On the expressive power of programming languages

Matthias Felleisen*
Department of Computer Science, Rice University, Houston, TX 77251-1892, USA

Communicated by N. D. Jones
Revised March 1991

Abstract

Felleisen, M., On the expressive power of programming languages, Science of Computer Programming 17 (1991) 35–75.

Presented by

Shriram Krishnamurthi
Brown University

What is this paper about?

We have sharp, mathematical distinctions between some classes of language features

These offer advice to language designers

Beyond a point (the “Turing Threshold”), we have none…

A Little Personal History

What’s Expressive?

$$\mathcal{L}$$

$$+$$

$$\mathcal{F}$$

Loop/Function-Free Language
+
Loops

`while` language

\+

`for`

for language

+

while

Regular language

+

Context-free grammar

Two-Armed `if`

+

Multi-Armed `if`

Pure Language

+

State

Language w/ binary –

+

Unary –

Language w/out Exceptions

+

`halt`

$\mathcal{L}$ $\mathcal{L} + \mathcal{F}$

Does $\mathcal{F}$ add expressive power to $\mathcal{L}$?

Intuition:
something about compilation

If we can translate/compile
$\mathcal{L}$ + $\mathcal{F}$ down to $\mathcal{L}$,
$\mathcal{F}$ is probably not expressive…

JavaScript

(Church)

x86

(Turing)

The genius of this paper:

Extracting us from the
“Turing tar-pit” (Perlis)

Better intuition:
local translation

"local" means "nobody else needs to know"

```
(for i lo hi body) ➔ (define i lo)
                     (while (< i hi)
                       body
                       (increment i))
```

Essentially, simple Lispy macro systems

(The Las Vegas principle)

```
x + 2
```

```
x.__add__(2)
```

```
[x*x | x <- l]
```

```
map (\x -> x*x) l
```

```
x or y
```

```
let t = x in
  if t then t else y
```

Two Sides of Expressiveness

$\mathcal{L}$ $\mathcal{L} + \mathcal{F}$

When is $\mathcal{F}$ not expressive relative to $\mathcal{L}$?

When a macro for $\mathcal{F}$ to $\mathcal{L}$ exists!

Given semantics for $\mathcal{L}$ and $\mathcal{L}$ + $\mathcal{F}$,
for *all* programs P in $\mathcal{L}$ + $\mathcal{F}$:
say $P_{\mathcal{L}}$ (in $\mathcal{L}$) is the result of "expanding" $\mathcal{F}$
then P (in $\mathcal{L}$ + $\mathcal{F}$) is "equal" to $P_{\mathcal{L}}$ (in $\mathcal{L}$)

$\mathcal{L}$ $\mathcal{L} + \mathcal{F}$

When is $\mathcal{F}$ expressive relative to $\mathcal{L}$?

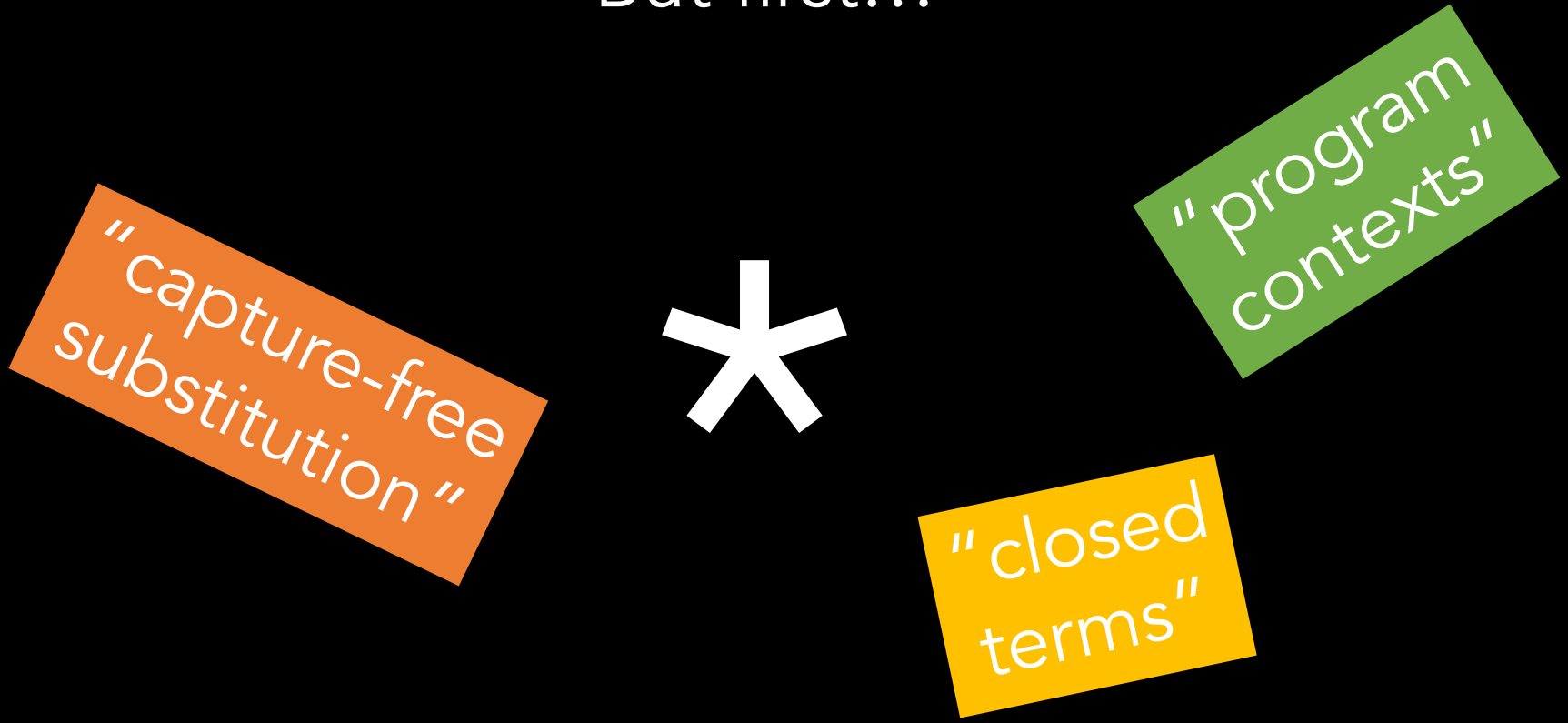
Need to show that
no macro can possibly exist

That is the interesting part of this paper!

Both parts rely on a definition of “equality”

Equality is Hard

But first…



Equality is hard…
Counterpoint: equality is easy!

For two expressions e_1 and e_2,
run(e_1) → v_1
run(e_2) → v_2
compare v_1 and v_2

Comparing as strings

What about
1 and 0.9999999…?

What about functions?

```
(λ (x) (* 2 x))
(λ (x) (+ x x))
```

What about free variables?

```
(* x 3)
(+ x x x)
```

What about closures?

Not just code,
also environments, etc.

What if you can measure
running time?
power consumption?
...

These aren't just #TheoryWorldProblems…

Every compiler optimization needs to replace terms with ones "equal" to them

Observational equivalence

LAMBDA-CALCULUS MODELS OF PROGRAMMING LANGUAGES

James H. Morris

Is there a way in the language
of telling the two answers apart?

If so, some program might use it!

E :: = v
| c
| (op E …)
| (E E …)
| (λ (v …) E)

What are all the ways of using a piece of code in a program?

We'll treat ourselves to more language…

```
(+ 1 •)
(f x • y)
(λ (x) (+ x •))
```

A context C[•] is an expression E with some sub-expression replaced with a •

For all contexts C,
if $C[e_1] = C[e_2]$,
$e_1 \cong e_2$

Can you in code (i.e., with a context):

- … tell apart 1 and 0.999…?
- … inspect program source?
- … measure time/power?

more "observations" ➔ fewer equivalences

Even more general definition:

$e_1 \cong e_2$ if,
for all contexts C,
$C[e_1]$ halts iff $C[e_2]$ halts

Ω is a canonical non-terminating term

A small "trick": programs with errors don't terminate

$e_1 \cong_{\mathcal{A}} e_2$ if,
for all contexts C in language $\mathcal{A}$,
$C[e_1]$ halts iff $C[e_2]$ halts

Is this okay?

Doesn’t this imply `5` ≅ `6`?

C[•] ≜ (`if (=? • 5) “halt” Ω`)

The definition is for all contexts,
and that’s one of ’em!

What if `(=? • 5)` is not a language operation?
Maybe it has only `0?`

C[•] ≜ `(if (0? (- • 5)) “halt” Ω)`

What if the language doesn’t have <u>any</u> way
of telling `5` and `6` apart?

Back to Expressiveness!

Motivation for the definition

Suppose we have
$3 \cong_{\mathcal{L}} (+\ 1\ 2)$

Could adding $\mathcal{F}0$ leave $3 \cong_{\mathcal{L}+\mathcal{F}0} (+\ 1\ 2)$?
Could an $\mathcal{F}1$ leave <u>all</u> $\cong_{\mathcal{L}}$ pairs $\cong_{\mathcal{L}+\mathcal{F}1}$?
Could adding $\mathcal{F}2$ make $3 \ncong_{\mathcal{L}+\mathcal{F}2} (+\ 1\ 2)$?

Key Theorem

Suppose $\mathcal{F}$ can be written as a local macro
Then for all e_1 and e_2
such that $e_1 \cong_{\mathcal{L}} e_2$,
$e_1 \cong_{\mathcal{L} + \mathcal{F}} e_2$

That is, $\mathcal{F}$ has not added power to $\mathcal{L}$

Theorem 3.14. *Let* $\mathcal{L}_1 = \mathcal{L}_0 + \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ *be a conservative extension of* $\mathcal{L}_0$. *Let* $\cong_0$ *and* $\cong_1$ *be the operational equivalence relations of* $\mathcal{L}_0$ *and* $\mathcal{L}_1$, *respectively.*

(i) *If the operational equivalence relation of* $\mathcal{L}_1$ *restricted to* $\mathcal{L}_0$ *expressions is not equal to the operational equivalence relation of* $\mathcal{L}_0$, *i.e.,* $\cong_0 \neq (\cong_1 | \mathcal{L}_0)$, *then* $\mathcal{L}_0$ *cannot macro-express the facilities* $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$.[9]

(ii) *The converse of* (i) *does not hold. That is, there are cases where* $\mathcal{L}_0$ *cannot express some facilities* $\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots$, *even though the operational equivalence relation of* $\mathcal{L}_1$ *restricted to* $\mathcal{L}_0$ *is identical to the operational equivalence relation of* $\mathcal{L}_0$, *i.e.,* $\cong_0 = (\cong_1 | \mathcal{L}_0)$.

How to show expressiveness?

Start with $\mathcal{L}$ terms e_1 and e_2 such that $e_1 \cong_{\mathcal{L}} e_2$

If we can find a C in $\mathcal{L} + \mathcal{F}$
that can distinguish e_1 from e_2
(i.e., $e_1 \ncong_{\mathcal{L}+\mathcal{F}} e_2$)

Then, $\mathcal{F}$ has added power to $\mathcal{L}$
(and can't be expressed as a local macro)

Language w/out Exceptions

+

`halt`

Reminder

Start with $\mathcal{L}$ terms e_1 and e_2 such that $e_1 \cong_{\mathcal{L}} e_2$

If we can find a C in $\mathcal{L}$ + `halt`
that can distinguish e_1 from e_2
(i.e., $e_1 \ncong_{\mathcal{L} + \texttt{halt}} e_2$)

…

$e_1 \triangleq$ `(λ (f) Ω)`

$e_2 \triangleq$ `(λ (f) ((f 0) Ω))`

$C[\bullet] \triangleq$ `(• halt)`

$C[e_1] = \Omega$

$C[e_2] = \texttt{0}$

$e_1 \triangleq$ `(λ (f) Ω)`

$e_2 \triangleq$ `(λ (f) ((f 0) Ω))`

$C[\bullet] \triangleq$ `(call/cc •)`

$C[e_1] = \Omega$

$C[e_2] =$ `0`

Pure Language

+

State

$e_1 \triangleq$ `(λ (_) (f 0))`

$e_2 \triangleq$ `(λ (_) (f 0) (f 0))`

$C[\bullet] \triangleq$

```
(define (f x)
  (set! f (λ (_) Ω))
  x))
(• 0)
```

take a parameter and return it changing `f` to a diverging function

$C[e_1] = 0$

$C[e_2] = \Omega$

Pure w/ Boolean-only `if` (`Bif`)

+

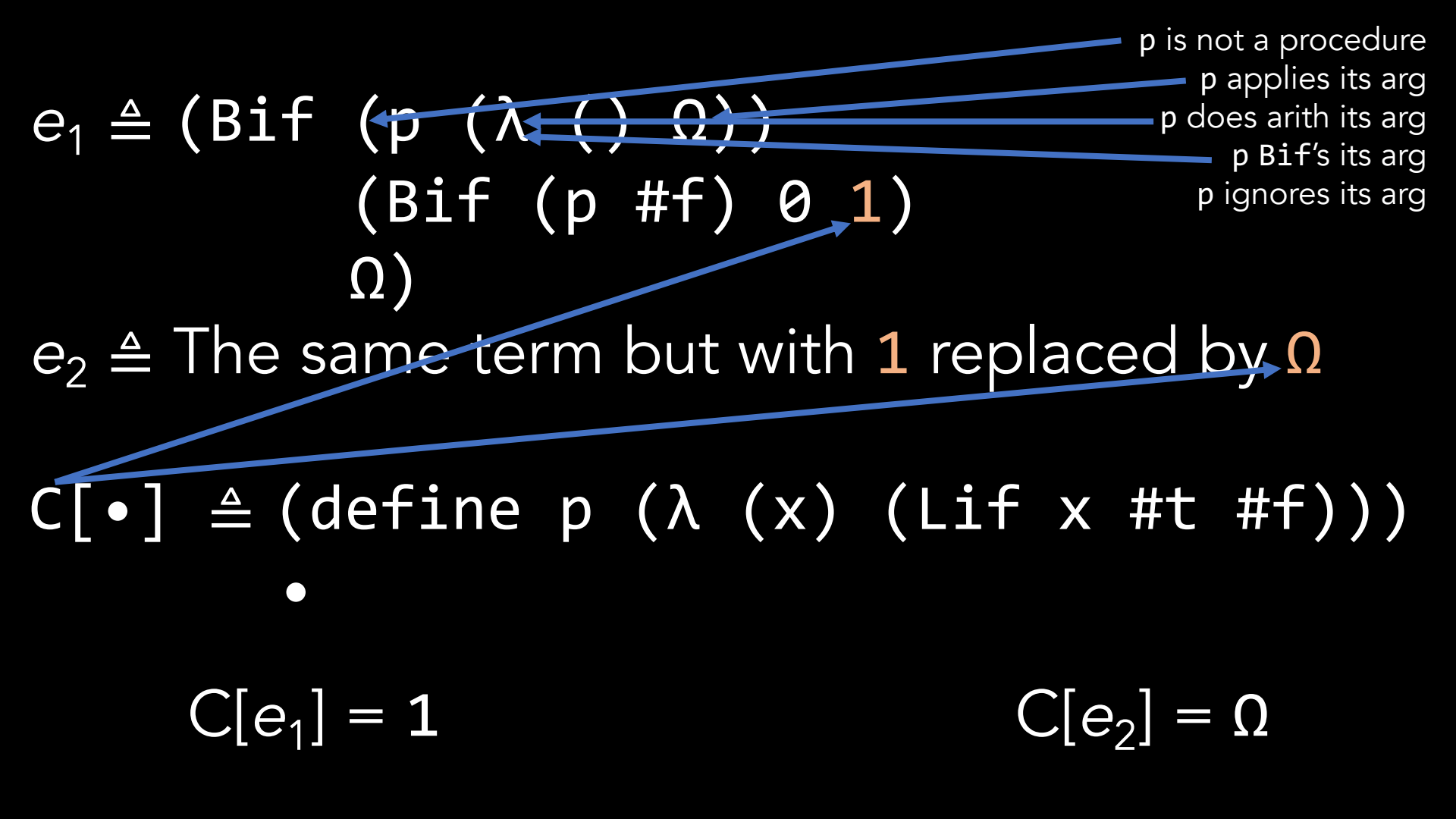
Truthy/Falsy `if` (`Lif`)

Semantics of `Lif`:

`(Lif #f` A B) → B

`(Lif` <any other value> A B) → A

(Similar argument for other truthy/falsy)


p is not a procedure
p applies its arg
p does arith its arg
p Bif's its arg
p ignores its arg
e1 ≜ (Bif (p (λ () Ω))
(Bif (p #f) 0 1)
Ω)
e2 ≜ The same term but with 1 replaced by Ω
C[•] ≜ (define p (λ (x) (Lif x #t #f)))
•
C[e1] = 1
C[e2] = Ω

Local vs global transformations

pure language + state: store-passing style

control operators: continuation-passing style

They can't be local transformations
They do add expressive power

Wrapping Up

What have we learned?

A beautiful, practical definition of equality

A clever definition of expressiveness

Proof sketches that show us it matches intuition

What else is in the paper?

Multiple notions of expressiveness

Proof of *that* theorem (not trivial at all!)

Relationship to logic

Relationship to other formalizations

Implications for language design

Desugaring/macros are now everywhere

Macros can have a variety of powers

Allowing macros that increase expressiveness…
is something to be done with care

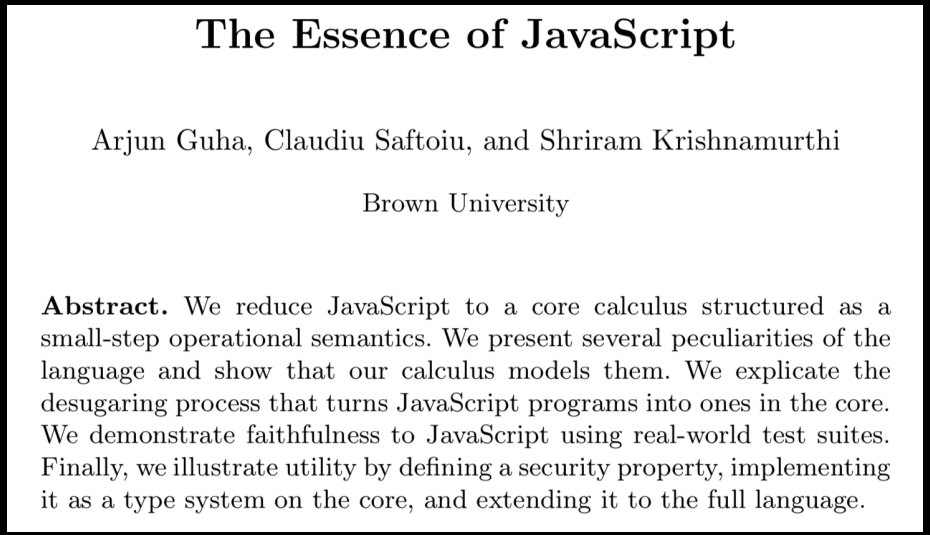
Benefits of expressive features

Avoids non-local/global transformation

Greater modularity
otherwise
Patterns…which can be misused
Patterns…which hide intent

Homework

Is the `with` construct of JavaScript expressive w.r.t. the rest of JavaScript?



The Essence of JavaScript

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Brown University

Abstract. We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

Homework

Is the generator construct of Python expressive w.r.t. the rest of Python?

Python: The Full Monty
A Tested Semantics for the Python Programming Language

Joe Gibbs Politz
Providence, RI, USA
joe@cs.brown.edu

Alejandro Martinez
La Plata, BA, Argentina
amtriathlon@gmail.com

Matthew Milano
Providence, RI, USA
matthew@cs.brown.edu

Sumner Warren
Providence, RI, USA
jswarren@cs.brown.edu

Daniel Patterson
Providence, RI, USA
dbpatter@cs.brown.edu

Junsong Li
Beijing, China
ljs.darkfish@gmail.com

Anand Chitipothu
Bangalore, India
anandology@gmail.com

Shriram Krishnamurthi
Providence, RI, USA
sk@cs.brown.edu