Pico: Scheme for Mere Mortals.

Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker
{wdmeuter, tjdhondt, jededeck}@vub.ac.be

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium

Abstract. In this paper we present the results of an extensive experiment to adapt a Scheme-like programming language called Pico to the requirements of a less-than-willing contemporary computing audience. Pico is a Scheme derivative that is equally powerful in the sense that everything – including programs and continuations – is first class. Yet, Pico uses classic infix syntax and fixed size arrays without losing the elegance and expressiveness of Scheme's list-based approach. Pico has no special forms but replaces Scheme's keywords by ordinary function calls. The paper enumerates Pico's major design concerns and speculates on their usefulness in a more general setting.

1 Introduction.

Scheme is arguably one of the most concise, expressive and elegant programming languages ever built. It is also the vehicle for SICP [sicp], one of the most remarkable visions on computing written down on paper. The application of both the language and the vision in an educational context have generally led to a kind of computational power-training unrivaled by other approaches.

Nevertheless, the use of Scheme in teaching has been in constant decline for the past decennium. The reason is not difficult to identify: the resistance of the general computing public to a less than *canonical* language has been rising during this same period. History seems to be at the recurring point where many are focused on solving problems using one *universal language* and insofar it is statically typed, class-based and garbage collected most seem satisfied. The result of course is that we have reached a low point in linguistic diversity; fewer and fewer professionals master fundamental concepts (ask recent computer science graduates to explain the difference between a closure and a continuation and prepare to be surprised) and they go happily about re-inventing a concept every time the need for one manifests itself.

On the other hand, the internet is a breeding ground for whole new generations of software applications. And we cannot but question the suitability of current development practice to many of the new requirements. Consider for instance the painful application of reflection in the Java programming model or the poor grade of data interchange formats introduced by XML.

Enter Scheme, or at least a programming model similar to Scheme that can lure people away from their day-to-day practice and expose them to the rich diversity of programming concepts in an easily recognizable setting. In the spirit of Scheme we aim for a simple set of basic constructs; a language emerges as their closure. But this process is effectively a balancing act: the expressiveness and elegance of the language semantics should not trivialize the external syntax to the point where it becomes unattractive. On the other hand, syntactic constructs should not invalidate the semantic power of the language.

This paper describes Pico[1], a language in which we seek such a balance. It is the result of several years of experimentation with generations of non-CS-major students at undergraduate and graduate level who unwittingly became co-designers. Pico is very close to Scheme in spirit; it features strong and dynamical typing, static scope, first class everything. However, it has a significantly different syntax without special forms, thanks to a richer parameter binding semantics. Also, Pico uses tables rather than pairs as basic data structure. Consequently Pico has a look-and-feel which is different from Scheme and which would seem to make it more appealing to a contemporary audience.

A coarse estimate[2] indicates that essential programming concepts can be acquired using Pico in less than half the time it takes using Scheme. But although initially Pico was only used for teaching [pico], its expressiveness made it a suitable candidate for research oriented experiments relative to code mobility [borg], prototype based inheritance [lmo] and distribution [soap].

Pico is indeed the result of a balancing act and an analytical mind may consider certain of its features as somewhat redundant. However – slightly misquoting Leibniz – we chose that which is the most simple in hypotheses and the most rich in phenomena. And *simple* does not necessarily signify *least in number*.

2 Pico basics.

Consider the Pico code fragment in figure 1. It features a higher order function `set` that accepts an arbitrary list of items (actually either numbers or strings). `set` returns a `member` predicate that decides whether its argument `Item` belongs to the originally specified set of `Items`.

The implementation is fairly straightforward: `set` constructs a binary tree (using the function `add`) which is stored in the local variable `TREE`. The returned predicate `member` uses a local function `get` to traverse `TREE`.

This example illustrates several of Pico's features:

– constant, variable and function declarations using `:` and `::`
– table access using standard `[ ... ]` notation
– variable and table assignment using `:=`
– compound expressions using { and } delimiters

[1] as in 10^{-12}, i.e. very small.
[2] comparing first year programming courses for mathematics (Pico) and computer science (Scheme) undergraduates.

– compound values using [and] delimiters
– canonical and applicative parameter lists (the latter using the @ symbol)
– first class argument lists via the use of @
– call-by-name parameter binding in the definition of add
– the use of infix operators
– the use of control structures such as if and for in standard function notation

```
set @ Items:
   { nbr_idx:: 1;
     lft_idx:: 2;
     rgt_idx:: 3;
     get(Item, Node):
       if(is_table(Node),
         if(Item > Node[nbr_idx],
           get(Item, Node[rgt_idx]),
           get(Item, Node[lft_idx])),
         Item = Node);
     add(Item, Node, Thunk(Tree)):
       if(is_table(Node),
         if(Item > Node[nbr_idx],
           add(Item, Node[rgt_idx], Node[rgt_idx]:= Tree),
           add(Item, Node[lft_idx], Node[lft_idx]:= Tree)),
         if(Item > Node,
           Thunk([Node, Node, Item]),
           if(Item < Node,
             Thunk([Item, Item, Node]))));
     if(size(Items) = 0,
       member(Item):: false,
       { TREE: Items[1];
         for(idx: 2, idx<=size(Items), idx:=idx+1,
           add(Items[idx], TREE, TREE:= Tree));
         member(Item):: get(Item, TREE) }) }
```

Fig. 1. example of Pico code

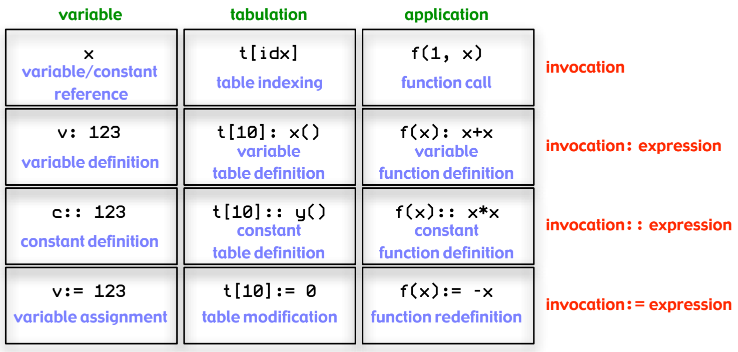
Figure 1 illustrates what Pico code typically looks like. It is syntactically close to a generally accepted format and it is conceivably more accessible than Scheme for a broad range of individuals, be they Java programmers or calculus adepts. But anyone familiar with Scheme will not fail to recognize its impact on the design of Pico, even from this small example.

For the rest of this section we will address the top ten concerns that have driven us in our design of Pico.

2.1 Concern #1: a straightforward syntax using a regular grid.

Pico features a fairly rich syntax that nevertheless takes less than one page to formulate. It incorporates a conventional operator syntax (see concern #2) but does not require the artifact of special forms (see concern #3). A limited set of syntactic tokens fixes the structure of a Pico program (to wit: matched quotes, parentheses, brackets and braces, period, comma and semicolon separators, completed by `@`, `:`, `::` and `:=`). A fundamental notion is the invocation which is either a reference, a tabulation or an application. An invocation is used in four modes: access, variable definition, constant definition and assignment.

This structure is captured in the grid in figure 2. It illustrates how twelve different Pico program expression types are constructed by combining the three invocation types into the four modes. An actual Pico evaluator is essentially required to provide specialized interpreters for these twelve expressions[3].



variable	tabulation	application	
`x` variable/constant reference	`t[idx]` table indexing	`f(1, x)` function call	invocation
`v: 123` variable definition	`t[10]: x()` variable table definition	`f(x): x+x` variable function definition	invocation: expression
`c:: 123` constant definition	`t[10]:: y()` constant table definition	`f(x):: x*x` constant function definition	invocation: : expression
`v:= 123` variable assignment	`t[10]:= 0` table modification	`f(x):= -x` function redefinition	invocation:= expression

Fig. 2. Pico syntax grid

2.2 Concern #2: operator notation as syntactical construct.

Pico recognizes prefix and infix operator syntax. Essentially the following syntactic equivalences hold:

operator(expression) ≡ operator expression

operator(expr1, expr2, ...) ≡ expr1 operator expr2 operator ...

for all applications of an operator, i.e. a concatenation of characters from the set:

{ < , = , >, #, ˜, $, %, +, - , | , &, * , /, \ , !, ? , ˆ }

[3] a metacircular definition for these twelve routines requires about 200 lines

This set is partitioned into:

Rel = { `<` , `=` , `>`, `#`, `~` }
Add = { `$`, `%`, `+`, `-` , `|` }
Mul = { `&`, `*` , `/`, \ }
Pow = { `!`, `?` , `^` }

with Rel $\ll$ Add $\ll$ Mul $\ll$ Pow. Operator precedence is defined as follows:

$$\begin{aligned} &p\ \phi\ q\ \psi\ r \\ &\quad = (p\ \phi\ q)\ \psi\ r \text{ iff } \phi \geq \psi \\ &\quad = p\ \phi\ (q\ \psi\ r) \text{ iff } \phi < \psi \end{aligned}$$

The precedence between 2 operators ϕ and ψ is defined by

$\phi < \psi$ iff initial(ϕ) ϵ V, initial(ψ) ϵ W and V $\ll$ W

Consider as an example the following definitions:

```
a++b: a+b+1
a**b: a*b*2
p<=>q: abs(p-q)<1
```

Applying the above precedence rules, `1++2<=>1**2`, will return `true`.

2.3 Concern #3: avoiding special forms via call-by-function.

Pico does not require special forms in order to support block structure, recursion, conditionals, etc. It features a richer parameter binding semantics than Scheme in order to introduce an applicative order mechanism; but this particular extension of formal parameters is completely integrated in the Pico spirit.

Whenever a Pico function is defined, the programmer has the option of specifying a formal parameter as an invocation (see concern #1). A parameter specified as a reference will be bound by value at application time; but if it is specified as an application we will have an extension of call-by-name, which we will label call-by-function. Consider the example in figure 3.

```
map(f(val), tab)::
  { idx: 0;
    res[size(tab)]:: f(tab[idx:=idx+1]) }
```

Fig. 3. example of call-by-function

The function `map` implements a Scheme-like map: a call to `map(val*val, [1, 2, 3, 5, 7])` generates `[1, 4, 9, 25, 49]`. In this particular example, the call-by-value parameter `tab` will be bound to the value of `[1, 2, 3, 5, 7]` while the call-by function parameter `f(val)` will be bound to the expression `val*val`. This actually means that during the application of `map` a local variable `f` will be bound to a closure consisting of the parameterlist `(val)`, the body `val*val` and the calling environment of `map`[4].

The package in figure 4 seeks inspiration in the λ-calculus in order to provide basic booleans in Pico. Note that `true` and `false` are binary functions and that conjunction, disjunction and negation are effectively implemented as Pico operators. Both `&` and `|` perform lazy evaluation, thanks to the use of call-by-function.

```
{ true(consequent(), alternative())::
    consequent();
  false(consequent(), alternative())::
    alternative();
  p&q()::
    p(q(),false);
  p|q()::
    p(true,q());
  !p:: p(false,true) }
```

Fig. 4. a boolean package

We conclude this subsection with a simple implementation of a Pico while iterator (see figure 5).

```
while(predicate(), expression())::
  { loop(value, boolean)::
      boolean(loop(expression(), predicate()), value);
    loop(void, predicate()) }
```

Fig. 5. a Pico while iterator

2.4 Concern #4: variable arity through first-class parameter lists.

Pico features first-class parameter lists by means of the `@` construct[5]. Whenever a function is defined with a `@`-based application, the reference (or other invocation) following the `@` will be bound to a table of argument values during application.

[4] note that `val` has dynamic scope
[5] reminiscent of the Scheme `apply`, but in this case `@` is a syntactical construct

```
begin@tab:: tab[size(tab)]                table@tab:: tab
```

Fig. 6. variable arity in Pico

Two simple examples in figure 6 suffice to illustrate the usefulness of `@`. The function `begin` accepts one or more arguments, evaluates them one by one, and returns the last value. `begin` serves as a compound expression constructor which is typically used to such an extent that the Pico parser provides syntactic sugar to minimize parenthesis paralysis:

`begin(exp1, exp2, ... , expN)` ≡ `{exp1; exp2; ... ; expN}`

The function `table` accepts zero or more arguments, evaluates them one by one, and returns a table containing all of the resulting values. `table` serves as a compound value constructor; in analogy to `begin`, the Pico parser provides syntactic sugar:

`table(exp1, exp2, ... , expN)` ≡ `[exp1, exp2, ... , expN]`

2.5 Concern #5: recursion through function naming.

Pico requires that all functions be named[6] – and consequently, λ-expressions are absent from Pico. The evaluation of a function definition:

```
fun(par, ...): body
```

results in a closure being bound to `fun`. The static environment present in this closure will specifically contain this binding in order to support recursion[7].

Anonymous functions (λ-expressions) are typically used as a filter in collection oriented operations or are passed around as thunks (see figure 1). Figure 3 is an example of the former; it could however also have been written as in figure 7 and used as in `map(f(val): val*val, [1, 2, 3, 5, 7])`. We use the first-class-everything property of Scheme and pass a function definition as argument to `map`[8].

[6] which proves beneficial to debugging.

[7] this is reminiscent of the Scheme `letrec` construct; but contrary to Scheme which features four variants of the `let`-structure, functions are the only scope constructor in Pico

[8] contrary to Scheme, where a `define` is not allowed as an argument to a function. However, in `map(f(val): val*val, ...)` the scope of `f` is no longer the body of `map`, which is why we prefer the call-by-function solution

```
map(f, tab)::
  { idx: 0;
    res[size(tab)]:: f(tab[idx:=idx+1]) }
```

Fig. 7. figure 3 revisited

2.6 Concern #6: composite values using tables.

Pico proposes tables for the composition of values. A table is instantiated as an in-line value (see concern #4) or by using basic syntax [9]. Figure 8 for instance defines a 3-dimensional unit matrix `[[1, 0, 0], [0, 1, 0], [0, 0, 1]]`.

```
{ unity[3, 3]: 0;
  for(i: 1, i<=3,  i:= i+1, unity[i, i]:= 1) }
```

Fig. 8. a unit matrix

The fact that Pico tables require dedicated syntax constructs is inspired by common practice; but at the same time it provides a richer semantics. This is illustrated by figure 9 which binds `[[1], [2, 3], [4, 5, 6]]` to `triangle`.

```
{ i: j: 0;
  triangle[3]: t[i:= i+1]: j:= j+1 }
```

Fig. 9. a triangular matrix

This example shows that during the definition of a table, the initialization expression is re-evaluated for each table slot. Actually, figure 8 could have been written as follows:

```
{ idx: 0; unity[3,3]: if((i:=i+1)\\4 = 0, 1, 0)}
```

Fig. 10. the unit matrix revisited

Pico tables are sufficiently expressive as to support list structures. Consider the Pico code in figure 11: it provides an implementation for Pico's own environ-

[9] contrary to Scheme, where native functions are used to create and manipulate vectors

```
{ add(nam, val, env):
    [ nam, val, env ];
  get(nam, env):
    if(is_void(env),
      error("undefined identifier: ",nam),
      if(env[1] = nam,
        env[2],
        get(nam, env[3])));
  set(nam, val, env):
    if(is_void(env),
      display("undefined identifier: ", nam),
      if(env[1] = nam,
        env[2]:= val,
        set(nam, val, env[3]))) }
```

Fig. 11. an environment implementation

ments (see concern #7) using a linear list and `add`, `get` and `set` functions that support creation, access and modification of bindings[10].

2.7 Concern #7: abstraction through first-class environments.

Pico environments are implemented as name–value association lists (see concern #6). Defining a variable or a constant pushes a new association onto the current environment; a reference to a variable or a constant requires a sequential search of the current environment. Hence, there are no explicit scope levels, but static scope with hiding of homonyms is effectively supported – however, duplicate variables or constants are allowed within the body of a function.

```
Stack(n):
  { T[n]: void;
    t: 0;
    empty():: t = 0;
    full():: t = n;
    push(x):: { T[t:= t+1]:= x; void };
    pop():: { x: T[t]; t:= t-1; x };
    capture() }
```

Fig. 12. a stack abstraction

Contrary to Scheme, Pico environments are first class: the current environment is accessible to a Pico program by way of the `capture` native function.

[10] this is actually very close to the actual implementation of Pico environments; it provides a very simple and clean view on an essential part of Pico semantics

Using the dot-notation, invocations may be qualified by an environment and the invocation identifier will be looked up in said environment and used, provided it was defined as a constant[11]. First-class environments introduce data abstraction: see figure 12. The environment `S: Stack(10)` is accessible via expressions such as `S.push(123)`; `S` effectively implements a stack.

First-class environments provide a very simple module system that constitutes a first step towards objects (see [lmo] for the description of an implementation of prototype-based inheritance on top of Pico).

2.8 Concern #8: control through first-class continuations.

Pico features first-class continuations, i.e. values with a unique type, that hold the computational state of the Pico evaluator [12]. Pico provides a native function:

```
call(expression(continuation)):: ...
```

which evaluates the argument of an application of `call` with respect to a temporary extension of the current environment with a binding to `coninuation` of the current continuation (i.e. the current Pico computational state). A second native function:

```
continue(continuation, expression):: ...
```

reactivates `continuation` substituting the value of `expression` for the value of the current computation[13].

```
{ raise(id, retval): error("UNCAUGHT EXCEPTION");
  trycatch(try(), filter(exception), catch(exception, value))::
     call({ keep:: raise;
            raise(id, retval):=
              { raise:= keep;
                if(filter(id),
                   continue(continuation, catch(id, retval)),
                   raise(id, retval)) };
            result: try();
            raise:= keep;
            result }) }
```

Fig. 13. an exception handler

[11] visibility is effectively enforced by choosing between the `:` and the `::` syntax
[12] contrary to Scheme, where closures are overloaded in order to support continuations
[13] this is the same semantics as in Scheme

Pico continuations can be used to build familiar constructs; in figure 13 we suggest a Pico implementation for a conventional exception handler.

As an example of using this exception handler, consider figure 14 where a negative discrimant of a quadratic equation is caught.

```
{ root(a, b, c)::
    { d:: b^2 - 4*a*c;
      if(d = 0,
        -b/2/a,
        if(d > 0,
          [ (-b + sqrt(d))/2/a, (-b - sqrt(d))/2/a ],
          raise("noRoots", d))) };
  safe_root(a, b, c)::
    trycatch(root(a, b, c),
             exception = "noRoots",
             display("discriminant was negative: ",value)) }
```

Fig. 14. using the exception handler

2.9 Concern #9: code as data through a unified abstract grammar.

The Pico evaluator is based on a unified abstract Pico grammar [14]. In the spirit of the selection of tables as composite value constructors (see concern #6), abstract grammar nodes are a generalization of tables[15].

Figure 15 gives an overview of the Pico abstract grammar. Italics refer to meta-values, an `environment` is either `void`, a `constant` or a `variable`, an `invocation` is as in concern #1, `arguments` are mostly tables but can be any `expression` whenever the `@` application syntax is in effect. A `thread` is actually a list of frames and is more structured than indicated in figure 15.

In Pico, the abstract grammar of expressions and values is accessible via a set of native functions. `make`, `get` and `set` allow the creation and the access to the slots of abstract grammar nodes [16]. Tags (the upper-case labels in figure 15 are referred to as natural numbers [0, 1, ...].

In addition to these, Pico allows reflective access to `read`, `eval`[17] and `print` in addition to providing a quotation syntax.

[14] unified refers to the fact that the abstract grammar incorporates both expressions and values
[15] generalization in the sense that a table is a particular kind of abstract grammar node
[16] these functions validate their arguments so as to avoid violating the Pico semantics
[17] contrary to Scheme

```
reference      ::= REF text
application    ::= APL expression arguments
tabulation     ::= TBL expression table
qualification  ::= QUA expression invocation
declaration    ::= DCL invocation expression
definition     ::= DEF invocation expression
assignment     ::= SET invocation expression
constant       ::= CST text expression environment
variable       ::= VAR text expression environment
continuation   ::= CNT environment thread
thread         ::= THR expression*
native         ::= NAT text number
function       ::= FUN text arguments expression environment
quotation      ::= QUO expression
table          ::= TAB expression*
text           ::= TXT text
fraction       ::= FRC fraction
number         ::= NBR number
void           ::= VOI
```

Fig. 15. Pico abstract grammar

2.10 Concern #10: uniformity through a single memory model.

This last concern is more of an implementation concern, but it crosscuts the design of Pico in its entirety. Pico uses a single memory model for the representation of all entities manipulated by its virtual machine[18] and uses a single memory space and a unified garbage collector . Every single Pico value or expression is stored in this unique memory space, according to the abstract grammar in figure 15. The latter can effectively be viewed as the instruction set for the Pico virtual machine.

Environments are first-class Pico values (see concern #7) and consist of linked lists of bindings, i.e. constant or variable abstract grammar nodes. In a similar vein, computation in the Pico virtual machine uses a linked list of frames, called a thread. Since threads are first-class, they can be garbage collected, combined with an environment into a continuation (see concern #8) and even used to introduce distributed programming into Pico featuring strong mobility [borg].

[18] contrary to Scheme, which typically uses several memory spaces to hold environments, pairs, vectors, strings, etc.

3 Assessment, conclusion and suggestions.

Originally, Pico was prototyped in Scheme and then ported to C. This prompted the idea of a single memory manager[19]. Next, a complete meta-circular[20] Pico version was developed, serving as human-readable specification[21] of the language.

Pico was designed with little regard for efficiency – and none at all if it compromised the semantical clarity of the language. Obviously, constructing environments as simple linked lists impacts the performance of Pico enormously. But we have chosen to maintain this concept because of its influence on a novice programmer's understanding.Moreover, the use of first-class environments as modules and allowing nested parameters in call-by-function parameters voids all possibility for introducing a lexical addressing scheme[22].

The introduction of Pico-styled call-by-function may possibly raise some eyebrows because of its – albeit limited-introduction of dynamic scoping. But in our opinion, and referring to [lmo], this may do more good than bad, certainly in a teaching or prototyping context.

But by and large the most successful contribution of Pico rests with its regular syntax and semantics. A novice Pico programmer masters the grid in figure 2 in matter of hours and the combination of the *first-class everything* and *functions-everywhere* goes a long way to ensure that. But an experienced programmer will also appreciate this basic Pico property: it is indeed possible to master all of Pico's features on short notice; and this is certainly not true for most main-stream languages – including Scheme.

Using a meta-circular implementation to document the semantics of a language is not new. And contrary to Scheme, where only a small subset op the language is involved (see for instance [sicp]) we propose a complete implementation of Pico in Pico. Thanks to the compactness of the language the result is still very accessible to Pico programmers – even novice ones. And short of a complete and formal specification[23] a meta-circular implementation is the only reasonable way to expose a member of the general programming public to the semantics of Pico in a more precise medium than natural language.

Also a word about Pico's virtual machine: in a time where *just-in-time compilation* compensates for the lack of performance of an interpreter, we would like

[19] the Pico virtual machine uses a single, contiguous memory space managed by a particularly effective compacting mark-and-sweep garbage collector

[20] a slight misnomer, because the base-level does not share the representation of program expressions with the meta-level

[21] a formally-minded individual may take offense at this, but remember that in this discussion *human* stands for *mere mortal*

[22] all of the techniques developed to optimize method lookup in dynamically typed languages can be put to use here. In fact, the implementation of a cache that shadows the global Pico environment (i.e. the native variables and function declarations) already led to a Pico implementation that is only slightly less efficient than DrScheme

[23] the R5RS report [r5rs] on Scheme contains a formal specification using denotational semantics

to argue in favor of a high-level VM instruction set. And we see no reason why this instruction set should not be based on some generic abstract grammar, as is illustrated with Pico.

Finally: few of the ideas that helped shape Pico are truly original. In fact, the real contribution of Pico is the merging into one package of a number of language concepts that are more or less well understood. We omitted an exhaustive list of bibliographical references as this would probably have doubled the size of this paper; we trust that anyone familiar with the domain of programming language engineering will identify the various concepts that we borrowed from the existing body of knowledge and will not fail to trace them back to the original contributors without our explicit help.

4 Bibliography.

[borg] Agent Mobility and Reification of Computational State, an experiment in migration
Van Belle W. and D'Hondt T.
Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems, Lecture Notes in Artificial Intelligence
Springer Verlag, 2000

[lmo] Of first-class methods and dynamic scope
D'Hondt T. and De Meuter W.
RSTI L'objet 9/2003. LMO 2003

[pico] `http://pico.vub.ac.be`

[r5rs] Revised^5 Report on the Algorithmic Language Scheme,
Kelsey R., Clinger W. and Rees J. (eds.),
Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998
and ACM SIGPLAN Notices, Vol. 33, No. 9, September, 1998

[sicp] Structure and Interpretation of Programming Languages
Abelson H. and Sussman G.
MIT Press (1996)

[soap] On the Performance of SOAP in a Non-Trivial Peer-to-Peer Experiment
Van Cutsem T., Mostinckx S., De Meuter W., Dedecker J. and D'Hondt, T.
2nd International Working Conference on Component Deployment (CD 2004)