Referential Transparency is Overrated

But let's keep this between us…

Didier Verna

didier@lrde.epita.fr
http://www.lrde.epita.fr/˜didier
http://www.facebook.com/didierverna
@didierverna

ACCU 2015 – Thursday, April 23rd

Table of contents

Natural Languages (Analytical Philosophy)

Origins

Quine

- reference $\approx$ meaning
- replacing an expression by another one which refers to the same thing doesn't alter the meaning

Example: "Wallace's dog" $\equiv$ "Gromit"

✔ Tomorrow, I'll go feed Wallace's dog.

✖ Gromit isn't Wallace's dog anymore.

Programming Languages (Semantics)

Inspired from Quine

Strachey[1]

If we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the sub-expression is its value.

Reade[2]

Only the meaning of immediate sub-expressions is significant in determining the meaning of a compound expression. Since expressions are equal if and only if they have the same meaning, [it] means that substitutivity of equality holds.

---

[1] Fundamental Concept of Programming Languages, 1967

[2] Elements of Functional Programming, 1989

Purely Functional Languages

A more extremist view

Take your pick

An expression which can be replaced with its value without changing the behavior of a program.

Evaluation of the expression simply changes the form of the expression but never its value.

All references to a value are equivalent to the value itself.

There are no other effects in any procedure for obtaining the value.

The original points

Quine and Strachey agreed

Quine's point

Natural languages are complicated because they need to be practical.

Strachey's point

The same! And BTW, he was talking about *imperative* languages.

A sound denotational semantics would render even imperative languages referentially transparent (by telling you when two expressions are equal).

Purity *vs.* Referential Transparency

Are we talking about the same thing ?

What purely functional programmers talk about

- values instead of meaning
- evaluation process becomes relevant
- side effects (or lack thereof)

The Typical PFP's argument

Which is refutable

This is referentially transparent

```
int plus_one (x) { return x + 1; } /* plus_one (1) is always 2 */
```

This is not

```
int foo = 10;
int plus_foo (x) { return x + foo; } /* plus_foo (1) depends on foo */
```

Really? What about this?

```
foo :: Int
foo = 10

plus_foo :: Int -> Int
plus_foo x = x + foo -- plus_foo 1 is always 11...

let foo = 20 in let plus_foo x = x + foo in plus_foo 1 -- 21. Woops!
```

One final definition

Gotta stop somewhere

Gelernter & Jagannathan

> *A language is referentially transparent if (a) every subexpression can be replaced by any other that's equal to it in value and (b) all occurrences of an expression within a given context yield the same value.*

- Applies to *languages*, not expressions
- Mostly rules mutation out

Intermediate Conclusion

Where to go from here

- There is *always* some form of context dependency (lexical / dynamic definitions, free / bound variables, side effects *etc*)
- PFPs disregard their own contexts (lexical and purity)
- PFPs reduce the notion of “meaning” to that of “value” (result of a λ-calculus evaluation process)

Consequently, I hereby claim that the expression “referential transparency” is not referentially transparent :-).

Optimization, Safety, Expressiveness. . .

Why referential transparency is profitable

- **Optimization:**
  - Memoization
  - Parallelism (Cf. Erlang)
- **Safety:**
  - Localized semantics (hence localized bugs)
  - Program reasoning and proof
- **Expressiveness:**
  - Lazy evaluation

Safety with Program Proof
Formal reasoning

Demonstrate (please) that $\forall n, ssq(n) > 0$

Purely functional

```
ssq :: Int -> Int
ssq 1 = 1
ssq n = n*n + ssq (n-1)
```

- True for $N = 1$
- Assuming it holds for $N - 1 \ldots$

Imperative

```
int ssq (int n)
{
  int i = 1, a = 0;

  while (i <= n)
    {
      a += i*i;
      i += 1;
    }

  return a;
}
```

- Ahem…

Explicit representation of infinite data structures

Haskell

```
intlist :: Int -> [ Int ]
intlist s = s : intlist (s + 1)

-- (intlist 0) !! 3 -> 3.
```

Lisp

```
(defun intlist (s)
  (cons s (intlist (1+ s))))

;; (elt (intlist 0) 3) -> ^C^C
```

Where to Look for Bindings?

Scoping

- **Lexical Scoping:** in the defining context
- **Dynamic Scoping:** in the calling context

Lexical Scope

```
(let ((x 10))
  (defun foo ()

    x))

(let ((x 20))
  (foo))         ;; -> 10
```

Dynamic Scope

```
(let ((x 10))
  (defun foo ()
    (declare (special x))
    x))

(let ((x 20))
  (foo))         ;; -> 20
```

- First Lisp was dynamically scoped
- Lexical scope since Scheme (except Emacs Lisp!)
- Common Lisp still offers both (Emacs Lisp now does)

Lexical Closures

Brought to you by lexical scope

- **Definition:**
  Combination of function definitions and their defining environment (free variables values at define-time)
- **Benefits:**
  - 1st order functional (anonymous) arguments
  - 1st order functional (anonymous) return values
  - …(*e.g.* encapsulation)
- **Lisp note:** lexical state is *mutable*

Why lexical closures are crucial

They're everywhere!

1st order functional (anonymous) arguments

```
(defun list+ (lst n)
  (mapcar (lambda (x) (+ x n))
          lst))
```

1st order functional (anonymous) return values

```
(defun make-adder (n)
  (lambda (x) (+ x n)))
```

Mutable lexical state

```
(let ((cnt 0))
  (defun newtag () (incf cnt))
  (defun resettag () (setq cnt 0)))
```

Why is dynamic scoping dangerous ?

But also useful

- **Problems:**
  - Name clashes on free variables
  - Very difficult to debug
  - Mc Carthy's first example of higher order function (1958) was wrong!
- **Advantages:**
  - Dynamic paradigms (*e.g.* COP)
  - Global variables!
    As per `defvar` and `defparameter`
    E.g. Emacs user options

Mc Carthy's bugged example
Transcribed in Common Lisp

The first mapping function

```
(defmacro while (test &rest body)
  `(do () ((not ,test))
     ,@body))

(defun my-mapcar (func lst)
  (let (elt n)
    (while (setq elt (pop lst))
      (push (funcall func elt) n))
    (nreverse n)))

(defun list+ (lst n)
  (my-mapcar (lambda (x)
               (declare (special n))
               (+ x n)) ;; Barf !!
             lst))
```

Intermediate Conclusion

Remember when I asked you about `(let ((x 10)) (foo))` ?

- **Duality of syntax (intentional):**

Lexical scope

```
(let ((lock nil))
  (defun lock () (test-and-set lock))
  (defun unlock () (setq lock nil)))
```

Dynamic scope

```
(let ((case-fold-search nil))
  (search-forward "^Bcc:␣"))
```

- **Going further**
  Semantics-agnostic macros (*e.g.* `being-true`)

Reader Macros
Hijacking the Lisp syntax

Hooking into the Lisp reader

- **readtable:** currently active syntax extension table
- **macro character:** special syntactic meaning
- **reader macro:** implements macro character behavior

Note: RTMP

Standard syntactic extensions

- `'` quote
- `#'` function
- `#c` complex
- ...

Why is RTMP useful?

Not *only* for syntactic sugar

Examples

```
;; Of course, the comment syntax is implemented with macro characters...
(asdf:defsystem :com.dvlsoft.clon
  :description "The␣Command-Line␣Options␣Nuker."
  :author "Didier␣Verna␣<didier@lrde.epita.fr>"
  :maintainer "Didier␣Verna␣<didier@lrde.epita.fr>"
  :license "BSD"
  :version #.(version :short)
  :depends-on (#+sbcl :sb-posix
               #+(and clisp com.dvlsoft.clon.termio) :cffi)
  :serial t
  #| ... |#)
```

Application to DSLs
The embedded homogeneous kind

From a previous talk:

```
;; Going from this:
{ option :foreground white
         :face { syntax :bold t :foreground cyan }
         :face { usage :foreground yellow }
}
;; To that:
(define-face option :foreground white
  :face (define-face syntax :bold t :foreground cyan)
  :face (define-face usage :foreground yellow))
```

What kind of underlying data structure would you like ?

```
{ :key1 val1 :key2 val2 #| ... |# }
```

What about this?

```
(defun random-ffi-bridge (foo bar) { struct winsize window; /* ... */ })
```

Symbol Macros
A special kind of macros

Macro-expanded symbols

```
(define-symbol-macro foo expansion-form)
;; Locally with SYMBOL-MACROLET
```

- Expansion then subject to regular macro-expansion

Example

```
(defun compute-thing () #|...|#)
(define-symbol-macro thing (compute-thing))

;; Using THING is cleaner than using (COMPUTE-THING).
```

Generalized Variables

l-values *vs.* r-values

The problem

```
(setq lst '(1 2 3)) ;; -> (1 2 3)
(nth 1 lst)          ;; -> 2

(defun setnth (nth lst newval)
  "Replace the NTH element in list LST with NEWVAL."
  (rplaca (nthcdr lst nth) newval)
  newval)

(setnth 1 lst 20)    ;; -> 20
lst                  ;; -> (1 20 3)
```

- Different setters for every data structure ?
- How boring…

The solution

```
(setf (nth 1 lst) 20)
```

Making your own
Setf expanders

- 50 or so expanders in the Lisp standard
- Accessors (struct or class instances)
- Make your own with
  - `(defun (setf foo) ...)`
  - `defsetf`
  - `define-setf-expander`

Application

Combining symbol macros and generalized variables

with-slots / with-accessors

```
(with-accessors ((origin circle-origin) (radius circle-radius)) circle
  ;; ...
  (setf origin (+ origin translation-factor))
  (incf radius 3)
  #| ... |#)
```

Crash Course

What are Lisp macros exactly?

- Ordinary Lisp functions (almost)
- Macro arguments: chunks of code (seen as data)
- Non-strict: arguments not evaluated
- Transform expressions into new expressions

Why are macros useful?

CTMP, factoring, non-strict idioms *etc*

Will this work?

```
(defun ifnot (test then else)
  (if test else then))

;; (ifnot t (error "Kaboum!") 'okay) -> Kaboum!
```

This will

```
(defmacro ifnot (test then else)
  (list (quote if) test else then))

;; (ifnot t (error "Kaboum!") 'okay) -> (if t 'okay (error "Kaboum!"))
```

Even better, and even more better

```
(defmacro ifnot (test then else)
  (list 'if test else then))

(defmacro ifnot (test then else)
  `(if ,test ,else ,then))
```

Macro pitfalls

Evaluation control, unwanted variable capture

Does this work?

```
(defmacro maybe-push (object place)
  `(when ,object (push ,object ,place)))
```

And this?

```
(defmacro maybe-push (object place)
  `(let ((obj ,object))
     (when obj (push obj ,place))))
```

At last!

```
(defmacro maybe-push (object place)
  (let ((the-object (gensym)))
    `(let ((,the-object ,object))
       (when ,the-object (push ,the-object ,place)))))
```

Intentional variable capture I

By example

This screams for abstraction

```
(defun signs (list)
  (mapcar (lambda (x) (if (= -1 (signum x)) '- '+))
          (remove-if (lambda (x)
                       (or (not (numberp x)) (complexp x)))
                     list)))
```

This screams a little less

```
(defun filter-map (term filter list)
  (mapcar term (remove-if filter list)))

(defun signs (list)
  (filter-map (lambda (x) (if (= -1 (signum x)) '- '+))
              (lambda (x) (or (not (numberp x)) (complexp x)))
              list))
```

Intentional variable capture II
By example

This doesn't scream anymore

```
(defmacro filter-map (term filter list)
  `(mapcar (lambda (x) ,term) (remove-if (lambda (x) ,filter) ,list)))

(defun signs (list)
  (filter-map (if (= -1 (signum x)) '- '+)
              (or (not (numberp x)) (complexp x))
              list))
```

- **Exercise:** write a Haskell-like list comprehension facility.

Side Note: Alternatives with Syntax Extension

More than one way…

With capture

```
(defun brace-reader (stream subchar arg)
  (declare (ignore subchar))
  (let ((body (read-delimited-list #\} stream t)))
    (push (cond ((or (null arg) (= 1 arg)) '(x))
                ((= 2 arg) '(x y))
                ((= 3 arg) '(x y z))
                ((= 4 arg) '(x y z t)))
      body)
    (push 'lambda body)
    body))
(set-dispatch-macro-character #\# #\{ #'brace-reader)

;; (#2{ (* x y) } 3 4) -> 12
```

Without capture (unicode Lisp)

```
(set-macro-character #\λ (lambda (stream char) 'lambda))

;; ((λ (x y) (* x y)) 3 4) -> 12
```

Anaphora

In the grammatical sense

Graham's classical examples

```
(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))

;; awhen, acond, awhile, aand etc.

(defmacro alambda (args &body body)
  `(labels ((self ,args ,@body))
     #'self))
```

- And the all-mighty and highly controversial `loop` macro!

Pure (Free Variable) Injection
Lexical trojans

In its simplest form

```
(defmacro inject () 'x)
```

Application: Lexical Communication Channels

Under the hood. . .

Principle:

- Two or more macros communicating with each other by injecting / capturing lexical bindings (variables, macros, symbol macros *etc*)
- This lexical communication channel does not even have to be visible in the source code

Examples

Cf. live demo (if it works. . .)

Tracing anaphora

```
(tracing-conditionals
  ;; ...
  (if this do-this do-that)
  #|...|#)
```

Alternate version

```
(tracing-conditionals...
  ;; ...
  (if this (progn do-this ...here) do-that)
  #|...|#)
```

Conclusion

Bringing programming languages closer to natural ones

- Referential transparency is useful
- Breaking it is also useful (readability, concision)
- Breaking it is dangerous (safety *vs.* expressiveness)