Secure and Efficient Parsing via Programming Language Theory

Neel Krishnaswami & Jeremy Yallop

parsing & security

types & algebras

$\Gamma \vdash e : \tau$

staging & speed

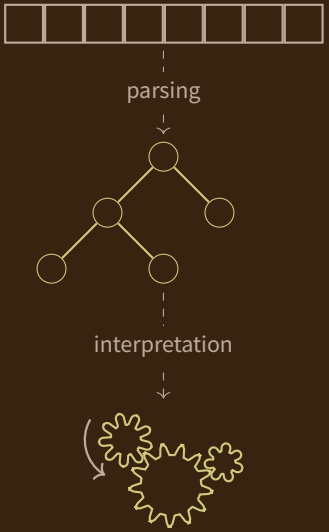
$\ll e \gg$

speed & correctness

✓✓

Parsing and security

parsing & security



Parser combinators: appeal

parsing & security

declarative:
parsers resemble BNF

```
sexp = (lparen >>
        star sexp >>
        rparen)
    ⊕ atom
```

```
sexp ::= LPAREN sexp* RPAREN
       | ATOM
```

simplicity:
parsers are functions

```
star :: Parser a → Parser [a]
star p = ps ⊕ empty
  where
    empty = return []
    ps = do x ← p
            xs ← star p
            return (x : xs)
```

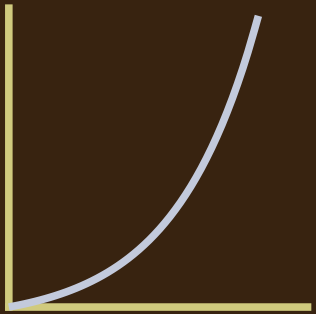
Parser combinators: pitfalls

parsing & security

declarative?
not in practice

$$p \oplus q \not\equiv p \oplus q$$

complexity:
exponential (or worse)



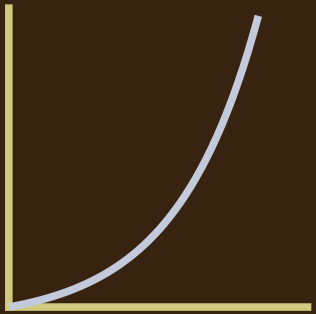
Parser combinators: pitfalls

parsing & security

declarative?
not in practice

$$p \oplus q \not\equiv p \oplus q$$

complexity:
exponential (or worse)



(demonstration)

ASP and its aims

parsing & security

types & algebras

asp: a combinator library with an unusual combination of features

Conventional Interface

Unsurprising Semantics

Guaranteed Determinism

Competitive Performance

ASP: interface

parsing & security

types & algebras

staging & speed

speed & correctness

Abstract grammar interface (context-free expressions)

```
type α t
val chr: char → char t
val eps: unit t
val seq: α t → β t → (α * β) t
val bot: α t
val alt: α t → α t → α t
val fix: (α t → α t) → α t
val map: (α → β) → α t → β t
```

User-defined functions

```
let option r =  alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser: α t → (char Stream.t → α)
```

parsing & security

types & algebras

staging & speed

speed & correctness

Abstract grammar interface (context-free expressions)

```
type α t
val chr: char → char t
val eps: unit t
val seq: α t → β t → (α * β) t
val bot: α t
val alt: α t → α t → α t
val fix: (α t → α t) → α t
val map: (α → β) → α t → β t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r =  alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser: α t → (char Stream.t → α)
```

parsing & security

types & algebras

staging & speed

speed & correctness

Abstract grammar interface (context-free expressions)

```
type α t
val chr: char → char t
val eps: unit t
val seq: α t → β t → (α * β) t
val bot: α t
val alt: α t → α t → α t
val fix: (α t → α t) → α t
val map: (α → β) → α t → β t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r =  alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also `star`, `plus`, `infix`, &c.)

Parsers from grammars

```
val parser: α t → (char Stream.t → α)
```

ASP: interface

parsing & security

types & algebras

Abstract grammar interface (context-free expressions)

```
type α t
val chr: char → char t
val eps: unit t
val seq: α t → β t → (α * β) t
val bot: α t
val alt: α t → α t → α t
val fix: (α t → α t) → α t
val map: (α → β) → α t → β t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r =  alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser: α t → (char Stream.t → α)
```

parsing & security

types & algebras

Abstract grammar interface (context-free expressions)

```
type α t
val chr: char → char t
val eps: unit t
val seq: α t → β t → (α * β) t
val bot: α t
val alt: α t → α t → α t
val fix: (α t → α t) → α t
val map: (α → β) → α t → β t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r =  alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also `star`, `plus`, `infix`, &c.)

Parsers from grammars

```
val parser: α t → (char Stream.t → α)
```

Imperative stream

ASP: interface

parsing & security

types & algebras

Abstract grammar interface (context-free expressions)

```
type α t
val chr: char → char t
val eps: unit t
val seq: α t → β t → (α * β) t
val bot: α t
val alt: α t → α t → α t
val fix: (α t → α t) → α t
val map: (α → β) → α t → β t
```

Real interface: arbitrary tokens

User-defined functions

```
let option r =  alt (map (fun _ → None) eps)
                    (map (fun x → Some x) r)
```

(also star, plus, infix, &c.)

Parsers from grammars

```
val parser: α t → (char Stream.t → α)
```

May fail!

Imperative stream

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism ✗

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism ✗

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism ✗

```
seq (chr 'a')
    (option (chr 'b'))
```

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

sequential non-determinism ✗

(also reject: left recursion, non-left-factored)

accepted or rejected?

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'b'))
```

✓

```
alt (map (fun _ → 1) (chr 'a'))
    (map (fun _ → 2) (chr 'a'))
```

disjunctive non-determinism ✗

```
seq (chr 'a')
    (option (chr 'b'))
```

✓

```
seq (option (chr 'a'))
    (option (chr 'a'))
```

sequential non-determinism ✗

(also reject: left recursion, non-left-factored)

Plan: use a **type system** to decide

parsing & security

types & algebras ⊢ ≡

staging & spec ⇒ ∀

proofs & correctness

Context-free expressions (CFEs)

$$g \quad ::= \quad \bot \quad | \quad g \vee g' \quad | \quad \epsilon \quad | \quad c \quad | \quad g \cdot g' \quad | \quad x \quad | \quad \mu x.\, g$$

Semantics of CFEs

$$\begin{array}{lcl}
[\![\bot]\!]\,\gamma & = & \emptyset \\
[\![g \vee g']\!]\,\gamma & = & [\![g]\!]\,\gamma \;\cup\; [\![g']\!]\,\gamma \\
[\![\epsilon]\!]\,\gamma & = & \{\varepsilon\} \\
[\![c]\!]\,\gamma & = & \{c\} \\
[\![g \cdot g']\!]\,\gamma & = & \{w \cdot w' \mid w \in [\![g]\!]\,\gamma \wedge w' \in [\![g']\!]\,\gamma\} \\
[\![x]\!]\,\gamma & = & \gamma(x) \\
[\![\mu x.\, g]\!]\,\gamma & = & \mathit{fix}(\lambda X.\; [\![g]\!]\,(\gamma, X/x))
\end{array}$$

$$\mathit{fix}(f) \quad = \quad \bigcup_{i \in \mathbb{N}} L_i \text{ where } \begin{array}{lcl} L_0 & = & \emptyset \\ L_{n+1} & = & f(L_n) \end{array}$$

parsing & security

types & algebras ⊢ ≡

staging & speed

speed & correctness

CFEs form an *idempotent semiring*

$$
\begin{aligned}
g_1 \vee (g_2 \vee g_3) &= (g_1 \vee g_2) \vee g_3 \\
g \vee g' &= g' \vee g \\
g \vee \bot &= g \\
g \vee g &= g \\
g_1 \cdot (g_2 \cdot g_3) &= (g_1 \cdot g_2) \cdot g_3 \\
g \cdot \epsilon &= g \\
(g_1 \vee g_2) \cdot g &= (g_1 \cdot g) \vee (g_2 \cdot g) \\
g \cdot (g_1 \vee g_2) &= (g \cdot g_1) \vee (g \cdot g_2) \\
g \cdot \bot &= \bot \\
\bot \cdot g &= \bot
\end{aligned}
$$

(along with some equations for μ)

Types for languages

$$\text{Types} \quad \tau \quad \in \quad \{\, \textsc{Null} : 2;\ \textsc{First} : \mathcal{P}(\Sigma);\ \textsc{FLast} : \mathcal{P}(\Sigma) \,\}$$

$$\tau_1 \vee \tau_2 \quad = \quad \begin{cases} \textsc{Null} & = & \tau_1.\textsc{Null} \vee \tau_2.\textsc{Null} \\ \textsc{First} & = & \tau_1.\textsc{First} \cup \tau_2.\textsc{First} \\ \textsc{FLast} & = & \tau_1.\textsc{FLast} \cup \tau_2.\textsc{FLast} \end{cases}$$

Type predicates

$$\tau_1 \# \tau_2 \quad \triangleq \quad (\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset) \wedge \neg(\tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null})$$

Properties of types

If $L \models \tau$ and $M \models \tau'$ and $\tau \# \tau'$, then $L \cup M \models \tau \vee \tau'$.

parsing & security

types & algebras ⊢ ≡

staging & speed

speed & correctness

Syntactic type system for LL(1) grammars

$$\frac{\Gamma;\Delta \vdash g : \tau \qquad \Gamma;\Delta \vdash g' : \tau' \qquad \tau \# \tau'}{\Gamma;\Delta \vdash g \vee g' : \tau \vee \tau'}$$

Semantic soundness

If $\Gamma;\Delta \vdash g : \tau$ and $\gamma \models \Gamma$ and $\delta \models \Delta$ then $[\![g]\!]\ (\gamma, \delta) \models \tau$

Type inference

No type annotations needed (even for fixed points)

A simple parsing algorithm

$$\mathcal{P}\,(\Gamma;\Delta \vdash g : \tau) \in \mathrm{Env}(\Gamma) \to \mathrm{Env}(\Delta) \to \Sigma^* \rightharpoonup \Sigma^*$$

$$\mathcal{P}\,(\Gamma;\Delta \vdash g \vee g' : \tau \vee \tau')\ \hat{\gamma}\ \hat{\delta}\ [] = \begin{cases} [] & \text{when } (\tau \vee \tau').\textsc{Null} \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{P}\,(\Gamma;\Delta \vdash g \vee g' : \tau \vee \tau')\ \hat{\gamma}\ \hat{\delta}\ ((c :: \_)\ \text{as}\ s) = \begin{cases} \mathcal{P}\,(\Gamma;\Delta \vdash g : \tau)\ \hat{\gamma}\ \hat{\delta}\ s & \text{when } c \in \tau.\textsc{First} \\ & \text{or } \tau.\textsc{Null} \wedge c \notin (\tau \vee \tau').\textsc{First} \\ \mathcal{P}\,(\Gamma;\Delta \vdash g' : \tau')\ \hat{\gamma}\ \hat{\delta}\ s & \text{when } c \in \tau'.\textsc{First} \\ & \text{or } \tau'.\textsc{Null} \wedge c \notin (\tau \vee \tau').\textsc{First} \\ \text{fail} & \text{otherwise} \end{cases}$$

The parsing algorithm is **sound** and **complete**
i.e. it parses exactly the words of the language.

ASP: linear-time guarantee

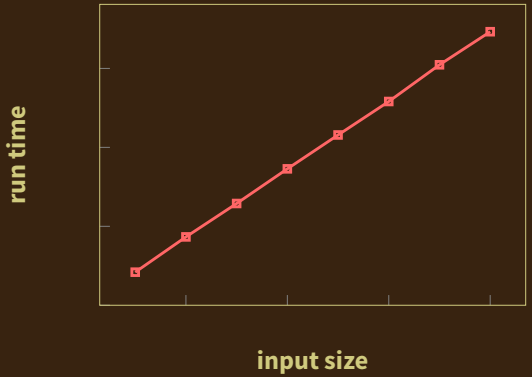
parsing & security

types & algebras

staging & speed

speed & correctness

Guarantee: well-typed parsers don't back-track



ASP: linear-time guarantee

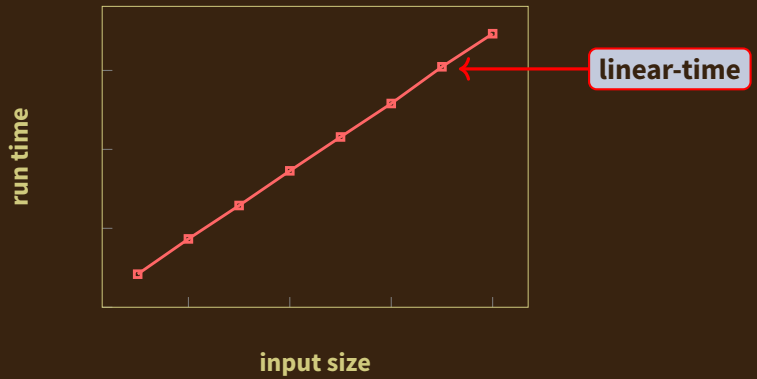
parsing & security

types & algebras

staging & speed

speed & correctness

Guarantee: well-typed parsers don't back-track

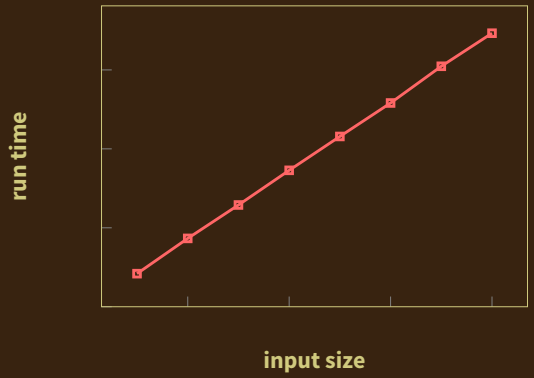


Speed: combinators vs yacc

parsing & security

types & algebras

staging & speed $\ll e \gg$

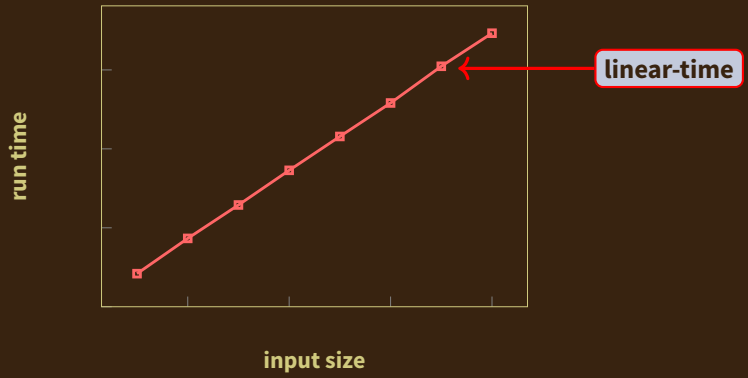


Speed: combinators vs yacc

parsing & security

types & algebras

staging & speed ≪e≫

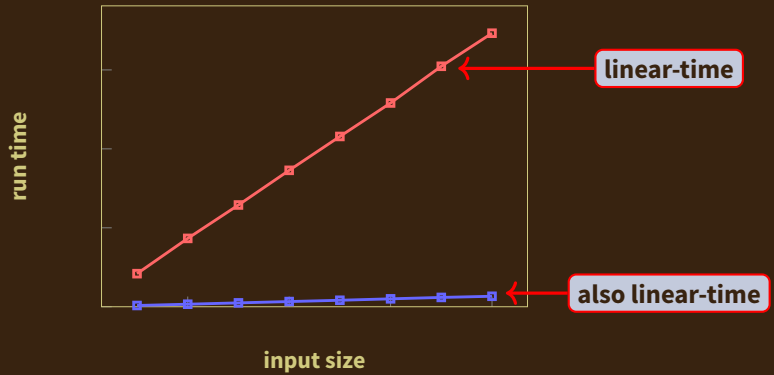


Speed: combinators vs yacc

parsing & security

types & algebras ⊢ ≡

staging & speed ≪e≫

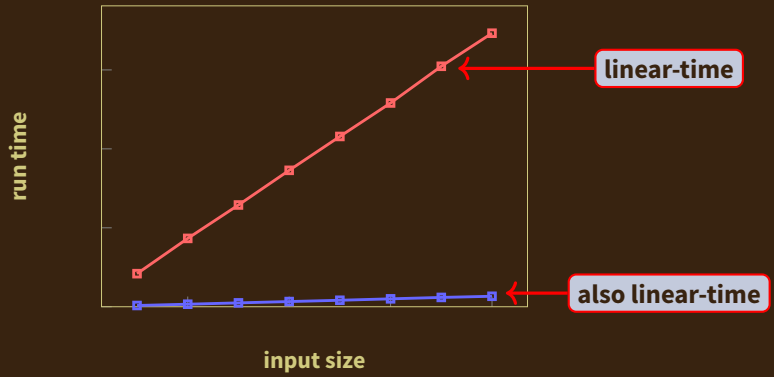


Speed: combinators vs yacc

parsing & security

types & algebras

staging & speed ≪e≫



How can we close the gap? **Staging**

staging removes overhead

parsing & security

types & algebras $\vdash \equiv$

staging & speed $\ll e \gg$

speed & correctness

abstraction $\Longrightarrow$ overhead

Parser **combinators abstract** over the grammar

Abstraction carries a **performance penalty**

abstraction + staging $\Longrightarrow$ no overhead

Use **staging** to specialize code once grammar is known

Delay (quote) code that accesses the input stream

$\ll$ peek stream == 'a' $\gg$

Evaluate code that depends only on the grammar

staging removes overhead

parsing & security

types & algebras $\vdash \equiv$

staging & speed $\ll e \gg$

abstraction $\Longrightarrow$ overhead

Parser **combinators abstract** over the grammar

Abstraction carries a **performance penalty**

abstraction + staging $\Longrightarrow$ no overhead

Use **staging** to specialize code once grammar is known

Delay (quote) code that accesses the input stream
$\ll$ peek stream == 'a' $\gg$

Evaluate code that depends only on the grammar

less-naive staging

parsing & security

types & algebras ⊢ ≡

staging & speed ⟪e⟫

Binding-time improvements turn dynamic terms static

```
if peek stream == 'a' then e
```

CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```

```
f c
```

c is a char

```
match c with
| 'a' → f 'a'
| 'b' → f 'b'
| 'c' → f 'c'
| ...
```

prune using types

```
match c with
| 'a' → f 'a'
| _   → f 'c'
```

less-naive staging

parsing & security

types & algebras ⊢ ≡

staging & speed ≪e≫

speed & correctness

Binding-time improvements turn dynamic terms static

```
if peek stream == 'a' then e
```

CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```

```
f c
```

c is a char

```
match c with
| 'a' → f 'a'
| 'b' → f 'b'
| 'c' → f 'c'
| ...
```

prune using types

```
match c with
| 'a' → f 'a'
| _   → f 'c'
```

less-naive staging

parsing & security

types & algebras ⊢ ≡

staging & speed ≪e≫

less-naive staging

parsing & security

types & algebras ⊢ ≡

staging & speed ≪e≫

less-naive staging

parsing & security

types & algebras ⊢ ≡

staging & speed ≪e≫

Binding-time improvements turn dynamic terms static

```
if peek stream == 'a' then e
```

CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```

```
f c
```

c is a char

```
match c with
| 'a' → f 'a'
| 'b' → f 'b'
| 'c' → f 'c'
| ...
```

prune using types

```
match c with
| 'a' → f 'a'
| _   → f 'c'
```

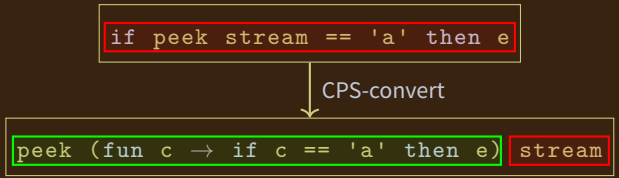
parsing & security

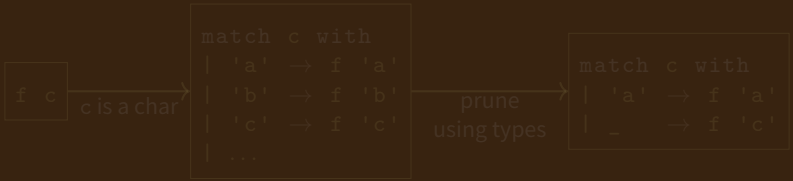
types & algebras ⊢ ≡

staging & speed ≪e≫

speed & correctness ✓

Binding-time improvements turn dynamic terms static





parsing & security

types & algebras

staging & speed ≪e≫

less-naive staging

parsing & security

types & algebras ⊢ ≡

staging & speed ≪e≫

less-naive staging

parsing & security

types & algebras

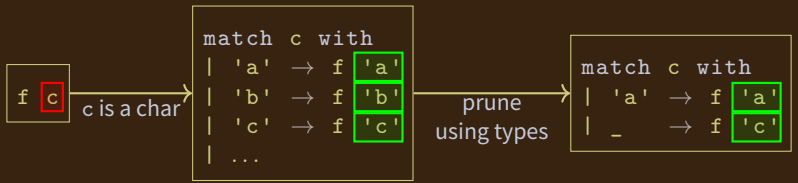
staging & speed ≪e≫

Binding-time improvements turn dynamic terms static

```
if peek stream == 'a' then e
```

CPS-convert

```
peek (fun c → if c == 'a' then e) stream
```

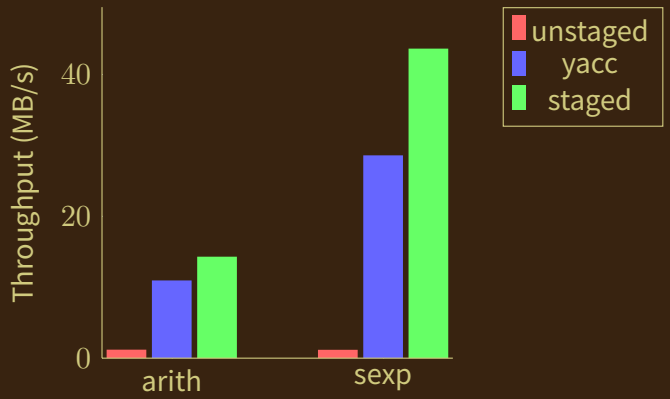


faster than yacc!

parsing & security

types & algebras $\vdash \equiv$

staging & speed $\ll e \gg$



Future: heterogeneous staging

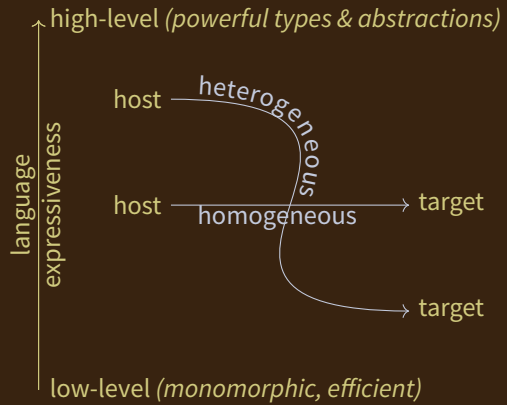
parsing & security

types & algebras ⊢ ≡

staging & speed ≪e≫

speed & correctness ✓✓

Homogeneity limits both guarantees and performance



Future: verified staging

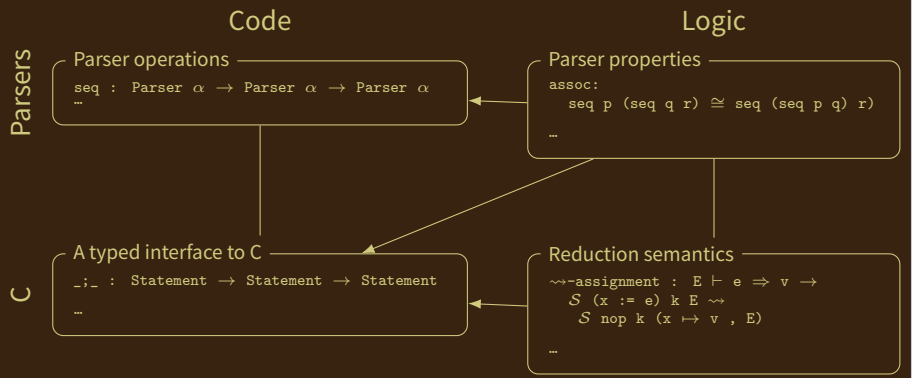
parsing & security

types & algebras $\vdash$ $\equiv$

staging & speed $\ll e \gg$

speed & correctness ✓✓

Challenge: verifying a staged program




parsing & security
types & algebras
$\Gamma \vdash e : \tau$
staging & speed
$\ll e \gg$
speed & correctness
✓✓