GNU epsilon

an extensible programming language

Luca Saiu <saiu@lipn.univ-paris13.fr>

LIPN, Université Paris 13

SPO Talks — Imperial College, London — 2011-06-20

Hello, I'm Luca Saiu

I work on programming languages and compilers

Hello, I'm Luca Saiu

I work on programming languages and compilers

- Bachelor's and Master's Degrees at the University of Pisa, Italy...
  - met my good friend Carlo Bertolli there
  - Marco Vanneschi is one of my great masters

Hello, I'm Luca Saiu

I work on programming languages and compilers

- Bachelor's and Master's Degrees at the University of Pisa, Italy...
  - met my good friend Carlo Bertolli there
  - Marco Vanneschi is one of my great masters
- ...now at LIPN, Institut Galilée, Université Paris 13, France:
  - Finishing my PhD thesis
  - *Attaché Temporaire d'Enseignement et Recherche*

Hello, I'm Luca Saiu

I work on programming languages and compilers

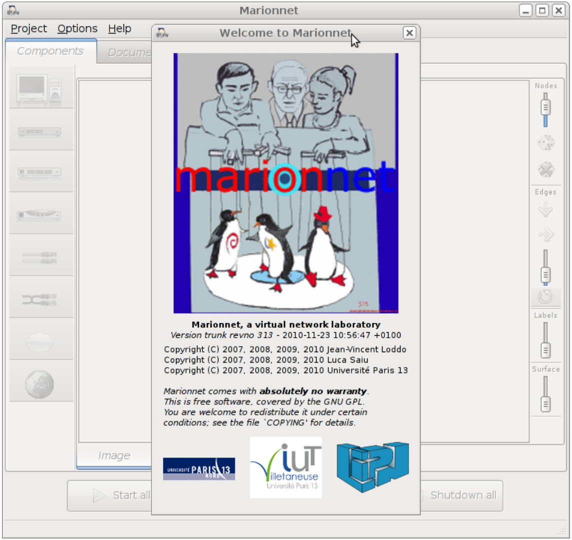
- Bachelor's and Master's Degrees at the University of Pisa, Italy...
  - met my good friend Carlo Bertolli there
  - Marco Vanneschi is one of my great masters
- ...now at LIPN, Institut Galilée, Université Paris 13, France:
  - Finishing my PhD thesis
  - *Attaché Temporaire d'Enseignement et Recherche*
- Free software activist
  - GNU maintainer since 2002

Hello, I'm Luca Saiu

I work on programming languages and compilers

- Bachelor's and Master's Degrees at the University of Pisa, Italy...
  - met my good friend Carlo Bertolli there
  - Marco Vanneschi is one of my great masters
- ...now at LIPN, Institut Galilée, Université Paris 13, France:
  - Finishing my PhD thesis
  - *Attaché Temporaire d'Enseignement et Recherche*
- Free software activist
  - GNU maintainer since 2002
- Lisper and functional programmer
  - Co-wrote Marionnet, in OCaml

Functional programming *in practice*: I co-wrote Marionnet



Quick history of ε

- 2001: a toy, my first functional language implementation and second compiler; static type checking; reference counter; no I/O; custom virtual machine; all written in C
- 2002-2005: rewritten from scratch; ML-style; static type inference; my first two garbage collectors; `epsilonlex` and `epsilonyacc` (bootstrapped); purely functional with I/O monad; new custom virtual machine; all written in C; $\sim 40,000$ LoC; approved as official GNU project in 2002
- 2006-2007: macros; user-defiend primitives; incomplete
- 2007-2009: reductionism: kernel based on λ-calculus; macros; user-defiend primitives; incomplete
- 2010-: reductionism: imperative first-order kernel macros and transformations; user-defiend primitives; s-expression syntax; advanced OCaml prototype, about to be bootstrapped

Language research

A crude chronology of common programming language features

- 1960s: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- 1970s:
- 1980s:
- 1990s:
- 2000s:

Language research

A crude chronology of common programming language features

- 1960s: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- 1970s: first-class continuations, quasiquoting, [type inference (suggested by Professor Kelly)]
- 1980s:
- 1990s:
- 2000s:

Language research

A crude chronology of common programming language features

- 1960s: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- 1970s: first-class continuations, quasiquoting, [type inference (suggested by Professor Kelly)]
- 1980s: logic programming, purely functional programming
- 1990s:
- 2000s:

Language research

A crude chronology of common programming language features

- 1960s: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- 1970s: first-class continuations, quasiquoting, [type inference (suggested by Professor Kelly)]
- 1980s: logic programming, purely functional programming
- 1990s: monads in programming; *err...* components?
- 2000s:

Language research

A crude chronology of common programming language features

- 1960s: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- 1970s: first-class continuations, quasiquoting, [type inference (suggested by Professor Kelly)]
- 1980s: logic programming, purely functional programming
- 1990s: monads in programming; *err...* components?
- 2000s: *err...*

More Language research would be nice

A crude chronology of common programming language features

- 1960s: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- 1970s: first-class continuations, quasiquoting, [type inference (suggested by Professor Kelly)]
- 1980s: logic programming, purely functional programming
- 1990s: monads in programming; *err...* components?
- 2000s: *err...*

More Language research would be nice

A crude chronology of common programming language features

- 1960s: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- 1970s: first-class continuations, quasiquoting, [type inference (suggested by Professor Kelly)]
- 1980s: logic programming, purely functional programming
- 1990s: monads in programming; *err...* components?
- 2000s: *err...*

No, we didn't solve the expressivity problem. Whoever thinks we did is **particularly** far from the solution.

"Modern" languages aren't expressive enough

- Program requirements get more and more complex
- Programs grow, too: $\sim 10^6$ LoC is not unusual
- But languages stopped evolving
  - Programs are hard to get right
  - Sometimes we *do* need to prove properties about programs (by machine, for realistic programs)...
    - ...so we need a *formal specification* (necessary but not sufficient)

"Modern" languages are *way* too complex for proofs

- *The Definition of Standard ML, Revised Edition*, 1997, **128 pp.** (**very dense** formal specification)
- *Revised^6 Report on the Algorithmic Language Scheme*, 2007 **187 pp.** (with a **non-normative and partial** formal specification in an appendix)
- *Haskell 98 Language and Libraries – The Revised Report*, 2003, **270 pp.** (**no** formal specification)
- *ISO/IEC 9899:201x Programming languages – C*, March 2009 draft, **564 pp.** (**no** formal specification)
- *The Java Language Specification*, Third Edition, June 2009, **684 pp.** (**no** formal specification)
- *ANSI INCITS 226-1994 (R2004) Common Lisp*, **1153 pp.** (**no** formal specification)
- *ISO/IEC 14882: Programming Language C++*, June 2009 draft, **1326 pp.** (**no** formal specification)

The silver bullet, in my opinion

What killer features do we need?

The silver bullet, in my opinion

What killer features do we need?

- Of course I've got opinions, but in general **I don't know**

The silver bullet, in my opinion: **reductionism**

What killer features do we need?

- Of course I've got opinions, but in general **I don't know**
- So, *delay decisions* and let users build the langauge

The silver bullet, in my opinion: **reductionism**

What killer features do we need?

- Of course I've got opinions, but in general **I don't know**
- So, *delay decisions* and let users build the langauge
  - Small kernel language
  - Syntactic abstraction
  - Formal specification

The silver bullet, in my opinion: **reductionism**

What killer features do we need?

- Of course I've got opinions, but in general **I don't know**
- So, *delay decisions* and let users build the langauge
  - Small kernel language
  - Syntactic abstraction
  - Formal specification
- We need radical experimentation again!
  - Many *personalities* on top of the same kernel

A Scheme demo

Have a look at an expressive language (it's not ε)

Raise your hand if you know some Lisp dialect

[Quick Scheme demo: McCarthy's `amb` operator, macros and `call/cc`]

Problems I see with Scheme

- High level kernel
    - *Very* hard to compile efficiently and analyze...
    - ...you pay for the complexity of `call/cc` even when you don't use it
        - performance, in some implementations
        - intellectual complexity
- Still relatively complex
    - Last standard (R^{6}RS, 2007): 187 pages *in English*
    - Too big to have a complete formal specification

What I call reductionism is not new. Can you recognize this?

"a language design of the old school is a pattern for programs. But now we need to 'go meta.' We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind. [...] My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language."
(my emphasis)

What I call reductionism is not new. Can you recognize this?

"a language design of the old school is a pattern for programs. But now we need to 'go meta.' We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind. [...] My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language."
(my emphasis)

—**Guy L. Steele Jr.**, ***Growing a Language***, 1998

What I call reductionism is not new. Can you recognize this?

"a language design of the old school is a pattern for programs. But now we need to 'go meta.' We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind. [...] My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language."
(my emphasis)

—**Guy L. Steele Jr.**, ***Growing a Language***, 1998

What kernel language did he plan to build on?

What I call reductionism is not new. Can you recognize this?

"a language design of the old school is a pattern for programs. But now we need to 'go meta.' We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind. [...] My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language."
(my emphasis)

—**Guy L. Steele Jr.**, ***Growing a Language***, 1998

What kernel language did he plan to build on? **Java** (!)

What I call reductionism is not new. Can you recognize this?

"a language design of the old school is a pattern for programs. But now we need to 'go meta.' We should now think of a language design as a pattern for language designs, a tool for making more tools of the same kind. [...] My point is that a good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language."
(my emphasis)

—**Guy L. Steele Jr.**, *Growing a Language*, 1998

What kernel language did he plan to build on? **Java** (!)
To Steele's credit, his later proposals based on Fortress are more realistic

Reflection

The program has to be able to *reason about* itself (*)

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*
- Analyses on the program state
  - `unexec`
  - Checkpointing
  - **Compiling** [the compiler is just a procedure!]

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*
- Analyses on the program state
  - `unexec`
  - Checkpointing
  - **Compiling** [the compiler is just a procedure!]

The program has to be able to *update* itself (**)

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*
- Analyses on the program state
  - `unexec`
  - Checkpointing
  - **Compiling** [the compiler is just a procedure!]

The program has to be able to *update* itself (**)

- Transformations à-la-CPS

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*
- Analyses on the program state
  - `unexec`
  - Checkpointing
  - **Compiling** [the compiler is just a procedure!]

The program has to be able to *update* itself (**)

- Transformations à-la-CPS
- **Compiler optimizations** [my idea: nondeterministic rewrite system, hill-climbing]

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*
- Analyses on the program state
  - `unexec`
  - Checkpointing
  - **Compiling** [the compiler is just a procedure!]

The program has to be able to *update* itself (**)

- Transformations à-la-CPS
- **Compiler optimizations** [my idea: nondeterministic rewrite system, hill-climbing]
- Compile-time garbage collection

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*
- Analyses on the program state
  - `unexec`
  - Checkpointing
  - **Compiling** [the compiler is just a procedure!]

The program has to be able to *update* itself (**)

- Transformations à-la-CPS
- **Compiler optimizations** [my idea: nondeterministic rewrite system, hill-climbing]
- Compile-time garbage collection

Point (**) is much more delicate

Reflection

The program has to be able to *reason about* itself (*)

- Good error reporting: *failed within the else branch of the conditional starting at line 35*
- Analyses on the program state
  - `unexec`
  - Checkpointing
  - **Compiling** [the compiler is just a procedure!]

The program has to be able to *update* itself (**)

- Transformations à-la-CPS
- **Compiler optimizations** [my idea: nondeterministic rewrite system, hill-climbing]
- Compile-time garbage collection

Point (**) is much more delicate

- Use syntax abstraction to rewrite into non-reflective programs where possible...
  - ...otherwise inefficient and unanalyzable (but *not* an "error")

ε_0 grammar

This is the *complete kernel language* grammar:

$$
\begin{aligned}
e ::= & \\
& x_h \\
\mid\ & c_h \\
\mid\ & [\texttt{let}\ x^*\ \texttt{be}\ e\ \texttt{in}\ e]_h \\
\mid\ & [\texttt{call}\ f\ e^*]_h \\
\mid\ & [\texttt{primitive}\ \pi\ e^*]_h \\
\mid\ & [\texttt{if}\ e \in \{c^*\}\ \texttt{then}\ e\ \texttt{else}\ e]_h \\
\mid\ & [\texttt{fork}\ f\ e^*]_h \\
\mid\ & [\texttt{join}\ e]_h \\
\mid\ & [\texttt{bundle}\ e^*]_h
\end{aligned}
$$

ε_0 grammar [This is the *kernel language* grammar!]

This is the *complete* *kernel language* grammar:

$$
\begin{array}{rl}
e ::= & \\
& x_h \\
| & c_h \\
| & [\texttt{let } x^* \texttt{ be } e \texttt{ in } e]_h \\
| & [\texttt{call } f\ e^*]_h \\
| & [\texttt{primitive } \pi\ e^*]_h \\
| & [\texttt{if } e \in \{c^*\} \texttt{ then } e \texttt{ else } e]_h \\
| & [\texttt{fork } f\ e^*]_h \\
| & [\texttt{join } e]_h \\
| & [\texttt{bundle } e^*]_h
\end{array}
$$

Parsing

A predefined parser, for bootstrapping reasons

- A predefined procedure parses s-expressions (like Scheme, nothing similar to ε_0)
  - Another predefined procedure expands s-expressions into *expressions*
    - Macro expansion and transformations, here
  - Easy to add new literals (lexicon only)

If you don't like s-expressions, write a new parser!

- Use the predefined frontend to make another one
  - Minimality not so important here: easy to replace

A feel of ε_0 dynamic semantics: sample rules

$$\frac{}{(x_h,\ \rho).S\ !V\ \Gamma \longrightarrow_{\mathbb{E}} S\ !c!V\ \Gamma}\ \Gamma(\texttt{global-environment})[\rho] : x \mapsto c$$

$$\frac{}{([\texttt{join}\ \square]_{h_0},\ \rho).S\ !\mathcal{F}(t)!V\ \Gamma \longrightarrow_{\mathbb{E}} S\ !c_t!V\ \Gamma}\ \Gamma(\texttt{futures}) : t \mapsto (\langle\rangle,\ !c_t!V_t)$$

$$\frac{S_t\ \ V_t\ \ \Gamma \longrightarrow_{\mathbb{E}} S'_t\ \ V'_t\ \ \Gamma'}{S\ \ V\ \ \Gamma \longrightarrow_{\mathbb{E}} S\ \ V\ \ \Gamma'[\texttt{futures}, t \mapsto (S'_t,\ V'_t)]}\ \Gamma(\texttt{futures}) : t \mapsto (S_t,\ V_t)$$

$$\frac{}{(x_h,\ \rho).S\ !V\ \Gamma \downarrow^{\rho}_{\text{☠}}}\ x \notin dom(\Gamma(\texttt{global-environment})[\rho])$$

The *complete* dyanamic semantics for ε_0 is two or three pages long.

A word against *mandatory* static checks

- You aren't *always* writing software for nuclear power plants, are you?
- Programmers know best
  - maybe the code is safe but the compiler can't prove it
  - maybe we want to test something unrelated to the problem
    - I'll take responsibility if it fails, but **let me run the damn thing**
- Refusing to compile or run is not rational
  - Silenceable warnings are fine
  - (Non-silenceable warnings will be overlooked and essentially ignored)

"Epiphenomena"

Compilation, optimizations, analyses, ... are **not** part of the language

- But they can be implemented with predefined building blocks
- A high-level pattern of lower-level objects
  - Interesting and useful, but not "fundamental"
  - Smaller language!

As an epiphenomenon, when extending ε_0 we distinguish:

- a *meta library*
- a *personality library*

ε_0 static semantics: dimension analysis

A form of type inference for ε_0
[Hint at the Math]

My ε_0 semantics is actually usable

- The full dynamic sementics of ε_0 fits in a few pages
- Dimension analysis *proved sound* with respect to dynamic semantics
  - *Well-dimensioned programs do not go wrong*

Analyses and personalities

- Some analyses must be performed on extended languages (example: type analysis with first-class continuations)
- Some analyses are better expressed on ε_0...
  - Dimension analysis, asymptotic complexity analysis, termination analysis...
  - We don't need the extended forms, so analyzing ε_0 is simple (example: type inference on pattern matching)

ε current status

- Advanced prototype in (a subset of) OCaml
  - To be bootstrapped with CamlP4
- Parallel garbage collector in C (see my LIPN home page)
- ε_0 compiler written in (a subset of) OCaml; ANF, liveness analysis
- *I need to write an extensible scanner in* ε_0 (relatively easy)
- Custom virtual machine written in low-level C (threaded code), native backends easy to add
- Bootstrapping code: lists, symbols, strings, hash tables..., **in** ε_0 ; not that uncomfortable
- Other bootstrapping code from the previous implementation based on λ-calculus

About ε

GNU epsilon is free software, to be released under the GNU GPL.

You're welcome to share and change it under certain conditions; please see the license text for details.

Conclusion

- Reductionism is a viable style of designing and implementing practical programming languages, leading to solutions which are easier to extend, experiment with and formally analyze.
- Strong syntactic abstraction makes easy what is *impossible* in other languages
  - An overlooked problem: non-Lisp languages are *severely* lacking
- Thanks to reflection we can build language tools as part of the program
- Performance doesn't need to be bad
  - I'll have measures soon

Conclusion

- Reductionism is a viable style of designing and implementing practical programming languages, leading to solutions which are easier to extend, experiment with and formally analyze.
- Strong syntactic abstraction makes easy what is *impossible* in other languages
    - An overlooked problem: non-Lisp languages are *severely* lacking
- Thanks to reflection we can build language tools as part of the program
- Performance doesn't need to be bad
    - I'll have measures soon

Thanks!