Validation Suite Generation

This work has been published in pages 134–144 of journal Bigre, volume 65 of July 1989.

Christian Queinnec*
Laboratoire d'Informatique de l'École Polytechnique
Équipe ICSLA**
91128 Palaiseau Cedex
France

Abstract

To validate big systems is a very complex task which has not been solved yet. Many researchers have addressed the more restricted problem of language validation [Goodenough 81, Whichmann & Sale 80] but still did not succeed. One reason is the lack of a semantical definition of the language to be validated, the other reason is that such a semantics is often considered as a theoretical object unable to provide more than meanings to programs.

To validate a Lisp-like system is still a large problem due to the number of its essential constructs (special forms and other unusual features like `eval` or multiple values) and to the size of its run-time support of which the garbage collector is the main and most complex part. However — since Scheme has only a few features [Rees & Clinger 86] and may be considered as an efficient λ-calculus [Steele & Sussman 75], — since λ-calculus is the support of Denotational Semantics [Stoy 77, Schmidt 86] and denotational semantics for Scheme has already been published [Muchnick & Pleban 80, Rees & Clinger 86] and eventually — since the equation "*program* = *data*" holds in Scheme, it seems therefore possible to make use of the denotational semantics of Scheme to generate validation suites.

The paper will present this technique, its implementation (via streams) and some experimental results.

Introduction

Languages are traditionally tested through a validation suite i.e., a wide set of programs each designed to exercize a few linguistic features. This set usually also contains the programs explicitely showing that former bugs are corrected: this property participates to what is called "non regression". To validate a language only means that the associated compiler (or interpreter) goes through all these programs and behaves as expected; erroneous programs must be rejected whilst correctly compiled programs are run and must fail or compute the right thing. To detect these status (failure or success) requires some cooperation between the language and the operating system: UNIX[1]is particularly well suited for that task.

Validation suites were produced for several languages
among which Pascal [Whichmann & Sale 80] and Ada[2][Goodenough 81]. They usually grow more and more as greater and greater

*This work has been partially funded by Greco de Programmation.

**Interprétation, Compilation et Sémantique des Langages Applicatifs

[1] UNIX is a trademark of AT&T.

[2] Ada is a trademark of the Ada Joint Program Office.

deviance must be avoided. Validation suites are often proprietary since their elaboration cost a lot to companies.

Testing through validation suites can only prove that on some programs (those of the suite) the implementation of the language seems to behave well. If tests are to be handmade, many problems will probably be encountered:

tests must be ordered they must only use "safe" features to test new ones.

tests must be exhaustive but whatever means "exhaustive" will be probably too boring for human writers.

expected results of tests must be correct since tests are programs, these programs must be adequate to what they are supposed to test: tests must be tested !

tests must be unbiased they must equally look for regular or erroneous behaviours.

To know how a compiler is done allows greater confidence in the validation suite. A modularly built compiler encourages modular tests, for example, stack-manageing constructs may be tested apart heap-manageing constructs and so forth. This confidence may be measured by the "coverage ratio" which counts the number of times each branch of alternatives were taken during the tests. A validation suite where all branches and all cycles of the implementation were followed at least once is probably better than one which has not this property !

Programs obey to the grammar of the language. To test a few linguistic features may be viewed as the generation of the set of programs following a sub-grammar containing only these features. One has also to write the necessary stuff to wrap them into fully complete and correct programs. Orthogonal languages allow to derive small and separate sub-grammars and thus modular tests.

Interesting grammars generate infinity of programs so we must stop the generation process where our confidence grows less than execution time. A cost function may be defined which given a test program returns its generation cost, for example, the number of syntactical nodes in its parse tree. Our method provides a breadth-first generation according to the cost function and thus avoids developping too far any infinite branch of the grammar.

Houssais analysed the errors encountered in the development of some Algol 68 compilers [Houssais 84]. He showed that

- most of the known bugs might have been found with a low generation cost,
- the bug discovery rate is quickly decreasing.

These facts joined to combinatorial explosion will therefore limit test generation.

All the previous points exhibit the empirical nature of validation suite. A better approach is probably to generate bug-free compilers from language semantics [Clinger 84]. But one cannot decide without tests if the (say denotational) semantics of a language is what s/he had in mind when designing it[3].

Compiler Generation is an area where much efforts are put on [Mosses 76, Appel 85]. The main idea is to consider the denotational semantics of a language as an object that can be analysed in order to derive some properties: separation of the static and dynamic semantics, type analysis, single-threadedness ... Some transformations may also be performed on it to obtain interpreters or compilers [Consel 88, Deutsch 89, Weis 87].

The case of Scheme may now seem easy. Given a semantics of a language, a program generator for this same language allow to generate validation suites i.e., lists of programs followed by the description of their expected behavior and result. A denotational semantics is a function which delivers denotations i.e., λ-terms which can be run with the initial settings (the initial environment ρ_{init}, the initial continuation κ_{init} ...) in order to compute the expected result of the test. Since results are generated, they are correct[4] with respect

[3] Gordon [Gordon 79] wrote in its thesis that "one has to debug the semantics equations s/he produced"

[4] provided the λ-evaluator is correct !

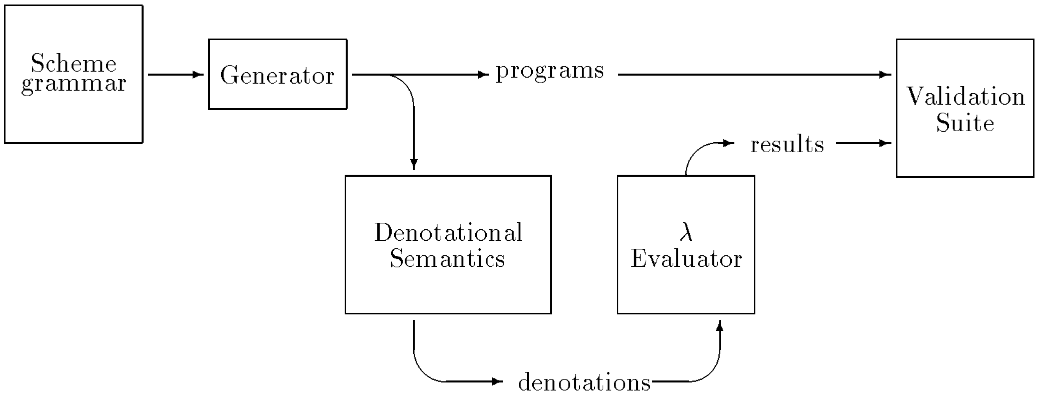

Figure 1: Validation Suite Generation (Synopsis)

to the semantics. Unbiased exhaustivity (under a cost limit) may be obtained thus sparing human efforts[5]. Tests are ranked by cost and thus provide a safe ordering according to the cost function.

Since the equation "*Program = Data*" is true in Lisp-like languages, it is therefore possible to synthesize new programs and very easy to generate a test, compute its expected result (via the semantics), run the test while trapping success or failure and compare it to the expected result. Another interest of Scheme is that its grammar is so close from abstract syntax that program generation is fairly easy. This process is summarized in figure 1.

The generator will be explained in the first section. Some of its uses, and mainly the Scheme grammar, will be found in section two. Section three will conclude with some commented experiments.

1 The Generator

The generation technique is due to a cocktail of streams and objects. The details of this process are worth to explain since — (i) the problem of generating ordered expressions (from simple to complex ones) is difficult — (ii) the proposed solution is short and neat — (iii) the solution may be adapted to other languages.

[5]But not computers efforts ! Non regression tests may be scheduled during nights or hollidays where most computer power is lost.

A *stream* is an ordered collection of data which may virtually be infinite provided you do not require all these data altogether. A stream may be *consumed* giving an element, known as its *head*, and a new stream representing its *tail* i.e., the original stream less its first initial element. In that way, streams are like `cons`es except that their tails are computed lazily, that is to say only when really needed. A thorough explanation of streams, implemented by the Scheme primitives `delay` and `force`, appears in [Abelson & Sussman 85, pp 242–272].

On the other hand, *objects* are entities with internal state. Behaviours may be associated to objects by means of *generic functions*. When such a function is applied, the classes (or types) of its arguments determine the *method* to be applied. Generic functions form the main part of CLOS [Bobrow et al. 88].

Objects are not part of Scheme but can be easily implemented with closures. For that project, we used SCOOPS [TI-Scheme].

Principles of Generation

When designing a piece of code, the author obeys to some principles or, at least, to some visions of the dreamt code. The generator was seen as a broad and intricated net of nodes producing and ordering expressions and eliminating duplicate computations. To cope with this requirements, we introduced an original technique with — (i) an unique view of streams associated with costs — (ii) a small set of stream composers and — (iii) two operations to make the net generate something.

A *normal stream* is made of a head (a generated expression) along with its generation cost and followed by its tail. Something we will represent as: $_1$X $_1$Y $_1$Z $_2$(X) $_2$(Y) ...

Where `X`, `Y` and `Z` have a cost of one and where the following terms `(X)` and `(Y)` have a cost of two. The dots represents the tail of the stream. As generation cost function, we will always take the total number of conses and symbols of the expression.

Two operations make streams evolve. `CStream-transform`[6] takes a stream and returns a new and equivalent stream. This new stream is a bit simpler since it is closer to be a normal stream. The second operation is `CStream-tail` which, given a normal stream, transforms its tail until it becomes a normal stream and then returns it. As a side-effect, the tail of the initial stream is replaced by the normalized equivalent tail. This side-effect avoids duplicate computations like Scheme's *promise* which are only `force` - d once, see [Rees & Clinger 86]. This side-effect does not compromise the functional behaviour of these two operations since only equivalent streams (by `CStream-transform`) are exchanged. We only use objects to implement functional streams, to share methods and to increase modularity.

Two kinds of streams exist, of which *real streams* can be empty (represented by `<>`) or normal (represented by an expression pre-indexed by its cost and followed by its tail). The other streams are compositions of streams.

`Joint` streams orderly merge two streams.

<JOINT $_1$X $_2$(X) $_2$(Y) <>
    + $_1$Y $_2$(Y) $_3$((Y)) ... >
;*is, for example, equivalent to*
$_1$X $_1$Y $_2$(Y) $_2$(X) <JOINT $_2$(Y) <>
    + $_3$((Y)) ... >

The main use of Joint streams is to order streams. While other streams are more directed toward generation, Joint streams do not produce anything but regulate others production. The regulation is fair and does not favour any of its input streams.

`Transduce` streams convert a stream into a new one applying a given function on each of its element. A cost converting function is also supplied. The following example uses `(lambda (elt) (cons elt 'A))` and `1+`

<TRANSDUCE $_1$X $_1$Y $_1$Z ... >
;*is equivalent to*
$_2$(X . A) $_2$(Y . A) <TRANSDUCE $_1$Z ... >

`Product` streams take two streams and output their "products". Each possible pair of elements of the two input streams are submitted to a binary composition function. A cost function must also be supplied, it is assumed to be monotonic with respect to the order. Suppose we want to multiply two streams with `cons` and `(lambda (cost1 cost2) (+ 1 cost1 cost2))`: <PRODUCT $_1$X $_1$Y $_3$(X Y) ...
    * $_1$A $_2$(A) $_3$((A)) ... >
;*is equivalent to*
$_3$(X . A) $_3$(Y . A) $_4$(X A) $_4$(Y A) ...

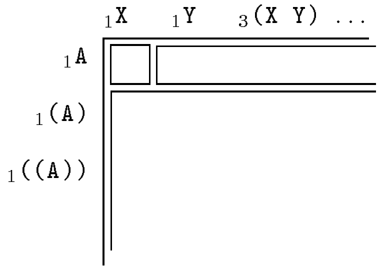

Figure 2: Product Decomposition

A product stream is recursively implemented. Just consider the result of a single `CStream-transform` applied on the previous example:

$_3$(X . A) <JOINT <PRODUCT $_1$X $_1$Y $_3$(X Y) ...
    * $_2$(A) $_3$((A)) ... >
  + <PRODUCT $_1$Y $_3$(X Y) ...
    * $_1$A <> > >

One can easily recognize in the previous expansion that the product of every term of the first stream by every term of the second one follows the diagram of figure 2.

`Kleene` streams take an input stream and deliver the stream of every sequence made of the input stream. The mere effect of a Kleene stream is, for instance, to have <KLEENE $_1$X <>>

[6] Since the word "`stream`" is often used within Scheme implementations and cannot usually be overloaded, we used the word "`CStream`" for *Cost-Stream* which appears to be free. Nevertheless we will continue to name "streams" the entities which belong to the `CStream` class.

;*generates*

```
₂(X) ₃(X X) ₄(X X X) ...
```

A Kleene stream is recursively (and circularly) implemented on top of the previous stream-composers. Let us show how is transformed a Kleene stream after just a single `CStream-transform`: `<KLEENE 1X 1Y ...>`

;*is equivalent to*

```
#1=2(X) <JOINT <TRANSDUCE 1Y ...>
              + <PRODUCT 1X 1Y ...
                        * #1# > >
```

The trick is to pipe the output of a Kleene stream expansion on itself. The whole output stream (as labelled by `#1=`) is given as second argument of the inner product (noted by `#1#`). One may recognize in that expansion the classical identity

$$list = (symbol) + (symbol\ \ .\ \ list)$$

where *symbol* is the kleene stream of symbols.

The previous expansion requires the Kleene operator to be only applied on an input stream which has a normal head (or is able to be transformed in such a stream).

The three following streams are purely technical but are given here for sake of completeness since they will be used in the sequel. `Concatenated` streams take two streams and output the first one followed by the second one. The resulting stream may not respect order and is primarily used to boot up Kleene streams which require their argument to be a normal stream i.e., has a first element.

`Relay` streams are indirections on other streams. This kind of stream is only used for technical reasons to keep a handle on successive transformations of a stream. Let us give an example

```
<RELAY <RELAY <RELAY 1X 1Y ... >>>
    ;is equivalent to
<RELAY 1X 1Y ... >
```

`Delay` streams enclose a closure which will be invoked only when the stream will be asked to transform itself. This technical stream is used when a highly recursive or cyclic stream is under construction. It can also be used to implement streams produced by *generators* i.e., closures producing different values according to their internal state. The only use we will make hereafter of Delay streams is to delay the evaluation of a variable by closing it. The printed representation will show the variable name.

As shown by the previous examples, streams may be quite quickly rather intricated since the transformation of Product or Kleene streams, though simpler in a sense, generate a lot of interconnected streams. A special function for printing these structures showing explicitely the types of the nodes and the cycles has been devised. All results shown in this paper are pretty-printed outputs of `CStream-display`.

This kind of data structure is at its best in Scheme since memory allocation is automatic. The main problem while porting this code to another more classical language will lie with the dynamic memory management of streams: a rather arduous problem if one needs efficiency.

Grammars as Streams

If streams may seem to be adequate for expression generation, their use is not obvious. We thus need an easy way to devise interesting streams. The syntactical form `make-Recursive-CStream` takes care of that.

`(make-Recursive-CStream` *name*
stream-expression `)`

The first parameter: *name*, names the resulting stream and encourages its recursive use throughout the second parameter. The latter parameter *stream-expression*, describes the stream to be produced; it may have the following form:

`(+` *stream-expression*$_i$... `)`
`makes a Joint stream`

`(++` *stream-expression*$_i$... `)`
`makes a Concatenated stream`

`(*` *stream-expression*$_i$... `)`
`makes a Product stream`

`(Kleene` *stream-expression*`)`
`makes a Kleene stream`

If ones wants to produce all the expressions composed of symbols `X` and `Y`, something that would be produced by the equivalent grammar:

```
<self> ::= <var> | (<self>*)
<var>  ::=   X   |    Y
```

Then one has only to write the stream expression:

```
(let ((var (make-enumerated-cstream
              1 '(x y) )))
```

```
  ;;var is now bound to 1X 1Y <>
  (make-Recursive-CStream self
          (++ var (kleene self)) ) )
<RELAY
 <CONCAT <RELAY <DELAY var>>
     ++ <KLEENE <RELAY <DELAY self>>> > >
```

The stream expression is close from its grammatical counterpart and thus easy to write, used we are to Backus-Naur Form. The Relay and Delay streams appearing in the expanded form are only there to take care of mutually recursive references to streams: Relay allows to have a handle on those streams which might be transformed before being used while Delay differ the evaluation of a closed variable thus providing, as in a `letrec` form, a reference to a location before using its content.

Synthesis of new streams is only possible via combination of existing streams. Another *stream-expression* allows to describe how streams are produced on a per element basis.

(letrec (($name_1$ *stream-expression*$_1$)
 ...
 ($name_n$ *stream-expression*$_n$))
 term-expression)

The `letrec` stream specification builds all the involved (and possibly mutually recursive) *stream-expression*$_i$; the transducing function `(lambda` $(\ldots name_i \ldots)$ *term-expression*`)` is then applied on every resulting tuple: its results form the output stream. Let us give an example taken from $\lambda-$calculus

```
(make-Recursive-CStream self
     ;;var is a stream of variables
     (++ var
         (+ (letrec ((variable var)
                     (body self) )
               `(lambda ,variable ,body) )
            (letrec ((fn self)
                     (arg self) )
               `(,fn ,arg) ) ) ) )
<RELAY
 <CONCAT <RELAY <DELAY var>>
     ++ <JOINT
           <PRODUCT
              <RELAY <DELAY variable>>
             * <RELAY <DELAY body>> >
         + <PRODUCT
              <RELAY <DELAY fn>>
             * <RELAY <DELAY arg>>>>>>
```

This stream outputs λ-terms which may be variables, abstractions or combinations.

`letrec` also allows locally recursive streams such as

```
(make-Recursive-CStream self
       (letrec ((exp (++ symbols
                         (* exp exp) )))
          exp ) )
<RELAY
 <RELAY
   <CONCAT <RELAY <DELAY symbols>>
        ++ <PRODUCT
              <RELAY <DELAY exp>>
            * <RELAY <DELAY exp>>>>>>>
```

This stream outputs S-expressions based on the symbols appearing in the base stream `symbols`. Note also that the previous form may have been directly written, taking advantage of the implicit `letrec` offered with the first parameter of `make-Recursive-CStream`, as

```
(make-Recursive-CStream self
       (++ symbols (* self self)) )
<RELAY
 <CONCAT <RELAY <DELAY symbols>>
      ++ <PRODUCT <RELAY <DELAY self>>
                * <RELAY <DELAY self>> > >>
```

Note that the two previous expansions, if not strictly equal, are equivalent.

2 Uses of Streams

The previous generator can be exercized on different topics. Since its main ability is to generate expressions patterned after a grammar, one can use it for example to test simple functions. An attempt for `member` may be tested against each term of the stream

```
(define member-stream
 (let ((symb (make-Enumerated-CStream
               1 '(A B C) )))
  (make-Recursive-CStream ignore
    (letrec ((exp1 (++ symb
                       (* exp1 exp1) ))
             (exp2 (++ symb
                       (* exp2 exp2) )) )
      `(,exp1 ,exp2) ) ) ) )
```

This simple grammar covers well the main potential erroneous situations that may be given to `member` – an atomic or dotted list as second argument, – a first argument which may be a symbol or a dotted pair, – a first argument appearing or not in the toplevel terms of the second argument.

All these situations are well-known but who really cares to write, without omission, all these checks? Here the power of the generator can be exercized, without much effort, to cover all these cases and some others that may have been neglected.

The Scheme Stream

It is now possible to express the whole Scheme grammar. By virtue of `make-Recursive--CStream` it can be straightforwardly written as shown in figure 3.

One can recognize the productions concerning the special forms of Scheme as well as reference and normal application. With this "`Scheme-CStream`", one may obtain, one at a time and in the natural order, correct Scheme forms.

Iteration on Streams

Infinite streams cannot be totally consumed. For all afore mentionned uses, one will probably only consume the first generated expressions. The function `iterate` allows to apply a function on the first elements of a stream. For instance (iterate

```
  (lambda (head cost)
     (display `(member . ,head))
     (newline)
     (display (apply member head))
     (newline) )
  member-stream
  83 )
```

will test `member` against the first 83 arguments generated by `member-stream`.

Following the purest data-driven style of which Lisp always makes great use, the whole net is transformed again and again by two operators implemented by generic functions. `CStream-transform` inspects the head node (and usually its input streams) and rewrites it more simply. If the result is not a normal stream, the net is one more time transformed. When the head is a normal stream, its tail is then transformed by `CStream-tail` and the final equivalent version is recorded as the new tail of the original stream. The stream appears as a linear sequence of expressions followed by a mess of intricated nodes circularly linked when Product or Kleene streams are used. To give an *aperçu* of the complexity of the net, see figure 4 where we developped only twice `member-stream`.

3 Experimentation

We exercize our generator on sub-grammars of Scheme, and mainly on nested `lambda` and `call/cc`. Since indefinite extent continuations are difficult to implement efficiently, it is likely to be some errors with them. Our experimentation was worthwile because it leads to some results.

• As expected, combinatorial explosion arrives. Depending on the number of symbols initially present in the "`var`" stream appearing in the `Scheme-CStream`, there is a rapidly increasing number of generated programs. Not all of them are interesting since due to $\alpha-$ or $\eta-$ conversions, many are, in fact, equivalent. This is not really true of all symbols since `(lambda (x) (call/cc x))`

and

`(lambda (lambda) (call/cc lambda))`

are not equivalent: variables exclude keywords.

• The cost function we adopt: counting the number of conses and symbols of an expression, is adequate. Similarly, the composition function taken in Product or Kleene nodes i.e., `cons` and `list` suit well the domain of S-expressions generation. Therefore in the Scheme realm, the default cost and composition functions need not be changed and may as well be attached to classes rather than to instances.

• To link the generator with a denotational semantics is straightforward. Two solutions exist

- generate a program, evaluate it, denote it, evaluate its denotation, compare the two results and continue
- generate a program, denote it, evaluate it and store the program with its expected result in a file which may subsequently be read by a special toplevel loop which read a program, evaluate it and compare its result with the following expression in the file.

The second solution allow to perform the generation only once; the resulting file: the *validation suite*, may also be edited before submission to an evaluator. When designing a new evaluator, it is interesting to have this file

```
(let ((var (make-enumerated-CStream 1 some variables...)))
  (make-Recursive-CStream self
    (++ var                              ;reference
        (+ (kleene self)                 ;application
           (let ((body (kleene self)))
             `(begin . ,body) )          ;sequence
           (let ((c self)
                 (th self)
                 (el self) )
             `(if ,c ,th ,el) )          ;alternative
           (let ((v var)
                 (form self) )
             `(set! ,v ,form) )          ;assignment
           (let ((variables (kleene var))
                 (body self) )
             `(lambda ,variables ,body) ) ;abstraction
           (let ((form self))
             `(quote ,form) )            ;quotation
           ) ) ) )
```

Figure 3: The Scheme Stream

since the new evaluator may be unable to generate its own tests.

• Among synthesized programs were found some interesting ones. For instance, we discover the following Scheme pearl `(call/cc call/cc)`

Although not complex, do you guess what it does? It is likely that, without systematic synthesis, it would have been unrevealed for a longer time.

• Another important problem lies with nonterminating programs. The denotation is reachable in a finite time but the result cannot be finitely obtained. To generate such a program breaks the suite ! Moreover one is quickly found: `((call/cc call/cc) (call/cc call/cc) )`

If you like this puzzle, see [Queinnec & Séniak 89] for details.

If such objects were in the file then we do not know other methods than to manually remove them from the validation suite.

4 Conclusions

We show and demonstrate a simple technique for testing Scheme-like language evaluators (interpreters or compilers). Given a syntactical grammar and the denotational equations defining the meaning of the programs conforming to this very grammar, validations suites can be easily generated. These validation suites are sets of programs associated to their expected behavior i.e., — compilation error, — execution error or — correct with given result. The set is ordered according to the size of the generated programs and provides exhaustive testing of the features present in the grammar. The number of generated programs can be arbitrarily long.

We describe an implementation for a syntax directed generator made of only a few components. Given a grammar, a recursive stream is patterned after it. Streams are built on finitely enumerated streams which are then composed by four main composers: Joint, Transduce, Product and Kleene. Two operators transform the original stream and makes it develop itself in a lazy way. The associated code is written according Object Oriented Paradigms and is pretty short.

Natural extensions can be thought of, each times a process can be verified. The generation can then be exercised only to generate tests which results can be checked by this specialized function.

```
    (define member-stream
     (let ((symb (make-Enumerated-CStream 1 '(A B C))))
      (make-Recursive-CStream ignore
        (letrec ((exp1 (++ symb (* exp1 exp1)))
                 (exp2 (++ symb (* exp2 exp2))) )
          `(,exp1 ,exp2) ) ) ) )
<RELAY <PRODUCT <RELAY <DELAY exp1>>
              * <RELAY <DELAY exp2>>> >

    (begin (CStream-transform member-stream) member-stream)
<RELAY {3}(A A)
       <JOINT <PRODUCT {1}A #2=<CONCAT #1={1}B {1}C <>
                                    ++ <PRODUCT <RELAY <DELAY exp1>>
                                              * <RELAY <DELAY exp1>>>>
                     * <CONCAT #1#
                            ++ <PRODUCT <RELAY <DELAY exp2>>
                                      * <RELAY <DELAY exp2>>>>>
            + <PRODUCT #2# * {1}A <>>>>

    (begin (CStream-tail (CStream-tail member-stream)) member-stream)
<RELAY {3}(A A)
       {3}(A B)
       <JOINT <JOINT <PRODUCT {1}A #4=<CONCAT {1}B #3={1}C <>
                                           ++ #2=<PRODUCT <RELAY <DELAY exp1>>
                                                        * <RELAY <DELAY exp1>>>>
                            * <CONCAT #3#
                                   ++ <PRODUCT <RELAY <DELAY exp2>>
                                             * <RELAY <DELAY exp2>>>>>
                   + <PRODUCT #4# * {1}B <>>>
            + {3}(B A) <JOINT <PRODUCT {1}B #1=<CONCAT #3#
                                                ++ #2#>
                                    * <>>
                            + <PRODUCT #1# * {1}A <>>>>>
```

Figure 4: The `member-stream` developped only twice

References

[Abelson & Sussman 85] Harold Abelson, Gerald Sussman, with Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge MA, 1985.

[Appel 85] Appel A., *Compile-time Evaluation and Code generation for Semantics-directed Compilers*, PhD thesis, Carnegie-Mellon University, July 1985.

[Bobrow et al. 88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales and David A. Moon, *Common Lisp Object System Specification*, SIGPLAN Notices, September 1988.

[Clinger 84] William Clinger, *The Scheme 311 compiler: An exercise in denotational semantics*, 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp 356 – 364.

[Consel 88] Consel Charles, *Realistic Compiler Generation using Partial Evaluation*, Rapport d'activités ICS-LA 88, rapport LIX 2-89.

[Deutsch 89] Deutsch Alain, *Génération automatique d'interpréteurs et compilation à partir de définitions dénotationnelles*, Rapport LITP 89-17 RXF, janvier 1989.

[Goodenough 81] Goodenough B., *The Ada Compiler Validation Capability*, IEEE Computer, June 1981.

[Gordon 79] Michael J. C. Gordon, *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, 1979.

[Houssais 84] Bernard Houssais, *Analyse d'erreurs dans des compilateurs Algol 68*, Technique et Science Informatiques, Vol 3, N 4, 1984, pp 289–295.

[Muchnick & Pleban 80] Steven S. Muchnick, Uwe F. Pleban, *A Semantic comparison of Lisp and Scheme*, Lisp Conference 1980, pp 56–64.

[Mosses 76] Mosses P. D., *Compiler generation using denotational semantics*, in *Mathematical Foundations of Computer Science*, Mazurkievicz (ed.), pp 436–441, Lecture Notes on Computer Science, Springer-Verlag 1976.

[Queinnec & Séniak 89] Queinnec Christian, Séniak Nitsan, *Puzzling with current puzzles*, Lisp Pointers, vol. 2, 89.

[Rees & Clinger 86] Jonathan A. Rees, William Clinger, *Revised^3 Report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, 21, 12, Dec 86, pp 37 – 79.

[Schmidt 86] David A. Schmidt, *Denotational Semantics, A Methodology for Language Development*, Allyn and Bacon, Inc., Newton, Mass., 1986.

[Steele & Sussman 75] Guy L. Steele Jr., Gerald J. Sussman, *Scheme: An interpreter for the extended lambda calculus*, MIT AI Memo 349.

[Stoy 77] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Mass., 1977.

[TI-Scheme] TI-Scheme reference Manual.

[Weis 87] Weis Pierre, *Le système Sam, métacompilation très efficace à l'aide d'opérateurs sémantiques*, Thèse d'Université, Paris VII, Novembre 1987.

[Whichmann & Sale 80] Whichmann B. A., Sale A. H. J., *A Pascal Processor Validation Suite*, Report CSU 7/80, NPL, Teddington, 1980.