Course Conclusion

Jonathan Aldrich

17-396: Language Design and Prototyping

Spring 2020

institute for SOFTWARE RESEARCH

Today's Topics

- Types: safety and expressiveness
- Perspectives on PL design
- Recap of course themes

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)

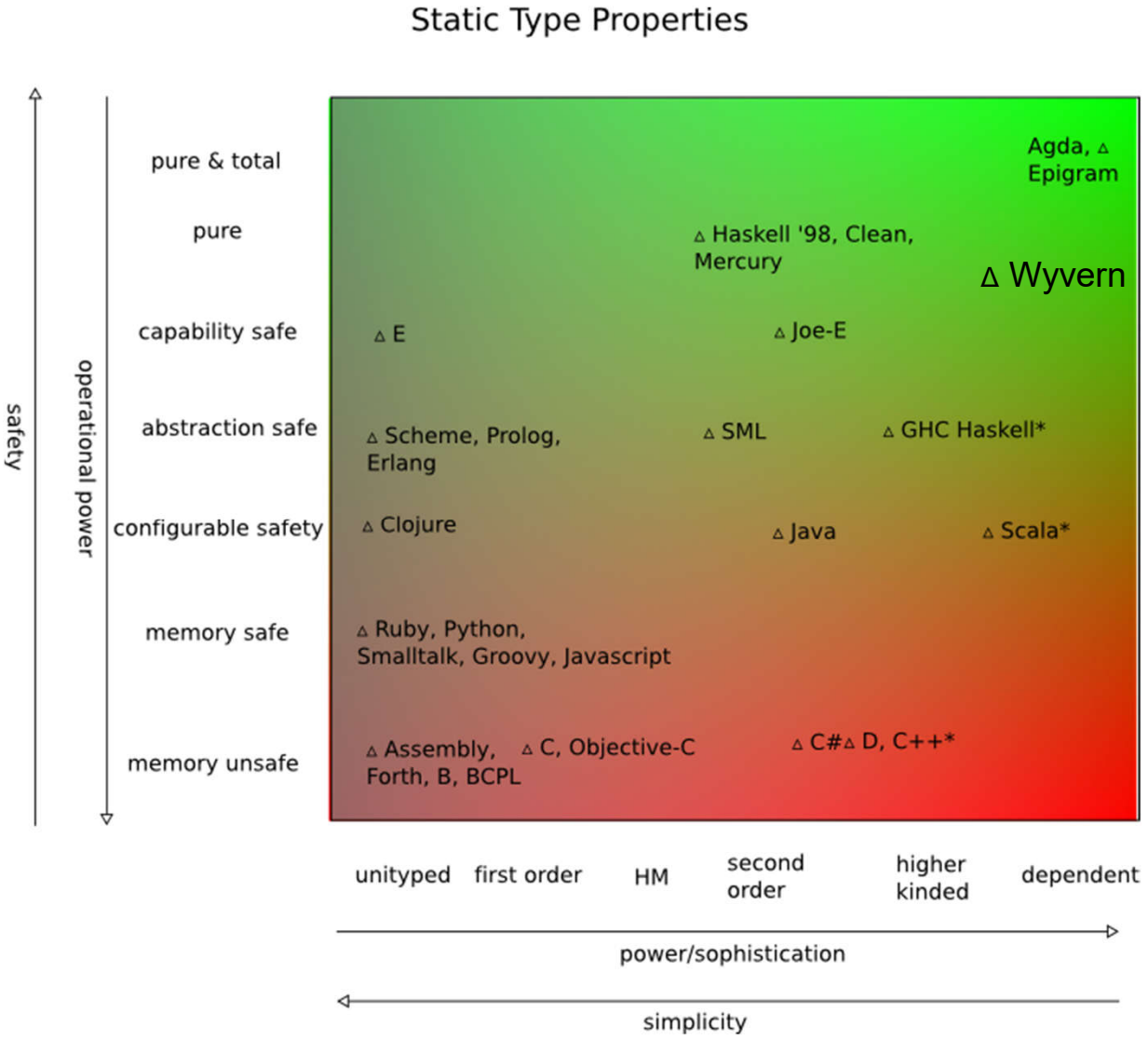

Diagram due to James Iry, currently available via the Internet Archive at https://web.archive.org/web/20140531013059/http://www.pogofish.com/types.png

Perspectives on PL Design

- Preparing for 17-396, I met with many CMU PL faculty
  - "How do you design a language?"
- Responses were quite varied!
- Some themes
  - The "Type Theory" perspective on PL design
  - Goal-oriented design
  - Tradeoffs: control vs. automation
  - Literature connection: "Worse is Better" vs. "The Right Thing"

The “Type Theory” perspective on PL design

- A strong (though not universal) theme in CMU PL
  - I am an exception! But I will try to represent the viewpoint

Search for Mathematical Beauty

- View that clean math makes good designs
  - Example: Curry-Howard Isomorphism
    - Used in many of the best theorem proving systems, including Coq, Twelf, Agda, … Extends to category theory too ("computational trinitarianism")

$$\frac{\Gamma, x{:}P \vdash e : Q}{\Gamma \vdash \lambda x{:}P.e : P \Rightarrow Q}$$

How do we prove $P \Rightarrow Q$?

We assume P and come up with a proof e of Q

Then $\lambda x{:}P.e$ proves $P \Rightarrow Q$

Later, with a proof of P we can apply the λ to prove Q

(c) 2018 Jonathan Aldrich

Search for Mathematical Beauty

- View that clean math makes good designs
  - Example: Curry-Howard Isomorphism
    - Used in many of the best theorem proving systems, including Coq, Twelf, Agda, … Extends to category theory too ("computational trinitarianism")
- Broadly, a useful criterion
  - Things hard to formalize are often design flaws
  - Thinking carefully about designs helps to get them right
  - Math seeks generality; making the PL follow the math helps make the PL general, too.
- Corollary: claim that PL concepts should have logical analogs
  - …even for PLs that are not for theorem proving
- My view: Curry-Howard less useful outside theorem provers
  - Drives PL design away from abstractions useful to programmers

(c) 2018 Jonathan Aldrich

Themes in the Type Theory Perspective

- Type and memory safety
  - Well-typed programs do not go wrong
- Variable binding
  - Does alpha-renaming work?
- Reasoning
  - What can you show is equal?
  - Is equality preserved when you add features? (cf. Felleisen)
- Clear specification
  - Key to memory safety – nothing about language is unspecified
  - Spec of Standard ML led the way – more recently WebAssembly

Themes in the Type Theory Perspective

- "Atoms, not molecules" – Bob Harper
  - PLs should have expressive primitives, not complex constructs that combine them – leads to duplication
  - Some hard cases, like datatypes – combination of primitives, but encodes idea of structural induction
- Uniformity and compositionality
  - Can anything be nested in anything else?
    - Counterexample: Hoare Logic
  - Are semantics compositional?
- Finding the right logical basis
  - Complete enough to express what you want
  - Non-redundant
    - If you remove a feature, does something break?

Goal-Oriented Design of C0

- Frank Pfenning's process for designing C0
- Approach directed by key goals
  - Keep focus on imperative algorithms and data structures
    - Requirement to keep the language simple – no time to teach PL features
  - Teach informal but rigorous reasoning re: data structure invariants
    - Prior to 122, 213 students did poorly in the malloc lab
  - Look and feel of C (for transition to 213)
  - Safe
    - No undefined behavior
    - Reasonable error messages
    - Facilitates testing student HW

Goal-Oriented Design of C0

- Key decisions
  - Almost completely a subset of C
    - Exception: [] vs. *, necessary to provide type safety
  - Assertions as effect-free Boolean functions
    - Avoid teaching a custom assertion language
  - Leave out many features: short, byte, … not needed for algorithms
  - Left out void* at first, then added it
    - Hard to talk about generic algorithms without C-style polymorphism

Tradeoffs: Control vs. Automation

- Memory management
  - Garbage collection increases productivity, eliminates some errors
  - Manual memory management more efficient for many programs
    - Rust avoids the errors, at the cost of type-system complexity
- Type inference
  - Automates local variable assignment
  - But when it doesn't work, error messages are horrible
  - Also complicates tools
- Similar tradeoffs in many aspects of PL
  - Theorem provers: Hilbert-style vs. Gentzen style
    - Gentzen's sequent calculus helps automate theorem provers
    - But Hilbert-style proofs are shorter, more readable for humans, and provide a simpler metatheory

Worse is Better [Gabriel]

Contrasting design styles:

- The "Right Thing"
  - Correctness is an absolute requirement
  - Consistency is also required, even at the cost of simplicity/completeness
  - Completeness: cover as many important situations as is practical
  - Simplicity is desirable, especially in the interface – but OK to sacrifice
- "Worse is Better"
  - Simplicity, especially in the implementation, is most important
  - Correctness is essential, but it is better to be simple than correct
  - Consistency is a goal; OK to sacrifice for simplicity, but better to leave out functionality
  - Completeness is desirable, but can be sacrificed for any other quality

Worse is Better: ***Simplicity***

- Memorable, but oxymoronic slogan
  - Meaning is that less functionality ("worse") is often preferable ("better")

- Rationale
  - If the initial program is good, it will be easier to implement and adapt
  - Use will spread rapidly, but users are conditioned to expect imperfection
  - Pressure, and capacity, to improve due to widespread use

"Therefore, the worse-is-better software first will gain acceptance, second will condition its users to expect less, and third will be improved to a point that is almost the right thing. In concrete terms, even though Lisp compilers in 1987 were about as good as C compilers, there are many more compiler experts who want to make C compilers better than want to make Lisp compilers better."

Course Themes

Why we design languages

- Expressiveness
  - Describe (a certain kind of) code more cleanly
  - Improve understandability, evolvability
- Constraints
  - Support optimization for performance
  - Support analysis for correctness

How to Design PLs in a Principled Way

- Multiple criteria; 3 of the big ones are
  - Correctness guarantees
  - Performance
  - Usability (performance/mistakes)

- Iterative design, guided by evaluation techniques
  - Concept-based design – capture key ideas, connect to goals
  - Theory – what is the design, exactly? Is it sound? Is it expressive?
  - Prototyping – can we build it, to get experience?
  - Qualitative human studies – feedback on design usability
  - Corpus studies – how might a new construct affect existing code?
  - Case studies – how well does the language work on a real problem?
  - User experiments – does it help people in practice?

Other Course Topics

- Effective prototyping techniques
  - Parsing, Interpreters, Transpilers, Internal DSLs
- Examples of research applying various design techniques
- Discussion and perspective on paradigms – FP, OO
- A few compilation and optimization topics

Your turn

- I look forward to seeing how you have applied these principles in your projects!

- Whether or not you do language design in the future…
  - I hope the principles learned in this course will make you
    - Better able to design and evaluate complex software systems
    - Better able to evaluate and select languages that you use
    - Better able to use language abstractions in your software development