Università degli Studi di Pisa

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

Tesi di Laurea Specialistica

The epsilon project
A functional language implementation

Candidato

Luca Saiu

Relatore

Prof. Marco Bellia

Controrelatore

Prof.ssa Francesca Levi

Anno Accademico 2006-2007

University of Pisa

Faculty of Mathematical, Physical and Natural Sciences

Master Degree course in Computer Science

Master Degree thesis

The epsilon project
A functional language implementation

Candidate

Luca Saiu

Advisor

Professor Marco Bellia

Opponent

Professor Francesca Levi

Academic Year 2006-2007

No dedication

Abstract

The design and implementation of a functional language are presented, with particular emphasys on expressivity and performance; alternative designs and implementations of abstract machines for functional lanauges of differing complexity and maturity are also shown, and comparisons are drawn accordingly.

Acknowledgements

I guess that any attempt of avoiding being rhetorical would be doomed to fail here. But this being the situation I could also take the risk of being laughed at, just forget about it and write my well deserved half page of sugary, sentimental literature. I guess I show myself as what I am here, before switching (more or less) to an appropriately cold and impersonal style.
I'm in an advantage position here. I may also try to fake some anxiety, but we all know that this situation is one of the few ones in which mostly everything I could say would be forgiven.

Let me set one issue clear first: I've never given a damn about being politically correct and I've no intention to start now. I'm *not* going to pay my "due" homage to whoever is supposed to automatically get it just for some unwritten social obligation. It doesn't work that way with me.
By the way, unless it's still February 2007 and you share the questionable fortune of belonging to the little committee examining me in Pisa, you're probably not compelled in any way to read this. You may skip this section and jump right to the first chapter — Or you can even do a favor to yourself, close this book altogether and spend your time in some more constructive way, if you prefer.
There's nothing bad in that: languages are beautiful, but life is full of other interesting things. I assure you I'll understand.

This work is for people loving Computer Science and the few who have not yet forgotten the art of programming. Other people are welcome to try, but I doubt they'll ever *get* it.

epsilon has more than some significance for me. When you spend such a long time on a project or an idea I guess it's normal that something develops and grows, either an affective binding, or possibly even a strong desire of detachment.
I recognize myself in both images.
Several years after its beginning epsilon still boggles my mind, with periods of frustration and sudden unforeseen enlightenments. It's my personal quest for perfection where every decision has to be carefully pondered, where rough edges must be polished, entire parts destroyed and rebuit ever again just for an aesthetic sense of minimality, to get just one step closer to my illusion of total control.
It's not easy to explain to an uninitiated why one would want to do this; it's even harder to write about something so symbolically important; it requires commitment and attention, but not only that. Sometimes it's nearly painful.

I can't start but thanking all the people who spurred me to continue, first of all my friendly and patient advisor, professor Marco Bellia. I wouldn't write this if it weren't true.

I won't forget all the evenings spent with my good friend Carlo Bertolli in front of a couple of good beers, and all our talking about the world, Computer Science, politics, books, women, and the parallel implementation of epsilon which I haven't found the time to write yet.

I spent enought time with my fellows at the labs for them to become first coworkers, then friends. Forgive me for the alphabetical sorting, but I really would't know how to make you justice otherwise, buddies; you've all been important.
Thanks to Massimo Cecchi, Alessio Mazzanti, Francesco Nidito, Marco Righi, Erika Rossi, Dario Russo, Federico Ruzzier, Riccardo Vagli, Andrea Venturi.

I can't but remember my few long-time friends Alessio Baldaccini, Manuela Bellandi, Dimitri Dini, Cristina Favini and Mirco Giuntoli.

Thanks to Marco Riccio and Martino Sabia, who shared with me an apartment in Pisa, some good and bad moments, and sometimes my experimental cusine and insomnia.

Richard Stallman and the Free Software Foundation people, whose vision and unbreakable moral coherence are beginning to change the world, deserve more than a casual mention. As free software and part of the GNU Project epsilon is also part of my personal, still unworthy contribution to their honorable cause.

A sincere thank for their work in the battle against software patents goes to all the FFII people and especially to Jan Macek — I really hope I could find the time to contribute some more in the future. And I dearly remember the other people I met in my short experience of "active politics", particularly Enrico Rubboli.
You are writing history. Don't give up.

I greet my coworkers at ISTI-CNR: my roommates Michele Artini, Federico Biagini and Andrea Manzi, and our benevolent coordinator Lino Pagano.
I can't forget the very few CNR people who went through the pain of sharing with me a moment of sincere humanity when I needed it. Thanks.

I've met several outstanding professors at University in these years, who changed my way of thinking for ever; the best ones were also the most able to communicate their enthusiasm and love for Computer Science while teaching. With this work I'd also like to show them how their message reached me, in the hope of being able to propagate it in my turn.

And looking at the future I can but renew my deepest, most sincere thank to Jean-Vincent Loddo for trusting my work so much, to Massimiliano Brocchini for having remembered me, and to them both for providing me with an opportunity

which may give a completely new turn to my life.

Last but not least, I wish to greet some of the nice people I met at or around University in these years: Adrenalina, Aggino, Aggione, Michele Albano, Luca Bartoletti, Fabio Buono, Giulio Caravagna, Grazia Cimino, Patrizio Dazzi, Matteo Golfarini, Carmen Heger, Rosy Maggio, Giovanni Manunta, Antonio Mirarchi, Alessio Mochi, Giandomenico Napolitano, Marco Peccianti, Serena Ricci, Rioda, Rose Rossiello, Davide Taccola, Sandra Zimei.
Thank you all.

Of course I may have forgetten someone, and I apologize in advance if that's the case; but any friend whose name I unwillingly omitted wouldn't think for a second not to belong here.

Contents

Chapter 1

Introduction

epsilon[1] [2] is a *purely functional*, *statically typed*, *ω-order eager* language. The language design goals include expressivity, safety and minimality, while implementation efficiency and portability are primary concerns.

epsilon was initially conceived in late 2001 as an exercise in compiler implementation; both the language and the code have significatively matured since then but the openness to experimentation is all but changed, and everything in the implementation is to be considered "in flux".

As epsilon is meant to be –and has already been– a language actually used in production, its semantic simplicity is more a consequence than a cause of its practical effectiveness.
The language can be considered a very minimal tool for expressing algorithms and *is* suitable for formal reasoning, but the main concern has always been its practical suitability for building real systems, with the aim of extending to tasks not yet commonly associated with functional programming.

[1] In an effort to emphasize the language's stated minimality, the convention of always writing the name "epsilon" with a small "e" was adopted.

[2] epsilon is free software, released under the GNU General Public License [FSF91], and an official part of the GNU Project [Sta02]. See also [Sai07] and [FSF07] for more information and access to source code.

While the project matured and results were obtained epsilon grew more ambitious, and goals shifted somewhat. The current focus is on developing a set of modular and reusable tools suitable for experimentation with language implemantation (not limited to functional languages), without sacrificing execution efficiency.

1.1 Functional programming

The *functional paradigm* is a programming style where computation is based on side-effect free function evaluation.

In contrast with the more usual *imperative paradigm* based on state mutation, all *purely* functional languages do not support any form of assignment or flow control.

The natural way of expressing repeated computation is by *recursion*, and indeed it should be noted that in absence of state change and flow control the very idea of iteration becomes ill-defined.

The semantic core of all functional languages lies in the λ-calculus developed in the Thirties by Alonzo Church and Stephen Kleene ([Chu36], [Bar84]), which is well-studied and mathematically clean; as a consequence of this many functional programs tend to be smaller and easier to understand than their imperative counterparts.

Purely functional programs also satisfy desirable mathematical properties such as *referential transparency* making them particularly amenable to formal reasoning and automatic analysis and transformation by meta-programming.

Some features, albeit not characterizing, are available in all existing practical functional languages[3] and have come to be associated to functional programming in general: it is worth to mention at least *first-class functions*, *higher-order*, *type inference*, *algebraic data types* and *tuples*.

Again, despite this not being a characterizing feature, most implementations are *interactive* or at least include a *Read-Eval-Print Loop*.

[3] epsilon included.

Functional langugages have several advantages over traditional ones: they are especially well-suited to *symbolic computation*, which includes compilers and interpreters: functional lanaguages make good *metalanguages*.
The interactive interface of most implementations allow users to experiment with definitions and calls on the command line, thus shortening developement type and making debugging easier.

On the other hand, functional languages are not trivial to implement efficiently and pose some problems with the expression of *I/O* and *stateful computations*.

The most mature purely functional language is Haskell ([J+99]), especially notable as a *lazy* language adopting *monads* ([Wad95]) for I/O and state.
The family of impure functional languages includes Lisp ([KCR98], [ANSIITIC96]), the prototypical *homoiconic dynamically-typed* language originally developed by John McCarthy ([McC60]), and ML ([Har86], [LDG+03]), with its efficient implementations.

For an introduction to modern functional programming see, for example, [CM98] or [FFFK01].

The rest of this work assumes some familiarity with the basic concepts of functional programming.

1.1.1 Functional programming acceptance and significance

The functional paradigm is a radical departure from the conventional imperative programming style. Despite its introduction dates back at least to the Seventies ([Bac78], [Mil78]) with its roots firmly planted in McCarthy's work from the Fifties and Sixties ([McC60]) and despite its merits, the acceptance of functional programming is still hindered by the fear of shifting paradigms and by inefficient implementations feeding the misperception of inherent inefficiencies due to the

model itself.

One of the goals of this work is showing how a functional language can be implemented in a relatively simple way retaining very good execution speed.

Functional languages are typically developed in academia, without many connections with the industrial world[4]. A question naturally springs to mind: *if the "real world" has not embraced it, how is functional programming relevant today?*

A possible answer may be found examining the most recently introduced general purpose languages in common use. They are, in approximate chronologic order: Tcl ([Ous90]), Perl ([W+07]), Java ([GJSB07]), Python ([vR+96]), and C# ([HGW03]). All of them adopt the traditional impertative or impure object-oriented paradigm. No connection is immediately apparent between functional programming and the above languages, until they are looked at more closely:

- All the above mentioned languages rely on a form of automatic memory management at runtime.
- The so-called "dynamic languages"[5], i.e. Tcl, Perl and Python, are *dynamically typed*.
- The "static" languages Java and C# support some form of *parametric polymorphism*.
- Tcl and Perl support higher-order procedures. Java can be extended with libraries to also support them ([Bri07]), and C# delegates come close.
- All "dynamic" languages provide a form of `eval`. No "static" language does.

[4] But this trend may be slowly changing or at least there are exceptions: see for example [AAA97].

[5] Although widely used, the term "dynamic language" appears to lack a single agreed upon definition; for example compare [Wik] with [Asc04]. However there seems to be a general agreement on the facts that a dynamic language must be dynamically typed, that Tcl, Perl and Python are in fact dynamic, and that Java and C# are not. To avoid any ambiguity the term will be used here in this strictly *extensional* acceptation, and *static* will be taken to mean "not dynamic".

- C# supports a limited form of type inference. Of course "dynamic" languages do not need it.

Two interesting trends are visible here.
The first one was already observed by Paul Graham in [Gra02]. Paraphrasing: **"dynamic" mainstream languages are converging to Lisp**.
Graham could not notice the second trend when he was writing back in 2002, but it is very evident now: **"static" mainstream languages are acquiring more sophisticated type systems and higher order, converging to statically typed (impurely) functional languages**.

The "static" and "dynamic" approach are two different solutions, each with its pros and cons[6] but both are reasonable; the important fact to notice is that mainstream languages are slowly evolving to overcome their lack of expressivity, and **they are heading for the direction of functional programming**.
Even this reason alone would be enough to make functional programming "relevant".

1.2 Contribution of this thesis

This work shows how a practically-usable functional language can be implemented detailing the choices and trade-offs involved, and hints at its possible interesting applications especially as a *meta-language*.

From another point of view, a programming language implementation where efficiency is a real concern constitues an ideal example of programming in the widest scope of different abstraction levels, from the *highest* (modules, higher-order functions) to the *lowest* level (cache effects in garbage collectors: [BÖ00]).

The project is presented in its evolution across three particular snapshots of widely differing maturity and complexity.

[6] Pun intended.

The last snapshot in particular, although incomplete, hints at how *one* language implementation, however ambitious by itself, can be expanded into a radically more general system meant to serve as a group of components usable *for* languages –be they functional or not– and in fact not even restricted to languages.

Finally, showing the evolution of a realistic-size project (about 60,000 lines) and the decisions driving such evolution may be, in the author's hope, instructive for future endeavors by highlighting the choices which revealed themselves to be correct, and preventing some wrong ones.

1.3 Plan of the work

The rest of this thesis is organized as follows.
Chapter 2 presents the first implementation of epsilon as it was initially conceived for experimentation with functional languages and compilers, also showing the language core features.
The current implementation of epsilon with its features and shortcomings is discussed in some detail in Chapter 3, which also introduces some noteworthy tools and applications implemented in epsilon.
Chapter 4.1 just sketches an approximation of what the third implementation of the *epsilon Abstract Machine* may end up being, before hinting at some of the possible directions which the project may take in the near or not so near future.

'Tis pleasant, sure, to see one's name in print;
A book's a book, although there's nothing in 't.
— Lord Byron, *English Bards and Scotch Reviewers* (1809)

Chapter 2

The first implementation of epsilon

2.1 History

Although epsilon has not been officially released as stable yet, *three* distinct phases are already recognizable in its writing.
Of course there is a continuum in this process; in a couple of cases one part of an implementation has been used as a starting point for its matching part in the next implementation — but sharing of code between successive versions has always been very limited.

The best criterion for drawing a line distinguishing an implementation from the next one is based on shifting goals and raising ambitions. The decision to build a more elaborate implementation has always been driven by a conscious acknowledgement of the limits of one solution, and the will to overcome them by starting from scratch in a clean way[1].

The first implementation was started in late 2001 as an experiment with functional languages and compilers in general.

[1] This amounts to an endorsement of *The Right Thing* as a philosophycal position: [Gab91], [Bou92].

2.2 Getting a snapshot

A snapshot of the first implementation can be retrieved from the CVS repository with the command line:

```
cvs -z3 -d:pserver:anonymous@cvs.savannah.gnu.org:/sources/epsilon checkout -D 2002-03-01 epsilon
```

From now on any reference to a source file or directory in the first implementation is implicitly intended as relative to the snapshot above.

2.3 Goals

The first implementation was not ambitious. The main idea was just building a close-to-minimal functional language in a simple way but *from scratch*, using only C ([ANSI99]) with the scanner and parser generators flex and Bison ([Nic93], [FSF06]). No particular attention was given to efficiency or even expressivity: early on it was decided to leave out important features like type inference, concrete types and separate compilation. Even tracing garbage collection was completely avoided, for simplicity and for taking the opportunity of experimenting with a *reference counter*.

2.4 Language features

As all functional languages epsilon is *expression-based* and *statically-scoped*. For simplicity the first implementation did away with global definitions, and only allowed programs consisting of *single*, possibly large, expressions.

The language featured *abstraction*, *application*, an `if...then...else` *conditional* and *blocks*. Functions were single-argument only, but *currying* easily allowed for multi-argument functions and partial application.
The usual block constructs `let` and `letrec` enabled the user to write a set of possibly nested definitions in a relatively comfortable way. `letrec` was one[2] way to

[2]The other way was the `rec` operator, providing for "anonymous" recursive functions such as "`rec fact : integer -> integer . \ n : integer . if n = 0 then 1 else n`

express recursion.

Starting right from the first version imported into the CVS repository at [FSF07] in early 2002, epsilon supported *parametric polymorphism*, *first-class functions*, *ω-order*, *lists*, *tuples* and *promises* [3].

Ground types were *integer*[4], (double-precision only) *float*[5], *boolean*, *character*, *array* and *string*. The essential primitive operations were provided: *arithmetic* on integers and floats, *boolean operators*, basic *string* and *array operations* like concatentation, *cons* (infix "`::`"), `head` and `tail` to work on lists, *tuple selectors*, and little more.

I/O operations, only accessing the terminal, were expressed as side effects; this made the language not *purely* functional in its first incarnation.

Concrete syntax was inspired by ML and Haskell.

`* (fact (n - 1))`".

Note that the identifier bound to the function required a type declaration, as the first implementation compiler did type checking but not type inference. See also the use of `letrec` in the example at section 2.5.

[3]Promises were inspired by Scheme and Common Lisp: a succint definition can be found in [ANSIITIC96].

[4]Integers were implemented as C `int`s, i.e. "efficient" signed integers. The extremely lax requirements in [ANSI99] (few restrictions on word size, no fixed binary representation, undefined behavior on overflow) on this point allow very few guarantees in strictly portable code. In practice, even if this is *not* mandated by the Standard, it is normally assumed that integers are at least 32 bit wide on modern machines; not much more can be said, and behavior on overflow and underflow remains undefined also in epsilon, which "inherits" this intentionally incomplete specification.

[5]Floats were implemented as C `double`s, whose specification by the C Standard is intentionally lax. See the footnote above.

2.5 A sample program

The epsilon program which follows is written in the original syntax, and has been tested with a CVS snapshot from 1st March 2002:

```
letrec interval-with-accumulator : integer -> integer -> (list of integer) -> list of integer be
  \ x : integer . \ y : integer . \ acc : list of integer .
  if x > y then
    acc
  else
    interval-with-accumulator x (y - 1) (y :: acc)
in let interval be
  \ x : integer . \ y : integer .
  interval-with-accumulator x y [ ]
in letrec interval : integer -> integer -> list of integer be
  \ x : integer . \ y : integer .
  if x > y then
    [ ]
  else
    x :: (interval (x + 1) y)
in letrec reverse-with-accumulator : list of a -> list of a -> list of a be
  \ x : list of a . \ acc : list of a .
  if empty x then
    acc
  else
    reverse-with-accumulator (tail x) ((head x) :: acc)
in let reverse be
  \ x : list of a . reverse-with-accumulator x [ ]
in
  reverse (interval 1 (input_integer "Please write a number> "))
```

The program may be executed from the directory `epsilon/` with the command line:

```
epsilon/epsilon < SOURCEFILE > e.lvm && as/LVMas < e.lvm  > x && lvm/lvm
```

As in Haskell, "\" stands for "λ".

Note the type declarations after `letrec` bindings, and the nested blocks. The program is a single large expression.

2.6 Implementation features

The first implementation was based on two main components: a *compiler* translating the input program into a lower-level language, and a bytecode interpreter (henceforth called *abstract machine*[6]) executing it.
The only other tool included was an *assembler*, a simple program with the only purpose of turning the textual output of the compiler into a binary bytecode, more compact and suitable for interpretation.

The implementation ran on GNU/Linux systems and was fairly portable[7] and simple.
As a user interface was considered little more than a nuisance for such a small an experiment, it was kept to a bare minimum.

The implementation totaled about 5,000 source lines. Source code quality was quite good for an experiment.

2.6.1 Compiler

The compiler was written in a quite straighforward and conventional style for a *single-pass oblivious* translator implemented in C with flex and Bison; the bulk of the code resided in the *actions* ([ASU86]) from the parser attributed grammar.
Yacc-compatible parser generators like Bison somewhat encourage oblivious compilers, implementing actions as blocks of C statements with side-effects ([FSF06]);

[6] This does not accurately reflects the terminology used at the time. Anyway, even if the first implementation abstract machine was called a "virtual machine" in the original documentation, the term "abstract machine" is always used in this work. First for consistency's sake, then because it does not risk to be unintentionally bound in the reader's mind to *totally unrelated* ideas like bytecode interpreatation, which nowadays are becoming increasingly associated to the term.

[7] The author normally develops on a PowerPC ([Mot97]) laptop, but epsilon has been routinely tested on x86 ([Int95]) machines. The first implementation has never been tested on 64-bit platforms due to lack of available machines, but the source were written with portability in mind and would have been easy to port even to non-POSIX ([Wal95]) systems.

this may be reasonable for small- to moderate-sized[8] compilers like the first epsilon compiler.
Bison actions included type-checking, with type information propagated via attributes, other semantic checks and code generation proper.
All the *uses*[9] of predefined operators were recognized at a lexical and syntactical level as special cases, and code was accordingly generated.
At slightly more than 1,000 lines the parser source code (see `epsilon/epsilon.y`) was quite large, but still manageable.

Other relevant modules in the compiler dealt with term structures and *unification*, needed for type checking. See `epsilon/term.[ch]` and `epsilon/types.[ch]`.

Local and *non-local* variable resolution is discussed below, as it involves the environment representation at runtime.

2.6.2 Abstract machine

The original abstract machine was a quite straightforward and clean *imperative stack machine* with *explicit flow control*, and no particular provisions for efficiency.

A *reference-counted heap* was employed to store all epsilon data including *closures* and chained structures implementing environments via *deep binding*.
At the abstract machine level objects were limited to *integer*, *floats*, *arrays* (of pointers to objects) and *strings*: single characters were implemented as integers, while lists, closures and environments were represented by arrays.
Environments just held lists of arrays of values[10]: all identifiers were resolved at compile time and their uses translated into instructions performing the appropriate

[8] However it's worth highlighting once more the hardly-learned lesson that this solution does *not* scale to more complex compilers. See section 2.9 and especially section 3.11.

[9] "Uses" means "calls" and only calls in this case. Section 2.9 explains why this is a problem.

[10] Such arrays had most often length 1, as functions were restricted to one argument. However the `let` construct could bind more than one variable in the same local environment.

number of *steps* through the *static links* and eventually looking up an element (see `epsilon/environment.[ch]` for the implementation of this mechanism in the compiler). No identifiers survived in compiled programs.

Closures were represented in memory as simple two-element arrays whose first element was an instruction index, and the second and environment.

All objects were tagged with their (abstract machine-level) *type* and a *reference count*, always kept up-to-date by abstract machine instructions. This was accomplished by implementing each object as a different C `struct` *with the same two initial fields*[11][12], as this excerpt from `lvm/lvm.c` shows:

```
1  struct base_object{
2    int pins_number;
3    enum type_ID type;
4  };
5  struct integer_object{
6    int pins_number;
7    enum type_ID type;
8    int value;
9  };
10 struct float_object{
11   int pins_number;
12   enum type_ID type;
13   double value;
14 };
15 struct string_object{
16   int pins_number;
17   enum type_ID type;
18   char* value;
19   size_t size; /* this does not include the trailing '\0' */
20 };
21 struct array_object{
```

[11] Such a solution is not strictly portable according to [ANSI99] due to the theoretical possibility of padding introduced by the C compiler. This was not a problem in practice, as the fields were at the *beginning* of the `struct`s and always in the same order. "Unsafe" casts to `struct base_object*` were also not a problem, and in fact cases like that are usual practice in C.

[12] This case is a clear instance of inheritance, and could have been readily implemented in C++. However an object-oriented solution at this level was not considered appropriate as its first implication would have been wasting *another* word per object. Also note that using `struct`s *within* other `struct`s would not have solved the problem without sacrificing the very handy `struct base_object`.

```
22    int pins_number;
23    enum type_ID type;
24    struct base_object* value;
25    size_t size;
26  };
```

Such a uniform representation allowed for a relatively simple management at destruction time. This did not came without a cost, as it can be deduced by a look at the code above.

Recursive functions were implemented as closures whose environment contained a pointer to the closures themselves: the so-called *circular closures* created several problems, as reference counters are not suitable to manage *cyclic structures*. Some dirty workarounds were implemented to destroy them.

A program was a linear sequence of mostly very simple low-level stack instructions, with typically popped their parameters from the top elements of the *control stack* and pushed a result back onto it. Other instructions performed conditional or unconditional jumps, accessed the current *environment*, and performed *I/O*.

Subprogram calling conventions were stack-based, and some instructions were available to perform *call* and *return*.

Instructions were stored at runtime using a linear array of `structs`, each one restricted to an opcode and a single[13] integer argument, possibly an index relative to a table of immediates.
Note in particular that instructions were *not* normal heap-allocated objects[14].

[13] Not used for all opcodes.

[14] This choice, despite speeding up execution and simplifying the implementation, had the unfortunate side effect of preventing the machine from running self-modifying code, the only programming style actually more entertaining than the functional paradigm.

The *interpretation loop* was implemented in C as a `for` loop whose body consisted in single large `switch` discriminating on the current instruction opcode. The interpretation loop alone took nearly half of the 2,300 lines of C code making the abstract machine. See `lvm/lvm.[ch]`.

2.7 A translation example

The epsilon program

```
(\ x : integer . x + 1) 2
```

is translated into the following bytecode, expressed here in the same *textual notation* used in the compiler output:

```
1  main:
2          cls     FUN_1:  // make a closure with the label FUN_1 and the current environment
3          ldc     2       // push the constant 2
4          call    1       // apply the closure on the top
5  /* Output the result: */
6          outf            // output a primitive object
7  /* Exit with success: */
8          ldc     0       // push the "success" exit code onto the stack
9          exit            // exit, using the stack top as the exit code
10
11 /* Code for unnamed lambda-abstraction #1: */
12 FUN_1:
13         lcl     1       // access the first local named x
14         ldc     1       // push the constant 1
15         add             // sum the two top elements
16         retv            // return and push the value which is currently on top
```

2.8 Achievements

The experiment had some success.

epsilon has had some important features since its very beginning including *parametric polymorphism*, *ω-order* and *first-class functions*.

Lists were not implemented as algebraic objects in the first implementation, but nonetheless they were available and usable.

2.9 Limitations

This may be too much.

Despite the achievements, at the time of what could be called the end of the first implementation, say Spring 2002, some shortcomings became more and more evident.

First of all the oblivious style was beginning to prevent the compiler from scaling up. Executing scanning, parsing, semantic checks and code generation *in a single pass* prevented analyses and optimizations.

A similar problem in the compiler was due to the *unification* implementation in C; manual memory management is too complex to be used for such a task.
Some long standing bugs, including a serious one breaking the support for *nested tuples*, was extremely hard to fix due to the complex implementation of terms with C pointers, `malloc()` and `free()`.

Other minor problems remained in the compiler, for example the fact that predefined operators like "`head`", "`+`" and "`::`" were not implemented like ordinary functions. This implied that they could not be passed as parameters, returned or partially applied[15]. And of course each primitive operator contributed to clutter the compiler source code.

It became clear that C had not been not the most appropriate choice as a tool for writing compilers: a compiler is a complex program for symbolic manipulation, and some higher-level language –such as a functional language, including a more mature version of epsilon itself– could have been much more effective.

[15] However such limitations were easy to overcome in practice: even if the user could not pass "+" as a parameter, for example, it was always possible to pass "λ x . λ y . x + y". Nonetheless primitive operators were not *first-class objects* in a strict sense.

A full rewrite of the compiler was first considered around that time.

The lack of type inference was particularly annoying, as it forced the user to write long type declarations which could be obtained automatically, and had the additional effect of obscuring the code.

Concrete types were also essential for a truly usable language, and well as modules, abstract types, and global environment.
The abstract machine had been a nice experiment, but something much more efficent, and possibly extendable, could be built without a great effort.

Chapter 3

Current implementation

Around the Spring of 2002 the problems mentioned at section 2.9 became evident, and in order to overcome them –and on the other hand because of raising ambitions– a new implementation was undertaken with the intent of gradually replacing components one by one.
The implementation described in this chapter is a nearly complete rewrite.

3.1 Getting a snapshot

An up-to-date snapshot of the second implementation can be retrieved from the CVS repository with the command line:

```
cvs -z3 -d:pserver:anonymous@cvs.savannah.gnu.org:/sources/epsilon checkout epsilon
```

As in Chapter 2, references to individual files or directories are henceforth to be intended as relative to the source code snapshot above.

3.2 History and goals

The primary goal of epsilon, beyond experimantation, is still the implementation of a *practically usable* and simple functional language.
To be practically usable as a language epsilon still lacked some important features like *algebraic types*, *global environment* and *separate compilation*.
The features above were deemed essential to continue, but in this phase it was

decided to also expand the language in other directions, in part also for aesthetic reasons: it was decided that the new implementation would be *purely functional*, with *monadic I/O*.

A central idea was the development of a *self-hosting* compiler; but this required a mature implementation of epsilon for the bootstrap phase, which of course was not available. However a *subset* of the language could do the job if expressive enough, and a realistic path to that goal consisted in *upgrading the current compiler to the level required for writing a realistic compiler*.
Hence the "old" compiler written in C was updated to support a larger and more complex language, initially with the aim of eventually replacing it with a new self-hosting implementation.

Goals have somewhat shifted in the meantime, and although epsilon has now reached and surpassed the required level of maturity, its implementation is not yet self-hosting.

In the last years the language has evolved and has been heavily used[1] to build some applications and more tools, including some quite elaborate ones.
New needs and goals emerged from this experience, and the ultimate form which the project will take is still far from clear.

Meanwhile the abstract machines has grown more and more into a project by itself, extending its potential scope also to non-functional languages. An ideal abstract machine for epsilon should be a good basis for *any* kind of language implementation, general, portable and efficient.

[1] By the author, of course — there can not be a real user base until the language somewhat stabilizes. But some interested strangers leave an e-mail every now and then.

3.3 Language improvements

Several major features were added to the language, the foremost of which include *global environment*, *type inference*, *algebraic types* (called "concrete types" in epsilon), *abstract types*, *synonym types* (simple abbreviations for possibly long type names), *separate compilation* and *exceptions*. Side-effects I/O was replaced by a *monadic purely-functional I/O system*.
The language concrete syntax was changed to a degree, and the possibility of defining functions callable with *infix* and *postfix* syntax was provided.

Despite being much needed most changes were inspired by ML or Haskell, and as such they are not particularly interesting or innovative.

3.3.1 State of the compiler

As the changes mentioned above were simply thought as temporary kludges enabling to bootstrap a new compiler no particular care was taken in keeping the code "beautiful", and the compiler grew to its current 10,000 lines (see `compiler/epsilonparser.y`).
Anyway, despite the complexity of the current implementation being well past the limit reasonable for an oblivious translator, the compiler proved to be quite solid and in fact is still being used — before an alternative is carefully designed and implemented.

The main focus, instead, shifted to the abstract machine. The first implementation was self-contained and easy to replace altogether, requiring only minimal changes in the compiler for *retargeting*.

3.3.2 Sample programs

What follows is the content of an *interface file*[2]:

```
1  // An exported type definition:
2  define concrete type expression =
3    Number of integer
4  | Variable of string
5  | Plus of (expression * expression)
6  | Minus of (expression * expression)
7  | Times of (expression * expression)
8  | Divided of (expression * expression)
9  | Sin of (expression)
10 | Cos of (expression);
```

And this is the matching *implementation file*:

```
1  // A function definition:
2  define derivative = fix \ derivative . \ e . \ x .
3    match e with
4      Number(n) -> Number(0)
5    | Variable(v) -> if x =s v then Number(1) else Number(0)
6    | Plus(e1_e2) -> Plus((derivative (e1_e2 ^ 1) x),
7                          (derivative (e1_e2 ^ 2) x))
8    | Minus(e1_e2) -> Minus((derivative (e1_e2 ^ 1) x),
9                            (derivative (e1_e2 ^ 2) x))
10   | Times(e1_e2) -> Plus(Times((derivative (e1_e2 ^ 1) x), (e1_e2 ^ 2)),
11                         Times((derivative (e1_e2 ^ 2) x), (e1_e2 ^ 1)))
12   | Divided(e1_e2) -> Divided(Minus(Times((derivative (e1_e2 ^ 1) x), (e1_e2 ^ 2)),
13                                Times((derivative (e1_e2 ^ 2) x), (e1_e2 ^ 1))),
14                                Times(e1_e2 ^ 2, e1_e2 ^ 2))
15   | Sin(e1) -> Times(Cos(e1), (derivative e1 x))
16   | Cos(e1) -> Times(Number(-1), Times(Sin(e1), (derivative e1 x)));
17
18 // Another function definition:
19 define successor = \ x . x + 1;
20
21 // An object definition:
22 define n = successor 41;
```

A module is now a sequence of global definitions.

Note how the "`match`" at row 3 serves only to discriminate among constructors, and

[2]Interface files contain public declarations and are identified by the extension `.epi`. Definitions are contained in *implementation files*, which by convention have extension `.epb`

does not provide for full *parttern matching*. Also interesting are the occurrences of the *tuple selector* operator "^". The "`fix \`" at row 2 is just syntactic sugar for "`rec`".

No types are explicitly declared; it's now the compiler to dump information about the inferred types to the standard error, while translating:

```
derivative : (expression -> (string -> expression))
successor : (integer -> integer)
```

This exceprt from `examples/mubasic/mubasic.epb` makes a good example of I/O code written in an "imperative" style:

```
define main = begin
  assign args := get_command_line_arguments;
  assign program_text := read_whole_file_as_8bit_string (head args);

  try_io
    eval (parse_mubasic program_text);
  catch_io mubasic_parse_error into se -> begin
    output_string "row ";
    output_integer (se ^ 6); output_string ", column ";
    output_integer (se ^ 5); output_string ": ";
    output_string ("Parse error near `" @@s (se ^ 2) @@s "'\n");
    throw_io mubasic_parse_error se; // rethrow
  end
        | scan_error_exception into se -> begin
    output_string "row ";
    output_integer (se ^ 4); output_string ", column ";
    output_integer (se ^ 3); output_string ": ";
    output_string ("Scan error near `" @@s
                   (program_text from (se ^ 1) to (se ^ 2)) @@s "'\n");
    throw_io scan_error_exception se; // rethrow
  end;
end;
```

Many other examples of various complexity and maturity can be found in the subdirectory `examples/`.

3.4 The epsilon Abstract Machine

The *epsilon Abstract Machine* (*eAM* from now on) constitutes a particularly interesting component of the second implementation.

The eAM was designed with the goal of being a portable and efficient abstract machine to execute functional *and non-functional* programs[3].

The eAM operates at about the same abstraction level compared to the abstract machine of the first implementation: the model is imperative, instructions tend to be very simple, and as a rule they work on a stack. Flow control is explicit. The language interpreted by the epsilon Abstract Machine Language is abbreviated into *eAML*.

3.4.1 Compilation model

The eAM supports two distinct modes of operation:

Interpreter mode

In the first of its two modes of operation the eAM interprets programs in bytecode form, just like the original abstract machine. However such programs can be more easily manipulated via external tools such a *linker* allowing for separate compilation at the eAML level and also supporting *libraries* of eAML code, and a *peephole*

[3]Several already implemented alternatives are readily available and they have been considered, of course: popular abstract machines used as compiler targets include the *JVM*, the *CLR*, *Parrot* and *Neko*.

The reason why they were not used, beyond the desire of maximizing the possibility of control and the number of knobs to turn, includes disappointing performance results such as [SS02], [BSS04] and [DHR01].

It is evident that any "high-level" machine hiding such details as calling conventions is necessarily coupled to a programming style or idiom (typically object-oriented languages) and is not flexible enough to support a very different language with adequate performance.

On the other hand JITs, however complex and difficult to exploit they can be, usually *do* provide a performance advantage over simpler solution such the one described here; the current eAM can not offer a good answer to JITs, but a different approach based on traditional *ahead-of-time compilation* like the new *eAM* outlined in Chapter 4.1 might become, in the future, a convincing alternative.

optimizer.

The interpreter mode is the only practical way to use the eAM in most cases.

Compiler mode

In its second mode of operation the eAM translates an already li6nked eAML program into a *single* C source file which, when compiled and linked to a runtime library, can run with a substantial speedup (when appliable a speedup around 60% is fairly consistent along different small programs).

The speedup is obtained by *eliminating the virtual machine interpretation loop*, to a lesser degree with optimizations performed by the C compiler, and with an efficient implementation of jumps as *computed gotos* (see below).

The obvious problems caused by the potentially large size of the generated source code were initially understimated. Such an overlook revealed itself as particularly serious, as the condition is further exacerbated by an implementation technique consisting in using the same C source code for *implementing the eAM in interpreter mode*[4], and *as target code emitted in compiler mode*. Such a trick succeeds in avoiding code duplication and gives good results when appliable, but it involves heavy use of the C preprocessor, thus pushing memory requirements even higher.

The implementation of this essentially failed but interesting experiment can be found in `eam/` together with the needed shell scripts and an incomplete backend generating Scheme code.

3.4.2 A notable implementation feature

When in compilation mode the eAM uses *computed goto*, a non standard GCC extension ([S+03]) allowing C labels to be computed as results in an expression and held in memory.

Note however that even a *compouted goto* is subject to the usual restrictions on `goto` imposed by [ANSI99], which in particular *forbids* to jump out of the currently

[4]Some shell scripts are automatically called by `make` at build time to glue together the implementation of each instruction into the interpretation loop

enclosing block. This is the reason why a single large block must be generated in compiling mode — independently from the sort of `goto` which is employed.

3.4.3 Memory management and garbage collection

The eAM only works with *word-sized* or larger objects; this incurs practically no overhead, and makes the implementation simpler and more portabile: conditional compilation is employed at configuration time to discover the machine word size, which is then used as the size of all atomic objects. It's an advantage, for example, to be able to always store either a pointer or an integer in an array slots, as they are guaranteed to be of the same size.

The object types at the abstract machine level are exactly the same as in the first implementation; but note that in this case *no runtime tagging of objects is needed*, thanks to the garbage-collected heap.

The eAM employs a set of registers[5], a control stack, an exceptions stack[6], and a garbage-collected heap.

3.4.4 Garbage collector

The heap is automatically managed by a quite efficient mark-and-sweep garbage collector ([Wil94]) employing conservative pointer-finding, as it is needed in practice for interfacing with C code ([BW88]).

As the roots are always in known memory locations (i.e. stack and registers[7]), no non-portable code is needed to trace them.

The collector source code is in `eam/gc/`. It totals 2,000 lines.

3.4.5 Extendability

The eAM has some support for dynamically-linked binary code, using a wrapper of the `dlopen()` family of functions. Dynamic libraries can be loaded at runtime from

[5] Not currently used by the instructions generated by the epsilon compiler.

[6] This allows for more efficient location of the topmost `catch` block on a `try`

[7] Abstract machine registers are implemented in the main memory of the physical machine.

eAML and even epsilon has some rudimentary support for defining low-level interfaces to them[8][9]: for a simple example see this excerpt from `library/floats.epb`:

```
1 define pi = c_object "c_pi" from "float_math";
2 define e = c_object "c_e" from "float_math";
3 define sin = c_function "c_sin" from "float_math";
4 define cos = c_function "c_cos" from "float_math";
```

`library/floats.epi` specifies the signatures for all exported functions, so that modules using them can only interact with the library in a type-safe way:

```
1 declare pi : float;
2 declare e : float;
3 declare sin : float -> float;
4 declare cos : float -> float;
```

The eAM can be extended with new instructions in an extremely simple way, by adding its implementation in C as new file under the appropriate subdirectory[10] within `eam/c_instructions/`. Recompiling the eAM automatically regenerates the interpretation loop for the *interpreter mode* and updates the code to emit in *compiler mode*.
This sort of extension, however, requires recompilation of the eAM and of all the code to interpret.

3.5 Library

The current implementation of epilon includes a small library, developed piece by piece as the need arised for a particular application or experiment.
Other times some code was simply extracted from an epsilon application and moved into the library. Higher-order and polymorphism appear to make *code reuse*

[8] Of course type inference can not be used in these cases. Types must be carefully declared, and when they are wrongly specified programs react by *crashing*.

[9] Libraries are normally wrapped by other simple C libraries (`float_math` in this example) with word-sized parameters and results, for compatibility with the eAM mamory model.

[10] Instructions are classified according to the type and number of their parameters.

particularly natural.

A brief description of the most important library modules is provided here also because of the interesting role of several such modules as "case studies" about the *practical* usability of a functional language.

The code can be found in the `library/` subdirectory.

3.5.1 Associative data structures

Several associative containers are developed around a single polymorphic implementation of AVL trees ([AL62]). The implementation is very well tested as both `epsilonlex` and `epsilonyacc` heavily rely on it, but usability suffers from the need of explicitly supplying comparison functions at container creation time:
as the operations on AVL trees –and on *binary search trees* in general– depend on the ordering of the elements, two comparison predicates "less" and "equal" of type τ `->` τ `->` `boolean` are needed to be able to work on a `tree of` τ.

Using containers like these shows the practical need for some sort of subtype polymorphism *in addition to* parametric polymorphism, in a very compelling way.

Included containers include `set`, `multiset`, `associative_array`, and `multimap`.

3.5.2 Utility functions

A collection of the "usual" *higher-order* functions like `compose` and `iterate` is supplied.
Another set of widely used functions have the purpose of easing the work with lists. They include functions like `length`, `map`, `append`, `fold_left` and `fold_right`.
An implementation of the usual *trigonometric* and other *transcendent* functions on floats is provided.

3.5.3 S-expressions and terms

An implementation of *S-expressions* ([McC60]) including reader and printer was obtained from *μ-lisp* (see section 3.9).
This module has revealed itself useful several times for experimenting with an interpreter without a set-in-the-stone concrete or abstract syntax, and also as glue for exchanging data with Lisp implementations[11].

An implementation of terms with variables and *unification* derived from *μ-prolog* is also provided. It is suitable for being used in type checkers but suffers from the *state problem* discussed at subsection 3.11.

3.5.4 Channels

Channels are generic communication interfaces similar to Scheme ports ([KCR98]), performing some communication with some external entity.
After being created with a primitive appropriate to the channel type, each channel exposes the exact same interface.
The currently implemented channel types are files and terminal I/O. With the same idea other interesting communication endpoints like network sockets could be implemented.

3.5.5 Other interesting modules

Bignums built on the GMP library ([Gra04]) are provided, supporting basic arithmetics, reading and printing.
A small module built on SDL ([SDL]) enables access to some simple graphic primitives.

[11] S-expressions would also make a good data interchange format. This idea is far from new ([McC90]) but today violating the XML orthodoxy appears to be unthinkable.

3.6 REPL

A quick-and-easy REPL implementation is provided. The REPL works by transparently invoking the epsilon compiler and the eAM in interpretation mode; the REPL is often handy for quick tests, but a real interpreter would have some value for developing.

3.7 `epsilonlex` and `epsilonyacc`

The `epsilonlex` scanner generator and the `epsilonyacc` parser generator are the most mature and complex, if not very large, epsilon applications.
Both follow the algorithms in [ASU86]; `epsilonlex` is quite common in being based on regular grammars. `epsilonyacc` makes up a more daring experiment, being a *canonical LR(1)* parser generator.

Both tools employ complex algorithm and show examples of *the state problem* in several cases. See section 3.11.
`epsilonlex` and `epsilonyacc` are *mutually self-hosting*: each of them depends on itself and on the other one for its frontend.

3.7.1 A frontend example

This is referred to the same interface shown at .

An `epsilonlex` scanner follows:

```
1  interface_code {}
2  implementation_code {}
3  grammar
4    <digit>      ::= from 0 to 9;
5    <letter>     ::= (from a to z) | (from A to Z);
6    <identifier> ::= <letter> (<letter> | <digit> | _ )*;
7    <comment>    ::= ' [ \n ]* \n;
8    <whitespace> ::= (\  | \t | \n)+;
9  rules
10   <whitespace>        ignore
11   <digit>+            { integer_constant_token }
12   \+                  { plus_token }
```

```
  -                 {minus_token}
  \*                {times_token}
  /                 {divided_token}
  sin               { sin_token }
  cos               { cos_token }
  \(                { openpar_token }
  \)                { closepar_token }
  <identifier>      { variable_token }
```

This is an `epsilonyacc` parser:

```
interface_code { import derivative; }
implementation_code {}
nonterminals
  <expression> : { expression }
terminals
  integer_constant, variable, plus, minus, sin, cos, times, divided
  openpar, closepar
<expression> ::=
  integer_constant { Number(string_to_integer $1_text) }
| variable       { Variable($1_text) }
| <expression> plus <expression> {Plus($1, $3) }
| <expression> minus <expression> {Minus($1, $3) }
| <expression> times <expression> {Times($1, $3) }
| <expression> divided <expression> { Divided($1, $3) }
| sin openpar <expression> closepar { Sin($3) }
| cos openpar <expression> closepar { Cos($3) }
| openpar <expression> closepar { $2 }
```

3.8 Recognition

In December 2002 the author had the honor to see epsilon officially accepted into the GNU Project by Richard Stallman ([GNU], [Sta02]).

3.9 Applications

Several applications were developed and are distributed with in epsilon.

Several programming language interpreters are provided: as their name say they are minimal versions of popular languages: in growing order of difficulty: *μ-basic*, *μ-lisp*, *μ-prolog*. They can all be found under `examples/`.

An implementation of call-by-name λ-calculus can be found at `examples/lambda_calculus/`.

Three classic graphic hacks were implemented: *Wander*, the *Sierpinsky's triangle* and a simple *function plotter*, whose most complicated part is the parser.

3.9.1 ICFP 2004 and 2005

The author took part to the ICFP programming contest (see [ICF06]) using epsilon. The 2004 problem dealt with compilers, and the result was quite good. A writeup can be found here: [Sai04].

3.10 Achievements

All the stated goals regarding the language were essentially met. The implementation, although not completely mature and polished, is usable. Efficiency is good, and the system is very portable. The new implementation was also ported on Sparc and UltraSparc processor with no problems.
The abstract machine can be extended by the user.

3.11 Shortcomings

3.11.1 The "state problem"

The lack of some epsilon construct to express stateful computation has became evident in several cases in the author's experience.
An example can be found in the term implementation in `library/term.epb`, where the author was forced to manually "sequence" stateful operation, not having a monad available.
But Haskell-style "state monads" ([J+99]) would provide some relief without solving the core problem. Some particularly clear examples involve the generation of a fresh variable or of a random number. When the user decides to compute an object based on some implicit "state" information, hence using a monad, a very large

part of the program text *surroounding* the changed part has to be restructured to accommodate for a cascade of the type changes.
Said in another way, introducing a monad in a local effect tends to have e *global* effect.
Despite all the problems it implies, mutable state appears to be more powerful as an abstraction tool.

The problem remains open.

3.11.2 Other shortcomings

The need for subtype polymorphism is well expressed by the example at section 3.5.1.

Pattern matching[12] would made the language much more concise and easy to use.

A new compiler is needed, but first it should be decided exactly what kind of language should be supported.
The current eAM has an acceptable performance, but it can be made faster. Its source code optimized for performance is quite difficult to follow.

[12] Or much, much better: some way to allow the user to implement pattetn matching in a functional setting. See [Ste98]

Chapter 4

Conclusions and future work

4.1 A more radical experiment

After a hiatus a new, *third* implementation of an epsilon Abstract Machine was began in late 2006 as a way to overcome the inefficiencies of the current eAM. The implementation is still immature, and the new implementation was conceived *bottom-up* as a conscious design decision: the abstract machine must be still lower-level and more efficient, even at the cost of sacrificing simplicity.

It is not clear yet what kind of language this infrastructure will be used for. Ideally this should be a work *for* languages more than *on a language*.
A snapshot can be retrieved from CVS at

```
cvs -z3 -d:pserver:anonymous@cvs.savannah.gnu.org:/sources/epsilon checkout eam
```

Many choices, constraints and possibilities are still nebulous, but below some *already implemented* ideas are quickly sketched. Of course some may be discarded in the future.

- No predefined primitive data and operations (the user can define them in C).
- Register-based, for maximum efficiency. Abstract machine registers can and should be allocated in hardware registers even if programming in C. This can be done, as the implementation shows.

- No predefined runtime structures: stacks and heap can be added by the user *if* needed.
- No fixed calling conventions: different languages and even different function may require ad-hoc calling conventions.
- eAML should support macros to be easier to generate code for, and for human programmers.
- No bytecode, no interpretation loop in software: only native-code compilation
- new mark-and-don't-sweep ([Wil94]) garbage collector optimized for tagged data, hence ideal for concrete types

The new experimental compiler at `stuff/benchmark/mubasic/`, running on the old eAM and generating code for the new one, seems to generate very high-performance code.

4.2 Future directions

A project like epsilon can be expanded in most any conceivable direction and lends itself to very creative experimentations. But implementations should be first of all usable and solid, and wild experiments should also make place to consolidation work.
A library should be built and possibly managed in a collective way if a user base of any dimension; documentation should be written. Some public repository organized like [CPA] might work.

One possible interesting pursuit would be a system for distributed programming, with either implicit or explicit communication — or even both, at different levels.

A functional language tends to be the right tool for manipulating other languages — and nowadays this implies work on the web, on databases, on biological

data.

Functional languages can do all of this and more, but some theorethical questions are still to be settled — and usability questions. Linear logic ([Cle]) could be a promising direction, for example.

Bibliography

[AAA97] Joe Armstrong, Thomas Arts, and Ericsson Telecom Ab. Erlang and its applications, December 18 1997.

[AL62] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1962.

[ANSI99] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1999.

[ANSIITIC96] American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. American National Standards Institute, 1996.

[Asc04] David Ascher. Dynamic languages — ready for the next challenges, by design. http://activestate.com/Company/NewsRoom/whitepapers_ADL.plex, July 2004.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[BÖ0] Hans-Jürgen Böhm. Reducing garbage collector cache misses. In *ISMM*, pages 59–64, 2000.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978.

[Bar84] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[Bou92] Nick Bourbaki. Worse is better is worse. *AI Expert*, 7(5):18–21, May 1992. Note that Nick Bourbaki is Richard P. Gabriel's pseudonym, and compare with [Gab91].

[Bri07] Björn Bringert. HOJ - Higher-Order Java. http://www.cs.chalmers.se/~bringert/hoj/, February 2007.

[BSS04] Yannis Bres, Bernard P. Serpette, and Manuel Serrano. Bigloo.NET: compiling scheme to.NET CLR. *Journal of Object Technology*, 3(9):71–94, 2004.

[BW88] Hans-Jürgen Böhm and Mark Weiser. Garbage collection in an uncooperative environment. *Software–Practice and Experience*, 18(9):807–820, September 1988.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics,*, 58:345–363, 1936.

[Cle] The Clean language. http://www.cs.kun.nl/~clean.

[CM98] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[CPA] Comprehensive Perl Archive Network. http://www.cpan.org.

[DHR01] Tyson Dowd, Fergus Henderson, and Peter Ross. Compiling Mercury to the .NET Common Language Runtime, November 05 2001.

[FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shiram Krishnamurthi. *How to Design Programs*. The MIT Press, 2001.

[FSF91] Free Software Foundation. GNU General Public License. http://www.gnu.org/licenses/gpl.html, 1991. Version 2.

[FSF06] Free Software Foundation. The Bison homepage, 2006. http://www.gnu.org/software/bison.

[FSF07] Free Software Foundation. The epsilon project page on GNU Savannah. http://savannah.gnu.org/projects/epsilon, 2001-2007.

[Gab91] Richard P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, June 1991. This includes the famous discussion about *The Right Thing* and *Worse is better*. Compare with [Bou92].

[GJSB07] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. The Java language specification, 2007.

[GNU] The GNU Project. http://www.gnu.org/gnu.

[Gra02] Paul Graham. Revenge of the nerds. http://www.paulgraham.com/icad.html, May 2002.

[Gra04] Tobjörn Granlund. *The GNU multiple precision arithmetic library*, 2004. Version 4.1.3.

[Har86] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, November 1986.

[HGW03] A. Hejlsberg, P. Golde, and S. Wiltamuth. *C# Language Specification.* Addison_Wesley, October 2003.

[ICF06] The ICFP programming contest home page. http://www.icfpcontest.org, 2006.

[Int95] Intel. *Pentium Family: Developer's Manual*, 1995. Pentium Processor.

[J+99] Simon Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available from http://www.haskell.org/onlinereport, February 1999.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[LDG+03] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon). *The Objective Caml system, Documentation and user's manual*, release 3.07 edition, 2003. http://caml.inria.fr/ocaml/htmlman.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. ACM*, 3(4):184–195, April 1960.

[McC90] John McCarthy. The Common Business Communication Language. In Vladimir Lifschitz, editor, *Formalizing Common Sense: Papers by John McCarthy*, pages 175–186. Ablex Publishing Corporation, Norwood, New Jersey, 1990.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17(3):348–375, December 1978.

[Mot97] Motorola. *PowerPC microprocessor family: the programming environments for 32-bit microprocessors*, 1997.

[Nic93] G. T. Nicol. *Flex: The Lexical Scanner Generator, for Flex Version 2.3.7.* Free Software Foundation, Inc., 1993.

[Ous90] J. K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 133–146, Berkeley, CA, 1990. USENIX Association.

[S+03] Richard Stallman et al. *Using GCC: the GNU compiler collection reference manual*. GNU Press, pub-GNU-PRESS:adr, 2003.

[Sai04] Luca Saiu. Writeup documenting Luca Saiu's entry at the 2004 ICFP programming contest, with source code. http://www.di.unipi.it/~saiu/icfpc/2004, 2004.

[Sai07] Luca Saiu. The epsilon home page on the GNU project web site. http://www.gnu.org/software/epsilon, 2001-2007.

[SDL] Simple Directmedia Layer. http://libsdl.org/.

[SS02] Bernard Paul Serpette and Manuel Serrano. Compiling Scheme to JVM bytecode: a performance study, 2002.

[Sta02] Richard M. Stallman. Personal communication, December 2002.

[Ste98] Guy L. Steele. Growing a language. In *OOPSLA '98 Addendum: Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, New York, NY, USA, 1998. ACM Press.

[vR+96] Guido van Rossum et al. *Python Reference Manual*. Stichting Mathematisch Centrum, Amsterdam, 1996.

[W+07] Larry Wall et al. Perl online documentation, 2007.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.

[Wal95] Stephen R. Walli. The POSIX family of standards. 1995. http://portal.acm.org/citation.cfm?id=210315\&coll=portal\&dl=ACM.

[Wik] Wikipedia. Dynamic programming language. http://en.wikipedia.org/wiki/Dynamic_programming_language.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.