BRICS

Basic Research in Computer Science

BRICS RS-03-26 Danvy & Schultz: Lambda-Lifting in Quadratic Time

Lambda-Lifting in Quadratic Time

Olivier Danvy
Ulrik P. Schultz

BRICS Report Series **RS-03-26**

ISSN 0909-0878 **August 2003**

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

Lambda-Lifting in Quadratic Time [*]

Olivier Danvy [†]
BRICS [‡]

Ulrik P. Schultz [§]
Center for Pervasive Computing

Department of Computer Science, University of Aarhus

August 15, 2003

Abstract

Lambda-lifting is a program transformation used in compilers and in partial evaluators and that operates in cubic time. In this article, we show how to reduce this complexity to quadratic time, and we present a flow-sensitive lambda-lifter that also works in quadratic time.

Lambda-lifting transforms a block-structured program into a set of recursive equations, one for each local function in the source program. Each equation carries extra parameters to account for the free variables of the corresponding local function *and of all its callees.* It is the search for these extra parameters that yields the cubic factor in the traditional formulation of lambda-lifting, which is due to Johnsson. This search is carried out by a transitive closure.

Instead, we partition the call graph of the source program into strongly connected components, based on the simple observation that *all functions in each component need the same extra parameters and thus a transitive closure is not needed.* We therefore simplify the search for extra parameters by treating each strongly connected component instead of each function as a unit, thereby reducing the time complexity of lambda-lifting from $\mathcal{O}(n^3 \log n)$ to $\mathcal{O}(n^2 \log n)$, where n is the size of the program.

Since a lambda-lifter can output programs of size $\mathcal{O}(n^2)$, we believe that our algorithm is close to optimal.

---

[*] A preliminary version of this article was presented at FLOPS'02 [16].

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

[‡] Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

[§] Address: IT-Byen, Aabogade 34, DK-8200 Aarhus C, Denmark.
Phone: (+45) 89 42 56 66. Fax: (+45) 89 42 56 24.
E-mail: `ups@daimi.au.dk`
Home page: `http://www.daimi.au.dk/~ups`

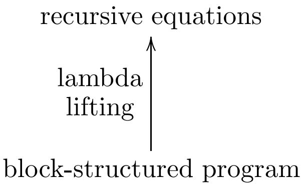
Contents

List of Figures

1 Lambda-lifting

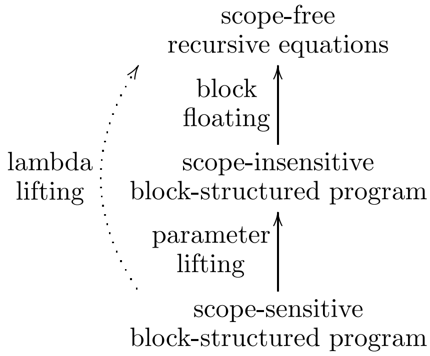
1.1 Setting and background

Lambda-lifting: what. In the mid 1980's, Augustsson, Hughes, Johnsson, and Peyton Jones devised 'lambda-lifting', a meaning-preserving transformation from block-structured programs to recursive equations [7, 23, 24, 32].



Recursive equations provide a propitious format because they are scope free. Today, a number of systems use lambda-lifting as an intermediate phase. For example, partial evaluators such as Schism, Similix, and Pell-Mell lambda-lift source programs and generate scope-free recursive equations [9, 11, 28]. Compilers such as Larceny and Moby use local, incremental versions of lambda-lifting in their optimizations [10, 33], and so did an experimental version of the Glasgow Haskell Compiler [35]. Hanus's compiler for the functional logic language Curry also uses lambda-lifting (personal communication to the first author, FLOPS'02). Program generators such as Bakewell and Runciman's least general common generalization operate on lambda-lifted programs [8].

Lambda-lifting: how. Lambda-lifting operates in two stages: *parameter lifting* and *block floating*.



A block-structured program is scope-sensitive because of free variables in local functions. Parameter lifting makes a program scope-insensitive by passing extra variables to each function. These variables account both for the free variables of each function but also for variables occurring free further in the call path. Block floating erases block structure by making each local function a global recursive equation.

Parameter lifting. Parameter-lifting a program amounts to making all the free variables of a function formal parameters of this function. All callers of the function must thus be passed these variables as arguments as well. A set of solutions is built by traversing the program. A solution pairs each function with a least set of additional parameters. Each block of locally defined functions gives rise to a collection of set equations describing which variables should be passed as arguments to its local functions. The names of functions, however, are not included in the sets, since all functions become globally visible when the lambda-lifting transformation is complete. The solution of each set equation extends the current set of solutions, which is then used to analyze the header and the body of the block.

Block floating. After parameter lifting, a program is scope insensitive. Block floating is thus straightforward: the program is merely traversed, all local functions are collected and all blocks are replaced by their bodies. The collected function definitions are then appended to the program as global mutually recursive functions, making all functions globally visible.

Lambda-lifting: when. In a compiler, the effectiveness of lambda-lifting hinges on the tension between passing many actuals vs. passing few actuals, and between referring to an actual parameter vs. referring to a free variable.

In practice, though, programmers often stay away both from recursive equations and from maximally nested programs. Instead, they write in a mixed style that both abides by Perlis's epigram "If you have a procedure with ten parameters, you probably missed some." and by Turner's recommendation that good Miranda style means little nesting. In this mixed style, and to paraphrase another of Perlis's epigrams, one man's parameter is another man's free variable.

1.2 Three examples

We first illustrate lambda-lifting with the classical `foldr` functional, and then with two examples involving multiple local functions and mutual recursion. Throughout, we use Standard ML.

Example 1: We consider the classical fold function for lists, defined with a local function.

```
fun foldr f b xs
    = let fun walk nil
              = b
            | walk (x :: xs)
              = f (x, walk xs)
      in walk xs
      end
```

Lambda-lifting this block-structured program yields two recursive equations: the original entry point, which now serves as a wrapper to invoke the other function, and the other function, which has been extended with two extra parameters.

```
fun foldr f b xs
    = foldr_walk f b xs
and foldr_walk f b []
    = b
  | foldr_walk f b (x :: xs)
    = f (x, foldr_walk f b xs)
```

Example 2: The following token program adds its two parameters.

```
fun main x y
    = let fun add p
                = add_to_x p
          and add_to_x q
                = q + x
      in add y
      end
```

Lambda-lifting this block-structured program yields three recursive equations:

```
fun main x y
    = main_add x y
and main_add x p
    = main_add_to_x x p
and main_add_to_x x q
    = q + x
```

As a local function, `add_to_x` has a free variable, `x`, and thus it needs to be passed the value of `x`. Since `add` calls `add_to_x`, it needs to pass the value of `x` to `add_to_x` and thus to be passed this value, even though `x` is not free in the definition of `add`. During parameter lifting, each function thus needs to be passed not only the value of its free variables, but also the values of the free variables of all its callees.

Example 3: The following token program multiplies its two parameters with successive additions, using mutual recursion.

```
fun mul x y
    = let fun loop z
                = if z=0 then 0 else add_to_x z
          and add_to_x z
                = x + loop (z-1)
      in loop y
      end
```

Again, lambda-lifting this block-structured program yields three recursive equations:

```
fun mul x y
    = mul_loop x y
and mul_loop x z
    = if z=0 then 0 else mul_add_to_x x z
and mul_add_to_x x z
    = x + mul_loop x (z-1)
```

As before, the free variable `x` of `add_to_x` has to be passed as a formal parameter, through its caller `loop`. When `add_to_x` calls `loop` recursively, it must pass the value of `x` to `loop`, so that `loop` can pass it back in the recursive call.

This third example illustrates our key insight: during parameter lifting, mutually recursive functions must be passed the same set of variables as parameters.

1.3 Overview

Lambda-lifting, as specified by Johnsson, takes cubic time (Section 2). In this article, we show how to reduce this complexity to quadratic time (Section 3). We also present a flow-sensitive extension to lambda-lifting, where flow information is used to eliminate redundant formal parameters generated by the standard algorithm (Section 4).

Throughout the main part of the article, we consider Johnsson's algorithm [24, 25]. Other styles of lambda-lifting, however, exist: we describe them as well, together with addressing related work (Section 5).

2 Lambda-lifting in cubic time

2.1 Johnsson's parameter-lifting algorithm

Johnsson's algorithm descends recursively through the program structure, calculating the set of variables that are needed by each function. This is done by solving set equations describing the dependencies between functions. These dependencies may be arbitrarily complex, since a function can depend on any variable or function that is lexically visible to it. In particular, mutually recursive functions depend upon each other, and so they give rise to mutually recursive set equations.

The mutually recursive set equations are solved using fixed-point iteration. A program containing m function declarations gives rise to m set equations. In a block-structured program the functions are distributed across the program, so we solve the set equations in groups, as we process each block of local functions. Each set equation unifies $\mathcal{O}(m)$ sets of size $\mathcal{O}(n)$, where n is the size of the program. However, the total size of all equations is bounded by the size of the program n, so globally each iteration takes time $\mathcal{O}(n \log n)$. The number of set union operations needed is $\mathcal{O}(n^2)$, so the time needed to solve all the set equations is $\mathcal{O}(n^3 \log n)$, which is the overall running time of lambda-lifting.

2.2 An alternative specification based on graphs

Rather than using set equations, one can describe an equivalent algorithm using graphs. We use a graph to describe the dependencies between functions. Peyton Jones names this representation a *dependency graph* [32], but he uses it for a different purpose (see Section 5.1). Each node in the graph corresponds to a function in the program, and is associated with the free variables of this function. An edge in the graph from a node `f` to a node `g` indicates that the function `f` depends on `g`, because it refers to `g`. Mutually recursive dependencies give rise to cycles in this graph. Rather than solving the mutually recursive equations using fixed-point iteration, we propagate the variables associated with each node backwards through the graph, from callee to caller, merging the variable sets, until a fixed point is reached.

2.3 Example

Figure 1 shows a small program, defined using three mutually recursive functions, each of which has a different free variable.

We can describe the dependencies between the local block of functions using set equations, as shown in Figure 2. To solve these set equations, we need to perform three fixed-point iterations, since there is a cyclic dependency of size three. Similarly, we can describe these dependencies using a graph, also shown in Figure 2. The calculation of the needed variables can be done using this representation, by propagating variable sets backwards through the graph. A single propagation step is done by performing a set union over the variables associated with a node and the variables associated with its successors. Similarly to the case of the set equations, each node must be visited three times before a fixed point is reached.

When the set of needed variables has been determined for each function, solutions describing how each function must be expanded with these variables

```
fun main x y z n
    = let fun f1 i
              = if i=0 then 0 else x + f2 (i-1)
          and f2 j
              = let fun g2 b = b * j
                in if j=0 then 0 else g2 y + f3 (j-1)
                end
          and f3 k
              = let fun g3 c = c * k
                in if k=0 then 0 else g3 z + f1 (k-1)
                end
      in f1 n
      end
```

Figure 1: Three mutually recursive functions

$$\begin{pmatrix} S_{\texttt{f1}} & = & \{\texttt{x}\} \cup S_{\texttt{f2}} \\ S_{\texttt{f2}} & = & \{\texttt{y}\} \cup S_{\texttt{f3}} \\ S_{\texttt{f3}} & = & \{\texttt{z}\} \cup S_{\texttt{f1}} \end{pmatrix}$$

$$\begin{aligned} S_{\texttt{g2}} &= \{\texttt{j}\} \\ S_{\texttt{g3}} &= \{\texttt{k}\} \end{aligned}$$

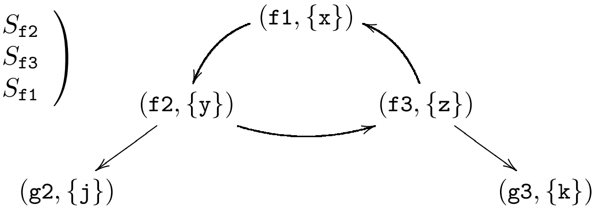


Figure 2: Dependencies between the local functions in Figure 1

```
fun main x y z n
    = f1 x y z n
and f1 x y z i
    = if i=0 then 0 else x + f2 x y z (i-1)
and f2 x y z j
    = if j=0 then 0 else g2 y j + f3 x y z (j-1)
and g2 b j
    = b * j
and f3 x y z k
    = if k=0 then 0 else g3 z k + f1 x y z (k-1)
and g3 c k
    = c * k
```

Figure 3: Lambda-lifted counterpart of Figure 1

are added to the set of solutions. The result is shown in Figure 3.

3 Lambda-lifting in quadratic time

We consider the variant of the parameter-lifting algorithm that operates on a dependency graph. It propagates needed variables backwards through the graph since the caller needs the variables of each callee.

It is our observation that functions that belong to the same strongly connected component of the call graph must be parameter-lifted with the same set of variables (as was illustrated in Section 1.2). We can thus treat these functions in a uniform fashion, by coalescing the strongly connected components of the dependency graph. Each function must define at least its free variables together with the free variables of the other functions of the strongly connected component. Coalescing the strongly connected components of the dependency graph produces a DAG with sets of function names for nodes. A breadth-first backwards propagation of variables can then be done in linear time, which eliminates the need for a fixed-point computation.

3.1 Complexity analysis

The parameter-lifting algorithm must first construct the dependency graph, which takes time $\mathcal{O}(n \log n)$, where n is the size of the program. The strongly connected components of the graph can then be computed in time $\mathcal{O}(n)$. The ensuing propagation requires a linear number of steps since we are now operating on a DAG. Each propagation step consists of a number of set-union operations, each of which takes $\mathcal{O}(n \log n)$ time, i.e., the time to unify two sets of variables of size $\mathcal{O}(n)$. Globally, a number of set-union operations linear in the size of the program needs to be performed, yielding a time complexity of $\mathcal{O}(n^2 \log n)$. The overall running time is thus $\mathcal{O}(n^2 \log n)$, where n is the size of the program.

3.2 Lower bound

Consider a function with m formal parameters $\{v_1, \ldots, v_m\}$ that declares m mutually recursive local functions, each of which has a different variable from $\{v_1, \ldots, v_m\}$ as a free variable. The size of the program n is $\mathcal{O}(m)$. The output program contains the m functions, each of which needs to be expanded with the m formal parameters of the enclosing function. The output program is therefore of size $\mathcal{O}(m^2)$, which is also $\mathcal{O}(n^2)$. One thus cannot perform lambda-lifting faster than $\mathcal{O}(n^2)$. Since one needs $\mathcal{O}(n \log n)$ time to compute the sets of free variables of the program, our complexity of $\mathcal{O}(n^2 \log n)$ must be close to optimal.

3.3 Contribution

Our contribution is

- to characterize the fixed-point operations on the set equations as propagation through the dependency graph, and
- to recognize that functions in the same strongly connected component require the same set of variables.

We can therefore first determine which variables need to be known by each function in a strongly connected component, and then add them as formal parameters to these functions. In each function, those variables not already passed as parameters to the function should be added as formal parameters.

This approach can be applied locally to work like Johnsson's algorithm, processing each block independently. It can also be applied globally to the overall dependency graph. The global algorithm, however, must explicitly limit the propagation of free variables, so that they are not propagated beyond their point of definition.

3.4 The new algorithm

We operate on programs conforming to the simple syntax of Figure 4.

$$
\begin{array}{lcl}
p \in \mathsf{Program} & ::= & \{d_1, \ldots, d_m\} \\
d \in \mathsf{Def} & ::= & f \equiv \lambda v_1 \boldsymbol{.} \ldots \lambda v_n \boldsymbol{.} e \\
e \in \mathsf{Exp} & ::= & e_0 \, \ldots \, e_n \\
 & | & \mathrm{LetRec}\,\{d_1, \ldots, d_k\}\; e_0 \\
 & | & \mathrm{If}\, e_0\, e_1\, e_2 \\
 & | & f \\
 & | & v \\
 & | & \underline{\mathrm{literal}} \\
v \in \mathsf{Variable} & & \\
f \in \mathsf{FunctionName} \cup \mathsf{PredefinedFunction} & &
\end{array}
$$

Figure 4: Simplified syntax of source programs

The set FV(f) denotes the set of free variables in the function f, and the set FF(f) denotes the set of free functions in f (note that $\mathrm{FV}(f) \cap \mathrm{FF}(f) = \emptyset$). In our algorithm, we assume variable hygiene, i.e., that no name clashes can occur. Figure 5 shows our (locally applied) $\mathcal{O}(n^2 \log n)$ parameter-lifting algorithm. It makes use of several standard graph and list operations that are described in the appendix. Figure 6 shows the standard linear-time (globally applied) block-floating algorithm. Johnsson's original lambda-lifting algorithm includes steps to explicitly name anonymous lambda expressions and replace non-recursive let blocks by applications. These steps are trivial and omitted from the figures.

When parameter-lifting a set of mutually recursive functions $\{f_1, \ldots, f_k\}$, and some function f_i defines a variable x that is free in one of its callees f_j, a naive algorithm expands the parameter list of the function with x. The sets P_{f_i} used in our parameter-lifting algorithm serve to avoid this problem.

3.5 Revisiting the example of Section 2.3

Applying the algorithm of Figure 5 to the program of Figure 1 processes the `main` function by processing its body. The letrec block of the body is processed by first constructing a dependency graph similar to that shown in Figure 2 (except that we simplify the description of the algorithm to not include the sets of free variables in the nodes). Coalescing the strongly connected components of this graph yields a single node containing the three functions. Since there is only a single node, the propagation step only serves to associate each function in the node with the union of the free variables of each of the functions in the component. These variable sets directly give rise to a new set of solutions.

Each of the functions defined in the letrec block and the body of the letrec block are traversed and expanded with the variables indicated by the set of solutions. Block floating according to the algorithm of Figure 6 yields the program of Figure 3.

```
parameterLiftProgram :: Program → Program
parameterLiftProgram $p$ = map (parameterLiftDef $\emptyset$) $p$
parameterLiftDef :: Set(FunName,Set(Variable)) → Def → Def
parameterLiftDef $S$ $(f \equiv \lambda v_1 \ldots \lambda v_n . e)$ =
                    applySolutionToDef $S$ $(f \equiv \lambda v_1 \ldots \lambda v_n .$(parameterLiftExp $S$ $e$))
parameterLiftExp :: Set(FunName,Set(Variable)) → Exp → Exp
parameterLiftExp $S$ $(e_0 \ldots e_n)$ =
                    let $e'_i$ = parameterLiftExp $S$ $e_i$, for each $e_i \in \{e_0, \ldots, e_n\}$
                    in $(e'_0 \ldots e'_n)$
parameterLiftExp $S$ (LetRec $\{d_1, \ldots, d_k\}$ $e_0$) =
    let $G$ = ref $(\emptyset, \emptyset)$
        $V_{f_i}$ = ref (FV($f_i$)), for each $(d_i = (f_i \equiv \lambda v_1 \ldots \lambda v_n . e)) \in \{d_1, \ldots, d_k\}$
        $P_{f_i} = \{v_1, \ldots, v_n\}$, for each $(d_i = (f_i \equiv \lambda v_1 \ldots \lambda v_n . e)) \in \{d_1, \ldots, d_k\}$
    in foreach $f_i \in \{f_1, \ldots, f_k\}$ do
           foreach $g \in$ FF($f_i$) $\cap \{f_1, \ldots, f_k\}$ do
               Graph.add-edge $G$ $f_i$ $g$
       let $(G'$ as $(V', E'))$ = Graph.coalesceSCC $G$
           $\mathrm{succ}_p = \{q \in V' | (p, q) \in E'\}$, for each $p \in V'$
           $F_p = \bigcup_{q \in \mathrm{succ}_p} q$, for each $p \in V'$
           propagate :: List(Set(FunName)) → ()
           propagate [] = ()
           propagate $(p :: r)$ =
               let $V = (\bigcup_{f \in p} V_f) \cup (\bigcup_{g \in F_p} V_g)$
               in foreach $f \in p$ do
                      $V_f := V \backslash P_f$;
                  (propagate $r$)
       in (propagate (List.reverse (Graph.breadthFirstOrdering $G'$)));
          let $S' = S \cup \{(f_1, V_{f_1}), \ldots, (f_k, V_{f_k})\}$
              $f_s$ = map (parameterLiftDef $S'$) $\{d_1, \ldots, d_k\}$
              $e'_0$ = parameterLiftExp $S'$ $e_0$
          in (LetRec $f_s$ $e'_0$)
parameterLiftExp $S$ (If $e_0$ $e_1$ $e_2$) =
                    let $e'_i$ = parameterLiftExp $S$ $e_i$, for each $e_i \in \{e_0, e_1, e_2\}$
                    in (If $e'_0$ $e'_1$ $e'_2$)
parameterLiftExp $S$ $f$ = applySolutionToExp $S$ $f$
parameterLiftExp $S$ $v$ = $v$
parameterLiftExp $S$ ($x$ as literal) = $x$
applySolutionToDef :: Set(FunName,Set(Variable)) → Def → Def
applySolutionToDef ($S$ as $\{\ldots, (f, \{v_1, \ldots, v_n\}), \ldots\}$) $(f \equiv \lambda v'_1 \ldots \lambda v'_n . e)$ =
                      $(f \equiv \lambda v_1 \ldots \lambda v_n . \lambda v'_1 \ldots \lambda v'_n . e)$
applySolutionToDef $S$ $d$ = $d$
applySolutionToExp :: Set(FunName,Set(Variable)) → Exp → Exp
applySolutionToExp ($S$ as $\{\ldots, (f, \{v_1, \ldots, v_n\}), \ldots\}$) $f$ = $(f\, v_1 \ldots v_n)$
applySolutionToExp $S$ $e$ = $e$
```

Figure 5: Parameter lifting: free variables are made parameters

```
blockFloatProgram :: Program → Program
blockFloatProgram $p$ = foldr makeUnion (map blockFloatDef $p$) $\emptyset$

blockFloatDef :: Def → (Set(Def),Def)
blockFloatDef ($f \equiv \lambda v_1.\ldots\lambda v_n.e$) = let ($F_{\text{new}}$,$e'$) = blockFloatExp $e$
                                         in ($F_{\text{new}}$, $f \equiv \lambda v_1.\ldots\lambda v_n.e'$)

blockFloatExp :: Exp → (Set(Def),Exp)
blockFloatExp ($e_0 \ldots e_n$) =
        let ($F_i, e'_i$) = blockFloatExp $e_i$, for each $e_i \in \{e_0, \ldots, e_n\}$
            $F_{\text{new}}$= foldr ($\cup$) $\{F_1, \ldots, F_n\}$ $\emptyset$
        in ($F_{\text{new}}$,$e'_0 \ldots e'_n$)
blockFloatExp (LetRec $\{d, \ldots\}$ $e_0$) =
        let ($F_{\text{new}}, e$) = blockFloatExp (LetRec $\{\ldots\}$ $e_0$)
        in ($\{d\} \cup F_{\text{new}}, e$)
blockFloatExp (LetRec $\emptyset$ $e_0$) = blockFloatExp $e_0$
blockFloatExp (If $e_0$ $e_1$ $e_2$) =
        let ($F_i, e'_i$) = blockFloatExp $e_i$, for each $e_i \in \{e_0, e_1, e_2\}$
        in ($F_0 \cup F_1 \cup F_2$,(If $e'_0$ $e'_1$ $e'_2$))
blockFloatExp $f$ = ($\emptyset, f$)
blockFloatExp $v$ = ($\emptyset, v$)
blockFloatExp ($x$ as literal) = ($\emptyset, x$)

makeUnion :: ((Set(Def),Def),Set(Def)) → Set(Def)
makeUnion (($F_{\text{new}}, d$), $S$) = $F_{\text{new}} \cup \{d\} \cup S$
```

Figure 6: Block floating: block structure is flattened

4 Flow-sensitive lambda-lifting in quadratic time

The value of a free variable might be available within a strongly connected component under a different name. Johnsson's algorithm (and therefore our algorithm as well), however, includes all the variables from the outer scopes as formal parameters because it only looks at their name. It therefore can produce redundant lambda-lifted programs with aliasing.

4.1 A simple example of aliasing

The following token program adds its parameter to itself.

```
fun main x
    = let fun add y
              = x + y
      in add x
      end
```

In the definition of `add`, the free variable `x` is an alias of the formal parameter `y`.

Lambda-lifting this program yields two recursive equations:

```
fun main x
    = main_add x x
and main_add x y
    = x + y
```

The extraneous parameter afflicting the second recursive equation corresponds to the aliasing mentioned above.

In extreme cases, the number of extraneous parameters can explode: consider the lower bound example of Section 3.2, where if the n formal parameters had been aliases, there would have been $\mathcal{O}(n^2)$ extraneous parameters. Such extra parameters can have a dramatic effect. For example, Appel's compiler uses algorithms that are not linear on the arity of source functions [2]. Worse, in partial evaluation, one of Glenstrup's analyses is exponential in the arity of source functions [20].

4.2 A solution

Improving the lambda-lifting algorithm would require us to look at the flow graph, as we did for lambda-dropping [15]. Variables coming from an outer scope that are present in a strongly connected component and that retain their identity through all recursive invocations do not need to be added as formal parameters. Doing so would solve the aliasing problem and yield what we conjecture to be "optimal lambda-lifting."

When performing lambda-lifting we do not need to take into account applications of higher-order functions, as illustrated in Example 1 of Section 1.1. (Doing so would lead us towards defunctionalization [14, 34].) Therefore, a simple first-order flow-analysis which can be computed in time $\mathcal{O}(n \log n)$, where n is the size of the program, is sufficient for flow-sensitive lambda-lifting.

4.3 An algorithm

The parameter-lifting algorithm presented in Figure 5 can be modified to perform flow-sensitive lambda lifting. Given a program annotated with the results of a first-order flow-analysis, parameter lifting proceeds as in the flow-insensitive case, except that a free variable already available as a formal parameter is not added to the set of solutions, but is instead substituted with the formal parameter that it aliases. The block-lifting algorithm remains unchanged. Since first-order flow analysis information can be computed in time $\mathcal{O}(n \log n)$, the time complexity of the complete lambda-lifting algorithm remains unchanged.

4.4 Revisiting the example of Section 4.1

Getting back to the token program of Section 4.1, a flow-sensitive lambda-lifter would yield the following recursive equations.

```
fun main x
    = main_add x
and main_add x
    = x + x
```

This lambda-lifted program does not have redundant parameters.

5 Related work

We review alternative approaches to handling free variables in higher-order, block-structured programming languages, namely supercombinator conversion, closure conversion, lambda-dropping, and incremental versions of lambda-lifting and closure conversion. Finally, we address the issues of formal correctness and typing.

5.1 Supercombinator conversion

Peyton Jones's textbook describes the compilation of functional programs towards the G-machine [32]. Functional programs are compiled into supercombinators, which are then processed at run time by graph reduction. Supercombinators are closed lambda-expressions. Supercombinator conversion [17, 23, 31] produces a series of closed terms, and thus differs from lambda-lifting that produces a series of mutually recursive equations where the names of the equations are globally visible.

Peyton Jones also uses strongly connected components for supercombinator conversion. First, dependencies are analyzed in a set of recursive equations. The resulting strongly connected components are then topologically sorted and the recursive equations are rewritten into nested letrec blocks. There are two reasons for this design:

1. it makes type-checking faster and more precise; and
2. it reduces the number of parameters in the ensuing supercombinators.

Supercombinator conversion is then used to process each letrec block, starting outermost and moving inwards. Each function is expanded with its own free variables, and made global under a fresh name. Afterwards, the definition of each function is replaced by an application of the new global function to its free variables, including the new names of any functions used in the body. This application is mutually recursive in the case of mutually recursive functions, relying on the laziness of the source language; it effectively creates a closure for the functions.

Peyton Jones's algorithm thus amounts to first applying dependency analysis to a set of mutually recursive functions and then to perform supercombinator conversion. As for dependency analysis, it is only used to optimize type checking and to minimize the size of closures.

In comparison, applying our algorithm locally to a letrec block would first partition the functions into strongly connected components, like dependency analysis. We use the graph structure, however, to propagate information, not to obtain an ordering of the nodes for creating nested blocks. We also follow Johnsson's algorithm, where the names of the global recursive equations are free in each recursive equations, independently of the evaluation order. Instead, Johnsson's algorithm passes all the free variables that are needed by a function and its callees, rather than just the free variables of the function.

To sum up, Peyton Jones's algorithm and our revision of Johnsson's algorithm both coalesce strongly connected components in the dependency graph, but for different purposes, our purpose being to reduce the time complexity of lambda-lifting from cubic to quadratic.

5.2 Closure conversion

The notion of closure originates in Landin's seminal work on functional programming [26]. A closure is a functional value and consists of a pair: a code pointer and an environment holding the denotation of the variables that occur free in this code. Efficient representations of closures are still a research topic today [37].

Closure conversion is a key step in Standard ML of New Jersey [5, 6], and yields scope-insensitive programs. It is akin to supercombinator conversion, though rather than creating a closure through a mutually recursive application, the closure is explicitly created as a vector holding the values of the free variables of the possibly mutually recursive functions.

In his textbook [32], Peyton Jones concluded his discussion between lambda-lifting and supercombinator/closure conversion by pointing out a tension between

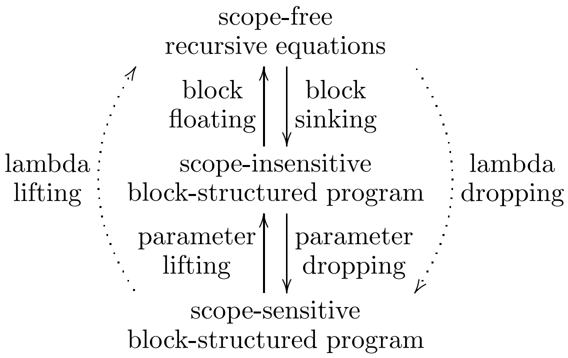
- passing all the [denotations of the] free variables of all the callees but not the values of the mutually recursive functions (in lambda-lifting), and
- passing all the values of the mutually recursive functions but not the free variables of the callees (in closure conversion).

He left this tension unresolved, stating that future would tell which algorithm (lambda-lifting or closure conversion) would prevail.

Today we observe that in the compiler world (Haskell, ML, Scheme), closure conversion has prevailed, with only one exception in Scheme [10]. Conversely, in the program-transformation world [9, 11, 28], lambda-lifting has prevailed. We also observe that only for lambda-lifting has an inverse transformation been developed: lambda-dropping.

5.3 Lambda-dropping

Lambda-dropping is the inverse of lambda-lifting [15]:



Block floating is reversed by block sinking, which creates block structure by making functions used in only one function local to this function. Parameter lifting is reversed by parameter dropping, which removes redundant formal parameters that are originally defined in an outer scope and that always take on the same value.

Lambda-lifting simplifies the structure of a program. However, a program transformation that employs lambda-lifting as a preprocessing phase tends to output a lambda-lifted program rather than a block-structured one. For one point, the resulting programs are less readable. For another point, compilers are often geared for source programs with few parameters.[1] Therefore, increased numbers of formal parameters often form a major overhead in procedure invocation at run time. Against these odds, lambda-dropping can be applied to re-create block structure and reduce the number of formal parameters.

A few years ago, Appel has pointed out a correspondence between imperative programs in SSA form and functional programs using block structure and lexical scope [3]. Specifically, he has shown how to transform an SSA program into its functional representation.[2] We were struck by the fact that this transformation corresponds to performing block sinking on the recursive equations defining the program. As for the transformation into optimal SSA form (which diminishes the number of Φ-nodes), it is equivalent to parameter dropping. This made us conclude that lambda-dropping can be used to transform programs in SSA form into optimal SSA form [15].

This conclusion prompted us to improve the complexity of the lambda-dropping algorithm to $\mathcal{O}(n \log n)$, where n is the size of the program, by using the dominance graph of the dependency graph. We then re-stated lambda-lifting in a similar framework using graph algorithms, which led us to the result presented in the present article.

---

[1] For example, the magic numbers of parameters, in OCaml, are 0 to 7.

[2] The point is made comprehensively in his SIGPLAN Notices note, which is also available in his home page [4].

Even with the improvement presented in this article, we are still left in an asymmetric situation where lambda-lifting and lambda-dropping do not have the same time complexity. With some thought, though, this asymmetry is not so surprising, since lambda-dropping is applied to the output of lambda-lifting, and the complexity is measured in terms of the size of the output program. Measuring the complexity of lambda-dropping in terms of the size of the program before lambda-lifting yields a relative time complexity of lambda-dropping of $\mathcal{O}((n^2)\log(n^2))$, which is $\mathcal{O}(n^2 \log n)$, a fitting match for the $\mathcal{O}(n^2 \log n)$ time complexity of lambda-lifting.

5.4 Flow sensitivity, revisited

We observe that lambda-dropping is flow sensitive, in the sense that it removes the aliased parameters identified as a possible overhead for lambda-lifting in Section 4. Therefore flow-sensitive lambda-lifting can be achieved by first lambda-dropping the program, and then lambda-lifting the result in the ordinary flow-insensitive way. Since the time complexity of lambda-dropping is lower than the time complexity of lambda-lifting, using lambda-dropping as a preprocessing transformation does not increase the overall time complexity of lambda-lifting.

5.5 Mixed style

In order to preserve code locality, compilers such as Twobit [10] or Moby [33] often choose to lift parameters only partially. The result is in the mixed style described at the end of Section 1.1.

In more detail, rather than lifting all the free variables of the program to become formal parameters, parameter lifting is used incrementally to transform programs by lifting only a subset of the free variables of each function. If a function is to be moved to a different scope, however, it needs to be passed the free variables of its callees as parameters. As was the case for global lambda-lifting, propagating the additional parameters through the dependency graph requires cubic time. To improve the time complexity, our quadratic-time parameter-lifting algorithm can be applied to the subsets of the free variables instead. The improvement in time complexity for incremental lambda-lifting is the same as what we observed for the global algorithm.

We note that a partial version of closure conversion also exists, namely Steckler and Wand's [38], that leaves some variables free in a closure because this closure is always applied in the scope of these variables. We also note that combinator-based compilers [41] could be seen as using a partial supercombinator conversion.

5.6 Correctness issues

Only idealized versions of lambda-lifting and lambda-dropping have been formally proven correct. Danvy has related lambda-lifted and lambda-dropped functionals and their fixed point [13]. Fischbach and Hannan have capitalized on the symmetry of lambda-lifting and lambda-dropping to formalize them in a logical framework, for a simply typed source language [18].

Overall, though, and while there is little doubt about Johnsson's original algorithm, its semantic correctness still remains to be established.

5.7 Typing issues

Fischbach and Hannan have shown that lambda-lifting is type-preserving for simply typed programs [18]. Thiemann has pointed out that lambda-lifted ML programs are not always typeable, due to let polymorphism [40]. Here is a very simple example. In the following block-structured program, the locally defined function has type `'a -> int`.

```
fun main ()
    = let fun constant x
              = 42
      in (constant 1) + (constant true)
      end
```

The corresponding lambda-lifted program, however, is not typeable because of ML's monomorphic recursion rule [30]. Since `constant` is defined recursively, its name is treated as lambda-bound, not let-bound:

```
fun main ()
    = (constant 1) + (constant true)
and constant x
    = 42
```

The problem occurs again when one of the free variables of a local recursive function is polymorphically typed.

To solve this problem, one could think of making lambda-lifting yield not just one but several groups of mutually recursive equations, based on a dependency analysis [32]. This would not, however, be enough because a local polymorphic function that calls a surrounding function would end up in the same group of mutually recursive equations as this surrounding function.

There is no generally accepted solution to the problem. Thiemann proposes to parameter-lift some function names as well, as in supercombinator conversion [40]. Fischbach and Hannan propose to use first-class polymorphism instead of let-polymorphism [18]. Yet another approach would be to adopt a polymorphic recursion rule, i.e., to shift from the Damas-Milner type system to the Milner-Mycroft type system, and to use a dependency analysis as in a Haskell compiler. Milner-Mycroft type inference, however, is undecidable [22] and in Haskell, programmers must supply the intended polymorphic type; correspondingly, a lambda-lifter should then supply the types of lifted parameters, when they are polymorphic.

6 Conclusion and future work

We have shown that a transitive closure is not needed for lambda-lifting. In this article, we have reformulated lambda-lifting as a graph algorithm and improved its time complexity from $\mathcal{O}(n^3 \log n)$ to $\mathcal{O}(n^2 \log n)$, where n is the size of the program. Based on a simple example where lambda-lifting generates a program of size $\mathcal{O}(n^2)$, we have also demonstrated that our improved complexity is close to optimal.

The quadratic-time algorithm can replace the cubic-time instances of lambda-lifting in any partial evaluator or compiler, be it for global or for incremental lambda-lifting.

As for future work, we are investigating lambda-lifting in the context of object-oriented languages. Although block structure is instrumental in object-oriented languages such as Java, Beta and Simula [12, 21, 27], existing work on partial evaluation for object-oriented languages has not addressed the issue of block structure [36]. Problems similar to those found in partial evaluation for functional languages appear to be unavoidable: residual methods generated in a local context may need to be invoked outside of the scope of their class. Side effects, however, complicate matters.

Acknowledgements: We are grateful to Lars R. Clausen, Daniel Damian, and Laurent Réveillère for their comments on an earlier version of this article, and to Andrzej Filinski for a clarification about ML typing. Thanks are also due to the anonymous referees for very perceptive and useful reviews.

This work is supported by the ESPRIT Working Group APPSEM (`http://www.appsem.org`).

A Graph and list utilities

The algorithm for parameter lifting, in Figure 5, makes use of a number of graph and list procedures. These procedures are specified in Figure 7.

Graph.add-edge :: $\mathsf{Graph}(\alpha) \rightarrow (\alpha, \alpha) \rightarrow (\alpha, \alpha)$
Graph.add-edge G (n_1, n_2) : Updates G to contain the nodes n_1 and n_2 as well as an edge between the two.

Graph.coalesceSCC :: $\mathsf{Graph}(\alpha) \rightarrow \mathsf{Graph}(\mathsf{Set}(\alpha))$
Graph.coalesceSCC G : Returns G with its strongly connected components coalesced into sets [1].

Graph.breadthFirstOrdering :: $\mathsf{Graph}(\alpha) \rightarrow \mathsf{List}(\alpha)$
Graph.breadthFirstOrdering G : Returns a list containing the nodes of G, in a breadth-first ordering.

List.reverse :: $\mathsf{List}(\alpha) \rightarrow \mathsf{List}(\alpha)$
List.reverse L : Returns L with its elements reversed.

Figure 7: Graph and list procedures

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* World Student Series. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[3] Andrew W. Appel. *Modern Compiler Implementation in {C, Java, ML}.* Cambridge University Press, New York, 1998.

[4] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[5] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM Press.

[6] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Małuszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[7] Lennart Augustsson. A compiler for Lazy ML. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, Texas, August 1984. ACM Press.

[8] Adam Bakewell and Colin Runciman. Automatic generalisation of function definitions. In Middeldorp and Sato [29], pages 225–240.

[9] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[10] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In Talcott [39], pages 128–139.

[11] 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.

[12] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *Simula: Common Base Language.* Norwegian Computing Center, October 1970.

[13] Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In Middeldorp and Sato [29], pages 241–250. Extended version available as the technical report BRICS RS-99-21.

[14] 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.

[15] 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.

[16] Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. In Zhenjiang Hu and Mario Rodriguez-Artalejo, editors, *Sixth International Symposium on Functional and Logic Programming*, number 2441 in Lecture Notes in Computer Science, pages 134–151, Aizu, Japan, September 2002. Springer-Verlag.

[17] Gilles Dowek. Lambda-calculus, combinators and the comprehension scheme. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications*, number 902 in Lecture Notes in Computer Science, pages 154–170, Edinburgh, UK, April 1995. Springer-Verlag. Extended version available as the INRIA research report 2535.

[18] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. *Journal of Functional Programming*, ?(?):??–??, 200? Journal version of [19], to appear.

[19] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. In Walid Taha, editor, *Proceedings of the First Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, number 1924 in Lecture Notes in Computer Science, pages 108–128, Montréal, Canada, September 2000. Springer-Verlag.

[20] Arne John Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, June 1999.

[21] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[22] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[23] 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.

[24] 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.

[25] Thomas Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[26] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[27] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygård. *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, Reading, MA, USA, 1993.

[28] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In John Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Rapport de recherche N^o 2265, INRIA*, pages 112–119, Orlando, Florida, June 1994. Also appears as Technical report CMU-CS-94-129.

[29] Aart Middeldorp and Taisuke Sato, editors. *Fourth Fuji International Symposium on Functional and Logic Programming*, number 1722 in Lecture Notes in Computer Science, Tsukuba, Japan, November 1999. Springer-Verlag.

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

[31] Simon L. Peyton Jones. An introduction to fully-lazy supercombinators. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, number 242 in Lecture Notes in Computer Science, pages 176–208, Val d'Ajol, France, 1985. Springer-Verlag.

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

[33] John Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2/3):161–180, 2002.

[34] 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.

[35] André Santos. *Compilation by transformation in non-strict functional languages*. PhD thesis, Department of Computing, University of Glasgow, Glasgow, Scotland, 1996.

[36] Ulrich P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25:452–499, July 2003.

[37] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In Talcott [39], pages 150–161.

[38] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

[39] Carolyn L. Talcott, editor. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[40] Peter Thiemann. ML-style typing, lambda lifting, and partial evaluation. In *Proceedings of the 1999 Latin-American Conference on Functional Programming, CLAPF '99*, Recife, Pernambuco, Brasil, March 1999.

[41] Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985. Springer-Verlag.

Recent BRICS Report Series Publications

RS-03-26 Olivier Danvy and Ulrik P. Schultz. *Lambda-Lifting in Quadratic Time*. August 2003. 23 pp. Extended version of of a paper appearing in Hu and Rodríguez-Artalejo, editors, *Sixth International Symposium on Functional and Logic Programming*, FLOPS '02 Proceedings, LNCS 2441, 2002, pages 134–151. This report supersedes the earlier BRICS report RS-02-30.

RS-03-25 Biernacki Dariusz and Danvy Olivier. *From Interpreter to Logic Engine: A Functional Derivation*. June 2003.

RS-03-24 Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. *A Functional Correspondence between Call-by-Need Evaluators and Lazy Abstract Machines*. June 2003.

RS-03-23 Korovin Margarita. *Recent Advances in Σ-Definability over Continuous Data Types*. June 2003. 26 pp.

RS-03-22 Ivan B. Damgård and Mads J. Jurik. *Scalable Key-Escrow*. May 2003. 15 pp.

RS-03-21 Ulrich Kohlenbach. *Some Logical Metatheorems with Applications in Functional Analysis*. May 2003. 55 pp.

RS-03-20 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *Fast Partial Evaluation of Pattern Matching in Strings*. May 2003. 16 pp. Final version to appear in Leuschel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '03 Proceedings, 2003. This report supersedes the earlier BRICS report RS-03-11.

RS-03-19 Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. *Static Analysis of XML Transformations in Java*. May 2003. 29 pp.

RS-03-18 Bartek Klin and Paweł Sobociński. *Syntactic Formats for Free: An Abstract Approach to Process Equivalence*. April 2003. 41 pp.

RS-03-17 Luca Aceto, Jens Alsted Hansen, Anna Ingólfsdóttir, Jacob Johnsen, and John Knudsen. *The Complexity of Checking Consistency of Pedigree Information and Related Problems*. March 2003. 31 pp. This paper supersedes BRICS Report RS-02-42.