Pastis, a Polymorphic Application System to Transform Internal Signatures

Antoine Amarilli, Pablo Rauzy

June 18, 2010

Abstract

In this note, we present Pastis, a Scheme toolkit which produces, from a given payload, a new program which behaves in the same way as the payload but additionally produces a rewritten version of itself on standard output. The rewriting applies a set of rules to change the form of the source code while keeping it functionally equivalent. We finally present the rewriting rules and some possible extensions.

Contents

1 General principles

1.1 Objectives

The general aim of Pastis is to make programs able to rewrite themselves in a way which does not alter their functionality but changes their code as much as possible. Such techniques can already be seen in polymorphic viruses as a way to foil signature-based detection attempts by anti-virus software; they also appear in code obfuscation systems.

1.2 Tools

Pastis is written in Scheme, for the MIT Scheme implementation. The payload to transform also has to be written in Scheme, and is restricted to the subset of the Scheme syntax which the rewriting system is able to understand. (Of course, since the rewriting engine has to rewrite itself, it is itself written using this limited subset of Scheme.)

2 Structure

2.1 General design

The top-level Pastis function is `pastis-generator`. It creates the self-rewriting program out of the payload and out of a rewriting function (which takes code as input and produces functionally equivalent rewritten code as its return value).

The produced code behaves functionally like the payload function: it will be evaluated to the same value if it gets the same parameters. However, it will additionally print, during the evaluation, a rewritten, equivalent version of itself.

Of course, the rewritten version is still functionally equivalent to the original payload, and will also produce a rewritten version of itself, which can be run itself, and so on ad infinitum (forgetting about the growing size of the rewritten code, ie. assuming that we have an infinite amount of memory).

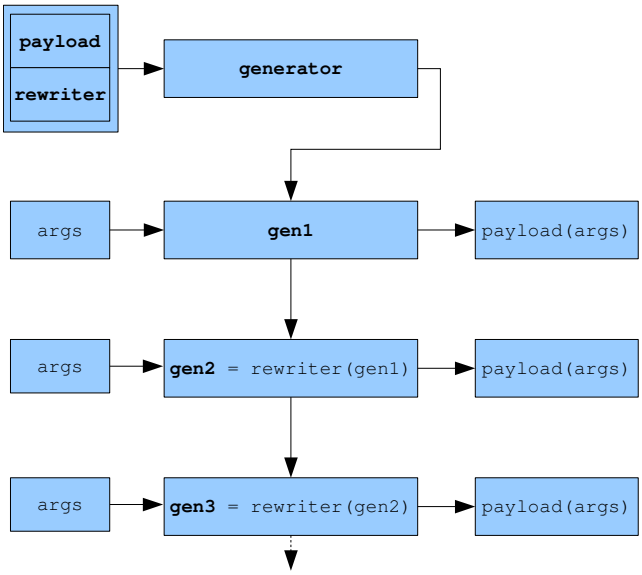

Figure 1: Summary of the use of Pastis

2.2 Internal structure

The use of the `pastis-generator` function is quite straightforward; its role is to provide a convenient mechanism to put the payload and rewriter together in such a way as to make self-rewriting possible.

Here is an example of the use of `pastis-generator`. The payload here is a simple function which adds 42 to its parameter, and the rewriter is the identity function.

```
(pastis-generator
 '((payload . (lambda (x) (+ 42 x)))
   (rewriter . (lambda (x) x))))
```

Here is the resulting code produced by the `pastis-generator` function.

```
(lambda (args)
  (define (pastis-rewrite x)
    ((lambda (x) x) x))
  (define (pastis-payload x)
    ((lambda (x) (+ 42 x)) x))
  (define (pastis-ls l)
    (map (lambda (x) (write (pastis-rewrite x)) (display " ")) l))
  (define (pastis-code l)
    (display "(")
    (pastis-ls l)
    (display "(pastis-code '(")
    (pastis-ls l)
    (display ")) (pastis-payload args))\n"))
  (pastis-code
   '(lambda (args)
      (define (pastis-rewrite x)
        ((lambda (x) x) x))
      (define (pastis-payload x)
        ((lambda (x) (+ 42 x)) x))
      (define (pastis-ls l)
        (map (lambda (x) (write (pastis-rewrite x)) (display " ")) l))
      (define (pastis-code l)
        (display "(")
        (pastis-ls l)
        (display "(pastis-code '(")
        (pastis-ls l)
        (display ")) (pastis-payload\nargs))\n"))))
  (pastis-payload args))
```

2.3 Step by step explanations

The code generated by `pastis-generator` seems complicated, but its structure is in fact very similar to that of the following classical quine.

```
(define (d l) (map write-line l))
(define (code l) (d l) (display "(code '(\n") (d l) (display "))\n"))
(code '(
(define (d l) (map write-line l))
(define (code l) (d l) (display "(code '(\n") (d l) (display "))\n"))
))
```

Adding a payload to this quine is quite straightforward.

```
(define (payload) (write "Hello, World!\n"))
(define (ls l) (map write-line l))
(define (code l) (ls l) (display "(code '(\n") (ls l) (display "))\n"))
(payload)
(code '(
  (define (payload) (write "Hello, World!\n"))
  (define (ls l) (map write-line l))
  (define (code l) (ls l) (display "(code '(\n") (ls l) (display "))\n"))
  (payload)))
```

Given Pastis's role, it is quite easy to see that it is related to quines. The only difference is that Pastis has to modify its code before printing it, instead of printing it verbatim as regular quines do. This is also quite easy to do.

However, some deeper changes are required if we want to be able to pass parameters to the payload: the classical quine's structure doesn't allow that. The solution to make a quine that is also a lambda expression (instead of a list of statements). This is possible, thanks to S-expressions.

The way the quine works relies on the fact that its code is a list of statements and that the last one can take a list of the previous ones as arguments. Making the whole quine a lambda expression in order to accept arguments for the payload means making it a single expression. But thanks to the language we are using, it appears that this single expression is *still* a list. This enables us to solve our problem. Here is the result:

```
(lambda (args)
  (define (payload x) (+ x 42))
  (define (ls l) (map write-line l))
  (define (code l)
    (display "(")
    (ls l) (display "(code '(")
    (ls l) (display "))\n(payload x))"))
  (code '(lambda (x)
           (define (payload x) (+ x 42))
           (define (ls l) (map write-line l))
           (define (code l)
             (display "(")
             (ls l) (display "(code '(")
             (ls l) (display "))\n(payload x))"))))
  (payload args))
```

3 Rewriter

In addition to `pastis-generator`, we also provide a `rewriter` function. Its role is to call specialized rewriters for each keyword, which will call `rewriters` recursively on their arguments if appropriate.

The specialized rewriters randomly choose a way to rewrite the top-level construct. Here is a list of implemented rewriting rules:

- Rewriting `if` as `case` or `cond` ;
- Rewriting `case` as `if` or `cond` ;
- Rewriting `cond` as `if` or `case` ;
- Adding a `not` to change the order of `if` branches.

4 Results

Pastis was tested with a simple payload and the example rewriter provided. It was possible to produce 25 generations before the rewritten code grew too big to be executed and failed with an "out of memory" error.

The code size of the generations increases steadily, which seems to demonstrate that the `rewriter` function provided often adds new constructs, but seldom simplifies out the useless ones.

The produced code is still fairly recognizable: the keywords aren't rewritten, and highly recognizable intermediate variables appear everywhere in the code. Furthermore, the numerous tautological conditional branches (of the form `(if #t foo #!unspecific)`) and useless nesting of operators are also a sure way to identify code produced by Pastis.

Here is an example of the code which Pastis produces after some iterations.

```
(lambda (args) (define (pastis-rewrite x) ((lambda (x) (define (rewrite-if s) (define
get-cond (rewrite (cadr s))) (define get-then (rewrite (caddr s))) (define get-else (let
((key906856151243052050955555138 (not (not (not (null?  (cdddr s)))))))) (let
((key450662953449777477537902121 key906856151243052050955555138)) (let
```

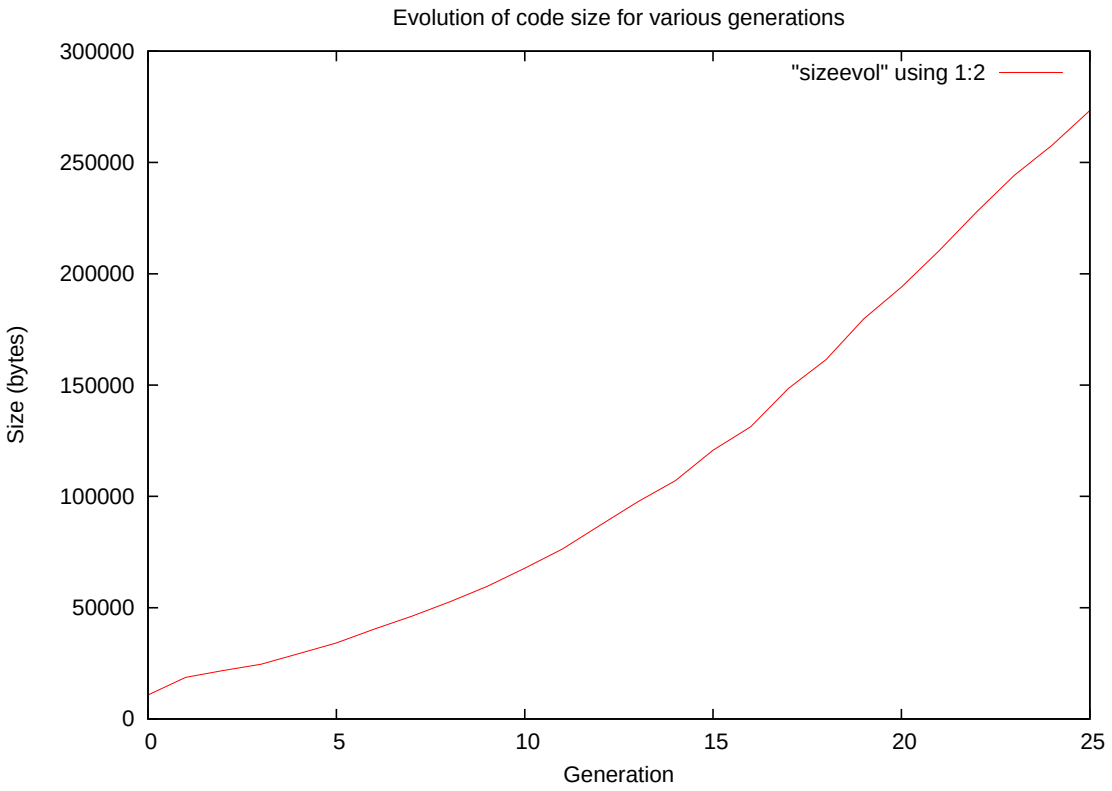

Figure 2: Evolution of size with generations

```
((key347530381576358560503334l3 (not (or (eq?  key450662953449777475379021l1 (quote
#f)))))) (let ((key74822769707555069929340259 (not (or (eq?  key347530381576358560503334l3
(quote #f)))))) (cond ((not (not (not (not (or (eq?  key74822769707555069929340259 (quote
#t))))))) (let ((key153009514049004536190920l6 #t)) (if (or (eq?
key153009514049004536190920l6 (quote #t))) (begin (case #t ((#t) (let
((key87838842585550845645406647 (not (or (eq?  key74822769707555069929340259 (quote
#t)))))) (case (not (or (eq?  key87838842585550845645406647 (quote #f)))) ((#t) (let
((key41701470274885460121759385 key87838842585550845645406647)) (if (not (not (or (eq?
key41701470274885460121759385 (quote #t))))) (if (not (or (eq?
key41701470274885460121759385 (quote #t)))) (let ((key98134142793119041861980707 #t)) (if
(or (eq?  key98134142793119041861980707 (quote #t))) (begin 42)
```

It is interesting to note that the self-referencing nature of Pastis makes it very hard to debug. When the third generation fails to run, for example, one needs to find the bug in the third generation code, identify what caused the second generation to produce it, and finally which code in the first generation causes the second generation to work this way. Several cases of bugs only occurring after several generations appeared during the development of Pastis.

5 Possible extensions

The current `rewriter` function only serves as an example. First, it leaves several recognizable features in the code. More importantly, the transformations it applies aren't very deep, since one could simply decide to only use `cond` constructs, systematically rewrite all `if` and `case` constructs to `cond`, and focus on the rewriting of `cond`. To be more precise, `if` and `case` can be seen as Scheme syntactic sugar; it would probably be better to restrict the rewriting to a bare bones subset of the Scheme syntax, convert everything to this subset before rewriting, and possibly convert some things back to syntactic sugar forms to make the rewritten code look more natural.

Several transformations could be applied instead of the simplistic operations done by our `rewriter` function. Here are a few ideas.

5.1 Alpha-renaming

The current `rewriter` does not rename variables at all. A way to do this would be to keep an environment indicating current renamings. When we encounter a definition, we change the name and add the original and rewritten name to the environment. When we encounter a name, we change it to the appropriate rewritten name by a simple lookup in the environment.

5.2 Beta-reduction and beta-expansion

A possibility for rewriting would be to perform beta-reductions (in the usual sense of the lambda-calculus). Conversely, it would also be possible to perform beta-expansions: select a subterm, replace it by a variable, and replace the whole expression by a function in this variable applied to the selected subterm, taking all necessary care to prevent variable capture problems (roughly, ensuring that the operation does not bind other occurrences of the new variable, and that the bindings of the subterms are still the same).

Of course, if we want to do such an operation without changing the semantics, we must ensure first that there is no breach of referential transparency. Indeed, if side effects are taking place, the planned modifications could change the order of evaluation, or even the number of evaluations of some subterms.

5.3 Adding and removing definitions

This would be the ability for the rewriter to add or remove local definitions when possible. When the rewriter sees an expression $E($*expr*$)$ it could replace it with `(let ((const` *expr*`))` $E($`const`$)$. This is very similar to the idea of beta-reduction and beta-expansion previously mentioned, and could be implemented in a similar way.

Source code

```
(define (rewrite-if s)
  (define get-cond (rewrite (cadr s)))
  (define get-then (rewrite (caddr s)))
  (define get-else
    (if (not (null? (cdddr s)))
        (rewrite (cadddr s))))
  (define get-else-beginned
    (if (and (pair? get-else)
             (eq? (car get-else) 'else))
        `(begin ,@(cdr get-else))
        (if (eq? get-else #!unspecific)
            '#!unspecific
            get-else)))

  (if (and (pair? get-cond)
           (eq? (car get-cond) 'not)
           (< (random 3) 2))
      (rewrite-if `(if ,@(cdr get-cond) ,get-else-beginned ,get-then)))
      (case (random 5)
        ((0) `(if (not ,get-cond) ,get-else-beginned ,get-then))
        ((1) `(cond (,get-cond ,get-then) (else ,get-else-beginned)))
        ((2) `(cond ((not ,get-cond) ,get-else-beginned)
                    (,get-cond ,get-then) (else 42)))
        ((3) `(case ,get-cond ((#f) ,get-else-beginned) ((#t) ,get-then)))
        ((4) (if (eq? get-else #!unspecific)
                 `(if ,get-cond ,get-then)
                 `(if ,get-cond ,get-then ,get-else)))))


(define (rewrite-cond s)
  (define get-cond
    (if (eq? (caadr s) 'else) #t (rewrite (caadr s))))
  (define get-then (rewrite (cdadr s)))
  (define get-rest
    (if (not (or (null? (cddr s))
                 (eq? (caddr s) #!unspecific)
                 ;(eq? (cadr s) 'else)
                 (eq? (caddr s) 'else)))
        (rewrite `(cond ,@(cddr s)))))

  (if (eq? get-rest #!unspecific)
      (case (random 3)
        ((0) `(if ,get-cond ,@get-then))
        ((1) `(cond (,get-cond ,@get-then)))
        ((2) `(case ,get-cond
                ((#t) ,@get-then))))
      (case (random 4)
        ((0) `(if ,get-cond ,@get-then ,get-rest))
        ((1) `(if (not ,get-cond) ,get-rest ,@get-then))
        ((2) `(cond (,get-cond ,@get-then) (#t ,get-rest)))
        ((3) `(case ,get-cond
                ((#t) ,@get-then)
                ((#f) ,get-rest))))))


(define (rewrite-case s)
```

```
  (define get-key (rewrite (cadr s)))
  (define get-next-case-object (caaddr s))
  (define get-next-case-exprs (cdaddr s))

  ; generate a random (supposedly unique) variable name to prevent
  ; capture problems when rewriting a code containing a variable with
  ; the same name (especially this very code)
  (define case-key
    (string->symbol
     (string-append "key"
                    (number->string (random 99999999999999999999999999)))))

  (define get-rest
    (if (not (or (null? (cdddr s))
                 (eq? (caddr s) 'else)))
        (rewrite `(case ,case-key ,@(cdddr s)))))

  (define (disjonction list)
    (define (op-list list)
      (if (null? list) '()
          `((eq? ,case-key (quote ,(car list)))
            ,@(op-list (cdr list)))))
    (if (list? list)
        `((or ,@(op-list list)))
        `((eq? ,case-key (quote ,list)))))

  (if (eq? get-next-case-object 'else)
      `(begin ,@get-next-case-exprs)
      (case (if (eq? get-rest #!unspecific)
                0
                (random 3))
        ((0) `(let ((,case-key ,get-key))
                (if ,@(disjonction get-next-case-object)
                    (begin ,@get-next-case-exprs)
                    ,get-rest)))
        ((1) `(let ((,case-key ,get-key))
                (cond
                 (,@(disjonction get-next-case-object)
                  ,@get-next-case-exprs)
                 (#t ,get-rest))))
        ((2) `(let ((,case-key ,get-key))
                (case ,case-key
                  (,get-next-case-object ,@get-next-case-exprs)
                  (else ,get-rest)))))))


(define (rewrite-else s)
  (define get-else (rewrite (cdr s)))

  (case 1;(random 2)
    ((0) `(#t ,@get-else))
    ((1) `(else ,@get-else))))


(define (rewrite-let s)
  `(let ,(cadr s) ,@(rewrite (cddr s))))
```

```
(define (rewrite-lambda s)
  `(lambda ,(cadr s) ,@(rewrite (cddr s))))


(define (rewrite-define s)
  `(define ,(cadr s) ,@(rewrite (cddr s))))


(define (rewrite-quote s) s)


(define (rewrite-quasiquote s) s)


(define (rewrite-unquote s)
  `(unquote ,@(rewrite (cadr s))))

(define (rewrite-unquote-splicing s) s)


;; remember not to want to add the possibility to swap or and and operands:
;; scheme haz lazy evaluation for this operator and the swap could result in
;; a change in semantics

(define (rewrite-or s)
  `(or ,@(map rewrite (cdr s))))


(define (rewrite-and s)
  `(and ,@(map rewrite (cdr s))))



(define (rewrite s)
  (if (pair? s)
      (case (car s)
        ((if) (rewrite-if s))
        ((else) (rewrite-else s))
        ((cond) (rewrite-cond s))
        ((case) (rewrite-case s))
        ((let) (rewrite-let s))
        ((lambda) (rewrite-lambda s))
        ((define) (rewrite-define s))
        ((quote) (rewrite-quote s))
        ((quasiquote) (rewrite-quasiquote s))
        ((unquote) (rewrite-unquote s))
        ((unquote-splicing) (rewrite-unquote-splicing s))
        ((and) (rewrite-and s))
        ((or) (rewrite-or s))
        (else (map rewrite s)))
      s))



(define (pastis-generator s)
  ;; s must be a quoted list of named items. A named item is a pair which have
  ;; a name (symbol) as car and a value as cdr.
```

```
;; For now the generator needs two items: "rewriter" and "payload". Both must
;; have a callable (for instance a lambda expression) value.

;; In the future there could be an item named "rewrite-features" whose value
;; would be a list of code rewriting features (choosen among a given list) to
;; use.
;; We also need a syntax to add custom rewriting features.

(define (pastis-get-item item s)
  ;; get the definition of item from s
  (if (eq? (caar s) item)
      (cdar s)
      (pastis-get-item item (cdr s))))

(let ((quine-code
       `((define (pastis-rewrite x)
           (,(pastis-get-item 'rewriter s) x))
         (define (pastis-payload x)
           (,(pastis-get-item 'payload s) x))
         (define (pastis-ls l)
           (map (lambda (x)
                  (write (pastis-rewrite x))
                  (display " ")) l))
         (define (pastis-code l)
           (display "(") (pastis-ls l)
           (display "(pastis-code '(") (pastis-ls l)
           (display ")) (pastis-payload args))\n")))))

  `(lambda (args) ,@quine-code
           (pastis-code '(lambda (args) ,@quine-code))
           (pastis-payload args))))
```