Language Expressiveness

Jonathan Aldrich

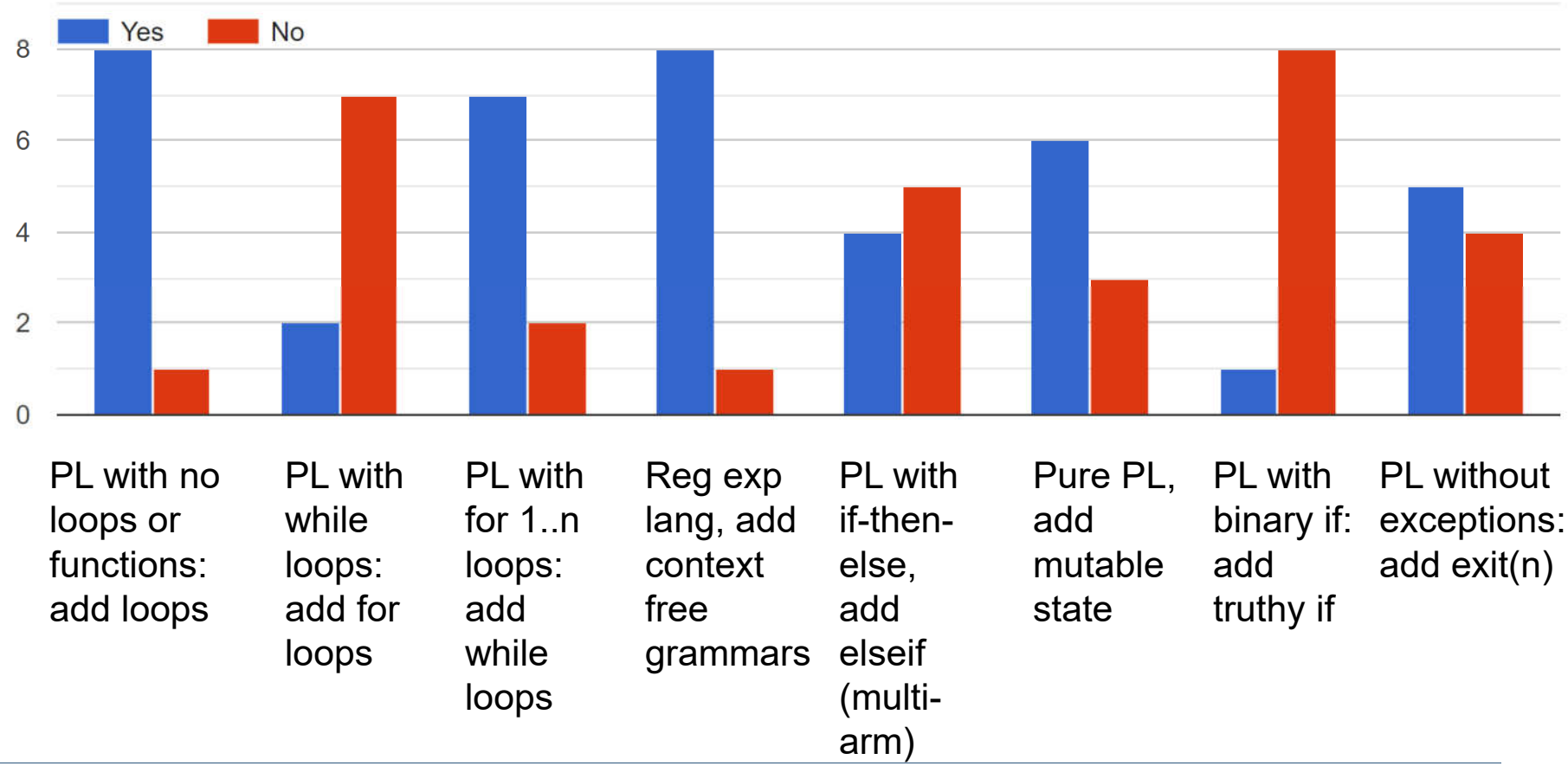
17-396/17-696/17-960: Language Design and Prototyping

Carnegie Mellon University

Acknowledgment: in addition to the sources in the bibliography slide, some of my presentation ideas and structure are inspired by Shriram Krishnamurthi's excellent Papers We Love 2019 talk: https://pwlconf.org/2019/shriram-krishnamurthi/

Results

Starting with the specified language, does the feature to be added result in greater expressiveness?



How to compare language expressiveness?

- One approach: what problems can the language solve?
- Draw from Automata theory
  - Finite state machines – linear time – e.g. parsing regular exps
  - Pushdown automata – cubic time – e.g. parsing context-free langs
  - Linear-bounded automata – exp. time – e.g. parsing cxt-sens langs
  - Turing machines – undecidable – general computation

Most real PLs fall in this category.
So does this actually answer the question?

Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy – Alan Perlis

Compilation Intuition

- If L+F can be compiled to L, then F probably doesn't add expressiveness
- But *by definition*, any Turing-complete language can be compiled to any other
  - So let's refine the intuition above
- If L+F can be *locally* compiled to L, then F probably doesn't add expressiveness
  - Local = doesn't affect parts of F or the surrounding context
  - Essentially, Lisp-style macros

This idea is explored in the paper *On the Expressive Power of Programming Languages* by Matthias Felleisen.

Local Translations

for i **in** n..m { S }	→	i **:=** n; **while** i<m { S; i **:=** i + 1 }
x + 2	→	x.__add__(2)
[x*x | x **<-** list]	→	map (\x **->** x*x) list

History: Kleene's Eliminable Symbols

- If you extend a logic L with a symbol $\otimes$, does $\otimes$ increase the expressiveness of the logic?

- F is *eliminable* if there is a mapping m from $L+\otimes$ to L s.t.
  - m is the identity on L
  - m is homomorphic in $\otimes$
    - Roughly, $m(f \otimes g) = m(f) \otimes m(g)$
  - If a formula f is true in $L+\otimes$, then $m(f)$ is true in L

    (note: I've made some minor simplifications)

- We can do something similar for programming languages

Felleisen's Eliminable PL Facilities

- Let L be a programming language, and L' be a conservative restriction that removes constructor F. Then F is *eliminable* if there is a computable mapping $\varphi : L \rightarrow L'$ such that
  - $\forall e \,.\, e \in L \Rightarrow \varphi(e) \in L'$
    (φ maps one language to the other)
  - For all constructors $C \in L'$ we have $\varphi(C(e_1, .., e_n)) = C(\varphi(e_1), .., \varphi(e_n))$
    (φ is homomorphic in all constructs of L' – thus φ is the identity on L')
  - $\forall e \in L \,.\, eval_L(e)$ holds iff $eval_{L'}(\varphi(e))$ holds
    (φ preserves semantics – technically e terminates whenever $\varphi(e)$ does)

Felleisen's Eliminable PL Facilities

- Let L be a programming language, and L' be a conservative restriction that removes constructor F. Then F is *eliminable* if there is a computable mapping $\varphi : L \rightarrow L'$ such that
  - $\forall e \,.\, e \in L \Rightarrow \varphi(e) \in L'$
    (φ maps one language to the other)
  - For all constructors $C \in L'$ we have $\varphi(C(e_1, .., e_n)) = C(\varphi(e_1), .., \varphi(e_n))$
    (φ is homomorphic in all constructs of L' – thus φ is the identity on L')
  - $\forall e \in L \,.\, eval_L(e)$ holds iff $eval_{L'}(\varphi(e))$ holds
    (φ preserves semantics)
- Can strengthen to *macro-eliminable* with one more condition:
  - $\varphi(F(e_1, .., e_n)) = A(\varphi(e_1), .., \varphi(e_n))$ for some syntactic abstraction A

Non-Expressiveness

- A feature F of a language does not add expressiveness if F is eliminable
  - or macro-eliminable, if you like

- **let** is macro-eliminable in the lambda calculus
  - $\varphi(\textbf{let } x = e_1 \textbf{ in } e_2) = (\lambda x.e_2)(e_1)$

Expressiveness

- A feature F adds expressiveness if F is not eliminable
- Need to show that no macro (or, more generally, no homomorphic mapping) can exist.
  - How do you show that?
- General strategy
  - Theorem: Let $L=L'+F$ be a conservative extension of L'.
    If $\forall \varphi : L \rightarrow L'$ that are homomorphic in all constructors of L',
    $\exists e_1..e_n$ such that $F(e_1, .., e_n) \neq \varphi(F(e_1, .., e_n))$,
    then L' cannot express L

What does (in)equality of programs mean?

Thinking About Program Equality

- One attempt to assess whether $e_1 = e_2$
  - $e_1 \rightarrow^* v_1$
  - $e_2 \rightarrow^* v_2$
  - Now see if $v_1 = v_2$
    - Easy for numbers
    - For strings: object equality? Value equality?
    - What about functions?
    - What about closures?
    - By the way, does execution time matter? Power?

- This is hard!
  - And compiler writers have to think about it all the time
    - Does my optimization preserve meaning?

Observational Equivalence

- Are two expressions e_1 and e_2 equal?
  - Jim Morris (1969): can any program tell them apart?
- A *context* C[•] is an expression e with some sub-expression replace with a *hole* •.
- Observational equivalence: $e_1 \cong e_2$ iff $\forall C, C[e_1]=C[e_2]$

We are still using program equivalence!

- Solution 1: evaluate to values, and compare primitives
  - Ok to avoid comparing functions because we are quantifying over all C's, so we can look at function behavior
- Solution 2: just consider termination

Observational Equivalence

- Are two expressions e_1 and e_2 equal?
  - Jim Morris (1969): can any program tell them apart?
- A *context* C[•] is an expression e with some sub-expression replace with a *hole* •.
- Observational equivalence (revised/simplified):

  $e_1 \cong e_2$ iff $\forall$C, C[e_1] halts whenever C[e_2] halts

  - Disadvantage: a bit theoretical, not decidable
  - Advantage: fully general – even works for the lambda calculus, which has no primitive values

Theorem, revised

- Theorem: Let $L=L'+F$ be a conservative extension of L'. If $\forall \varphi : L \rightarrow L'$ that are homomorphic in all constructors of L', $\exists e_1..e_n$ such that $F(e_1, .., e_n) \neq \varphi(F(e_1, .., e_n))$, **and there is a context C that witnesses this inequality,** then L' cannot express L

Using the Theorem

- Consider the lambda calculus with call-by-name and call-by-value
  - $\lambda_{\mathbf{v}} x.e$ – when called, we evaluate the argument and substitute
  - $\lambda_{\mathbf{n}} x.e$ – when called, we substitute the argument without evaluating it first

- Theorem: Neither call-by-name nor call-by-value is eliminable (in terms of the other)
  - Call-by-value is not eliminable – see Felleisen's paper
    - Complicated argument, but ultimately you can't force order of evaluation without a global rewriting.

Using the Theorem

- Theorem: Neither call-by-name nor call-by-value is eliminable (in terms of the other)
  - Call-by-value is not eliminable – see Felleisen's paper
  - **Call-by-name is not eliminable**
    - Consider Ω to be a diverging program (definition is in Felleisen)
    - Take $C(\alpha) = (\alpha(\Omega))$, $e = \lambda_n x.(\lambda_n x.x)$
    - Assume φ is a homomorphic translation
    - $C(\lambda_n x.(\lambda_n x.x)) = (\lambda_n x.(\lambda_n x.x))(\Omega) \rightarrow \lambda_n x.x$
    - But $C(\varphi(e)) = \varphi(e)(\Omega)$ which must diverge
    - So no φ can exist with the right characteristics

Expressiveness and Equivalence

- Expressive language features can also break existing equivalences
- Consider an example due to Krishnamurthi
  - $3 \cong_L 1 + 2$
  - Could a language feature leave this equivalence in place?
  - Could a language feature violate this equivalence?

Theorem: violating equalities adds power

- Theorem: Let $L_1 = L_0 + F$ be a conservative extension of L_0. Let $\cong_0$ and $\cong_1$ be the operational equivalence relations of L_0 and L_1, respectively.

  (i) If $\cong_1$ restricted to L_0 expressions is not equal to $\cong_0$ then L_0 cannot macro-express the facility F

  (ii) The converse does not hold

- So if we can find $e_1 \cong_0 e_2$ in the base language and a C in the extended language L_1 that can distinguish e_1 and e_2, then F adds expressiveness.

Operator Overloading

- Start in a base language L and add operator overloading (and overriding). Consider $3 \cong_L 1 + 2$

- Context $C(\alpha) =$

  let def Int.+(other) = 1 **in** α

- C witnesses the ability to violate the equality

- Observe: adding expressive power can reduce the ability to reason about code!

Exit

- Start in a base language L that does not have exceptions. Add an "**exit** n" construct, like Java's System.exit(n), that terminates the program with the result n.
  - Does this add expressive power? Can we show this with 2 expressions and a context?

- e_1 = **fn** f => **Ω**
- e_2 = **fn** f => **let** x = f(0) **in Ω**
- Context C(α) = α (**exit** 0)
  - C(e_1) does not terminate, C(e_2) = 0
  - **exit** adds the ability to jump out of a nonterminating program
  - call/cc (continuations) is similar

State example

- Start in a pure base language L. Add the ability to define and mutate variables.
  - Does this add expressive power? Can we show this with 2 expressions and a context?

- e_1 = **fn** _ => f(0)
- e_2 = **fn** _ => **let** x = f(0) **in** f(0)
- Context C(α) = **let var** f =

  fn x => { f := **fn** _ => Ω;
  x} **in** α
  - $C(e_1) = 0$, $C(e_2)$ does not terminate
  - State lets you count how many times a function is called

Global transformations

- Can we compile state in a pure functional language?
  - Of course! We create a functional model of the store and pass it around.
  - But this is a *global* transformation – we must rearrange the entire program
  - State adds expressive power
- What about control operators like **exit** and **call/cc**?
  - Can be compiled using continuation-passing style
  - Also a global transformation
  - Control operators add expressive power

Felleisen Expressiveness

- Argues that language expressiveness is about constructs that cannot be encoded *locally*
- Useful because:
  - Matches many intuitions about expressiveness
  - There's a crisp mathematical definition
    - Relying on important ideas like operational equivalence
  - Illustrates benefit of expressive features
    - Avoid the need for global transformations (by hand!)
    - Avoid patterns of programming which obscure intent (and can be implemented incorrectly)

Do Macros Increase Expressiveness?

- According to Felleisen, no!
  - At least for macros that are pure syntactic expansions, with no compile-time metaprogramming
  - **Takeaway: add other kinds of macros with care!**

- But maybe there is more to expressiveness?
  - Macros can help programmers avoid code duplication, especially of error-prone boilerplate code
  - Of course, a language with other excellent abstraction facilities may not benefit from macros in this way
    - A good test for how well a language supports abstraction!

Expressiveness as Reasoning

- One view
  - What do you know statically about a program?
- Two dims:
  - Exp. power
  - Safety
- Conflict
  - Felleisen's expressiveness reduces theorems (e.g. limits total, pure, capability safe, abstraction safe theorems)

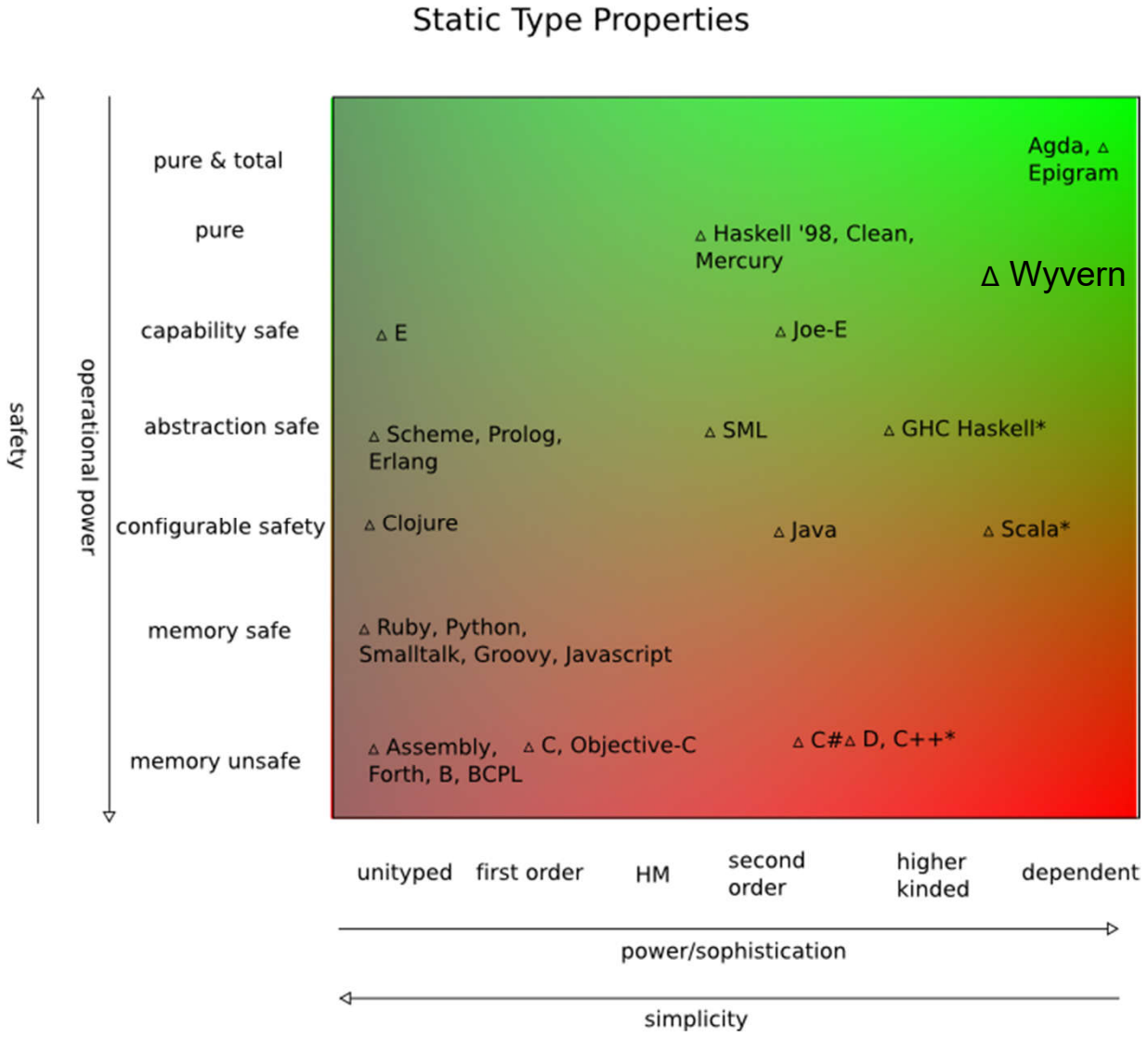


Diagram due to James Iry, currently available via the Internet Archive at https://web.archive.org/web/20140531013059/http://www.pogofish.com/types.png

More Type-Related Expressiveness

- Is there a (local) rewriting from one type system to another that preserves typeability?
  - Is a dynamic type system (or unityped, as some theorists would say) the most expressive?
- What global properties does a type system enforce?
  - Null safety, information flow, no resource loss, …
- What does a type system express about code?
  - This function returns an integer
  - This function must be the identity

Bibliography

- Matthias Felleisen. On the Expressive Power of Programming Languages. Science of Computer Programming 17:35-75, 1991.
- James Iry. Types a la Chart. http://james-iry.blogspot.com/2010/05/types-la-chart.html, 2010. Viewed April 2, 2020. The chart is currently missing, but is available at https://web.archive.org/web/20140531013059/http://www.pogofish.com/types.png