Semantics of Combinations in Scheme [*]

S. Anglade[†] J.J. Lacrampe[†] C. Queinnec[‡]
Laboratoire d'Informatique Fondamentale d'Orléans
École Polytechnique & INRIA-Rocquencourt

Abstract

This paper presents a denotational semantics for the combinations of the Scheme language. Scheme leaves unspecified the order of evaluation of the terms of a combination. Our purpose is to formally and denotationally characterize such indeterminacy. We achieve this by extending the denotation as well as the domain of final answers to take into account the various possible orders of evaluation.

1 Introduction

The Revised^4 Report [CR91] defines the Scheme language in English. This document also contains an appendix that provides a denotational definition for Scheme. The informal description of combinations is:

[R^4RS **4.1.3**] ... *The operator and operand expressions are evaluated (in an unspecified order) and the resulting procedure is passed the resulting arguments. ...*
In contrast to other dialects of Lisp, the order of evaluation is unspecified, and the operator expression and the operand expressions are always evaluated with the same evaluation rules.
Although the order of evaluation is otherwise unspecified, the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.

But the denotational appendix uses a trick to suggest the indeterminacy of the order of evaluation:

[R^4RS **7.2**] *The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations permute and unpermute, which must be inverses, to the arguments in a call before and after they are evaluated. This is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.*

The purpose of this paper is to present a denotational specification for the indeterminacy of the order of evaluation.

In section 2, we analyze combination and procedure invocation and illustrate some ambiguities. In section 3, we expose our solution which roughly allows all possible orders of evaluation and leaves to the implementation the final choice among all possible denotations. Section 4 will discuss related problems such as other aspects of indeterminacy and infinite evaluation. We will also emphasize the notion of store. We finally end with conclusion, in section 5.

2 Actual state

This section presents and discusses the actual meaning of combinations. We had two additional constraints: we wished to preserve as much as possible the actual denotational semantics as published in [CR91], we also wanted the resulting semantics to be still executable when translated into Scheme.

[*]Revision: 2.0

[†]Laboratoire d'Informatique Fondamentale d'Orléans, BP 6759, 45067 Orléans Cedex 2, France – Email: {anglade,lacrampe}@univ-orleans.fr

[‡]Laboratoire d'Informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France – Email: Christian.Queinnec@polytechnique.fr This work has been partially funded by GDR-PRC de Programmation du CNRS.

2.1 Formal semantics

In the Revised[4] Report, the semantics of procedure calls appears as[1]:

$$
\begin{array}{l}
\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] = \\
\quad \lambda\rho\kappa.\mathcal{E}^*\ [\![(\mathit{permute}\ (\langle\ \mathrm{E}_0\ \rangle\ \S\ \mathrm{E}^*))]\!] \\
\qquad\qquad \rho \\
\qquad\qquad \lambda\epsilon^*.(\ (\lambda\epsilon^*.\mathit{applicate}\ (\epsilon^* \downarrow 1)(\epsilon^* \dagger 1)\kappa\ ) \\
\qquad\qquad\qquad (\mathit{unpermute}\ \epsilon^*)\ \ )
\end{array}
$$

The valuation function $\mathcal{E}^*$ distributes $\mathcal{E}$ onto a sequence of expressions. Several problems exist with that definition:

1. As recalled in the introduction, *permute* is a global function that may incorrectly suggest that the generated permutation is constant.

2. *unpermute* is not exactly an inverse of *permute* since it only shuffles sequences of values and has nothing to do with syntax.

3. *permute* is a syntactic constructor taking syntactical terms and returning a sequence of these terms. This violates a tenet of denotational semantics and forces one to prove that E is well defined. This proof is straightforward (and always omitted) if the denotation of any term is only made of the denotations of its subterms [Sto77].

2.2 Informal semantics

The following example will be used to illustrate the many points we will discuss and, most of all, the various evaluation orders. It is written with an essential syntax.

```
(let ((ord 0))
  (let ((f (lambda ()
             (set! ord (+ 1 ord))
             (let ((tmp ord))
               (lambda l (cons tmp l))
             ))))
    ((f)((f)((f)))((f)((f))))))
```

Let us illustrate the various possible orders of evaluation with the associated sentences of Revised[4] Report.

[R^4RS **7.2**] *...Applying arbitrary permutations permute and unpermute ... is a closer approximation to the intended semantics than a left-to-right evaluation would be.*

- For an uniform left-to-right evaluation, the result is `(1 (2 (3)) (4 (5)))`.

- For an uniform right-to-left evaluation, the result is `(5 (4 (3)) (2 (1)))`.

- And, for the evaluation where the second, first and then, when necessary, third expressions are successively considered, the result is `(3 (2 (1)) (5 (4)))`.

[R^4RS **7.2**] *...Applying arbitrary permutations ... to the arguments in a call before and after they are evaluated, is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program (for any given number of arguments).*

If we adopt a left-to-right evaluation order for the external procedure call and, a right-to-left for the internal procedure calls then, the result is `(1 (3 (2)) (5 (4)))`.

[R^4RS **4.1.3**] *Although the order of evaluation is otherwise unspecified, the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation.*

This rule excludes, for instance, a result like `(2 (1 (5)) (3 (4)))`. This result might be produced by a tri-processor concurrently evaluating the three terms of the external procedure call. The second processor is the first to start evaluation as well as the last to finish. Meanwhile, the first and third processors interleave their reading of the shared variable **ord**.

[R^4RS **4.1.3**] *The order of evaluation may be chosen differently for each procedure call.*

This formulation is ambiguous and will be the subject of the next section.

2.3 Combinations *versus* invocations

The Revised[4] Report states that *combination* and *procedure call* are synonymous (section 4.1.3). A combination is a piece of text having a precise syntax, it is a syntactical concept. On another hand, an invocation represents a run-time concept where a function, obtained by evaluation of the operator part of a combination, is applied on the values that were obtained from the operands of the combination. Therefore a combination may lead to numerous invocations: this

[1]The denotations appearing in this paper were produced automatically with LiSP2TEX. This program may be fetched on `ftp.inria.fr:INRIA/Projects/Papers/`.

is the case of (cons tmp 1) which is a combination that the above example evaluates five times.

A combination may have more than one occurrence. This is the case, for instance, of ((f) ((f))) which appears twice. Each of these occurrences is evaluated only once.

In pure denotational style [Sto77, Sco82, Sch86], there is no means to distinguish between occurrences of a combination, therefore $[\![$((f) ((f)))$]\!]$ should have an unique denotation.

The last quoting from Revised[4] Report may be considered ambiguous since it does not precise whether it concerns combinations or invocations. If it concerns combinations then the evaluation order of a combination is unique even if that combination has multiple occurrences. In particular if one instruments a combination to determine the chosen order then, once determined, this order will never change (for that combination).

If the evaluation order concerns invocations then, a single occurrence of a combination may exhibit a different evaluation order any time it is evaluated. This is even more obvious for different occurrences of a combination.

Let us name *invocation approach* the latter and *combination approach* the former. Returning to our previous example, there are 24 different possible evaluation orders for the invocation approach, whereas there are only 12 possibilities in the combination approach. For instance, the result (1 (3 (2)) (4 (5))) cannot be obtained with the combination approach[2].

If a real compiler compiles differently different occurrences of a combination then it implicitly adopts the invocation approach.

3 All orders

The purpose of this section is to propose a denotational semantics for the invocation approach which is more general than the combination approach.

3.1 Towards denotational functions

This subsection defines the permutation machinery. We define two new functions, named *perm* and *unperm*. We will use the following notations for sequence manipulation:

$(s \dagger i)$	drops as usual the first i members of s,
$(s \downarrow i)$	is also the i^{th} member of s,
$(s \,\bar{\downarrow}\, i)$	keeps all the members of s but the i^{th},
$(s \overset{i}{\leftarrow} x)$	shifts right the members of s from the i^{th} and inserts x as the i^{th},

Let D be the domain U $\rightarrow$K $\rightarrow$C and let δ name elements of D.

The first trick is not to permute expressions but to permute their denotations. The second trick is to automatically produce the unpermutation from the definition of the permutation. Recall that N is the set of natural numbers. A sequence of items in D can be encoded as a function from N to D. Therefore a permuter *perm* is a function that maps sequences onto sequences; its signature is (N $\rightarrow$D) $\rightarrow$N $\rightarrow$D.

The *unperm* function reverses the effect of *perm*. Though *perm* is imposed by the implementation, *unperm* may be derived automatically from *perm*, with the signature $D^+ \rightarrow D$.

The role of *unperm* is to build a cascade of continuations that will insert values in the order which is appropriate for the final application. The *unperm* function is defined as follows:

```
$unperm =$
$\lambda\delta^+ .\ perm$
      $\lambda i\rho\kappa .\ (\delta^+ \downarrow i)$
            $\rho$
            $single$
            $\lambda\epsilon .\ \#\delta^+ = 1 \rightarrow \kappa\langle\epsilon\rangle,$
                $unperm$
                  $(\delta^+ \,\bar{\downarrow}\, \mathrm{i})$
                  $\rho$
                  $\lambda\epsilon^* .\kappa(\epsilon^* \overset{\mathrm{i}}{\leftarrow} \epsilon)$
    $\#\delta^+$
```

If δ^+ is the sequence of denotations of expressions, $(\delta^+ \downarrow i)$ represents the one *perm* chooses, and $(\delta^+ \,\bar{\downarrow}\, \mathrm{i})$ the sequence of all others. The chosen denotation is "run" with a continuation that will take the resulting value ϵ, and will insert it at the right place in the sequence of already obtained values ϵ^* coming from the other denotations $(\epsilon^* \overset{\mathrm{i}}{\leftarrow} \epsilon)$.

Let *Fix* be a fix-point operator [Sco82] then given a *perm* function, *invert* builds the associated *unperm* function. Its signature is therefore $((\mathrm{N} \rightarrow \mathrm{D}) \rightarrow \mathrm{N} \rightarrow \mathrm{D}) \rightarrow \mathrm{D}^+ \rightarrow \mathrm{D}$.

[2]If computations were allowed to be interleaved then, there are 120 imaginable permutations, most of them are ruled out by the necessary sequentiality of evaluation order.

```
invert =
  λperm. Fix
          λunperm.
           λδ⁺. perm
                 λiρκ. (δ⁺ ↓ i)
                         ρ
                         single
                           λε. #δ⁺ = 1 → κ⟨ε⟩,
                             unperm
                               (δ⁺ † i)
                               ρ
                               λε*.κ (ε* ←ⁱ ε)
                 #δ⁺
```

For a given *perm*, The semantics of procedure call can therefore be expressed as:

$$
\begin{aligned}
&\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] = \\
&\quad \lambda\rho\kappa.\ \mathit{invert} \\
&\qquad \mathit{perm} \\
&\qquad \mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] \\
&\qquad \rho \\
&\qquad \lambda\epsilon^+.\mathit{applicate}(\epsilon^+ \downarrow 1)(\epsilon^+ \dagger 1)\kappa
\end{aligned}
$$

The semantics of sequence of expressions is now defined as:

$$
\begin{aligned}
&\mathcal{E}^*[\![]\!] = \langle\ \rangle \\
&\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] = \langle\ \mathcal{E}[\![\mathrm{E}_0]\!]\ \rangle\ \S\ \mathcal{E}^*[\![\mathrm{E}^*]\!]
\end{aligned}
$$

3.2 An Axiomatization of evaluation order

The function *perm* cannot be any function with signature $((\mathrm{N} \to \mathrm{D}) \to \mathrm{N} \to \mathrm{D})$. To ensure that *unperm* visits all (and only once) the elements of a sequence δ^+, *perm* must verify:

$$\forall f, n, \exists\ i, 1 \le i \le n, \mathit{perm}\ f\ n = f\ i$$

This formula corresponds to the contract of *perm* which is to choose one element among n, then one element among the $n-1$ remainings etc.

With a denotational semantics point of view, an evaluation context is made of an expression, an environment, a continuation and a store. The exact choice of an evaluation order in a combination (E_0 E^*) may depend on all these informations. This order can be deduced by *invert* from a function *perm*, depending on the context.

Actually, one could axiomatize evaluation order of a combination by:

$$
\begin{aligned}
&\forall \mathrm{E}_0, \mathrm{E}^*, \rho, \kappa, \sigma, \\
&\quad \exists\ \mathit{perm}, \forall f, n, \exists\ i, 1 \le i \le n, \mathit{perm}\ f\ n = f\ i
\end{aligned}
$$

Such a specification does not fit our aim to translate the denotational semantics into a native Scheme.

3.3 A Meta-function for evaluation strategies

Let us name *Meta-perm* the function that defines the evaluation order strategy in a given implementation. To be realistic, we give to *Meta-perm* the whole context E_0, E^*, ρ, κ and σ to leave the choice of an evaluation order completely unrestricted. Therefore this function has: $\mathrm{Exp} \to \mathrm{Exp}^* \to \mathrm{U} \to \mathrm{K} \to \mathrm{S} \to (\mathrm{N} \to \mathrm{D}) \to \mathrm{N} \to \mathrm{D}$ as signature. It fully characterizes an evaluator which is correct if:

$$
\begin{aligned}
&\forall \mathrm{E}_0, \mathrm{E}^*, \rho, \kappa, \sigma, f, n, \\
&\quad \exists\ i, 1 \le i \le n, \mathit{Meta\text{-}perm}\ \mathrm{E}_0\ \mathrm{E}^*\ \rho\ \kappa\ \sigma\ f\ n = f\ i
\end{aligned}
$$

Meta-perm results from a skolemization of *perm* and the semantics of procedure call, depending on *Meta-perm* can therefore be expressed as:

$$
\begin{aligned}
&\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] = \\
&\quad \lambda\rho\kappa.\lambda\sigma.\ \mathit{invert} \\
&\qquad\quad \mathit{Meta\text{-}perm}\ \mathrm{E}_0\ \mathrm{E}^*\ \rho\ \kappa\ \sigma \\
&\qquad\quad \mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] \\
&\qquad\quad \rho \\
&\qquad\quad \lambda\epsilon^+.\mathit{applicate}(\epsilon^+ \downarrow 1)(\epsilon^+ \dagger 1)\kappa \\
&\qquad\quad \sigma
\end{aligned}
$$

Let us give some examples of specific *Meta-perms*.

1. *Meta-perm* E_0 E^* $\rho\ \kappa\ \sigma = \lambda\ f\ i.(f\ 1)$ characterizes left-to-right order.

   Meta-perm E_0 E^* $\rho\ \kappa\ \sigma = \lambda\ f\ i.(f\ i)$ characterizes right-to-left order.

   These behaviors are entirely independent from the context.

2. If the behavior of *Meta-perm* depends only on the expressions then it reflects the combination approach. A given combination always uses the same order but, two different combinations can use different orders.

3. If the behavior of *Meta-perm* depends on each component of the context then, it reflects the invocation approach[3].

[3]For instance, (π_1 π_2) can be compiled into `(if (amb) (let ((x π1)) (x π2)) (let ((x π2)) (π1 x)))`. This leaves the indeterminacy until run-time, the store under the random function amb being responsible for the choices.

3.4 Multiple answers

The *Meta-perm* function of the previous section is suitable to characterize implementations. We now want to characterize the language itself independently of its implementations i.e., to characterize the set of all possible answers yielded by *Meta-perm* instances satisfying the invocation approach.

The basic idea is to generate all permutations and to collect their results. This leads us to redefine C as S $\rightarrow$ A* rather than as S $\rightarrow$ A since the denotation now returns the sequence of answers rather than a single one. Then *Meta-perm* E_0 E^* $\rho\ \kappa\ \sigma$ will constantly yield *all* where *all* is a function that concatenates the answers of its first argument *f* applied to each natural positive number, less than the length of the submitted sequence.

The *all* function ensures that we get the results of all possible permutations using *invert all.*

$$\begin{aligned}
&all = \\
&\lambda f i \rho \kappa \sigma.\ i = 1 \rightarrow f\ i\ \rho\ \kappa\ \sigma, \\
&\qquad\qquad all\ f\ (i-1)\ \rho\ \kappa\ \sigma \\
&\qquad\qquad\ \S \\
&\qquad\qquad f\ i\ \rho\ \kappa\ \sigma
\end{aligned}$$

So the final semantics for combinations is:

$$\begin{aligned}
&\mathcal{E}[\![(E_0\ E^*)]\!] = \\
&\ \lambda\rho\kappa.\ invert \\
&\qquad all \\
&\qquad \mathcal{E}^*[\![E_0\ E^*]\!] \\
&\qquad \rho \\
&\qquad \lambda\epsilon^+.applicate(\epsilon^+ \downarrow 1)(\epsilon^+ \dagger 1)\kappa
\end{aligned}$$

Of course, an implementation is compelled to return only one of the possible values as predicted by the semantics.

4 Related problems

In this section, we consider various related problems such as — the specification of the *unspecified value*, — the writing of an evaluator for the above semantics and, — the potential for infinite evaluations [Lac92].

The Revised^4 Report explicitly allows an other kind of indeterminacy:

> [*R^{4}RS* **1.3.2**] *If the value of an expression is said to be "unspecified", then the expression must evaluate to some object without signaling an error, but the value depends on the implementation; this report explicitly does not say what value should be returned.*

> [*R^{4}RS* **7.2.2**] *Here and elsewhere, any expressed value other than undefined may be used in place of unspecified.*

The "unspecified" value is not required to be the same in every context. We can adopt the same techniques developed in this paper (see sections 3.2, 3.3, 3.4) for the specification of the meaning of this "unspecified value". For instance and similarly to section 3.3, the global semantics can be parameterized by another implementation-defined function which goal is to return an unspecified value that might be chosen according to the context of execution i.e. $\rho\ \kappa\ \sigma$.

A nice evaluator would try to show all the various possible answers an expression yields. But this evaluator would also have to return the multiple corresponding stores. In a toplevel-based implementation, different choices are possible, for instance — to let the user choose which solution is the base for the next toplevel iteration or, — to evaluate exhaustively the new expression in all the previously obtained stores. The latter leads to a combinatorial explosion but the former is already very expensive.

The previous nice evaluator requires an Answer domain able to represent collections of values. The simplest solution is to use sequences of values as in A*. But there is no reason to impose an order on the answers so a set would be more appropriate. But sets require elements (i.e., functions) to be comparable to ensure that elements do not belong more than once to a given set. Bags will be more useful but sequences are a good compromise.

To use sequences of answers also raises the problem of infinite evaluations. A clever and fair [LL92] evaluator should produce first the finitely computable values.

5 Conclusions

We showed in this paper how to precisely define an undefined evaluation order. We propose it for R^{n}RS which should at least clarify the meaning of "procedure call". Note that our approach does not impose (but does not exclude either) to choose the order of evaluation at run-time, it allows multiple occurrences of a single combination to have different but statically chosen order of evaluations. As such our proposal leaves entire freedom within indeterminacy.

Acknowledgements

We wish to thank John Ramsdell for his remarks and mainly the adaptation of our solution to the "unspecified value" problem.

Bibliography

[CR91] William Clinger and Jonathan A Rees. The revised^4 report on the algorithmic language scheme. *Lisp Pointer*, 4(3), 1991.

[Lac92] Jean-Jacques Lacrampe. S3L à tire d'ailes .... Technical Report 92-11, Laboratoire d'Informatique Fondamentale de l'Université d'Orléans, BP 6759 - 45067 Orléans Cedex 2, France, 1992.

[LL92] Jean-Jacques Lacrampe and Marianne Ligou. Une machine "équitable" pour implanter s3l. In *Journées Francophones des Langages Applicatifs*, pages 82–109, 1992.

[Sch86] David A Schmidt. *Denotational Semantics, a Methodology for Language Development.* Allyn and Bacon, 1986.

[Sco82] Dana Scott. Domains for denotational semantics. In M. Nielson and E.M. Scmidt, editors, *Automata, Languages and Programming, 9th Colloquium*, pages 577–623. Aarhus, Lecture Notes in Computer Science 140, Springer Verlag, 1982.

[Sto77] Joseph E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge Massachussetts USA, 1977.