GNU epsilon
an extensible programming language

Luca Saiu
positron@gnu.org
http://ageinghacker.net

LRDE, EPITA — Télécom ParisTech
GNU Project

Séminaire *Performance et Généricité*
Laboratoire de Recherche et Développement de l’EPITA

Le Kremlin-Bicêtre, October 9th 2013
Hello, I’m Luca Saiu

I’m starting work on Vaucanson.

I’ve mostly worked on programming languages and compilers:

- Master’s at the University of Pisa;
- PhD at Université Paris 13;
  - advisors: C. Fouqueré, J.-V. Loddo;
  - reviewers: E. Chailloux, M. Mauny;
- Just finished a post-doc at Inria, on OCaml multi-core support;
- Free software activist, GNU maintainer;
- Lisper and functional programmer:
  - Co-wrote Marionnet, in OCaml
Functional programming *in practice*: I co-wrote Marionnet

http://www.marionnet.org
We want more expressive languages

A crude chronology of programming language features:

- **1960s:** structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- **1970s:** relational programming, first-class continuations, quasiquoting, type inference
- **1980s:**
- **1990s:**
- **2000s:**
We want more expressive languages

A crude chronology of programming language features:

- **1960s**: structured programming, recursion, symbolic programming, higher order, garbage collection, meta-programming, object orientation, concatenative programming
- **1970s**: relational programming, first-class continuations, quasiquoting, type inference
- **1980s**: logic programming, constraint programming, purely functional programming
- **1990s**: monads in programming; err... components?
- **2000s**: err...

We should work harder to *improve expressivity.*
Motivations

The $\varepsilon_0$ and $\varepsilon_1$ languages

Status and conclusion

Mainstream languages aren’t sufficient

Reductionism

“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 don’t evolve fast enough
  - Programs are hard to get right
  - Sometimes we do need to prove properties about programs (by machine, for realistic programs)...
    - ...so we need formal specifications for languages (necessary but not sufficient)
“Modern” languages are way too complex for proofs

- *ISO/IEC 9899:201x Programming languages – C*, March 2009 draft, 564 pp. *(no formal specification)*
- *ISO/IEC 14882:2011: Programming Language C++*, 1324 pp. as per the N3337 draft *(no formal specification)*

Luca Saiu — positron@gnu.org  GNU epsilon — savannah.gnu.org/bzr/?group=epsilon
Motivations

The \( \varepsilon_0 \) and \( \varepsilon_1 \) languages

Status and conclusion

Mainstream languages aren’t sufficient

Reductionism

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 language
  - Small core language
  - Syntactic abstraction
  - Formal specification
- We need radical experimentation again!
  - Many *personalities* on top of the same *core language*
Minimalistic, extensible languages: Scheme [and Forth]

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

Revised¹ Report on the Algorithmic Language Scheme


Sample extension: McCarthy’s \texttt{amb} backtracking operator
Problems I see with Scheme

- **High-level core**
  - higher-order, closures, continuations
  - 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**
  - Latest official standard (R^6RS, 2007): 187 pages *in English*
    - R^7RS WG1 will be smaller: 88 pages as of November 2012
  - Too big to have a complete formal specification
The *reductionism* idea is not new.

“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*
The reductionism idea is not new.

“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]

“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]


He planned to build on **Java (!)**

To Steele’s credit, his later proposals based on Fortress are more realistic.
Reflection (1/2: self-analysis)

The program has to be able to **(1) access its own dynamic state:**

- **Analyses** on the program state:
  - **self-analysis**: in the style of static analyses (for example type inference);
  - **“unexec” operation**: dump the current dynamic state (to files, sockets...) — *definable as an ordinary procedure*;
  - **compilation** — *definable as an ordinary procedure*
Reflection (2/2: self-modification)

The program has to be able to (2) *update* its own state, including procedures, «à chaud»:

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

Point (2) is more delicate

- Use syntax abstraction to rewrite into non-self-modifying programs *where possible*...
  - ...otherwise inefficient and unanalyzable (but *not* an “error”)

Luca Saiu — positron@gnu.org

GNU epsilon — savannah.gnu.org/bzr/?group=epsilon
We call our core language $\varepsilon_0$.

$\varepsilon_0$ is a first-order imperative language of global recursive procedures, with threads. Here’s its \textit{complete} grammar:

$$e ::= \begin{cases} 
\chi_h \\
\psi_h \\
[\text{let } x^* \text{ be } e \text{ in } e]_h \\
[\text{call } x \ e^*]_h \\
[\text{primitive } x \ e^*]_h \\
[\text{if } e \in \{c^*\} \text{ then } e \text{ else } e]_h \\
[\text{fork } x \ e^*]_h \\
[\text{join } e]_h \\
[\text{bundle } e^*]_h 
\end{cases}$$
Our core language $\varepsilon_0$ [This is the core language grammar!]

We call our core language $\varepsilon_0$.

$\varepsilon_0$ is a first-order imperative language of global recursive procedures, with threads. Here’s its complete grammar:

$$e ::=$$

- $x_h$
- $c_h$
- $[\text{let } x^* \text{ be } e \text{ in } e]_h$
- $[\text{call } x \ e^*]_h$
- $[\text{primitive } x \ e^*]_h$
- $[\text{if } e \in \{c^*\} \text{ then } e \text{ else } e]_h$
- $[\text{fork } x \ e^*]_h$
- $[\text{join } e]_h$
- $[\text{bundle } e^*]_h$
Why $\varepsilon_0$ has no side effects or definitions

The $\varepsilon_0$ grammar lacks explicit *side effect* and *definition* operators. Our “initial state” (globals, primitives, procedures, memory, ...) will allow:

- memory side effects *by primitives*
  - *store* is a primitive among *load*, *allocate*, ...
- global and procedure definitions *by procedures*
  - Global tables for globals and procedures, in memory

So, programs can *self-modify*:

- if a program doesn’t, it can be compiled more efficiently
A feel of $\varepsilon_0$ dynamic semantics: sample rules

\[
([\text{call } f \ e_{h_1}...e_{h_n}]_{h_0}, \rho).S \ ?V \ \Gamma \rightarrow_{E} \ (e_{h_1}, \rho)...(e_{h_n}, \rho).([\text{call } f \ \Box]_{h_0}, \emptyset).S \ ?\!V \ \Gamma
\]

\[
([\text{bundle } \Box]_{h_0}, \rho).S \ ?c_n\!c_{n-1}...c_2?c_1?V \ \Gamma \rightarrow_{E} \ S \ ?c_n\!c_{n-1}...c_2?c_1?V \ \Gamma
\]

\[
([\text{join } \Box]_{h_0}, \rho).S \ ?T(t)?V \ \Gamma \rightarrow_{E} \ S \ ?c_t?V \ \Gamma \quad \Gamma_{\text{futures}}: t \mapsto (\langle\rangle, \ ?c_t?)
\]

The full dynamic semantics of $\varepsilon_0$ fits in two pages; three if we also include failure semantics.
My epsilon_0 semantics is actually usable

- Formally developed “dimension analysis”, as a sample static analysis on epsilon_0 programs — a form of type inference

\[
\begin{array}{ccccccc}
& & & & & & \\
& 1 & 2 & 3 & 4 & 5 & 6 \\
0 & & & & & & ...
\end{array}
\]

- Dimension analysis *proved sound* with respect to dynamic semantics:

  “well-dimensioned programs do not go wrong”

Luca Saiu — positron@gnu.org

GNU epsilon — savannah.gnu.org/bzr/?group=epsilon
User syntax: by s-expressions

Lisp-style s-expressions are a data structure convenient for encoding syntax.

- A “list” structure:

  (average x 10)
User syntax: by s-expressions

Lisp-style s-expressions are a data structure convenient for encoding syntax.

- A “list” structure:
  
  \((\text{average} \ x \ 10)\)

- The same structure, making conses explicit:
  
  \((\text{average} \ . \ (x \ . \ (10 \ . \ ()))))\)
Lisp-style *s-expressions* are a data structure convenient for encoding syntax.

- A “list” structure:

  \[(\text{average } x \ 10)\]

- The same structure, making conses explicit:

  \[(\text{average . (x . (10 . ()))))\]

- The same structure, graphically:

![Diagram of s-expression structure]
Expansion of s-expressions into $\varepsilon_0$ expressions (1/2)

A trivial encoding for $\varepsilon_0$ syntax into s-expressions.

We use the s-expression

\[
\text{form name} \\
(e0:if-in \ x (1 4 6) 10 50)
\]

sub-forms

\[
\text{to represent the } \varepsilon_0 \text{ conditional expression}
\]

\[
[\text{if } x_{h_2} \in \{1, 4, 6\} \text{ then } 10_{h_3} \text{ else } 50_{h_4}]_{h_1}
\]

\[
\text{for some fresh handles } h_1, h_2, h_3, h_4.
\]
Expansion of s-expressions into $\varepsilon_0$ expressions (2/2)

Default case, if the first element is not a form name:

We use the s-expression

\[
\text{operator} \quad \left( \text{average } x \ 10 \right) \quad \text{operands}
\]

to represent the $\varepsilon_0$ procedure call

\[
[\text{call average } x_{h_2} 10_{h_3}]_{h_1}
\]

for some fresh handles $h_1, h_2, h_3$. 
Extension mechanisms

Even with side effects and definitions, $\varepsilon_0$ is inconvenient to use directly.

We introduce two syntactic extension mechanisms:

- a **macro** rewrites an s-expression into an expression
  - [in case you’re wondering: *not homoiconic*, unlike Lisp]
  - “local”: it cannot access its surrounding s-expression
- a **transform** rewrites an expression into another expression
  - “global” syntactic abstraction (example: Closure Conversion)
Sample macroexpansion (s-expression to $\varepsilon_0$)

User-defined forms can also be encoded as s-expressions.

An example with the sequential composition macro $e_1$:begin:

$(e_1$:begin
  (string:write "The result is ")
  (fixnum:write $n$)
  (string:write "\n")$)

$\Rightarrow$

[let () be [call string:write "The result is "]$_{h_3}$]$_{h_2}$ in
[let () be [call string:write $n$]$_{h_6}$]$_{h_5}$ in [call string:write "\n"]$_{h_8}$]$_{h_7}$]$_{h_4}$]$_{h_1}$

for some fresh handles $h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8$.  

Luca Saiu — positron@gnu.org

GNU epsilon — savannah.gnu.org/bZR/?group=epsilon
A sample macro definition

A definition of \texttt{e1:begin}, as a quite simple (recursive) macro:

\begin{verbatim}
(e1:define-macro (e1:begin first-form . more-forms)
 (e0:if-in (sexpression:null? more-forms) (#t)
   first-form
   `(e0:let () ,first-form
    (e1:begin ,(sexpression:car more-forms)
     @(sexpression:cdr more-forms))))
\end{verbatim}

In case you're wondering:

\begin{itemize}
  \item \texttt{quasiquote} is itself a macro; quasiquoting (like quoting) yields an \textit{expression}
  \item \texttt{e1:define-macro} is itself a macro, built on \texttt{e1:destructuring-bind}, yet another macro
\end{itemize}
Transforms (à-la-CPS)

Expression-to-expression rewriting, to be applied to all toplevel forms from a certain point on, or to the whole program.

- define an ordinary procedure turning an expression into another expression
- “install” it so that it is automatically applied from now on (possibly even retroactively, as for CPS)

Ask me later if you want more details [presentation part 4]
Also including the syntax we’ve just shown, the $\varepsilon_1$ personality is a set of extensions to conveniently write other personalities.

- S-expression syntax à-la-Lisp
- *macroexpansion* and *transforms*
- many general-purpose syntactic forms to make the user’s life easier

$\varepsilon_1$ as a programming language:
- Lispy feel; low-level, potentially efficient
- *untyped* (*not even dynamically-typed*)
I implemented $\varepsilon_1$ on top of $\varepsilon_0$:

- I defined the *macroexpansion* and *transformation* machinery in $\varepsilon_0$
- then $\varepsilon_1$ syntactic forms, by macros and transforms
  - expressivity grows fast: I can use an extension to build the next one
Main \( \varepsilon_1 \) forms (defined over \( \varepsilon_0 \)) (1/2)

Just showing syntactic construct names:

\[
\begin{align*}
\end{align*}
\]
Main $\varepsilon_1$ forms (defined over $\varepsilon_0$) (2/2)

...,  
set-as-list:make, set-as-list:union,  
set-as-list:intersection, set-as-list:subtraction,  
record:define, sum:define, sum:define-open,  
sum:extend-open, $e_1$:lambda, closure:ml-lambda,  
e1:call-closure, e1:named-let, e1:do, e1:while,  
e1:dolist, e1:dotimes, e1:for, e1:let [including named let],  
pattern matching].

- Notice that we included closures ($e_1$:lambda).
Some $\varepsilon_1$ forms are defined with transforms

Some code-to-code transformations depend on the context.

- **Closure-conversion**
  - expression **non-locals** depend on context

- **First-class continuations with** $\varepsilon_1$:$\texttt{call/cc}$ (experimental)
  - inherently **global**: CPS-transformed expressions are incompatible with untransformed ones
Bootstrap: implementing $\varepsilon_1/\varepsilon_0$

$\varepsilon_0$ syntax encoded by s-expressions: using Guile Scheme, plus C for primitives.
- Data structures as untyped memory buffers, with pointers
  - primitives to allocate, load, store
- s-expression as a data structure: “open” sum type;
  - expressions (themselves an open sum!) as one case;
- Reliance on the s-expression parser from Guile’s frontend

Bootstrapping final step:
- Unexec
- exec into a different runtime implementation
  (final data representation more efficient than Guile’s)
Back to soundness proofs: $\varepsilon_1$ properties

The static semantics we proved sound was on $\varepsilon_0$.

How to do soundness proofs on $\varepsilon_1$ (or higher-level personalities):

- provide informal “abstract syntax” for $\varepsilon_1$ forms and mappings to $\varepsilon_0$. Example:
  - $[[\text{begin } e_{h_1}]_h] = [e_{h_1}]$
  - $[[\text{begin } e_{h_1} e_{h_2} \ldots e_{h_n}]_h] = [\text{let } \langle \rangle \text{ be } [e_{h_1}] \text{ in } [[\text{begin } e_{h_2} \ldots e_{h_n}]_h']]_h''$

- Use properties on $\varepsilon_0$ forms as lemmas for properties on $\varepsilon_1$ forms

Just an idea for future work.
Memory management may be a bottleneck in high-level parallel programs

- **parallel** mark-sweep, conservative pointer finding, no safe points
- **BiBOP**, efficient for programs where most heap-allocated objects have one of a few shapes
- scales well on multi-cores, on micro-benchmarks (8 cores)
- nontrivial — 5000 lines of (heavily commented) C
- currently not generational
  - promising as *the old generation* of a generation system
GNU epsilon project: current status

- bootstrapped from Guile Scheme
  - now I only use Guile for its s-expression parser/printer
- three different runtimes: untagged, tagged, based on Guile
- $\varepsilon_0$ interpreter in itself (slow), in C (fast)
- unexec
- closure-conversion as a transform
  - unexpected uses: imperative loops, friendly syntax with nonlocals for futures and unexec;
- experimental CPS transform (currently broken)
- quick-'n-dirty compilers (three backends: C, MIPS, x86_64): ~1000 lines (!)
- a few cool syntax hacks: keyword parameters

Luca Saiu — positron@gnu.org
GNU epsilon — savannah.gnu.org/bzr/?group=epsilon
Developed but not integrated yet:

- parallel BiBOP collector
  - another garbage collector, sequential semispase [suitable as the young generation when joined];
- extensible scanner (to be finished)
- custom virtual machine written in low-level C (threaded code), for bytecode execution;
GNU epsilon is free software, released under the GNU GPL version 3 or later.

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
- Thanks to reflection we can build language tools as part of the program
- Performance doesn’t need to be bad
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.

Thanks to reflection we can build language tools as part of the program.

Performance doesn’t need to be bad.

Thank you.
4 A transform definition in (some) detail
- Add new expression cases, and their syntax
- Define ordinary procedures
- Install transform procedures

5 Approximated tombstone diagrams
- Interpreters
- Runtimes
- Unexec
Sample transform (1/5: add new expression cases)

```
(sum:extend-open e0:expression
  (lambda handle formals body)
  (call-closure handle closure-expression actuals))

;; Define "builder" procedures like for ε₀ expression cases:
(e1:define (e1:lambda* formals body)
  (e0:expression-lambda (e0:fresh-handle) formals body))
(e1:define (e1:call-closure* closure-expression actuals)
  (e0:expression-call-closure (e0:fresh-handle)
   closure-expression actuals))
```

In case you’re wondering:

- expressions are a sum type à-la-ML, open to new cases (like exn in OCaml)
- sum types definition and extension operators are macros...
- ultimately just untyped memory structures: integers, pointers to buffers
The macro for our new forms will call the builder procedures at macroexpansion time:

```
(e1:define-macro (e1:lambda formals . body-forms)
  (sexpression:inject-expression
   (e1:lambda* (sexpression:eject-symbols formals)
     (e1:macroexpand `(e1:begin ,@body-forms)))))
```

```
(e1:define-macro (e1:call-closure closure-expression . actuals)
  (sexpression:inject-expression
   (e1:call-closure* (e1:macroexpand closure-expression)
     (e1:macroexpand-sexpressions actuals))))
```

In case you’re wondering:

- injection and ejection convert to and from s-expressions.
A transform definition in (some) detail

Approximated tombstone diagrams

Add new expression cases, and their syntax
Define ordinary procedures
Install transform procedures

Sample transform (3/5: ordinary recursive procedure)

(e1:define (closure-convert expression bound-variables)
 (e1:match expression
  ((e0:variable x)
   (e0:variable* x))
  ((e0:let let-variables bound-expression body)
   (e0:let* let-variables
    (closure-convert bound-expression
     bound-variables)
    (closure-convert body
     (set:union bound-variables
      let-variables))))
  ;; ... the actually interesting cases ...
 ))

In case you’re wondering:
  • e1:match is a macro (quite long, but no transforms are needed)
  • expressions are an ordinary sum type à-la-ML
    • sum types à-la-ML are defined with macros...

Luca Saiu — positron@gnu.org
GNU epsilon — savannah.gnu.org/bzr/?group=epsilon
Again ordinary procedure definitions, with the good “types”.

(e1:define (closure-convert-expression expression)
  (closure-convert expression expression set:empty))

(e1:define (closure-convert-procedure name formals body)
  (e0:bundle name
   formals
   (closure-convert body
    formals)))
(transform:prepend-expression-transform!
  (e0:value closure-convert-expression))

(transform:prepend-procedure-transform!
  (e0:value closure-convert-procedure))

From now on we can execute e1:lambda and e1:call-closure.

In case you’re wondering:

- Some transforms have to be applied retroactively (ex.: CPS)
- transform:transform-retroactively!
Tombstone diagrams: interpreters

Bootstrap $\varepsilon_0$ interpreter, $\varepsilon_0$ interpreter in C:

$\varepsilon_0$ interpreter, $\varepsilon_0$ interpreter in C:

$\varepsilon_0$

Guile

$\varepsilon_0$

C

$\varepsilon_1$ implementation:

$\varepsilon_1$

$\varepsilon_1$

$\varepsilon_0$

$\varepsilon_0$
Tombstone diagrams: runtimes

Guile runtime, efficient runtime:

```
  dmp
  Guile
dmp
  C
```
Unexec:

\[
\begin{array}{c}
\varepsilon_1 \\
\varepsilon_0 \\
\varepsilon_0 \rightarrow \text{dmp} \\
\varepsilon_0 \\
\varepsilon_0 \\
\varepsilon_1 \\
\varepsilon_0 \\
\varepsilon_0
\end{array}
\]

\(\varepsilon_1\) is built on top of \(\varepsilon_0\) by side effects, as a program. An interactive REPL is also effectively a program.