Struggle
The First Denotational Game

EuroPAL '90, Cambridge UK, pp 351–361.

Christian Queinnec*
Internet: queinnec@poly.polytechnique.fr
École Polytechnique — INRIA (ICSLA Team)

Keywords: denotational semantics, computer game, Lisp, Lisp interpreter, threads.

Abstract

Games are often complex enough that new winning strategies cannot be devised without a clear and precise knowledge of the rules of the game. This situation makes games candidate to clear specifications and denotational semantics an interesting framework to specify games.

This paper introduces a new game where two programs fight against each other by alteration of their code: program errors lead to death. Programs are written in a Scheme-like language with `lambda`, `setq`, `prog2`, `if` and `quote` special forms. Another unusual special form `fork` is offered which allows to run independent threads. Programs are interpreted in a very special way with continual references to the store which acts as the fighting arena. Data are restricted to functions (closures), symbols and dotted pairs. Primitives are only `car`, `cdr`, `consp`, `eq`, `rplaca` and `rplacd`. The two lattest primitives are the weapons that can alter the code structure of the opponent, conversely they can also be used to repair its own structure.

Although very simple, these capabilities must be precisely defined since they are the rules of the game. To know, for instance, the number of steps taken by the execution of some construct as well as the semantics of threads is answered by the denotational semantics of the game.

The paper is probably the first use of denotational semantics in the realm of games and as such may be viewed as an exercise in denotation. It also offers some insights on the meaning of the *data = program* equation inside interpreters. Eventually it follows the Lisp tradition and offer a world-wide arena for cleverness.

The *Scientific American* Journal once introduced a game [Dewdney84] named Core War which simulates the fight of two programs within a common memory. The way to destroy its opponent was to alter the memory where lie the code and data of the two fighters in order to place erroneous instructions in the opponent code. Instructions were written in a little assembly language (Red Code) with only eight instructions for reading and writing memory cells, performing some simple arithmetics, inconditional or conditional jumps as well as forking into two threads. All addresses are PC-relative and designate words in the memory considered cyclic. Instructions and data share an uniform size: the word. This game has some afficionados and holds an annual championship. News from Core War may regularly be found in *Scientific American*.

Although simply summarized this game suffers from some defaults. The first being its non-determinism. Initially, the two fighters are placed into the memory with some randomness, then the game starts. The relative position of the two fighters may be crucial for some of them and clearly not any configuration can be exercized. The second default is related to the definition of victory ? Apart some simple cases like, for example, the situation where there exist only the two initial threads (corresponding to the two original fighters) and one is aborted thanks to an illegal instruction exception, to know the winner is difficult. For

*Current Address: LIX, Palaiseau 91128 Cedex, France. This work has been partially supported by Greco de Programmation.

example, one can imagine rather than to kill opponent's threads, to patch them to execute one's code. These opponent's threads thus become friend threads. This question is not unimportant since if cpu-time must be shared equally between the two fighters, then to ignore the owner of a thread would result in an unfair policy. All these remarks show that the simple description of the RedCode assembly language is not sufficient to fully determine the game.

We therefore propose to describe a new version of this game which solves these ambiguities. The first section will present the language and its denotational semantics. The overall approach as well as some variants will be commented upon in section two. The ludic side of this game named STRUGGLE will appear in section three with some interesting fighters. Eventually the reader will find some hints for a speedy implementation in section four.

1 Rules

STRUGGLE is a game for two players[1]. Each player writes a program in the STRUGGLE language. All the programs are then inserted into a common memory and a thread is created on each of them. The fight proceeds by performing a single step of each thread on an equal per-fighter basis i.e.if a fighter has one thread θ_1 while its opponent has two θ_2 and θ_3 then threads will be served in the order $\theta_1, \theta_2, \theta_1, \theta_3, \theta_1, \theta_2, \theta_1, \theta_3 \ldots$ provided that no threads is killed or created. Since the first fighter may take advantage of being the first one, each fight will be done twice with the two possible orders.

Each program is initially given, under variables `me` and `it`, its own code and its opponent code. The two initial threads are based on: `` `((lambda (me it) ,first) ',first ',second) ``
`` `((lambda (me it) ,second) ',second ',first) ``

where the dotted pairs representing the code of the two fighters are simultaneously given, as they are and without any copy, to the two fighters.

The STRUGGLE language is derived from Scheme [Rees & Clinger 86]. It offers the same set of essential special forms, here named: `lambda`, `setq`, `if`, `quote` and `prog2` (for convenience). It also offers `fork` which allows to create new threads. The syntax of all these special forms is strict in order both to simplify the interpreter and to augment lethality: a syntactically erroneous form kills the thread. That is why `lambda` has a single form body, `if` is ternary while `prog2` and `fork` are binary. As in Scheme, `lambda` creates closures. All bindings can be mutated by `setq`.

The predefined functions deal with the usual S-expressions i.e. `consp`, `eq`, `car` and `cdr` for dotted pairs. The STRUGGLE language is not purely functional, dotted pairs are mutable data thanks to `rplaca` and `rplacd`. These allow both to alter the opponent's code or to repair its own code. For predicates, boolean values are identified to the predefined symbols `t` and `nil` which have themselves as predefined values.

The rules of the STRUGGLE game are denotational rules [Stoy 77, Schmidt 86] but before explaining them let us comment the necessitated domains (see table 1).

Since the store is the fighting arena it has to be passed from thread to thread (a kind of single-threadedness [Schmidt 86, p. 234]). The domains of STRUGGLE which deal with the linguistic side **Env**, **Cont**, **Fun**, **Cons**, **Val** are inherited from Scheme [Rees & Clinger 86, Annex] except that they lack the store component. Note also that dotted pairs has a **Player** component which identifies to which player a pair belongs: a pair belongs to the lattest player which modifies it. This unusual field will be exploited by threads which interpret dotted pairs, these threads will belong to the player owning the interpreted dotted pairs. The other component of the **Thread** domain is a **Seq** element which takes the current store, the metacontinuation[2] which represents the next scheduler state and the current player (in order to know the holder of newly created or altered dotted pairs). A metacontinuation will be called with the new store and the list of newly created threads. The metacontinuation will sort these new threads in its internal lists of waiting threads, will perform a round-robin selection and will resume the oldest thread of the opponent with the current store as more easily shown[3] in: `;;;` *schedule*:
Store $\times$ **Thread*** $\times$ **Thread*** $\times$ **Player** $\times$ **Natural** $\rightarrow$ **Player**$\cup\{0\}$
`(define (schedule s th1* th2* player max)`

[1] Extensions for more than two players can be straightforwardly devised.

[2] No relation is intended with the *metacontinuations* of [Wand & Friedman 86], it is just here another possible meta extension of the continuation concept.

[3] Denotational semantics usually makes use of small greek letters. We take an alternate more lispish syntax.

`w, nw` $\in$	**Seq**	=	**Store** $\times$ **MetaCont** $\times$ **Player** $\rightarrow$ **Player**
`m, nm` $\in$	**MetaCont**	=	**Store** $\times$ **Thread*** $\rightarrow$ **Player**
`p, np, nnp` $\in$	**Player**	=	$\{1, 2\}$
`th` $\in$	**Thread**	=	**Player** $\times$ **Seq**
`s, ns, nns` $\in$	**Store**	=	**Loc** $\rightarrow$ **Val**
`a` $\in$	**Loc**		
`e, ne` $\in$	**Val**	=	**Id** + **Cons** + **Fun**
	Cons	=	**Loc** $\times$ **Loc** $\times$ **Player**
`f` $\in$	**Fun**	=	**Val*** $\times$ **Cont** $\rightarrow$ **Seq**
`k, nk` $\in$	**Cont**	=	**Val** $\rightarrow$ **Seq**
`r, nr` $\in$	**Env**	=	**Id** $\rightarrow$ **Loc** $\cup$ $\{$*no-location*$\}$

Table 1: STRUGGLE domains

```
(if (< max 0)
    0                  ;;no clear winner !
    (if (null? th1*)
        (- 3 player)   ;;the other player wins !
        ((seq-of-thread (car th1*))
         s
         (lambda (ns th*)
           (if (null? th*)
               (schedule ns th2* (cdr th1*) (- 3 player) (- max 1))
               ;;All threads of th* belong to the same player
               (if (= player (player-of-thread (car th*)))
                   (schedule ns th2* (append (cdr th1*) th*)
                                (- 3 player) (- max 1) )
                   (schedule ns (append th2* th*) (cdr th1*)
                                (- 3 player) (- max 1) ) ) ) )
         player ) ) ) )
```

We will note *field*-`of`-*entity* the function that extracts the *field* component of an *entity*. Such entities are recognized thanks to the `is-a-`*entity* predicate and constructed with `make-`*entity*.

The `schedule` function monitors the fight, sharing time equally between the two opponents. The fight is limited to `max` rounds, a predefined constant that must be agreed upon by the players.

The semantics of STRUGGLE is now defined by the `meaning` function: ;;;*meaning*: **Val** $\times$ **Env** $\times$ **Cont** $\rightarrow$ **Seq**

```
(define (meaning e r k)
  (lambda (s m p)
    (cond ((is-a-function e)
           (notify-struggle-error "cannot eval function" e m s) )
          ((is-an-id e)
           (if (eq (r e) 'no-location)
               (notify-struggle-error "unknown variable" e m s)
               (m s (list (make-thread p (k (s (r e)))))) ) )
          ((is-a-cons e)
           (case (s (car-of-cons e))
             ((quote) (quote-meaning e r k s m p))
             ((prog2) (prog2-meaning e r k s m p))
             ((fork) (fork-meaning e r k s m p))
             ((if) (if-meaning e r k s m p))
             ((setq) (setq-meaning e r k s m p))
             ((lambda) (lambda-meaning e r k s m p))
             (else (application-meaning e r k s m p)) ) ) ) ) )
```

The `meaning` function is a classical interpreter which takes an element of the **Val** domain and interprets it according to its type. STRUGGLE is rather pure and thus does not offer an *autoquote* facility for functions. This decision slightly augments the lethality of the game but also corrects a situation which is meaningless and probably corresponds to a latent error: a function does not have to be reevaluated. Also on the side of purity, reference to an unknown identifier (or assignment to) is an error: STRUGGLE forms must be closed terms. When such a situation occurs, the function `notify-struggle-error` kills the current thread (as well as it can report the situation for debug purpose): ;;;*notify-struggle-error*: **String** × **Val** × **MetaCont** × **Store** → **Player**

```
(define (notify-struggle-error msg e m s)
  (m s (list)) )
```

To kill the current thread is just to call the metacontinuation with no threads at all since the normal behaviour is to call the metacontinuation with a new thread which represents the current thread after a progression of one step.

If the value to interpret is a dotted pair then its `car` is analysed and the appropriate interpretation function is called with all the available arguments i.e. `e`, `r`, `k`, `s`, `m` and `p`. Note that among these variables the last three are taken from the scheduler so that if the expression `e` is a dotted pair, it has to be interpreted in the current store. The *denotation* of a fighter is thus dependent of the store as continuously modified by all threads.

All the specialized functions `quote-meaning`, `prog2-meaning` etc. share a similar structure. They first check the syntax of the form i.e.a `quote` form must have exactly one parameter. Should the syntax be incorrect, the thread is killed. The syntax of the special forms was chosen to be very constrained and does not allow any variations. Alternative is ternary, abstraction body is made of a single form, `progn` was bannished and replaced by `prog2` which only admits two parameters and, as usually, `quote` is unary. After the syntactic check, the usual interpretation of the special form is undertaken but instead of computing a value to give to the current continuation, a thread is computed and given to the metacontinuation which when resumed on a new store will give the value to the original continuation. This complexity is due to the intricate relationship between domains that allow to mix threads while preserving a single store. Finally all threads created during the interpretation of dotted pair are owned by the initial owner of the original dotted pair.

Quotation is what anyone can expect, it returns its parameter `data`, as it is, to the continuation `k` by creating a new thread `(make-thread np (k data))` with the current owner `np` of the form `e`, this thread being given to the metacontinuation `m` along with the unmodified store `s`. ;;;*quote-meaning*: **Val** × **Env** × **Cont** × **Store** × **MetaCont** × **Player** → **Player**

```
(define (quote-meaning e r k s m p)
  (if (is-a-cons (s (cdr-of-cons e)))
      (let ((np (owner-of-cons e))
            (data (s (car-of-cons (s (cdr-of-cons e))))) )
        (m s (list (make-thread np (k data)))) )
      (notify-struggle-error "incorrect quotation" e m s) ) )
```

In the STRUGGLE model, quotation returns an element of **Val**. This element is, as any other object in **Val**, a mutable object. This is a quality in STRUGGLE but generally not in languages where quotations belong to the text of programs and therefore must be immutable[4].

The rule of `prog2` is straightforward:

```
(define (prog2-meaning e r k s m p)
  (if (and (is-a-cons (s (cdr-of-cons e)))
           (is-a-cons (s (cdr-of-cons (s (cdr-of-cons e))))) )
      (let ((np (owner-of-cons e))
            (form1 (s (car-of-cons (s (cdr-of-cons e)))))
            (form2 (s (car-of-cons (s (cdr-of-cons (s (cdr-of-cons e))))))) )
        (m s (list (make-thread np
                     (meaning form1 r
                              (lambda (ne)
                                (meaning form2 r k) ) ) ))) )
      (notify-struggle-error "incorrect sequence" e m s) ) )
```

[4] Classic languages like Pascal, only accept numbers (or constants derived from numbers like characters or enumerated types instances) as quotations and thus avoid the problem of mutable data inserted within text of programs.

One may also see, as in a real interpreter, the weight of syntactic verification and extraction of components which represent most of the part of the rule. Once again is the extraction of the owner of the dotted pair `e` necessary to determine the owner of the fresh thread.

The rule concerning `setq` shows that it is not possible to alter an inexistent binding.

```
(define (setq-meaning e r k s
    (if (and (is-a-cons (s (cdr-of-cons e)))
             (is-an-id (s (car-of-cons (s (cdr-of-cons e)))))
             (is-a-cons (s (cdr-of-cons (s (cdr-of-cons e))))) )
        (let ((np (owner-of-cons e))
              (id (s (car-of-cons (s (cdr-of-cons e)))))
              (form (s (car-of-cons (s (cdr-of-cons (s (cdr-of-cons e))))))) )
          (if (eq (r id) 'no-location)
            (notify-struggle-error "unknown variable" e m s)
            (m s (list (make-thread np
                         (meaning form r (lambda (ne)
                                           (lambda (ns nm nnp)
                                             ((k ne) (extend.s ns (r id) ne)
                                                     nm
                                                     nnp ) ) ) ) ))) ) )
        (notify-struggle-error "incorrect assignment" e m s) ) )
```

Once again this decision is rather tough but in fact corresponds to the usual behaviour of classic language where languages are designed to execute programs and not to incrementally debug them. A missing binding cannot be created on the fly: the global environment is not adjustable.

Functions (closures) are created by `lambda`. The rule is slightly complicated since the arity of functions is not constrained in the language. The syntactic check of the variable list must be finite even if some weird `rplac` have been performed on it: the interpreter must not loop ! Therefore to ease the syntactic check we restrict a variable list to contain at most `*struggle-maximum-number-of-variables*` variables which all must be symbols: the predicate `variable-listp` takes care of that. The element of **Val** which represents the list of variables is afterthat converted into an element of $\mathbf{Id}^\star$ by means of `make-variable-list`.

```
(define (lambda-meaning e r k s m p)
    (if (and (is-a-cons (s (cdr-of-cons e)))
             (variable-listp (s (car-of-cons (s (cdr-of-cons e))))
                             s
                             *struggle-maximum-number-of-variables* )
             (is-a-cons (s (cdr-of-cons (s (cdr-of-cons e))))) )
        (let* ((np (owner-of-cons e))
               (id* (make-variable-list
                      (s (car-of-cons (s (cdr-of-cons e)) )) s ))
               (length-ids (length id*))
               (body (s (car-of-cons
                          (s (cdr-of-cons (s (cdr-of-cons e)))) ))) )
          (m s (list (make-thread (owner-of-cons e)
                       (k (lambda (e* nk)
                            (lambda (ns nm nnp)
                              (if (= length-ids (length e*))
                                  (let ((a* (new-locations ns length-ids)))
                                    (nm (extend*.s ns a* e*)
                                        (list (make-thread nnp
                                                (meaning body
                                                  (extend*.r r id* a*)
                                                  nk ) ) ) ) ) )
                                  (notify-struggle-error
                                    "incorrect number of arguments"
                                    e nm ns ) ) ) ) ) ))) )
        (notify-struggle-error "incorrect abstraction" e m s) ) )
```

Some auxiliary functions are used: `extend*.s` extends a store with a set of locations and a set of values, `extend*.r` acts similarly on an environment with a set of identifiers and a set of locations, `new-locations` is implementation-dependent and provides fresh locations guaranteed not to appear in the store given as its first argument.

An associated rule concerns application where a closure or a predefined function is applied:

```
(define (application-meaning e r k s m p)
   (m s (list (make-thread (owner-of-cons e)
              (meaning (s (car-of-cons e))
                       r
                       (lambda (f)
                         (meaning* (s (cdr-of-cons e))
                                   r
                                   (lambda (e*)
                                     (if (is-a-function f)
                                       (f e* k)
                                       (notify-struggle-error "not a function"
                                                  e* m s ) ) ) ) ) ) ))) )
```

It makes use of `meaning*`, a reminiscence of the classical **evlis** [McCarthy et al. 62] ;;; *meaning**: **Val** $\times$ **Env** $\times$ **Cont** $\rightarrow$ **Seq**

```
(define (meaning* e r k)
  (lambda (s m p)
    (if (is-a-cons e)
        ((meaning (s (car-of-cons e))
                  r
                  (lambda (ne)
                    (meaning* (s (cdr-of-cons e))
                              r
                              (lambda (ne*)
                                (k (cons ne ne*)) ) ) ) )
         s m p )
        ((k (list)) s m p) ) ) )
```

The initial threads are run with an initial environment containing the definition of `t`, `nil` as well as the predefined functions such as `consp` and `rplacd`: ;;; *init.consp*: **Val*** $\times$ **Cont** $\rightarrow$ **Seq**

```
(define (init.consp e* k)
     (lambda (s m p)
       (if (= (length e*) 1)
           (m s (list (make-thread p
                        (k (if (is-a-cons (car e*))
                               boolean-true boolean-false )) )))
           (notify-struggle-error "incorrect consp form" e* m s) ) ) )
(define (init.rplacd e* k)
    (lambda (s m p)
      (if (and (= (length e*) 2)
               (is-a-cons (car e*)) )
          (m (extend.s s (cdr-of-cons (car e*)) (car (cdr e*)))
             (list (make-thread p (k (make-cons (car-of-cons (car e*))
                                                (cdr-of-cons (car e*))
                                                p ) ))) )
          (notify-struggle-error "incorrect rplacd form" e* m s) ) ) )
```

The semantical variables `boolean-true` and `boolean-false` hold the value `t` and `nil` representing truth and falsity in Struggle. The `extend.s` function extends a store with a location and a value. The other predefined functions wether monadic (`car`, `cdr`) or dyadic (`eq`, `rplaca`) are based on the same patterns.

The last rule to explain, skipping `if` which is uninteresting, is related to `fork` which creates threads:

```
(define (fork-meaning e r k s m p)
   (if (and (is-a-cons (s (cdr-of-cons e)))
            (is-a-cons (s (cdr-of-cons (s (cdr-of-cons e))))) )
       (let ((np (owner-of-cons e))
             (form1 (s (car-of-cons (s (cdr-of-cons e)))))
             (form2 (s (car-of-cons (s (cdr-of-cons (s (cdr-of-cons e))))))) )
```

```
      (m s (list (make-thread np (meaning form1 r k))
                 (make-thread np (meaning form2 r k)) )) )
    (notify-struggle-error "incorrect clonation" e m s) ) )
```

After the usual syntactic checks, two threads are created and returned to the metacontinuation, each of which are based on a parameter of the **fork** special form. Like **progn**, **fork** is restricted to be binary for ease of syntax.

We only have now to explain how to start the game: (define (struggle[5] one two)

```
(let ((init.k (lambda (e) (lambda (s m p) (m s (list)))))
      (store init.s)
      (one.r init.r) (two.r init.r)
      (none nil) (ntwo nil) )
  (set! one (incorporate 1 one store (lambda (e s) (set! store s) e)))
  (set! two (incorporate 2 two store (lambda (e s) (set! store s) e)))
  (set! none (embed 1 one two store (lambda (e s) (set! store s) e)))
  (set! ntwo (embed 2 two one store (lambda (e s) (set! store s) e)))
  (set! one.r (make-init.r store (lambda (r s) (set! store s) r)))
  (set! two.r one.r)
  (schedule store
          (list (make-thread 1 (meaning none one.r init.k)))
          (list (make-thread 2 (meaning ntwo two.r init.k)))
          1
          *struggle-steps* ) ) )
```

The fighters are held by variables **one** and **two**. Their values are actually values in the store of the implementing language and must therefore be **incorporate**-d in the STRUGGLE store. The two obtained fighters must afterthat be **embed**-ded in a STRUGGLE expression which will provide the correct values for the predefined **me** and **it** variables: `‘((lambda (me it) ,one) ’,one ’,two)` `‘((lambda (me it) ,two) ’,two ’,one)`

Separate environments **one.r** and **two.r** are provided to allow fighters to consider as safe the initial bindings for predefined entities. Eventually the scheduler is started with the appropriate threads. The number of rounds of the fight is limited by the value of ***struggle-steps*** which has to be agreed upon before the fight.

2 Discussion

We do not introduce **call/cc** in our model since we do not think that it is essential for STRUGGLE. It can be straightforwardly[6] denoted if needed.

It might be also useful to add some randomness in fighters since it seems that no winning strategy can be devised. For each fighter it is probably feasible to derive a new fighter which kills the first one. Randomness can be added with the **either** construct. **either** is binary and evaluates either its first or second parameter. To add some randomness may lead to better *stochastic* fighters.

We chose to share the initial environment between the two fighters i.e.attacks through modifications of the values held in the shared locations such as **t**, **nil**, **car** or even **rplaca** are allowed. A previous version of this paper keeps separate these two environments but we now find this new variant funnier.

One may also recognize that it is not necessary to provide fighters in a readable way: fighters may be computed results. To compute fighters does not change the semantics of STRUGGLE but allow fighters to be syntactically recursive programs i.e.their dotted pair representation is no more a tree nor a directed acyclic graph but an unrestricted graph. To have such fighters augment the possibility to have redundant structure which can be seen from the opponent's point of view as a more complex maze.

As in Scheme there is no **eval** feature in STRUGGLE. It is nevertheless very easy to have it since it is possible to graft code onto its own code thus providing an **eval** facility. Care should be

[5] Use of the assignment within the definition of `struggle` might have been avoided in this denotation but would have made it at least unreadable.

[6] That is an indirect offer of a not so trivial exercice.

taken to introduce legal programs and not arbitrary elements of **Value**. Such definition of `eval` is:

```
(setq eval (lambda (e)
             (prog2 (rplaca 'it e)
                    it:wait ) ))
```

The argument `e` is grafted in the location with the *it* label. To create such a structure is easy once fighters may be computed.

STRUGGLE can be extended to multiple fighters. We can simply give to a fighter the unordered list of the code of its opponents. Fighters may concert themselves to act as synergetic packs of fighters. The main problem is therefore to identify friends and foes in the list of opponents which is a difficult problem in such an hostile environment.

The denotation is important since it provides the meaning of ownership of pairs and threads but also the number of steps that takes any simple computation. We decided to take uniform speed for each primitive step, although this is not true in usual Lisp systems, since it is then easier to compute the number of steps that an attack will last.

The equations to which we arrive are a kind of denotation but are they really ? What they denote ? There is no such thing as texts of program. We cannot say that we denote the initial store but it is a changing entity which meaning (the identity of the winner) depends on itself. We can nonetheless remark that the functions we gave throughout this paper are purely functional, operate on domains and as such looks like regular denotational equations. STRUGGLE is in fact a definitional interpreter which can fail to determinate the winner in a finite time.

3 Some Fighters

We have not yet fully understood how to conceive perfect fighters. We hope that interested readers will invent their own. We will just provide two simple but non trivial fighters. Remember however that attack and defence cannot be done altogether since they both consumes some steps to be performed.

The first fighter protects itself as quickly as it can and then loops forever. It can only win if the other dies. It does not seem possible to kill it once it is detached from the initial variables `me` and `it`. It also detach itself as soon as possible hence the unusual nesting of forms:

```
((lambda (loop) ((setq loop (lambda () (loop)))))
 (rplaca (rplacd me nil) nil) )
```

The other is blind but often successful. It strikes as he can its opponent by grafting its own code in `it` and thus it may destroy itself either ! `((lambda (sweep)`

```
   (setq sweep (lambda (x a d)
                 (if (consp x)
                   (prog2 (prog2 (rplaca x me)
                                 (sweep a (car a) (cdr a)) )
                          (prog2 (rplacd x me)
                                 (sweep d (car d) (cdr d)) ) )
                   x ) ))
   (sweep it (car it) (cdr it)) )
  t )
```

By combining an endless loop with an offensive code one may investigate other strategies.

4 Direct Implementations

If STRUGGLE is restricted to two players then the ownership of dotted pairs can be coded with a single bit. Classical representation of dotted pairs on stock hardware usually offers some spare bits that can be used to represent ownership.

The other problem is the great consumption of threads since to run a thread in the denotation generally creates a new thread which is the old one, one step further. A direct implementation will represent a thread with a dotted pair or a structure and will update it to reflect the progression of the evaluation. To ensure that the two fighters have the same amount of cpu, threads cannot be implemented as lightweight process on the underlying operating system: the STRUGGLE scheduler must then be part of the implementation.

5 Conclusions

We present an original use of denotational semantics to specify a game. We also describe a rudimentary system with threads with a non trivial denotations. The description is accurate enough to allow direct and efficient implementation. We hope that such a game will offer both pleasure when playing and thinking on the hidden nature of evaluation when reading.

References

[Dewdney84] A. K. Dewney, Scientific American, May 1984.

[McCarthy et al. 62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin, *Lisp 1.5 Programmer's Manual*, MIT Press, Cambridge, Massachusetts, 1962.

[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.

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

[Wand & Friedman 86] Mitchell Wand, Daniel Friedman, *The Mystery of the Tower revealed: A non reflective Description of the Reflective Tower*, 1986 ACM Conference on Lisp and Functional Programming, pp 233–248.

Annex

The following functions are missing from the text but are necessary to complete the description of STRUGGLE.

```
;;; create a new initial environment in store s
(define (make-init.r s kk)
  (let ((r (lambda (n) 'no-location))
        (name* '(car cdr consp eq rplaca rplacd t nil))
        (f* (list init.car init.cdr init.consp
                  init.eq init.rplaca init.rplacd
                  boolean-true boolean-false )) )
    (mapc (lambda (name f)
            (let ((a (new-location s)))
              (set! r (extend.r r name a))
              (set! s (extend.s s a f)) ) )
          name* f* )
    (kk r s) ) )

;;; put the expression ee in store s with pairs belonging to p
(define (incorporate p ee s kk)
  (if (pair? ee)
    (incorporate p (car ee)
      s (lambda (the-car s)
          (incorporate p (cdr ee)
            s (lambda (the-cdr s)
                (let* ((a* (new-locations s 2))
```

```
                        (the-cons (make-cons (car a*)
                                             (car (cdr a*))
                                             p )) )
                   (kk the-cons
                       (extend.s (extend.s s (car a*) the-car)
                                 (car (cdr a*))
                                 the-cdr ) ) ) ) ) ) )
      (if (null? ee) (kk 'nil s)
          (if (symbol? ee) (kk ee s)
              (struggle-error "Not a program " ee) ) ) ) ) )
```

;;; *Check if value* `e` *in store* `s` *is a valid variable list.*

```
(define (variable-listp e s max)
  (if (< max 0)
      #f
      (if (is-a-cons e)
          (if (is-an-id (s (car-of-cons e)))
              (variable-listp (s (cdr-of-cons e)) s (- max 1))
              #f )
          #t ) ) )
```

;;; *convert a valid variable list into a sequence of identifiers*

```
(define (make-variable-list e s)
  (if (is-a-cons e)
      (cons (s (car-of-cons e))
            (make-variable-list (s (cdr-of-cons e)) s) )
      () ) )
```

;;; *consruct ((lambda (me it) ,me) ',me ',it)*

```
(define (embed p me it s kk)
  (let* ((a* (new-locations s 24))
         (final-two (make-cons (nth 0 a*) (nth 1 a*)  p))
         (quoted-two (make-cons (nth 2 a*) (nth 3 a*) p))
         (final-one (make-cons (nth 4 a*) (nth 5 a*) p))
         (quoted-one (make-cons (nth 6 a*) (nth 7 a*) p))
         (second-arg (make-cons (nth 8 a*) (nth 9 a*) p))
         (first-arg (make-cons (nth 10 a*) (nth 11 a*) p))
         (body (make-cons (nth 12 a*) (nth 13 a*) p))
         (second-var (make-cons (nth 14 a*) (nth 15 a*) p))
         (first-var (make-cons (nth 16 a*) (nth 17 a*) p))
         (var-list (make-cons (nth 18 a*) (nth 19 a*) p))
         (function (make-cons (nth 20 a*) (nth 21 a*) p))
         (form (make-cons (nth 22 a*) (nth 23 a*) p)) )
    (set! s (extend.s s (nth 0 a*) it))
    (set! s (extend.s s (nth 1 a*) 'nil))
    (set! s (extend.s s (nth 2 a*) 'quote))
    (set! s (extend.s s (nth 3 a*) final-two))
    (set! s (extend.s s (nth 4 a*) me))
    (set! s (extend.s s (nth 5 a*) 'nil))
    (set! s (extend.s s (nth 6 a*) 'quote))
    (set! s (extend.s s (nth 7 a*) final-one))
    (set! s (extend.s s (nth 8 a*) quoted-two))
    (set! s (extend.s s (nth 9 a*) 'nil))
    (set! s (extend.s s (nth 10 a*) quoted-one))
    (set! s (extend.s s (nth 11 a*) second-arg))
    (set! s (extend.s s (nth 12 a*) me))
    (set! s (extend.s s (nth 13 a*) 'nil))
    (set! s (extend.s s (nth 14 a*) 'it))
```

```
    (set! s (extend.s s (nth 15 a*) 'nil))
    (set! s (extend.s s (nth 16 a*) 'me))
    (set! s (extend.s s (nth 17 a*) second-var))
    (set! s (extend.s s (nth 18 a*) first-var))
    (set! s (extend.s s (nth 19 a*) body))
    (set! s (extend.s s (nth 20 a*) 'lambda))
    (set! s (extend.s s (nth 21 a*) var-list))
    (set! s (extend.s s (nth 22 a*) function))
    (set! s (extend.s s (nth 23 a*) first-arg))
    (kk form s) ) )
```

;;; *Rule for alternative*

```
(define (if-meaning e r k s m p)
   (if (and (is-a-cons (s (cdr-of-cons e)))
            (is-a-cons (s (cdr-of-cons (s (cdr-of-cons e)))))
            (is-a-cons (s (cdr-of-cons
                           (s (cdr-of-cons
                               (s (cdr-of-cons e)) )) ))) )
       (let ((np (owner-of-cons e))
             (form (s (car-of-cons (s (cdr-of-cons e)))))
             (then (s (car-of-cons (s (cdr-of-cons (s (cdr-of-cons e)))))))
             (else (s (car-of-cons
                       (s (cdr-of-cons
                           (s (cdr-of-cons
                               (s (cdr-of-cons e)) )) )) ))) )
         (m s (list (make-thread np
                      (meaning form r
                           (lambda (ne)
                             (when *struggle-debug*
                                   (print `(return to if
                                             ,(convert-to-Sexp ne s))) )
                             (if (samep.e ne boolean-false)
                                 (meaning else r k)
                                 (meaning then r k) ) ) ) ))) )
       (notify-struggle-error "incorrect alternative" e m s) ) )
```

;;; *Definition of car*

```
(define (init.car e* k)
    (lambda (s m p)
      (if (and (= (length e*) 1)
               (is-a-cons (car e*)) )
          (m s (list (make-thread p (k (s (car-of-cons (car e*)))))))
          (notify-struggle-error "incorrect car form" e* m s) ) ) )
```

;;; *Definition of cons*

```
(define (init.cons e* k)
    (lambda (s m p)
      (if (= (length e*) 2)
          (let ((a* (new-locations s 2)))
            (m (extend*.s s a* e*)
               (list (make-thread p
                        (k (make-cons (car a*) (car (cdr a*)) p)) )) ) )
          (notify-struggle-error "incorrect cons form" e* m s) ) ) )
```

;;; *Definition of eq*

```
(define (init.eq e* k)
    (lambda (s m p)
      (if (= (length e*) 2)
```

```
(m s (list (make-thread p
             (k (if (samep.e (car e*) (car (cdr e*)))
                    boolean-true boolean-false )) )))
(notify-struggle-error "incorrect eq form" e* m s) ) ) )
```