursday afternoon)



# Where are we in the schedule?

- Monday, Tuesday, Wednesday:
    - Tools for building languages with Racket
  - Thursday:
    - How to build typed languages
  - Thursday and Friday:
    - Let's look at some example languages built with Racket
- We are still here
- But we are also here
- 

The language of type rules ... is a language

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_{in} \rightarrow \tau_{out} \quad \Gamma \vdash e_2 \Leftarrow \tau_{in}}{\Gamma \vdash e_1 e_2 \Rightarrow \tau_{out}}$$

# A Macro-based Language in Racket

UNTYPED-LANG

```
#lang RACKET  
(provide-as [new-app #%app]  
            [new-λ λ])
```

```
(def-stx (new-app e1 e2)  
  i . . . .  
  (#%app e1 e2))
```

```
(def-stx (new-λ x e)  
  i . . . .  
  (λ x e))
```



UNTYPED-PROG

```
#lang UNTYPED-LANG  
  
; uses: new-app, new-λ
```

# IDEA: A Typed Macro-based Language

UNTYPED-LANG

```
#lang RACKET
(provide-as [new-app #%app]
            [new-λ λ])

(def-stx (new-app e1 e2)
  ; ....
  (host-app e1 e2))

(def-stx (new-λ x e)
  ; ....
  (host-λ x e))
```



UNTYPED-PROG

```
#lang UNTYPED-LANG

; uses: new-app, new-λ
```

TYPED-LANG

```
#lang RACKET
(provide-as [typed-app #%app]
            [typed-λ λ])

(def-stx (typed-app e1 e2)
  ; do typechecking
  (host-app e1 e2))

(def-stx (typed-λ [x : τ] e)
  ; do typechecking
  (host-λ x e))
```



TYPED-PROG

```
#lang TYPED-LANG

; uses: typed-app, typed-λ
```

“do typechecking” = expand + stx props

```
(define-syntax (typed-app stx)
  (syntax-parse stx
    [(_ e1 e2)
     #:with e3 (local-expand #'e1)
     #:with (→ τin τ) (stx-prop #'e3 'ty)
     #:with e4 (local-expand #'e2)
     #:with τarg (stx-prop #'e4 'ty)
     #:fail-unless (stx= #'τarg #'τin)
     (stx-prop #'(%app e3 e4) 'ty #'τ)])
```

Compute type

Check type

Assign type

Types checked when macro expands

# Leverage Domain-specific Syntax

$$\text{[T-App]} \frac{\begin{array}{l} \Gamma \vdash e_1 \Rightarrow \tau_{in} \rightarrow \tau_{out} \\ \Gamma \vdash e_2 \Leftarrow \tau_{in} \end{array}}{\Gamma \vdash e_1 e_2 \Rightarrow \tau_{out}}$$

Compute type

Check type

Assign type

# TURNSTILE: Type and Rewrite Rules

$$\begin{array}{c}
 \Gamma \vdash e_1 \Rightarrow \tau_{in} \rightarrow \tau_{out} \quad \Gamma \vdash e_2 \Leftarrow \tau_{in} \\
 \hline
 \Gamma \vdash e_1 e_2 \Rightarrow \tau_{out}
 \end{array}
 \quad
 \begin{array}{l}
 (\text{def-typed-stx } (\text{typed-app } e_1 \ e_2) \gg \\
 [\vdash e_1 \gg e_1' \Rightarrow (\rightarrow \tau_{in} \ \tau_{out})] \\
 [\vdash e_2 \gg e_2' \Leftarrow \tau_{in}] \\
 [\vdash (\#\%app \ e_1' \ e_2') \Rightarrow \tau_{out}])
 \end{array}$$

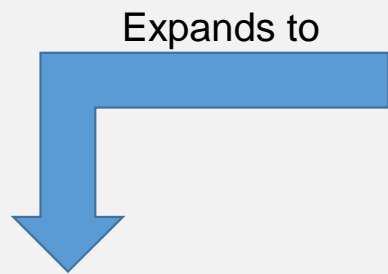
Conclusion: inputs

---

Conclusion: outputs



# TURNSTILE: Type and Rewrite Rules



```
(def-stx (typed-app e1 e2)  
  #:with e1' (expand e1)  
  #:with (→ τin τout) (detach e1')  
  #:with e2' (expand e2)  
  #:with τarg (detach e2')  
  #:fail-unless (stx= τarg τin)  
  (attach (#%app e1' e2') τout)
```

```
(def-typed-stx (typed-app e1 e2) »  
  [⊢ e1 » e1' ⇒ (→ τin τout)]  
  [⊢ e2 » e2' ⇐ τin]  
  
(def-typed-stx, (typed-app e1 e2) »  
[⊢ (#%app e1' e2') ⇒ τout]  
[⊢ e1 » e1' ⇒ (→ τin τout)]  
[⊢ e2 » e2' ⇐ τin]  
-----  
[⊢ (#%app e1' e2') ⇒ τout]
```

