On Reasoning about Recursive Programs using Structural Induction*

Olivier Danvy
Department of Computer Science
Aarhus University†

February 11, 2009

Abstract

This note was written at the occasion of the retirement of Professor Jean-François Perrot at the Université Pierre et Marie Curie (Paris VI). In an attempt to emulate his academic spirit, we revisit an example proposed by Patrick Greussay in his doctoral thesis: how to verify in sublinear time whether a Calder mobile is well balanced. Rather than divining one solution or another, we derive a spectrum of solutions, starting from the original specification of the problem. We also prove their correctness using structural induction.

Keywords: Calder mobiles. Continuations. Functional programming. Structural induction.

Contents

*English translation of "Sur un exemple de Patrick Greussay" [1].

†IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: danvy@cs.au.dk

1 Introduction

The original goal of this note was to revisit an example due to Patrick Greussay [7, pages 66-68]: how to determine, in sub-linear time, whether a mobile is balanced.

Prerequisites

We assume from the reader some familiarity with functional programming in general and the ML programming language in particular [8]. In addition, we reason over ML terms equationally to establish their observational equivalence (noted $\cong$).

2 Calder Mobiles

Definition 1 (Calder mobile) *A* mobile *is inductively defined as an* object *of some weight or a* bar *of some weight with two sub-mobiles.*

Definition 2 (representing a mobile in ML) *The following data type specifies the representation of a mobile:*

```
datatype mobile = OBJ of int
                | BAR of int * mobile * mobile
```

For example, let us note `m1` and `m2` the two following mobiles:

```
val m1 = BAR (1,
              BAR (1, OBJ 2, OBJ 2),
              OBJ 5)
val m2 = BAR (1,
              OBJ 6,
              BAR (1, OBJ 2, OBJ 9))
```

Definition 3 (structural induction over mobiles) *Given a predicate M, if*

1. *for any value* `n : int`*, M(*`OBJ n`*) holds, et*
2. *for all values* `n : int`*, and* `m1, m2 : mobile`*,*
   if M(`m1`*) and M(*`m2`*) hold then M(*`BAR (n, m1, m2)`*) holds,*

then for any value `m : mobile`*, M(*`m`*) holds.*

Definition 4 *(weight of a mobile) The weight of a mobile is the sum of the weight of its objects and of its bars.*

For example, the weight of the mobile denoted by `m1` is 11, and the weight of the mobile denoted by `m2` is 19.

Proposition 1 *The following recursive function computes the weight of a mobile:*

```
(*  weight : mobile -> int  *)
fun weight m
    = let (*  visit : mobile -> int  *)
          fun visit (OBJ n)
              = n
            | visit (BAR (n, m1, m2))
              = n + (visit m1) + (visit m2)
      in visit m
      end
```

Proof: By stri

```
datatype 'a option = SOME of 'a
                   | NONE
```

We also use two auxiliary functions to lift a unary function and a binary function:

```
(*  lift1 : ('a -> 'b option)
            -> 'a option -> 'b option  *)
fun lift1 f
    = (fn (SOME v)
          => f v
        | _
          => NONE)

(*  lift2 : ('a * 'b -> 'c option)
            -> 'a option * 'b option -> 'c option  *)
fun lift2 f
    = (fn (SOME v1, SOME v2)
          => f (v1, v2)
        | _
          => NONE)
```

Proposition 3 *The following function computes whether a mobile is balanced:*

```
(*  equil1 : mobile -> bool  *)
fun equil1 m
    = let (*  visit : mobile -> int option  *)
          fun visit (OBJ n)
              = SOME n
            | visit (BAR (n, m1, m2))
              = lift2 (fn (n1, n2) => if n1 = n2
                                      then SOME (n + n1 + n2)
                                      else NONE)
                      (visit m1, visit m2)
      in case visit m
           of (SOME _)
              => true
            | NONE
              => false
      end
```

(Readers uncomfortable with `lift2` *can mentally unfold its call.)*

Proof: By structural induction, using the following predicate:

$$M(\mathtt{m}) \equiv \mathtt{equil1.visit\ m} \cong \begin{cases} \mathtt{SOME\ n} & \text{if the mobile denoted by } \mathtt{m} \text{ is balanced} \\ & \text{and } \mathtt{n} \text{ denotes its weight} \\ \mathtt{NONE} & \text{otherwise} \end{cases}$$

□

This solution is linear since the mobile is traversed once. However, it is completely traversed, even when one of its sub-mobiles (and therefore the entire

mobile) is ill-balanced. We therefore break the symmetry of the solution so that the second sub-mobile of a bar is only traversed if the first is balanced:

Proposition 4 *The following function computes whether a mobile is balanced:*

```
(*  equil2 : mobile -> bool  *)
fun equil2 m
    = let (*  visit : mobile -> int option  *)
          fun visit (OBJ n)
              = SOME n
            | visit (BAR (n, m1, m2))
              = lift1 (fn n1
                          => lift1 (fn n2
                                       => if n1 = n2
                                          then SOME (n + n1 + n2)
                                          else NONE)
                                   (visit m2))
                      (visit m1)
      in case visit m
           of (SOME _)
              => true
            | NONE
              => false
      end
```

Proof: By structural induction, using the same predicate as in Proposition 3. □

This solution is sub-linear because the rest of the mobile is not traversed if one of its sub-mobiles is ill-balanced. However, in this case, the computation yields `NONE` and degenerates into a cascade of returns and of tests verifying whether `NONE` has been returned.

One can use an exception to short-cut this cascade:

```
(*  equil3 : mobile -> bool  *)
fun equil3 m
    = let exception STOP
          (*  visit : mobile -> int  *)
          fun visit (OBJ n)
              = n
            | visit (BAR (n, m1, m2))
              = let val n1 = visit m1
                    val n2 = visit m2
                in if n1 = n2
                   then n + n1 + n2
                   else raise STOP
                end
      in let val _ = visit m
         in true
         end handle STOP => false
      end
```

Instead of `mobile -> int option`, the type of `equi3.visit` is now `mobile -> int`, but this function is no longer pure because of the exception. Does this mean that a purely functional solution is out of reach? The answer is of course negative if one uses an exception monad [9], and doubly so if one passes intermediate results to a continuation, which we proceed to do.

Getting back to `equi2`, one can see that there are three calls to `visit`, each in a context. Let us represent this context as a unary function (the continuation) and let us pass this function to `visit`:

Proposition 5 *The following function computes whether a mobile is balanced:*

```
(*  equi4 : mobile -> bool  *)
fun equi4 m
    = let (*  visit : mobile * (int option -> bool) -> bool  *)
          fun visit (OBJ n, k)
              = k (SOME n)
            | visit (BAR (n, m1, m2), k)
              = visit (m1,
                       fn (SOME n1)
                          => visit (m2,
                                    fn (SOME n2)
                                       => if n1 = n2
                                          then k (SOME (n + n1 + n2))
                                          else k NONE
                                     | _
                                       => k NONE)
                        | _
                          => k NONE)
      in visit (m, fn (SOME _)
                      => true
                    | NONE
                      => false)
  end
```

In `equi2`, the type of `visit` was `mobile -> int option` and the type of `equi2` was `mobile -> bool`. In `equi4`, the type of `visit` is `mobile * (int option -> bool) -> bool` and the type of `equi4` is `mobile -> bool`, i.e., the co-domain of `equi4.visit` and of its continuation is the co-domain of `equi4`.

Proof: We show that for any value `m : mobile`, evaluating `equi2 m` yields a Boolean value `b` if and only if evaluating `equi4 m` yields the same Boolean value `b`. We proceed by structural induction using the following predicate:

$M(\texttt{m}) \equiv$ for all values `k : int option -> bool`
`equi2.visit m` $\cong$ `v` $\Leftrightarrow$ `equi4.visit (m, k)` $\cong$ `k v`
for some `v : int option`

□

Continuation-passing has not solved the problem—once the continuation is applied to `NONE`, the same test cascade occurs. Fortunately, we can use the type isomorphism

```
int option -> bool  ≅  (int -> bool) * (unit -> bool)
```

and split the continuation into two: one is applied if the current sub-mobile is balanced and the other is used otherwise:

```
(*  equil5 : mobile -> bool  *)
fun equil5 m
    = let (*  visit : mobile * (int -> bool) * (unit -> bool)
                    -> bool  *)
          fun visit (OBJ n, ki, ku)
              = ki n
            | visit (BAR (n, m1, m2), ki, ku)
              = visit (m1,
                       fn n1
                          => visit (m2,
                                    fn n2
                                       => if n1 = n2
                                          then ki (n + n1 + n2)
                                          else ku (),
                                    ku),
                       ku)
      in visit (m, fn _ => true, fn () => false)
      end
```

In the recursive call to `visit`, we write `ku` instead of `fn () => ku ()` (i.e., we η-reduce `ku`) to short-cut the cascade of returns in case of ill balance. This definition coincides with Patrick Greussay's solution [7, page 67].

Additionally, instead of schlepping `ku` across the recursive calls to `visit`, we can "lambda-drop" it [6] from its declaration site (the initial call to `visit`) to its use site (the alternative branch of the equality test) and β-reduce `(fn () => false) ()`:

```
(*  equil6 : mobile -> bool  *)
fun equil6 m
    = let (*  visit : mobile * (int -> bool) -> bool  *)
          fun visit (OBJ n, k)
              = k n
            | visit (BAR (n, m1, m2), k)
              = visit (m1,
                       fn n1
                          => visit (m2,
                                    fn n2
                                       => if n1 = n2
                                          then k (n + n1 + n2)
                                          else false))
      in visit (m, fn _ => true)
      end
```

(Alternatively, we could have used the type isomorphism between `unit -> bool` and `bool` and replaced `fn () => false` by `false`, which only works here because `false` is a value.)

We can now write this solution in direct style with `callcc` and `throw` [3] (found in the `SMLofNJ.Cont` library of Standard ML of New Jersey): `callcc` captures the current continuation and `throw` restores a captured continuation.

```
(*  equil7 : mobile -> bool  *)
fun equil7 m
    = callcc (fn k => let (*  visit : mobile -> bool  *)
                          fun visit (OBJ n)
                              = n
                            | visit (BAR (n, m1, m2))
                              = let val n1 = visit m1
                                    val n2 = visit m2
                                in if n1 = n2
                                   then n + n1 + n2
                                   else throw k false
                                end
                      in let val _ = visit m
                         in true
                         end
                      end)
```

The initial continuation of `equil7` is captured; it is only activated, during the computation, if the mobile is ill-balanced.

We can also defunctionalize this solution into a transition system, i.e., an abstract machine or again a pushdown automaton [4, 5, 10, 12]. To this end, we identify that an inhabitant of the function space `int -> bool` is an instance of three lambda-abstractions in the definition of `equil6` (this identification is the result of a control-flow analysis). We thus represent this function space by a sum (i.e., in ML, by a data type `cont`), each lambda-abstraction by the corresponding constructor in `cont`, and each application by a call to a function `apply_cont` that interprets the constructor:

Proposition 6 *The following function computes whether a mobile is balanced:*

```
(*  equil8 : mobile -> bool  *)
fun equil8 m
    = let datatype cont = C0
                        | C1 of int * mobile * cont
                        | C2 of int * int * cont
          (*  apply_cont : cont * int -> bool  *)
          fun apply_cont (C0, _)
              = true
            | apply_cont (C1 (n, m2, c), n1)
              = visit (m2, C2 (n, n1, c))
```

```
      | apply_cont (C2 (n, n1, c), n2)
        = if n1 = n2
          then apply_cont (c, n + n1 + n2)
          else false
    (*  visit : mobile * cont -> bool  *)
    and visit (OBJ n, c)
        = apply_cont (c, n)
      | visit (BAR (n, m1, m2), c)
        = visit (m1, C1 (n, m2, c))
in visit (m, C0)
end
```

Proof: We show that for any value `m : mobile`, evaluating `equi16 m` yields a Boolean value `b` if and only if evaluating `equi18 m` yields the same Boolean value `b`. We proceed by structural induction using the logical relation

$$K(\mathtt{k},\mathtt{c}) \equiv \text{for any value } \mathtt{n : int}\quad \mathtt{k\ n} \cong \mathtt{b} \ \Leftrightarrow\ \mathtt{equi18.apply\_cont\ (c,\ n)} \cong \mathtt{b} \quad \text{for some value } \mathtt{b : bool}$$

and the following predicate:

$$M(\mathtt{m}) \equiv \text{for all the values } \mathtt{k : int -> bool} \text{ and } \mathtt{c : cont} \text{ satisfying } K(\mathtt{k},\mathtt{c})$$
$$\mathtt{equi16.visit\ (m,\ k)} \cong \mathtt{k\ n} \ \Leftrightarrow\ \mathtt{equi18.visit\ (m,\ c)} \cong \mathtt{equi18.apply\_cont\ (c,\ n)} \quad \text{for some value } \mathtt{n : int}$$
$$\text{or}$$
$$\mathtt{equi16.visit\ (m,\ k)} \cong \mathtt{false} \ \Leftrightarrow\ \mathtt{equi18.visit\ (m,\ c)} \cong \mathtt{false}$$

Here is a representative case of the proof, to show that for all the values `n : int`, `m1 : mobile`, `m2 : mobile`, `k : int -> bool` and `c : cont` satisfying $M(\mathtt{m1})$, $M(\mathtt{m2})$ and $K(\mathtt{k},\mathtt{c})$,

$$\mathtt{equi16.visit\ (BAR\ (n,\ m1,\ m2),\ k)} \cong \mathtt{k\ n'} \ \Rightarrow$$
$$\mathtt{equi18.visit\ (BAR\ (n,\ m1,\ m2),\ c)} \cong \mathtt{equi18.apply\_cont\ (c,\ n')} \quad \text{for some value } \mathtt{n' : int}$$
$$\text{or}$$
$$\mathtt{equi16.visit\ (BAR\ (n,\ m1,\ m2),\ k)} \cong \mathtt{false}$$
$$\Rightarrow$$
$$\mathtt{equi18.visit\ (BAR\ (n,\ m1,\ m2),\ c)} \cong \mathtt{false}$$

By definition, `equi16.visit (BAR (n, m1, m2), k)` $\cong$ `equi16.visit (m1, fn n1 => ...)`. Since $K(\mathtt{k},\mathtt{c})$ holds, we simply verify that for any value `n1 : int`, the relation

$$K(\mathtt{fn\ n2 => if\ n1 = n2\ then\ k\ (n + n1 + n2)\ else\ false},\ \mathtt{C2\ (n,\ n1,\ c)})$$

also holds. By induction hypothesis on `m2`, for any value `n1 : int`,

`equil6.visit (m2, fn n2 => ...)` $\cong$ `(fn n2 => ...)  n2`
$\Rightarrow$
`equil8.visit (m2, C2 (n, n1, c))` $\cong$ `equil8.apply_cont (C2 (n, n1, c), n2)`
for some value `n2 : int`
or
`equil6.visit (m2, fn n2 => ...)` $\cong$ `false`
$\Rightarrow$
`equil8.visit (m2, C2 (n, n1, c))` $\cong$ `false`

We can thus verify that the relation

$$K(\texttt{fn n1 => equil6.visit (m2, fn n2 => ...)}, \texttt{C1 (n, m2, c)})$$

holds, which puts us in position to apply the induction hypothesis on `m1`. □

Alternatively to the data type `cont`, we can defunctionalize the continuation into a stack of frames and interpret it with a popping function:

```
(*  equil9 : mobile -> bool  *)
fun equil9 m
    = let datatype frame = F1 of int * mobile
                         | F2 of int * int
          type cont = frame list
          (*  pop_frame : cont * int -> bool  *)
          fun pop_frame (nil, _)
              = true
            | pop_frame ((F1 (n, m2)) :: c, n1)
              = visit (m2, (F2 (n, n1)) :: c)
            | pop_frame ((F2 (n, n1)) :: c, n2)
              = if n1 = n2
                then pop_frame (c, n + n1 + n2)
                else false
          (*  visit : mobile * cont -> bool  *)
          and visit (OBJ n, c)
              = pop_frame (c, n)
            | visit (BAR (n, m1, m2), c)
              = visit (m1, (F1 (n, m2)) :: c)
      in visit (m, nil)
      end
```

4 Analysis

The reader is now equipped to tackle the traditional multiplication example of integers in a tree, exploiting the absorption property of 0 and deriving a spectrum of solutions[1] rather than heroically inventing one of these solutions.

[1] Namely with a local function whose co-domain is `int option`, or that uses an exception, or a continuation, or a stack of sub-trees.

More generally, the derivation presented here illustrates a class of applications of continuations (and of exceptions) for functions of type $t_1 \to t_2$ that use an auxiliary function of type $t_3 \to t_4 + t_5$. Often, one can:

1. CPS-transform the auxiliary function [2], giving it the following type:

$$t_3 \times (t_4 + t_5 \to t_2) \to t_2$$

2. split the continuation into two:

$$t_3 \times (t_4 \to t_2) \times (t_5 \to t_2) \to t_2$$

   and

3. simplify one of the two continuations and its use (e.g., when t_5 is the unit type, as for Calder mobiles, for multiplying integers in the leaves of a tree, or for programming an substitution algorithm that preserves sharing).

References

[1] Olivier Danvy. Sur un exemple de Patrick Greussay. Research Report BRICS RS-04-41, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, December 2004.

[2] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[3] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

[4] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 2009. In press. Extended version available as the research report BRICS RS-08-04.

[5] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the research report BRICS RS-01-23.

[6] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000. A preliminary version was presented at the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1997).

[7] Patrick Greussay. *Contribution à la définition interprétative et à l'implémentation des λ-langages.* Thèse d'état, Université de Paris VII, Paris, France, 1977.

[8] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[9] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[10] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, September 1981. Reprinted in the Journal of Logic and Algebraic Programming 60-61:17-139, 2004, with a foreword [11].

[11] Gordon D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

[12] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword [13].

[13] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.