Basic Research in Computer Science

Sur un Exemple de Patrick Greussay

Olivier Danvy

BRICS Report Series **RS-04-41**

ISSN 0909-0878 **December 2004**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/04/41/`

Sur un exemple de Patrick Greussay

Olivier Danvy
BRICS*
Department of Computer Science
University of Aarhus†

le 31 décembre 2004

Résumé

Cette note a été écrite à l'occasion du départ en retraite de Jean-François Perrot à l'Université Pierre et Marie Curie (Paris VI). En tentant d'émuler son esprit pédagogique, nous revisitons un exemple de la thèse de Patrick Greussay sur les mobiles de Calder. Plutôt que d'en pressentir une solution ou une autre — certaines d'entre elles utilisent des continuations — nous en dérivons un éventail à partir des données du problème. Nous présentons aussi leur preuve.

Abstract

This note was written at the occasion of the retirement of 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.

Keywords: Calder mobiles. Functional programming. Continuations.

*Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.
†IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark. Email: `danvy@brics.dk`

Table des matières

[1] Ce texte est apparu dans les actes du colloque scientifique sur les objets en l'honneur de Jean-François Perrot, en octobre 2003 à Paris.

1 Introduction

Une continuation représente "le reste du calcul" [1, 13]. Depuis trente ans [11], les continuations ont été découvertes dans pratiquement tous les domaines de l'informatique, des plus abstraits — logique intuitionniste, mathématiques constructives, théorie des types, théorie des langages de programmation — aux plus concrets — implantation de langages de programmation, services dans les systèmes d'exploitation, processus et autres 'threads', interfaces graphiques, programmation du Web, calculs distribués ou nomades. Les continuations forment aussi des outils précieux pour programmer et pour analyser ou transformer des programmes. En général, elles offrent un vocabulaire uniforme pour énoncer des solutions à des problèmes de contrôle, pour raisonner sur ces solutions, et pour les implanter efficacement. Pourtant les continuations tendent à être perçues comme des objets exotiques, difficiles à gérer en C, C++, C# ou Java d'une manière native.

En résumé, pour paraphraser Jules Renard, les continuations mènent à tout, à condition d'y entrer. Hélas il semble que la simple mention de leur nom dissuade les meilleures volontés. Le forum européen ETAPS, par exemple, est le théâtre régulier d'exposés sur les continuations. L'intérêt de l'audience est frappant; est frappante tout autant sa perplexité. Invariablement, les questions, à la fin des exposés comme dans les couloirs et jusqu'à tard le soir, sont plus élémentaires que le contenu des exposés, indiquant une incompréhension qui perdure. Pourtant lorsque Jean-François Perrot enseignait les continuations à ses élèves de licence, le message passait. Dans l'esprit de cet enseignement, nous revisitons un exemple dû à Patrick Greussay [6, pages 66-68]: comment vérifier, en temps sous-linéaire, si un mobile est équilibré.

Postulat

Nous supposons du lecteur un peu de familiarité avec la programmation fonctionnelle en général et le langage ML en particulier [7]. En outre, nous raisonnons équationnellement sur des termes ML pour établir leur équivalence observationnelle (notée $\cong$).

2 Mobiles de Calder

Définition 1 (mobile de Calder) *Un mobile est défini inductivement comme un objet d'un certain poids ou une barre d'un certain poids avec deux sous-mobiles.*

Définition 2 (représentation d'un mobile en ML) *Le type de données suivant spécifie la représentation d'un mobile:*

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

Par exemple, notons `m1` and `m2` les deux mobiles suivants:

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

Définition 3 (induction structurale sur les mobiles) *Etant donné le prédicat M, si*

1. *pour toute valeur* `n : int`, M(`OBJ n`) *est satisfait, et*
2. *pour toutes valeurs* `n : int`, *et* `m1, m2 : mobile`, *si M*(`m1`) *et M*(`m2`) *sont satisfaits alors M*(`BAR (n, m1, m2)`) *est satisfait,*

alors pour toute valeur `m : mobile`, M(`m`) *est satisfait.*

Définition 4 *(poids d'un mobile) Le poids d'un mobile est la somme du poids de ses objets et de ses barres.*

Par exemple, le poids du mobile dénoté par `m1` est 11, et le poids du mobile dénoté par `m2` est 19.

Proposition 1 *La fonction suivante calcule récursivement le poids d'un 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
```

Preuve: Par induction structurale, en utilisant le prédicat suivant:

M(`m`) $\equiv$ `weight.visit m` $\cong$ `n`,
où `n` dénote le poids du mobile dénoté par `m`

□

Par exemple, évaluer `weight m1` donne `11`, et évaluer `weight m2` donne `19`.

3 Equilibre

Définition 5 *Un mobile qui consiste en un objet est équilibré. Un mobile qui consiste en une barre est équilibré si et seulement si ses deux sous-mobiles ont le même poids et sont équilibrés.*

Proposition 2 *La fonction suivante détermine si un mobile est équilibré:*

```
(*  equil0 : mobile -> bool  *)
fun equil0 m
    = let (*  visit : mobile -> bool  *)
          fun visit (OBJ n)
              = true
            | visit (BAR (n, m1, m2))
              = (weight m1) = (weight m2) andalso visit m1
                                          andalso visit m2
      in visit m
      end
```

Preuve: Par induction structurale, en utilisant le prédicat suivant:

$$M(\texttt{m}) \equiv \texttt{equil0.visit m} \cong \begin{cases} \texttt{true} & \text{si le mobile dénoté par m est équilibré} \\ \texttt{false} & \text{sinon} \end{cases}$$

□

Par exemple, évaluer `equil0 m1` donne `true`, et évaluer `equil0 m2` donne `false`.

Cette solution est inefficace à cause de ses traversées répétées du mobile. Pour réduire ces traversées à une unique descente récursive (eureka), `equil0.visit` devrait retourner non pas seulement un booléen, mais, si ce booléen est vrai, alors également le poids du mobile. A cette fin, nous utilisons un type de données optionnel:

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

(*  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 *La fonction suivante détermine si un mobile est équilibré:*

```
(*  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
```

(Le lecteur que `lift2` *fait renâcler peut mentalement déplier son appel.)*

Preuve: Par induction structurale, en utilisant le prédicat suivant:

$$M(\texttt{m}) \equiv \texttt{equil1.visit m} \cong \begin{cases} \texttt{SOME n} & \text{si le mobile dénoté par } \texttt{m} \text{ est équilibré} \\ & \text{et } \texttt{n} \text{ dénote son poids} \\ \texttt{NONE} & \text{sinon} \end{cases}$$

□

Cette solution est linéaire, puisque le mobile est traversé une seule fois. Toutefois, il est complètement traversé, même lorsque l'un des sous-mobiles (et donc le mobile tout entier) n'est pas équilibré. Désymétrisons donc la solution pour que le second sous-mobile ne soit traversé que si le premier est équilibré:

Proposition 4 *La fonction suivante détermine si un mobile est équilibré:*

```
(*  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
```

Preuve: Par induction structurale, en utilisant le même prédicat que pour la proposition 3. □

Cette solution est sous-linéaire parce que le reste du mobile n'est pas traversé si l'un des sous-mobiles n'est pas équilibré. Toutefois, dans ce cas, le calcul dégénère en une cascade de retours de `NONE` et de tests vérifiant si `NONE` a été retourné.

On peut utiliser une exception pour court-circuiter cette 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
```

Au lieu de `mobile -> int option`, le type de `equil3.visit` est maintenant `mobile -> int`, mais cette fonction n'est plus pure à cause de l'exception. Est-ce à dire qu'une solution purement fonctionnelle est hors de portée? La réponse est bien sûr négative si l'on utilise une monade d'exceptions [8] ou alors si l'on passe les résultats intermédiaires à une continuation, ce que nous faisons ci-après.

Revenant à `equil2`, on peut voir qu'il y a trois appels à `visit`, chacun dans un contexte. Représentons ce contexte comme une fonction unaire (la continuation) et passons cette fonction à `visit`:

Proposition 5 *La fonction suivante détermine si un mobile est équilibré:*

```
(*  equil4 : mobile -> bool  *)
fun equil4 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
```

Dans `equil2`, le type de `visit` était `mobile -> int option` et le type de `equil2` était `mobile -> bool`. Dans `equil4`, le type de `visit` est `mobile * (int option -> bool) -> bool` et le type de `equil4` est `mobile -> bool`, *i.e.*, le co-domaine de `equil4.visit` et de sa continuation est le co-domaine de `equil4`.

Preuve: On montre que pour toute valeur `m : mobile`, évaluer `equil2 m` donne une valeur booléenne `b` si et seulement si évaluer `equil4 m` donne la même valeur booléenne `b`. On procède par induction structurale en utilisant le prédicat suivant:

$M(\texttt{m}) \equiv$ pour toute valeur `k : int option -> bool`
`equil2.visit m` $\cong$ `v` $\Leftrightarrow$ `equil4.visit (m, k)` $\cong$ `k v`
pour une valeur `v : int option`

□

Le passage à la continuation n'a pas résolu le problème — une fois la continuation appliquée à `NONE`, la même cascade de tests se produit. Toutefois nous pouvons utiliser l'isomorphisme de types

`int option -> bool` $\cong$ `(int -> bool) * (unit -> bool)`

et diviser la continuation en deux: l'une est appliquée si le sous-mobile courant est équilibré et l'autre s'il ne l'est pas:

```
(*  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
```

Dans l'appel récursif à `visit`, on écrit `ku` plutôt que `fn () => ku ()` (*i.e.*, on η-réduit `ku`) pour court-circuiter la cascade de retours en cas de déséquilibre. Cette définition coïncide avec la solution de Patrick Greussay [6, page 67].

Additionnellement, au lieu de véhiculer `ku` au travers des appels récursifs à `visit`, on peut le "lambda-dropper" [5] de son point de déclaration (l'appel initial à `visit`) à son point d'utilisation (la branche alternative du test d'égalité) et β-réduire `(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
```

On peut maintenant écrire cette solution en style direct avec `callcc` et `throw` [3] (trouvés dans la librairie `SMLofNJ.Cont` de Standard ML of New Jersey): `callcc` capture la continuation courante et `throw` restaure une continuation capturée.

```
(*  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)
```

La continuation initiale de `equil7` est capturée; elle n'est activée, au cours du calcul, que si le mobile est déséquilibré.

On peut aussi défonctionnaliser cette solution en un système de transitions, *i.e.*, une machine abstraite ou encore un automate d'états fini à pile [4, 9, 12]. A cette fin, on identifie qu'un habitant de l'espace fonctionnel `int -> bool` est une instance d'une des trois lambda-abstractions dans la définition de `equil6` (cette identification est le résultat d'une analyse de flot de contrôle). On représente

donc cet espace fonctionnel par une somme (*i.e.*, en ML, par un type de données `cont`), chaque lambda-abstraction par le constructeur correspondant dans `cont`, et chaque application par un appel à une fonction `apply_cont` qui interprète le constructeur:

Proposition 6 *La fonction suivante détermine si un mobile est équilibré:*

```
(*  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
```

Preuve: On montre que pour toute valeur `m : mobile`, évaluer `equil6 m` donne une valeur booléenne `b` si et seulement si évaluer `equil8 m` donne la même valeur booléenne `b`. On procède par induction structurale en utilisant la relation logique

$$K(\mathtt{k},\mathtt{c}) \equiv \text{pour toute valeur } \mathtt{n : int}\quad \mathtt{k\ n} \cong \mathtt{b} \ \Leftrightarrow\ \mathtt{equil8.apply\_cont\ (c,\ n)} \cong \mathtt{b} \quad \text{pour une valeur } \mathtt{b : bool}$$

et le prédicat suivant:

$$M(\mathtt{m}) \equiv \text{pour toutes les valeurs } \mathtt{k : int -> bool} \text{ et } \mathtt{c : cont} \text{ satisfaisant } K(\mathtt{k},\mathtt{c})$$
$$\mathtt{equil6.visit\ (m,\ k)} \cong \mathtt{k\ n} \ \Leftrightarrow$$
$$\mathtt{equil8.visit\ (m,\ c)} \cong \mathtt{equil8.apply\_cont\ (c,\ n)}$$
pour une valeur `n : int`
ou
$$\mathtt{equil6.visit\ (m,\ k)} \cong \mathtt{false} \ \Leftrightarrow\ \mathtt{equil8.visit\ (m,\ c)} \cong \mathtt{false}$$

Voici un cas représentatif de la preuve: montrer que pour toutes les valeurs `n : int`, `m1 : mobile`, `m2 : mobile`, `k : int -> bool` et `c : cont` satisfaisant $M(\mathtt{m1})$, $M(\mathtt{m2})$ et $K(\mathtt{k},\mathtt{c})$,

`equil6.visit (BAR (n, m1, m2), k)` $\cong$ `k n'` $\Rightarrow$
`equil8.visit (BAR (n, m1, m2), c)` $\cong$ `equil8.apply_cont (c, n')`
pour une valeur `n' : int`
ou
`equil6.visit (BAR (n, m1, m2), k)` $\cong$ `false`
$\Rightarrow$
`equil8.visit (BAR (n, m1, m2), c)` $\cong$ `false`

Par définition, `equil6.visit (BAR (n, m1, m2), k)` $\cong$ `equil6.visit (m1, fn n1 => ...)`. Puisque $K($`k`, `c`$)$ est satisfaite, on vérifie simplement que pour toute valeur `n1 : int`, la relation

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

est aussi satisfaite. Par hypothèse d'induction sur `m2`, pour toute valeur `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)`
pour une valeur `n2 : int`
ou
`equil6.visit (m2, fn n2 => ...)` $\cong$ `false`
$\Rightarrow$
`equil8.visit (m2, C2 (n, n1, c))` $\cong$ `false`

On peut donc vérifier que la relation

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

est satisfaite, ce qui permet d'appliquer l'hypothèse d'induction sur `m1`. □

Alternativement au type de données `cont`, on peut défonctionnaliser la continuation en une pile d'articles et l'interpréter avec une fonction de désempilement:

```
(*  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 Analyse

Le lecteur est maintenant à même de s'attaquer à l'exemple traditionnel de la multiplication des entiers dans un arbre en exploitant l'absorbance de 0 pour la multiplication, et pour en dériver un éventail de solutions[2] plutôt que d'inventer une de ces solutions.

Plus généralement, la dérivation présentée ici illustre une classe d'application des continuations (et des exceptions) pour des fonctions de type $t_1 \rightarrow t_2$ qui utilisent une fonction auxiliaire de type $t_3 \rightarrow t_4 + t_5$. Souvent, on peut:

1. CPS-transformer la fonction auxiliaire [2], lui donnant le type suivant:

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

2. diviser la continuation en deux:

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

   et

3. simplifier l'une des deux continuations et son usage (par exemple lorsque t_5 est le type unité, comme pour les mobiles de Calder, pour multiplier les entiers d'un arbre, ou pour programmer un algorithme de substitution avec partage).

5 Conclusion

Le plus souvent, les continuations restent mystérieuses. Elles tendent à provoquer des réactions passionnées, mais la plupart du temps on ne les comprend pas. John von Neumann aurait probablement dit qu'à défaut de les comprendre, on peut au moins se familiariser avec elles.[3] Jean-François Perrot, lui, a fait tomber une génération d'étudiants tout de go dans la marmite des continuations. Je faisais partie de cette génération, et lui en suis encore reconnaissant.

[2] C'est à dire avec une fonction locale dont le co-domaine est `int option`, ou qui utilise une exception, ou une continuation, ou une pile de sous-arbres.

[3] "In mathematics one does not understand things. One just gets used to them."

Remerciements: A Jean-François Perrot et à Patrick Greussay, bien évidemment.

Je voudrais aussi remercier Jean-Pierre Briot pour avoir organisé le colloque et pour m'y avoir généreusement invité, malgré cette note hors sujet.

La version française de cette note a bénéficié de la relecture et du bon sens de Sandrine Chirokoff, Karoline Malmkjær et Florence Vasselin. Merci à elles enfin.

A Les continuations, vingt ans après

Ce texte est apparu dans les actes du colloque scientifique sur les objets en l'honneur de Jean-François Perrot, en octobre 2003 à Paris.

Printemps 1983. Jean-François Perrot (JFP, de son nom de login) donne un cours lumineux sur l'interprétation des langages applicatifs. Sa présentation de l'informatique est tout à la fois synthétique, cultivée, sensée, vivante et tout autant exigeante, à l'image — je l'ai réalisé depuis — de son enseignant. Suivre ce cours, c'est effectuer le même saut quantique que de passer du lycée à l'Université ou d'apprendre à conduire. Le but ne se réduit pas à passer l'examen au gré d'une houlette. On vous met les élements en main [10] et vous en prenez non seulement le contrôle mais aussi la direction. Le cours de JFP captive, inspire et donne presque le vertige: programmation fonctionnelle et par continuations,[4] programmes comme données, diagonalisation, et écriture de ses propres outils — interprète, générateur de code, machine virtuelle ou gérant de mémoire. Une fois là, c'est en avant vers toutes les directions: la programmation, les langages de programmation, leur sémantique et leur logique, leur implémentation, la représentation des programmes et de leurs processeurs, etc.

Et les objets. Ah, les objets. Ont-ils réussi, raté, dérivé, divergé ou même pourquoi pas trahi? Qu'importe au fond. Ce qui compte aujourd'hui, c'est le nombre de gens qui ont une opinion informée sur le sujet. Et à cet égard, JFP lui n'a pas raté au regard de l'influence qu'il a eu sur tous les gens qui ont eu la chance d'avoir bénéficié de son enseignement.

Aarhus, Danemark, mai 2003

References

[1] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk.

[4] Je ne sais pas si je me suis jamais remis du cours de JFP sur les continuations en mai 1983 et de la rencontre subséquente d'Emmanuel Saint-James qui les comprenait totalement. Ou alors, plus simplement, peut-être m'y suis-je tout bêtement mis.

[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 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 technical report BRICS RS-01-23.

[5] 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.

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

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

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

[9] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[10] Christian Queinnec. *Language d'un autre type: LISP*. Eyrolles, Paris, France, 1982.

[11] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[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] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000. Reprint of the technical monograph PRG-11, Oxford University Computing Laboratory (1974), with a foreword.

Recent BRICS Report Series Publications

RS-04-41 Olivier Danvy. *Sur un Exemple de Patrick Greussay*. December 2004. 14 pp.

RS-04-40 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *Fast Partial Evaluation of Pattern Matching in Strings*. December 2005. 22 pp. To appear in TOPLAS. Supersedes BRICS report RS-03-20.

RS-04-39 Olivier Danvy and Lasse R. Nielsen. *CPS Transformation of Beta-Redexes*. December 2004. ii+11 pp. Superseedes an article to appear in *Information Processing Letters* and BRICS report RS-00-35.

RS-04-38 Olin Shivers and Mitchell Wand. *Bottom-Up β-Substitution: Uplinks and λ-DAGs*. December 2004.

RS-04-37 Jørgen Iversen and Peter D. Mosses. *Constructive Action Semantics for Core ML*. December 2004. 68 pp. To appear in a special *Language Definitions and Tool Generation* issue of the journal *IEE Proceedings Software*.

RS-04-36 Mark van den Brand, Jørgen Iversen, and Peter D. Mosses. *An Action Environment*. December 2004. 27 pp. Appears in Hedin and Van Wyk, editors, *Fourth ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications*, LDTA '04, 2004, pages 149–168.

RS-04-35 Jørgen Iversen. *Type Checking Semantic Functions in ASDF*. December 2004.

RS-04-34 Anders Møller and Michael I. Schwartzbach. *The Design Space of Type Checkers for XML Transformation Languages*. December 2004. 21 pp. Appears in Eiter and Libkin, editors, *Database Theory: 10th International Conference*, ICDT '05 Proceedings, LNCS 3363, 2005, pages 17–36.

RS-04-33 Aske Simon Christensen, Christian Kirkegaard, and Anders Møller. *A Runtime System for XML Transformations in Java*. December 2004. 15 pp. Appears in Bellahsene, Milo, Rys, Suciu and Unland, editors, *Database and XML Technologies: Second International XML Database Symposium*, XSym '04 Proceedings, LNCS 3186, 2004, pages 143–157. Supersedes the earlier BRICS report RS-03-29.

RS-04-32 Philipp Gerhardy. *A Quantitative Version of Kirk's Fixed Point Theorem for Asymptotic Contractions*. December 2004. 9 pp.