On the Expressive Power of Programming Languages
Historical Context

Parametricity (1983)

R3R Scheme (1986)

Control delimiters (1990)

This paper (1991)

Reduction Semantics (1992)

Progress and Preservation (1994)

DrScheme (1997)

Revenge of the Son of the LISP Machine (1999)
Historical Context

1991: Writes this paper

1994: Shriram pivots from CompBio after reading it
1991: Writes this paper

1994: Shriram pivots from CompBio after reading it

2010: Essence of JavaScript

2019: nothing of note.
Expressivity

Language $L'$
Expressivity

Constructs $F$

Language $L'$

Language $L$
Expressivity

Language L

Language L'
Expressivity

Constructs $F$

$\phi$ (compiler)

Language $L'$
Expressivity

\[
\text{Let } x = \text{init} \ \text{in} \ \text{body}
\]

\[
\text{(fun } x \rightarrow \text{body) } \text{init}
\]

\[
\text{Let } x = \text{ref 0} \ \text{in} \ x++
\]

\[
\text{Let } x = \text{makeBox} () \ \text{in} \ x.\text{setRef}(x.\text{getRef} + 1)
\]
Eliminable Constructs
$E_1 \varphi(e)$ is an $\mathcal{L}'$-program for all $\mathcal{L}$-programs $e$;
E3 $eval_{\mathcal{L}}(e)$ holds if and only if $eval_{\mathcal{L}'}(\varphi(e))$ holds for all $\mathcal{L}$-programs $e$. 
E2 $\varphi(F(e_1, \ldots, e_a)) = F(\varphi(e_1), \ldots, \varphi(e_a))$ for all facilities $F$ of $\mathcal{L}'$, i.e., $\varphi$ is homomorphic in all constructs of $\mathcal{L}'$; and
Let $x = \text{init}$ in body

$\phi(\text{Let } x = \text{init } \text{in } \text{body}) \Rightarrow (\text{fun } x \rightarrow \phi(\text{body})) \phi(\text{init})$

(fun $x \rightarrow \text{body}$) $\text{init}$

Eliminable: Example
Contextual Equivalence

Or Observational Equivalence
Definition 3.5. (Operational Equivalence) Let $\mathcal{L}$ be a programming language and let $eval_{\mathcal{L}}$ be its operational semantics. The $\mathcal{L}$-phrases $e_1$ and $e_2$ are operationally equivalent, $e_1 \cong_{\mathcal{L}} e_2$, if there are contexts that are program contexts for both $e_1$ and $e_2$, and if for all such contexts, $C(\alpha)$, $eval_{\mathcal{L}}(C(e_1))$ holds if and only if $eval_{\mathcal{L}}(C(e_2))$ holds.
Definition 3.5. (*Operational Equivalence*) Let $\mathcal{L}$ be a programming language and let $eval_\mathcal{L}$ be its operational semantics. The $\mathcal{L}$-phrases $e_1$ and $e_2$ are *operationally equivalent*, $e_1 \equiv_\mathcal{L} e_2$, if there are contexts that are program contexts for both $e_1$ and $e_2$, and if for all such contexts, $C(\alpha)$, $eval_\mathcal{L}(C(e_1))$ holds if and only if $eval_\mathcal{L}(C(e_2))$ holds.
Definition 3.5. (*Operational Equivalence*) Let $\mathcal{L}$ be a programming language and let $\text{eval}_{\mathcal{L}}$ be its operational semantics. The $\mathcal{L}$-phrases $e_1$ and $e_2$ are operationally equivalent, $e_1 \cong_{\mathcal{L}} e_2$, if there are contexts that are program contexts for both $e_1$ and $e_2$, and if for all such contexts, $C(\alpha)$, $\text{eval}_{\mathcal{L}}(C(e_1))$ holds if and only if $\text{eval}_{\mathcal{L}}(C(e_2))$ holds.
Contextual Equivalence: Example

\( C(a) = (\text{fun } x, y \rightarrow a) \ 1 \ (\text{throw } 1) \)
Contextual Equivalence: Example

\[(\text{fun } x, y \to x) \ (\text{fun } x \to x) \ (\text{throw } 1)\]

\[(\text{fun } x, y \to y) \ (\text{fun } x \to x) \ (\text{throw } 1)\]

1

(throw 1)

⊥
Expressivity
Constructs $F$ is expressible in Language $L$ if

$\phi$ satisfies $E_1, E_2, E_3$

$F$ is eliminable

There is no *distinguishing context* for $F$ and $\phi(F)$. 
Macro-expressivity

**E4** For each $a$-ary construct $F \in \{F_1, \ldots, F_n, \ldots\}$ there exists an $a$-ary syntactic abstraction, $A$, over $\mathcal{L}'$ such that

$$\varphi(F(e_1, \ldots, e_a)) = A(\varphi(e_1), \ldots, \varphi(e_a)).$$
For each $a$-ary construct $F \in \{F_1, \ldots, F_n, \ldots\}$ there exists an $a$-ary syntactic abstraction, $A$, over $\mathcal{L}'$ such that

$$\varphi(F(e_1, \ldots, e_a)) = A(\varphi(e_1), \ldots, \varphi(e_a)).$$

**For (init, test, update, body)**

**While (test, body)**

Macro expressivity: Example
E4 For each $a$-ary construct $F \in \{F_1, \ldots, F_n, \ldots\}$ there exists an $a$-ary syntactic abstraction, $A$, over $\mathcal{L}'$ such that

$$\varphi(F(e_1, \ldots, e_a)) = A(\varphi(e_1), \ldots, \varphi(e_a)).$$

**For** (init, test, update, body)

For (init, test, upd, body) => init **in** While ($\phi$(test), $\phi$(body) ++ $\phi$(update))

**While** (test, body)

Macro expressivity: Example
Expressive but Macro-inexpressive

\[ A \vdash e : \tau ; A \vdash b[x/e] : \tau' \]

\[ A \vdash \text{let } x = e \text{ in } b : \tau' \]

Polymorphic let

Call-by-value STLC
Expressive but Macro-inexpressive

Polymorphic let

\[
\frac{A \vdash e : \tau; A \vdash b[x/e] : \tau'}{A \vdash \text{let } x = e \text{ in } b : \tau'}
\]

\[
\text{Let } (x, e, b) \Rightarrow (\text{fun } x \rightarrow \text{subst}(x, \phi(e), \phi(b))) \phi(e)
\]

Call-by-value STLC

Expressive!
Let \((x, e, b) => (\text{fun } x -> \text{subst}(x, \phi(e), \phi(b))) \phi(e)\) 

AST function, not a \textit{syntactic abstraction}!

Recursive macros are not a problem! Macro-based \texttt{subst} implementation will generate scoped macros. \texttt{subst} is truly performing a \textit{compile-time computation}.

Expressive but Macro-inexpressive
What do we get?

Eliminability

(Macro) expressivity
What do we get?

Eliminability

(Macro) expressivity

State

Continuations

Inexpressible

Lambda Calculus
What do we get?

Eliminability

(Macro) expressivity

State

Continuations

Bad?

Lambda Calculus
Case Study
let x = {
  a: 10,
  b: 20,
}

> { a: 10, b: 20 }

with (x) {
  a + b + 10
}

> 30

* The Essence of JavaScript (2010); Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi
let x = {
    a: 10,
    b: 20,
}

> { a: 10, b: 20 }

with (x) {
    a + b + 10
}

> 30

Lambda Calculus + objects

* The Essence of JavaScript (2010); Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi
let x = {
    a: 10,
    b: 20,
}

> { a: 10, b: 20 }

with (x) {
    a + b + 10
}

> 40

Lambda Calculus + objects

Not macro expressible

* The Essence of JavaScript (2010); Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi
Thanks!

(call/cc
  (lambda (return)
    (while (true)
      (return "Power Overwhelming!")))))
Discussion points

- Expressivity as a language design principle vs type directed language design.
- Why is this not the prevailing way of designing languages?
- Programming languages: isolated mathematical formalisms or complete ecosystems?

Put differently, interactive programming systems actually add expressive power to the programming language. Peter Lee [personal communication] pointed out another example of this phenomenon: The addition of a read-eval-print loop also introduces true, non-eliminable polymorphism into a language like $\Sigma + \text{let}$ by providing top-level let declarations with an open-ended body expression. The fact that such interactive programming environments add power to their underlying languages suggests that they should be specified as a part of the language standards!