On the Relevance of Programming Language Theory

Gabriel Scherer

Gallium – INRIA

Feburary 16th, 2013

What programming language researchers do

Our goal: help people design programming languages that help people write better programs.

What programming language researchers do

Our goal: help people design programming languages that help people write better programs.

In a good programming language:

- We understand well what the programs mean.
- Meaningless programs are rejected.
- Meaningful things we want to do can be expressed in the language.

What programming language researchers do

Our goal: help people design programming languages that help people write better programs.

In a good programming language:

- We understand well what the programs mean.
- Meaningless programs are rejected.
- Meaningful things we want to do can be expressed in the language.

Researchers use *mathematics* to define meaning, and study programs as mathematical objects.
They write papers about:

- how to better define some programming constructs
- how to reject more meaningless programs
- how to express more meanings as programs
- how to prove that all the above works (better proof techniques)

Motivation of this talk

Unfortunately, programming language research is often considered disconnected from the industrial practice. Does it really help people that we write papers about formal toy languages?

- Technology can change fast, research moves slowly.
- Researchers are not interested in design patterns, agile programming or XML.
- Who cares about typed λ-calculi?

Motivation of this talk

Unfortunately, programming language research is often considered disconnected from the industrial practice. Does it really help people that we write papers about formal toy languages?

- Technology can change fast, research moves slowly.
- Researchers are not interested in design patterns, agile programming or XML.
- Who cares about typed λ-calculi?

Thesis of this talk

Programming Language Theory (PLT) is relevant to real world practices.

What we cannot do

Produce industrial-strengh programming languages to sweep industrial practices. It is not a research problem: you need tools, libraries, good documentation, marketing/buzz efforts, etc. etc.

> " Social factors influence language adoption decisions more than intrinsic features do. Libraries, legacy code, and developer familiarity are the most important factors. "
>
> Leo Meyerovitch and Ariel Rabkin, "Social Influences on Language Adoption"

> " We are not fashion designers. "
>
> Tony Hoare

What can we do?

Not sure if there really is a link between our maths and your programming?

Just look at the (cool?) things we can do:

- Explain mistakes
  breaking mathematical properties also hurts usability
- Inspire new features
  once a concept is expressed in elegant maths, it Just Works

Variable scoping examples

Functions are a central concept to most programming language.

They allow to define a meaning somewhere, and use it somewhere else.

Transforming a complex expression in a call to a function (defined in the same environment) is a valid transformation.

Conversely, we can replace such a function by its definition to understand the program.

Getting scope wrong

```python
def sum(seq):
    result = 0
    for x in seq:
        result = result + x
    return result
```

```python
def sum(seq):
    result = 0
    def handle(x):
        result = result + x
    for x in seq:
        handle(x)
    return result
```

Getting scope wrong

```
def sum(seq):
    result = 0
    for x in seq:
        result = result + x
    return result
```

```
def sum(seq):
    result = 0
    def handle(x):
        result = result + x
    for x in seq:
        handle(x)
    return result
```

" [The well-known, elegant solution] yields a simple and consistent model, but it would be incompatible with all existing Python code. "

Ka-Ping Yee, PEP 3104

```
def handle(x):
  nonlocal result
  result = result + x
```

Getting scope wrong again

```
def fact(m):
    result = 1
    for i in range(1,m+1):
        result = result * i
    return result

def sum_facts(n):
    result = 0
    def handle(x):
        nonlocal result
        result = result + x
    for i in range(1,n+1):
        handle(fact(i))
    return result
```

Getting scope wrong again

```
def fact(m):
    result = 1
    for i in range(1,m+1):
        result = result * i
    return result

def sum_facts(n):
    result = 0
    def handle(x):
        nonlocal result
        result = result + x
    for i in range(1,n+1):
        handle(fact(i))
    return result
```

```
def sum_facts(n):
    result = 0
    def handle(x):
        nonlocal result
        result = result + x
    for i in range(1,n+1):
        result = 1
        for i in range(1,m+1):
            result = result * i
        handle(result)
    return result
```

Failure: no way to define local variables.

Inclusion (subtyping)

Natural refinement relation: $A \subseteq B$ when

- all values at type A also have type B
- you can always use an A when a B is expected

```
class Animal:
  get_older: time -> Animal
class Snake:
  get_older: time -> Snake
```

```
Animal a = new Bear
a.get_older(5 years)
Animal a = new Snake
a.get_older(5 years)
```

Inclusion (subtyping)

Natural refinement relation: $A \subseteq B$ when

- all values at type A also have type B
- you can always use an A when a B is expected

```
class Animal:                        Animal a = new Bear
  get_older: time -> Animal          a.get_older(5 years)
class Snake:                         Animal a = new Snake
  get_older: time -> Snake           a.get_older(5 years)
```

```
class Animal:                        Animal a = new Bear
  meet: Animal -> Reaction           react = a.meet(new Mouse)
class Snake:                         Animal a = new Snake
  meet: Snake -> Reaction            react = a.meet(new Mouse)
```

Another example: numerical hierarchies

$$\mathbb{N} \subseteq \mathbb{R}$$

All functions $\mathbb{N} \to \mathbb{N}$ are also functions $\mathbb{N} \to \mathbb{R}$. You may safely *widen* the return type of a function.

But all functions $\mathbb{N} \to \mathbb{R}$ are not functions $\mathbb{R} \to \mathbb{R}$. It's the other way around. You can only *narrow* the input type of a function.

Getting function inclusion right

Argument type inclusion goes "in the other direction". This is *contravariance*. Well-understood in the research community since the eighties.

Programming language designers repeatedly got this wrong: they assume that everythings can be widened.

Java (199x) made this mistake for mutable arrays. This lead to user bugs and performance issues.

Still considered too complex for practitioners. Dart (2011) willingly gives up on (generics) variance.

More positive interactions

Programming language researchers hired by the industry to work on:

- specifying the *meaning* of language constructs
- improving performance
- adding important but complex features (eg. generics/templates)
- language evolution: evolve an ugly language into something reasonable – while preserving compatibility

More positive interactions

Programming language researchers hired by the industry to work on:

- specifying the *meaning* of language constructs
- improving performance
- adding important but complex features (eg. generics/templates)
- language evolution: evolve an ugly language into something reasonable – while preserving compatibility

Prominent examples:

- Java (Sun/Oracle)
- C♯ (Microsoft)
- JavaScript (Mozilla, Google...)

Promising research/community/industry interaction: Rust at Mozilla.

The subject of my talk

Type systems: the static semantics of programming languages.
Weird math-looking formal syntax.

$$\textsc{Lambda}\ \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

Some example:

$$\dfrac{\dfrac{\alpha, x : \alpha \vdash x : \alpha}{\alpha \vdash (\lambda x.x) : \alpha \to \alpha}}{\vdash (\lambda x.x) : \forall \alpha.\alpha \to \alpha}$$

Beautiful formal properties: strong normalization, hierarchies of increasing logical power.

$$div2 : \forall n, \exists k, \quad \{2 * k = n\} \oplus \{2 * k + 1 = n\}$$

Difficult practical questions: type inference (principality?), type erasure...
Type systems for fine-grained properties: termination, resource analysis,

Thanks!

Bonus Slide: Some success stories of research ideas

Bonus Slide: Some success stories of research ideas

- 1930-201x: first-class anonymous functions demanded – **80** years

Bonus Slide: Some success stories of research ideas

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years

Bonus Slide: Some success stories of research ideas

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years

Bonus Slide: Some success stories of research ideas

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years
- 197x-200x: Generics (still source of trouble), type inference – **30** years

Bonus Slide: Some success stories of research ideas

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years
- 197x-200x: Generics (still source of trouble), type inference – **30** years
- 1989-2004: monads inspiring – **15** years

Bonus Slide: Some success stories of research ideas

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years
- 197x-200x: Generics (still source of trouble), type inference – **30** years
- 1989-2004: monads inspiring – **15** years

We're actually getting better!

Bonus Slide: Examples of challenges ahead

Constant push for static typing and analyses to be more expressive and simpler, less invasive at the same time.

Theoretical difficulties with modularity (we don't really know how to build large-scale systems).

Subtle cohabitation of proofs, static analyses and dynamic checks.

Lack of understanding of the long-term tradeoffs of concurrency models.

Under-represented areas: tooling, live programming/prototyping...