Basic Research in Computer Science

An Extensional Characterization of Lambda-Lifting and Lambda-Dropping

Olivier Danvy

BRICS Report Series **RS-99-21**

ISSN 0909-0878 **August 1999**

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/99/21/`

An Extensional Characterization of Lambda-Lifting and Lambda-Dropping *

Olivier Danvy

BRICS †
Department of Computer Science
University of Aarhus ‡

August 25, 1999

Abstract

Lambda-lifting and lambda-dropping respectively transform a block-structured functional program into recursive equations and vice versa. Lambda-lifting was developed in the early 80's, whereas lambda-dropping is more recent. Both are split into an analysis and a transformation. Published work, however, has only concentrated on the analysis parts. We focus here on the transformation parts and more precisely on their correctness, which appears never to have been proven. To this end, we define extensional versions of lambda-lifting and lambda-dropping and establish their correctness with respect to a least fixed-point semantics.

*Extended version of an article to be presented at FLOPS'99 (`http://www.score.is.tsukuba.ac.jp/flops99`)

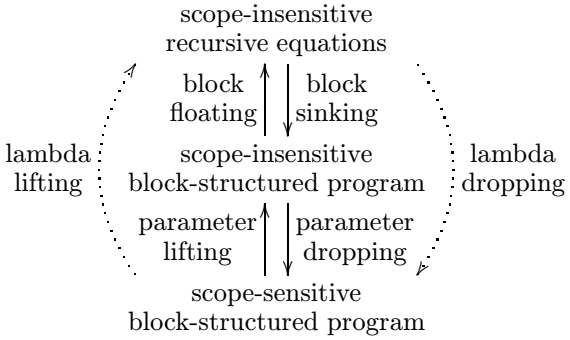
†Basic Research in Computer Science (`http://www.brics.dk`), Centre of the Danish National Research Foundation.

‡Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark.
Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55. E-mail: `danvy@brics.dk`
Home page: `http://www.brics.dk/~danvy`

1 Introduction and motivation

If procedural programming languages are out of the Turing tar pit today, it is largely due to the expressive power induced by block structure and lexical scope. However block structure and lexical scope are not essential: in the mid 80's, Hughes, Johnsson, and Peyton Jones showed how to *lambda-lift* any block-structured program into recursive equations, which can then be efficiently compiled on the G-machine [5, 6, 9]. Since then, lambda-lifting has had its ups and downs: it is used for example in at least one compiler and two partial evaluators for the Scheme programming language [1, 2, 3]; in an unpublished note, "Down with Lambda-Lifting" [7], Meijer severely criticizes it; and it is no longer systematically used to compile Haskell programs today [8]. In all cases, lambda-lifting is considered as an intermediate transformation in a compiler or a partial evaluator.

Our own stab at lambda-lifting is linguistic. We are interested in programming and in the expressive power induced by block structure and lexical scope, which lambda-lifting eliminates. This led us to devise an inverse transformation to *lambda-drop* recursive equations into a block-structured, lexically scoped program. Lambda-dropping was reported at PEPM'97 jointly with Schultz and implemented as the back-end of a partial evaluator [4, 10]. In that joint work, we tried to emphasize the symmetric aspects of lambda-lifting and lambda-dropping:



Let us start from a block-structured program. A priori, this program contains free variables and is thus scope-sensitive. To make it scope-insensitive, we pass extra parameters to each of its locally defined functions. These extra parameters account for the free variables of each function. Once the program is scope-insensitive, we globalize each block by making it float to the top level and defining each of its locally defined functions as a global recursive equation.[1] Conversely, lambda-dropping requires us to group recursive equations into blocks and make these blocks sink in the corresponding recursive equation, following

[1] Incidentally, we favor Johnsson's style of lambda-lifting [6], where recursive equations are named and names of recursive equations are free in their bodies. This makes it possible to see them as mutually recursive top-level functions in a functional programming language.

```
fun power_l (base, 0) = 1
  | power_l (base, expo) = base * (power_l (base, expo - 1))
(* power_l : int * int -> int *)

fun power_d (base, expo)
    = let fun loop 0 = 1
            | loop expo = base * (loop (expo - 1))
      in loop expo
      end (* loop : int -> int *)
(* power_d : int * int -> int *)
```

Figure 1: λ-lifted and λ-dropped versions of the power function

the edges of the source call graph. The resulting program is block-structured but scope insensitive, except for the names of the (ex-)recursive equations, of course. We make it scope sensitive by preventing each function from passing variables whose end use is lexically visible.

A simple example: Figure 1 displays the power function in ML. Many other examples exist [2, 4, 5, 6, 9, 10] but this one is simple and illustrative enough. One of its parameters is "inert," i.e., it does not change through the recursive calls. The lambda-lifted version carries the inert argument through each recursive call. The lambda-dropped version does not, making it instead a free variable in the actual traversal of the other argument. Lambda-lifting and lambda-dropping transform one definition into the other and vice-versa.

The anatomy of lambda-lifting and lambda-dropping: Both naturally split into an *analysis* and a *transformation*. Leaving aside block-sinking, which is specific to lambda-dropping, the analysis determines which parameters can be lifted and which parameters can be dropped.

It is however a fact that most of the work in lambda-lifting and lambda-dropping has concentrated on its analysis and neglected its transformation. Both at PEPM'97 and at a meeting of the IFIP Working Group on Functional Programming in June 1997, we ventured that the correctness of lambda-lifting was still an open problem and there was a general agreement that it was so.

Our goal: We address parameter lifting and dropping and their formal correctness. More precisely, we want to know whether a lambda-lifted program and the corresponding lambda-dropped program compute the same function.

Our means: We consider the meanings of the lambda-lifted and the lambda-dropped programs in a least-fixed point semantics [11]. Using fixed-point induction, we prove that the lambda-lifted version and the lambda-dropped version of `power` compute the same function.

Our point: We generalize this example and introduce *extensional versions* of lambda-lifting and lambda-dropping, i.e.,

Extensional lambda-lifting: a type-indexed mapping from the functional associated to a lambda-dropped function to the functional associated to the corresponding lambda-lifted function; and

Extensional lambda-dropping: a type-indexed mapping from the functional associated to a lambda-lifted function to the functional associated to the corresponding lambda-dropped function.

Overview: In Section 2, we present two extensional, type-indexed transformations respectively lambda-lifting and lambda-dropping a functional, and we show that the input and the output functionals share the same fixed point. These extensional transformations assume that one knows which parameter(s) to lift or to drop. In Section 3, we scale up extensional lambda-lifting and lambda-dropping to mutually recursive functions. Section 4 concludes.

In the appendix, we instantiate both a lambda-dropped and a lambda-lifted version of the append function out of the same fold function for lists. We had observed that instantiating (lambda-dropped) fold functionals naturally yields lambda-dropped functions, and it came as a surprise to us that it could also yield lambda-lifted functions as well.

2 Extensional lambda-lifting and lambda-dropping

Lambda-lifting and lambda-dropping are defined intensionally, i.e., they are textual transformations. Could we define their extensional counterparts, that we could apply to the corresponding meanings instead of to their text? We answer positively to this question by exhibiting two mappings between the functional corresponding to the lambda-dropped version of a function and the functional corresponding to its lambda-lifted counterpart.

2.1 Extensional lambda-lifting

Let us define an extensional lambda-lifter for unary functionals abstracted by a dropped parameter. The lambda-lifter lifts the abstracted parameter in first position in the resulting uncurried binary functional.

Definition 1 *Let A, B, and X denote pointed CPOs.*

$$\begin{array}{l}\mathrm{lift}_1 : (X \to (A \to B) \to A \to B) \to (X \times A \to B) \to X \times A \to B\\ \mathrm{lift}_1\ F\ f\ \langle x, a\rangle = F\ x\ (\lambda a'.f\ \langle x, a'\rangle)\ a\end{array}$$

This extensional version makes it possible to lambda-lift a functional prior to taking its fixed point.

Theorem 1 (lambda-lifting) *Let A, B, and X denote pointed CPOs and let $F \in X \to (A \to B) \to A \to B$. Then for any $x \in X$ and $a \in A$,*

$$fix_{A \to B}\,(F\ x)\ a = fix_{X \times A \to B}\,(\mathrm{lift}_1\ F)\ \langle x,\ a\rangle.$$

Proof: By fixed-point induction. Let us define R_x as an X-indexed family of admissible relations between $A \to B$ and $X \times A \to B$:

$$R_x = \{(d,\ \ell) \mid \forall a \in A. d\ a = \ell\ \langle x,\ a\rangle\}$$

Each R_x is pointed (contains $(\bot_{A \to B},\ \bot_{X \times A \to B})$) and admissible (it is defined as an intersection of inverse images by (continuous) application functions of the admissible equality relation). Now $\forall x \in X$ and $\forall (d,\ \ell) \in R_x$,

$$(F\ x\ d,\ \mathrm{lift}_1\ F\ \ell) \in R_x$$

$$\begin{array}{lrcl} \text{since } \forall a \in A,\ \mathrm{lift}_1\ F\ \ell\ \langle x,\ a\rangle & = & F\ x\ (\lambda a'.\ell\ \langle x,\ a'\rangle)\ a \\ & = & F\ x\ (\lambda a'.d\ a')\ a \\ & = & F\ x\ d\ a \end{array}$$

Therefore, by fixed-point induction, the least fixed points of the two functions are also related, i.e.,

$$(\mathrm{fix}_{A \to B}\ F\ x,\ \mathrm{fix}_{X \times A \to B}\ (\mathrm{lift}_1\ F)) \in R_x$$

which, expanding the definition of R_x, is precisely what we wanted to prove. □

Similarly, we can define an extensional lambda-lifter that lifts the abstracted parameter in second position in the resulting uncurried binary functional.

Definition 2 *Let A, B, and X denote pointed CPOs.*

$$\begin{array}{l} \mathrm{lift}_2 : (X \to (A \to B) \to A \to B) \to (A \times X \to B) \to A \times X \to B \\ \mathrm{lift}_2\ F\ f\ \langle a,\ x\rangle = F\ x\ (\lambda a'.f\ \langle a',\ x\rangle)\ a \end{array}$$

2.2 Extensional lambda-dropping

We now turn to defining an extensional lambda-dropper for uncurried binary functionals. The lambda-dropper drops the first parameter of the functional, assuming it to be inert.

Definition 3 *$F : (A \times X \to B) \to A \times X \to B$ is* inert in *X if and only if $\forall f \in A \times X \to B$, $\forall x \in X$, and $\forall a \in A$,*

$$F\ (\lambda\langle x',\ a'\rangle.f\ \langle x,\ a'\rangle)\ \langle x,\ a\rangle = F\ (\lambda\langle x',\ a'\rangle.f\ \langle x',\ a'\rangle)\ \langle x,\ a\rangle$$

Definition 4 *Let A, B, and X denote pointed CPOs.*

$$\begin{array}{l} \mathrm{drop}_1 : ((X \times A \to B) \to X \times A \to B) \to X \to (A \to B) \to A \to B \\ \mathrm{drop}_1\ F\ x\ f\ a = F\ (\lambda\langle x,\ a'\rangle.f\ a')\ \langle x,\ a\rangle \end{array}$$

This extensional version makes it possible to lambda-drop a functional prior to taking its fixed point.

Theorem 2 (lambda-dropping) *Let A, B, and X denote pointed CPOs and let $F \in (X \times A \to B) \to X \times A \to B$ be inert in X. Then for any $x \in X$ and $a \in A$,*

$$fix_{X \times A \to B}\, F\, \langle x,\, a\rangle = fix_{A \to B}\, (\mathrm{drop}_1\, F\, x)\, a.$$

Proof: By fixed-point induction. Let us define R_x as an X-indexed family of admissible relations between $X \times A \to B$ and $A \to B$:

$$R_x = \{(\ell,\, d) \mid \forall a \in A.\ell\, \langle x,\, a\rangle = d\, a\}$$

Each R_x is pointed and admissible. Now $\forall x \in X$ and $\forall(\ell,\, d) \in R_x$,

$$(F\, \ell,\, \mathrm{drop}_1\, F\, x\, d) \in R_x$$

$$\begin{array}{lll} \text{since } \forall a \in A,\, \mathrm{drop}_1\, F\, x\, d\, a & = & F\, (\lambda\langle x',\, a'\rangle.d\, a')\, \langle x,\, a\rangle \\ & = & F\, (\lambda\langle x',\, a'\rangle.\ell\, (x,\, a'))\, \langle x,\, a\rangle \\ & = & F\, (\lambda\langle x',\, a'\rangle.\ell\, (x',\, a'))\, \langle x,\, a\rangle \text{ since } F \text{ is inert in } X \\ & = & F\, \ell\, \langle x,\, a\rangle \end{array}$$

Therefore, by fixed-point induction, the least fixed points of the two functions are also related, i.e.,

$$(\mathrm{fix}_{X \times A \to B}\, F,\, \mathrm{fix}_{A \to B}\, (\mathrm{drop}_1\, F\, x)) \in R_x$$

which is what we wanted to prove. □

Similarly, we can define an extensional lambda-dropper that drops the second parameter of an uncurried binary functional, assuming it to be inert.

Definition 5 *Let A, B, and X denote pointed CPOs.*

$$\mathrm{drop}_2 : ((A \times X \to B) \to A \times X \to B) \to X \to (A \to B) \to A \to B$$
$$\mathrm{drop}_2\, F\, x\, f\, a = F\, (\lambda\langle a',\, x\rangle.f\, a')\, \langle a,\, x\rangle$$

2.3 Inverseness properties

It is a simple matter to check that $\mathrm{drop}_1 \circ \mathrm{lift}_1 =$ identity and that $\mathrm{lift}_1 \circ \mathrm{drop}_1 =$ identity over functionals that are inert in their first parameter.

3 Scaling up to mutual recursion

Extensional lambda-lifting and lambda-dropping scale up to mutual recursion, along the lines of Section 2.

For the record, Figure 2 displays an extensional lambda-lifter for a pair of unary functionals abstracted by a dropped parameter, using ML as a meta-language. The lambda-lifter lifts the abstracted parameter in first position in the resulting pair of uncurried binary functionals. Similarly, Figure 3 displays

```
signature LIFT2
= sig
    val lift1 : ('e -> (('a -> 'b) * ('c -> 'd) -> 'a -> 'b) *
                       (('a -> 'b) * ('c -> 'd) -> 'c -> 'd)) ->
                (('e * 'a -> 'b) * ('e * 'c -> 'd) ->
                 'e * 'a -> 'b) *
                (('e * 'a -> 'b) * ('e * 'c -> 'd) ->
                 'e * 'c -> 'd)
  end
structure Lift2 : LIFT2
= struct
    fun lift1 F
        = (fn (f,g) => fn (x1, x2) => let val (F1,F2) = F x1
                                      in F1 (fn a2 => f (x1, a2),
                                             fn a2 => g (x1, a2)) x2
                                      end,
           fn (f,g) => fn (x1, x2) => let val (F1,F2) = F x1
                                      in F2 (fn a2 => f (x1, a2),
                                             fn a2 => g (x1, a2)) x2
                                      end)
  end
```

Figure 2: Extensional lambda-lifting for mutually recursive functions

```
signature DROP2
= sig
    val drop1 : (('e * 'a -> 'b) * ('e * 'c -> 'd) ->
                 'e * 'a -> 'b) *
                (('e * 'a -> 'b) * ('e * 'c -> 'd) ->
                 'e * 'c -> 'd) ->
                'e -> (('a -> 'b) * ('c -> 'd) -> 'a -> 'b) *
                      (('a -> 'b) * ('c -> 'd) -> 'c -> 'd)
  end
structure Drop2 : DROP2
= struct
     fun drop1 (F1, F2) x1
         = (fn (f, g) => fn x2 => F1 (fn (a1, a2) => f a2,
                                      fn (a1, a2) => g a2) (x1, x2),
            fn (f, g) => fn x2 => F2 (fn (a1, a2) => f a2,
                                      fn (a1, a2) => g a2) (x1, x2))
  end
```

Figure 3: Extensional lambda-dropping for mutually recursive functions

```
signature FIX2
= sig
    val fix : (('a -> 'b) * ('c -> 'd) -> 'a -> 'b) *
              (('a -> 'b) * ('c -> 'd) -> 'c -> 'd) ->
              ('a -> 'b) * ('c -> 'd)
  end
structure Fix2 : FIX2
= struct
    fun fix (f1, f2)
        = (fn a1 => f1 (fix (f1, f2)) a1,
           fn a2 => f2 (fix (f1, f2)) a2)
  end
```

Figure 4: Applicative-order fixed-point operator for pairs of functionals

an extensional lambda-dropper for a pair of uncurried binary functionals. The lambda-dropper drops the first parameter of the two functionals. The dropped parameter is assumed to be inert.

These pairs of functionals require a fixed-point operator such as the one in Figure 4.

For example, here is a lambda-dropped pair of functionals computing the parity of a non-negative integer. Their fixed point is the pair of mutually recursive functions `even` and `odd`, parameterized with the value to test for the base case.

```
val mkFodd_even_d
    = fn b => (fn (ev, od) => fn n => (n = b) orelse od (n - 1),
               fn (ev, od) => fn n => (n > b) andalso ev (n - 1))
(*
  mkFodd_even_d : int -> ('a * (int -> bool) -> int -> bool) *
                         ((int -> bool) * 'b -> int -> bool)
*)
```

The corresponding two lambda-dropped parity functions are obtained by instantiating the base value and taking the fixed point of the result:

```
val (even_d, odd_d)
    = Fix2.fix (mkFodd_even_d 0)
(*
  even_d : int -> bool
   odd_d : int -> bool
*)
```

Applying `Lift2.lift1` to the lambda-dropped pair of functionals yields the corresponding lambda-lifted pair of functionals:

```
val Fodd_even_l
    = Lift2.lift1 mkFodd_even_d
```

```
(*
  Fodd_even_l : ((int * int -> bool) * (int * int -> bool) ->
                 int * int -> bool) *
                ((int * int -> bool) * (int * int -> bool) ->
                 int * int -> bool)
*)
```

And indeed, simplifying `Lift2.lift1 Fodd_even_d` yields:

```
(fn (ev, od) => fn (b, n) => (n = b) orelse od (b, n - 1),
 fn (ev, od) => fn (b, n) => (n > b) andalso ev (b, n - 1))
```

which is lambda-lifted.

We obtain two mutually recursive lambda-lifted functions by taking the fixed point of this pair. Instantiating their first argument yields the parity functions:

```
val (even_l, odd_l)
    = let val (even_aux, odd_aux) = Fix2.fix Fodd_even_l
      in (fn n => even_aux (0, n), fn n => odd_aux (0, n))
      end
(*
  even_l : int -> bool
   odd_l : int -> bool
*)
```

Finally, applying `Drop2.drop1` to the lambda-lifted pair of functionals yields a lambda-dropped pair of functionals which is extensionally equivalent to the original lambda-dropped pair of functionals.

```
val mkFodd_even_d'
    = Drop2.drop1 Fodd_even_l
(*
  mkFodd_even_d' : int -> ((int -> bool) * (int -> bool) ->
                           int -> bool) *
                          ((int -> bool) * (int -> bool) ->
                           int -> bool)
*)
```

4 Conclusion

Over the last ten years, only the analysis part of lambda-lifting and lambda-dropping have been investigated. Establishing the formal correctness of their transformation appeared to be still an open problem. We have introduced extensional versions of lambda-lifting and lambda-dropping to address this problem. For the sake of expository concision, we have concentrated on single recursive functions and only informally outlined how the approach scales up to mutually recursive functions.

Concentrating on single recursive functions makes it clear how both lambda-lifting and lambda-dropping connect to the static-argument transformation in

Haskell [4, 8]. One thing we have not reported here (because it is a little tedious) is that lambda-lifting and lambda-dropping are correct both in a call-by-name language and in a call-by-value language. To this end, we defined the denotational semantics of two functional languages, one following call-by-name and one following call-by-value, and we considered the denotations of a lambda-dropped function and of a lambda-lifted function such as the power function of Figure 1: lift_1 and drop_1 respectively map one into the other and vice-versa.

On the other hand, experimenting with mutually recursive functions reveals the practical limitations of extensional, type-directed lambda-lifting and lambda-dropping: handling block structure becomes daunting rather quickly. Some automated support is needed here to study extensional lambda-lifting and lambda-dropping further.

Acknowledgements

Glynn Winskel suggested that the equation of Theorem 1 should be a foundation for extensional lambda-lifting. I am also grateful to Ulrik Schultz for our enjoyable joint work and for many discussions about possible ways of formalizing lambda-dropping, to Belmina Dzafic and Karoline Malmkjær for comments on an earlier draft, and to Daniel Damian, Andrzej Filinski and Lasse R. Nielsen for further comments and suggestions. Thanks are also due to the anonymous referees.

A Lambda-dropping and lambda-lifting from fold

In daily programming practice, we have observed that fold-instantiated definitions naturally yield lambda-lifted programs when the fold function is lambda-lifted and lambda-dropped programs when the fold function is lambda-dropped. For example, let us consider the resident fold function in Standard ML of New Jersey:

```
fold : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b.
```

Figure 5 displays both its lambda-lifted and its lambda-dropped definitions.

The usual definitions of append from fold read as follows.

```
fun append_l xs ys = fold_l (op ::) xs ys
(* append_l : 'a list -> 'a list -> 'a list *)

fun append_d xs ys = fold_d (op ::) xs ys
(* append_d : 'a list -> 'a list -> 'a list *)
```

Simplifying them (using the usual unfold-fold technique) yields the following definitions:

```
fun append_l [] ys
    = ys
```

```
fun fold_l f [] b
    = b
  | fold_l f (x :: xs) b
    = f (x, fold_l f xs b)

fun fold_d f xs b
    = let fun walk [] = b
            | walk (x :: xs) = f (x, walk xs)
      in walk xs
      end
```

Figure 5: Lambda-dropped and lambda-lifted fold functionals for lists

```
  | append_l (x :: xs) ys
    = x :: (append_l xs ys)

fun append_d xs ys
    = let fun walk [] = ys
            | walk (x :: xs) = x :: (walk xs)
      in walk xs
      end
```

The former is lambda-lifted, and the latter is lambda-dropped.

It is, however, also possible to obtain the lambda-lifted version of `append`, using the *lambda-dropped* version of `fold` as follows:

```
fun append_alt xs ys = fold_d (fn (x, c) => fn ys => x :: (c ys))
                              xs
                              (fn ys => ys)
                              ys
(* append_alt : 'a list -> 'a list -> 'a list *)
```

Simplifying the definition of `append_alt` yields:

```
fun append_alt xs ys
     = let fun walk [] = (fn ys => ys)
             | walk (x :: xs) = (fn ys => x :: (walk xs ys))
       in walk xs ys
       end
```

Spreading some syntactic sugar makes it read:

```
fun append_alt xs ys
     = let fun walk [] ys = ys
             | walk (x :: xs) ys = x :: (walk xs ys)
       in walk xs ys
       end
```

which in effect is lambda-lifted.

The key idea here is to instantiate the type variable `'b`, in the type of `fold_d`, with a function `'a list -> 'a list`. This function carries the inert parameter `ys` until its point of use.

Conjecture 1 *One cannot obtain the lambda-dropped version of* `append`*, using the lambda-lifted version of* `fold`*.*

References

[1] Anders Bondorf. Similix manual, system version 3.0. Technical Report 91/9, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1991.

[2] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, pages 128–139, Orlando, Florida, June 1994. ACM Press.

[3] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

[4] Olivier Danvy and Ulrik Pagh Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 90–106, Amsterdam, The Netherlands, June 1997. ACM Press. Extended version available as the technical report BRICS-RS-97-6.

[5] John Hughes. Super combinators: A new implementation method for applicative languages. In Daniel P. Friedman and David S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, August 1982. ACM Press.

[6] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[7] Erik Meijer. Down with lambda-lifting. Unpublished note, April 1992.

[8] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 1–12, Philadelphia, Pennsylvania, May 1996. ACM Press.

[9] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

[10] Ulrik P. Schultz. Implicit and explicit aspects of scope and block structure. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1997.

[11] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

RS-99-21 Olivier Danvy. *An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*. August 1999. 13 pp. Extended version of an article to appear in *Fourth Fuji International Symposium on Functional and Logic Programming*, FLOPS '99 Proceedings (Tsukuba, Japan, November 11–13, 1999). This report supersedes the earlier report BRICS RS-98-2.

RS-99-20 Ulrich Kohlenbach. *A Note on Spector's Quantifier-Free Rule of Extensionality*. August 1999. 5 pp. To appear in *Archive for Mathematical Logic*.

RS-99-19 Marcin Jurdziński and Mogens Nielsen. *Hereditary History Preserving Bisimilarity is Undecidable*. June 1999. 18 pp.

RS-99-18 M. Oliver Möller and Harald Rueß. *Solving Bit-Vector Equations of Fixed and Non-Fixed Size*. June 1999. 18 pp. Revised version of an article appearing under the title *Solving Bit-Vector Equations* in Gopalakrishnan and Windley, editors, *Formal Methods in Computer-Aided Design: Second International Conference*, FMCAD '98 Proceedings, LNCS 1522, 1998, pages 36–48.

RS-99-17 Andrzej Filinski. *A Semantic Account of Type-Directed Partial Evaluation*. June 1999. To appear in Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, PPDP99 '99 Proceedings, LNCS, 1999.

RS-99-16 Rune B. Lyngsø and Christian N. S. Pedersen. *Protein Folding in the 2D HP Model*. June 1999. 15 pp.

RS-99-15 Rune B. Lyngsø, Michael Zuker, and Christian N. S. Pedersen. *An Improved Algorithm for RNA Secondary Structure Prediction*. May 1999. 24 pp. An alloy of two articles appearing in Istrail, Pevzner and Waterman, editors, *Third Annual International Conference on Computational Molecular Biology*, RECOMB 99 Proceedings, 1999, pages 260–267, and *Bioinformatics*, 15, 1999.

RS-99-14 Marcelo P. Fiore, Gian Luca Cattani, and Glynn Winskel. *Weak Bisimulation and Open Maps*. May 1999. To appear in Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, LICS '99 Proceedings, 1999.