BRICS

Basic Research in Computer Science

Lambda-Dropping:
Transforming Recursive Equations into
Programs with Block Structure

Olivier Danvy
Ulrik P. Schultz

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/99/27/`

Lambda-Dropping:
Transforming Recursive Equations into
Programs with Block Structure *

Olivier Danvy
BRICS [†]
Department of Computer Science
University of Aarhus [‡]

Ulrik P. Schultz
Compose Project
IRISA/INRIA
University of Rennes [§]

July 30, 1998

Abstract

Lambda-lifting a block-structured program transforms it into a set of recursive equations. We present the symmetric transformation: lambda-dropping. Lambda-dropping a set of recursive equations restores block structure and lexical scope.

For lack of block structure and lexical scope, recursive equations must carry around all the parameters that any of their callees might possibly need. Both lambda-lifting and lambda-dropping thus require one to compute Def/Use paths:

- for lambda-lifting: each of the functions occurring in the path of a free variable is passed this variable as a parameter;

- for lambda-dropping: parameters which are used in the same scope as their definition do not need to be passed along in their path.

A program whose blocks have no free variables is scope-insensitive. Its blocks are then free to float (for lambda-lifting) or to sink (for lambda-dropping) along the vertices of the scope tree.

---

[*]To appear in TCS.

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

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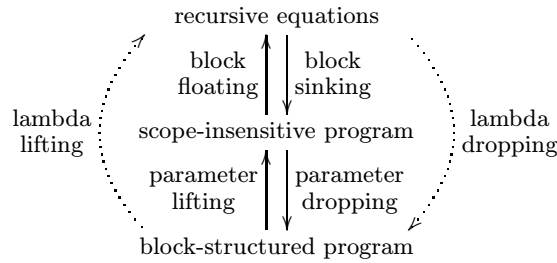E-mail: Olivier.Danvy@brics.dk
Home page: http://www.brics.dk/~danvy

[§]Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France.
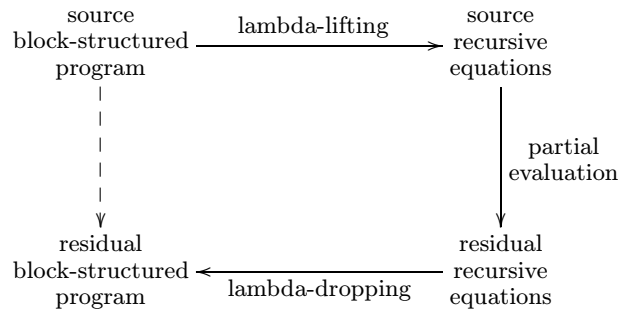Phone: (+33) 2 99 84 75 36. Fax: (+33) 2 99 84 71 71.
E-mail: Ulrik.Schultz@irisa.fr
Home page: http://www.irisa.fr/compose/ups

1

To summarize:

recursive equations

block   block
floating   sinking

lambda   lambda
lifting   scope-insensitive program   dropping

parameter   parameter
lifting   dropping

block-structured program

Our primary application is partial evaluation. Indeed, many partial evaluators for procedural programs operate on recursive equations. To this end, they lambda-lift source programs in a pre-processing phase. But often, partial evaluators [automatically] produce residual recursive equations with dozens of parameters, which most compilers do not handle efficiently. We solve this critical problem by lambda-dropping residual programs in a post-processing phase, which significantly improves both their compile time and their run time.

To summarize:

source   lambda-lifting   source
block-structured   ──────────────────────▶   recursive
program   equations

                                                partial
                                                evaluation

residual   residual
block-structured   ◀──────────────────────   recursive
program   lambda-dropping   equations

Lambda-lifting has been presented as an intermediate transformation in compilers for functional languages. We study lambda-lifting and lambda-dropping per se, though lambda-dropping also has a use as an intermediate transformation in a compiler: we noticed that lambda-dropping a program corresponds to transforming it into the functional representation of its optimal SSA form. This observation actually led us to substantially improve our PEPM'97 presentation of lambda-dropping.

Contents

List of Figures

1 Introduction

1.1 Motivation

As epitomized by currying, lexical scope stands at the foundation of functional programming. And as epitomized by Landin's correspondence principle [28], so does block structure. Yet block structure is not used that much in everyday programming. For example, the standard append function is rather expressed with recursive equations:

```
fun append_lifted (nil, ys)
    = ys
  | append_lifted (x :: xs, ys)
    = x :: (append_lifted (xs, ys))
```

Using block structure, this definition could have been stated as follows:

```
fun append_dropped (xs, ys)
    = let fun loop nil
              = ys
          | loop (x :: xs)
              = x :: (loop xs)
      in loop xs
      end
```

In the first version, **append_lifted**, the second argument is passed unchanged during the whole traversal of the first argument, only to be used in the base case. In the second version, **append_dropped**, the second argument is free in the traversal of the first argument.

This example might appear overly simple, but there are many others. Here are three. In the standard definition of map, the mapped function is passed as an unchanging parameter during the whole traversal of the list. A fold function over a list passes *two* unchanging parameters (the folded function and the initial value of the accumulator) during the traversal of the list. Lambda-interpreters thread the environment through every syntactic form instead of keeping it as an enclosing variable and making an appropriate recursive call when encountering a binding form.

1.2 Recursive equations vs. block structure

The above examples are symptomatic of an expressiveness tension.

Parameters vs. free variables: Recursive equations suffer from a chronic inflation of parameters. As Alan Perlis's epigram goes, "if you have a procedure with ten parameters, you probably missed some." Conversely, who can read a program with nine nested blocks and fifty free variables in the inner one? As Turner puts it in his Miranda manual, good style means little nesting.

Modularity: Block structure encourages one to define auxiliary functions locally. Since they are lexically invisible, they cannot be called with non-sensical initial values, e.g., for accumulators. Recursive equations offer no such linguistic support: auxiliary functions must be written as extra global equations.

In practice: Programmers tend to follow Perlis's and Turner's advices and stay away from extremes, adopting a partly lifted / partly dropped style. In addition, recursive equations are often provided modularity through an explicit module system ("scripts" in Miranda), which is a form of separate block structure.

Efficiency at compile time: We have observed that with most implementations, block-structured programs are faster to compile than the corresponding recursive equations. This is especially true for automatically produced recursive equations, which are often afflicted with dozens of parameters. For example, because it uses lightweight symbolic values, a partial evaluator such as Pell-Mell tends to produce recursive equations with spectacular arities [16, 29]. Compiling these residual programs de facto becomes a bottleneck because compilers are often tuned to typical handwritten programs. For example, Chez Scheme and Standard ML of New Jersey handle functions with few parameters better than functions with many parameters.

Efficiency at run time: We have also observed that with most implementations, block-structured programs are faster to run than the corresponding recursive equations. And indeed the procedure-calling conventions of several processors handle procedures with few parameters better than procedures with many parameters.

Simplicity of representation: Block-structured programs are more implicit than recursive equations, which are explicitly passed all of their live variables as actual parameters. Recursive equations are thus simpler to process.

Synthesis: There seems to be no consensus about the relative efficiencies of block-structured programs and recursive equations. For one example, Turner wrote against nested blocks because they are "slower to compile," even though block-structured programs appear to be faster to run in Miranda. For one converse example, the system code in Caml was deliberately block-structured to make it run faster (Xavier Leroy, personal communication to the first author, spring 1996).

1.3 Lambda-lifting and lambda-dropping

Lambda-lifting and lambda-dropping resolve the programming tension between a monolithic, block-structured style and a decentralized style of recursive equations: they transform programs expressed in either style into programs expressed in the other one. More generally, programs that are written in a partly lifted / partly dropped style can be completely lambda-lifted or lambda-dropped.

For example, recursive equations can be efficiently implemented on the G-machine. This led Hughes, Johnsson, and Peyton Jones, in the mid 80's, to devise lambda-lifting as a meaning-preserving transformation from block-structured programs to recursive equations [21, 23, 34].

Recursive equations also offer a convenient format in Mix-style partial evaluation [26]. Indeed, modern partial evaluators such as Schism and Similix lambda-lift source programs before specialization [9, 11].
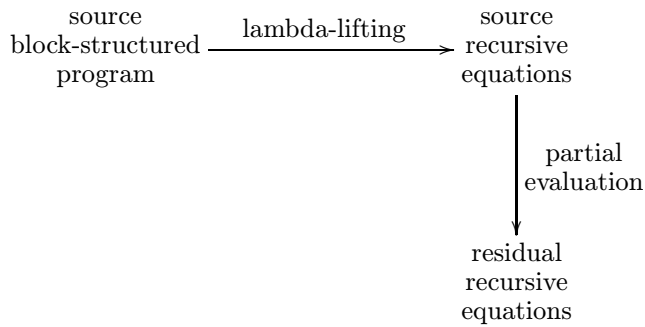
As a result, residual programs are also expressed as recursive equations. If partial evaluation is to be seen as a source-to-source program transformation, however, residual programs should be block structured. To this end, we devised lambda-dropping: the transformation of recursive equations into block-structured and lexically scoped programs.
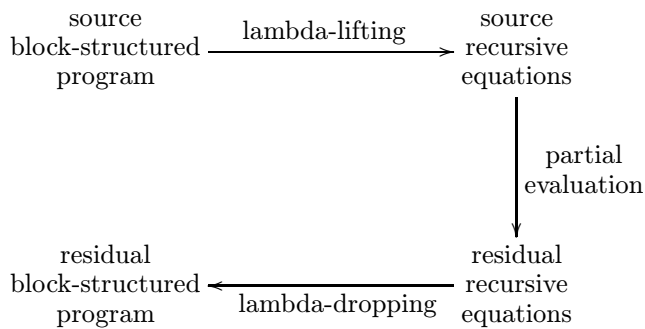
The anatomy of lambda-lifting and lambda-dropping: We present lambda-lifting and lambda-dropping in a symmetric way, using four transformations: parameter-lifting, block-floating, block-sinking and parameter-dropping.

$$
\begin{array}{ccc}
 & \text{scope-insensitive} & \\
 & \text{recursive equations} & \\
 \text{block} & \uparrow\downarrow & \text{block} \\
 \text{floating} & & \text{sinking} \\
\text{lambda} & \text{scope-insensitive} & \text{lambda} \\
\text{lifting} & \text{block-structured program} & \text{dropping} \\
 \text{parameter} & \uparrow\downarrow & \text{parameter} \\
 \text{lifting} & & \text{dropping} \\
 & \text{scope-sensitive} & \\
 & \text{block-structured program} &
\end{array}
$$
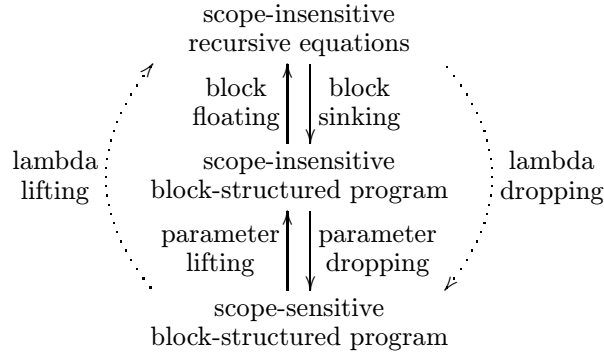
Parameter lifting makes a program scope-insensitive by passing extra variables to each function to account for variables occurring free further down the call path. Block floating eliminates block structure by globalizing each block, making each of its locally defined functions a global recursive equation. Block sinking restores block structure by localizing (strongly connected) groups of equations in the call graph. Parameter dropping exploits scope by not passing variables whose end use occurs in the scope of their initial definition.

1.4   Overview

The rest of this article is organized as follows. Section 2 addresses first-order programs. Section 3 generalizes the transformations to higher-order programs. Section 4 puts lambda-lifting and lambda-dropping into perspective, and Section 5 lists some applications. Section 6 reviews related work and Section 7 concludes.

Conventions and notations: Throughout, we assume variable hygiene, i.e., that no name clashes can occur. Also, we refer to letrec expressions as "blocks."

2   First-Order Programs

Section 2.1 provides a roadmap of lambda-lifting and lambda-dropping: we first review the two steps of lambda-lifting (Section 2.1.1) and the two steps of lambda-dropping (Section 2.1.2). Section 2.2 specifies lambda-lifting and illustrates it with an example. Section 2.3 outlines how to reverse lambda-lifting. Section 2.4 specifies lambda-dropping and illustrates it by revisiting the example of Section 2.2. Section 2.5 summarizes.

$$
\begin{array}{llll}
p \in \mathsf{Program} & ::= & \{d_1, \ldots, d_m\} \\
d \in \mathsf{Def} & ::= & f \equiv \lambda(v_1,\, ...,\, v_n).e \\
e, \cancel{e} \in \mathsf{Exp} & ::= & \ell \\
& | & v \\
& | & f\,(e_1,\, ...,\, e_n) \\
& | & \mathrm{LetRec}\,\{d_1, \ldots, d_k\}\,e_0 & \text{where } k > 0 \\
\ell \in \mathsf{Literal} \\
v, \cancel{v} \in \mathsf{Variable} \\
f \in \mathsf{FunctionName} \cup \mathsf{PredefinedFunction}
\end{array}
$$

<center>Figure 1: Simplified syntax of first-order programs.</center>

2.1 Introduction

Our first-order programs conform to the syntax of Figure 1. To simplify the presentation, we leave out conditional expressions and (non-recursive) let blocks. A program is represented as a set of functions. Each function may define local functions in blocks. Function names can only occur in function position. Applications involve either named functions or predefined operators (arithmetic and the like). The predefined operators have no influence on the correctness of our definitions and algorithms, and are included for use in examples only. Also, for the purpose of program transformations, some expressions and formal arguments are marked with a horizontal line. To make the informal descriptions more concise, we sometimes refer to the set of global functions as a "block." We use FV to denote the set of free variables declared as formal parameters, and FF to denote the set of free variables declared as function names.

2.1.1 The basics of lambda-lifting

Lambda-lifting is achieved by applying the following two transformations to a block-structured program:

1. *Parameter lifting.* The list of formal parameters of each lambda-abstraction is expanded with the free variables of the lambda, and likewise for the list of arguments at each application site, with the same variables.

2. *Block floating.* Local functions in a block are moved to the enclosing scope level whenever they do not use locally defined functions and do not refer to local free variables.

Example: the power function. To illustrate lambda-lifting, we use the block-structured definition of the power function shown in Figure 2(a). The

<center>10</center>

(a) The power function:

```
(define power
  (lambda (n e)
    (letrec ([loop (lambda (x)
                     (if (zero? x)
                         1
                         (* n (loop (1- x)))))])
      (loop n))))
```

(b) The lambda-lifted power function:

```
(define power
  (lambda (n e)
    (loop e n)))
(define loop
  (lambda (n0 x)
    (if (zero? x)
        1
        (* n0 (loop n0 (1- x))))))
```

Figure 2: A simple example: lambda-lifting the power function

variable `n` is free in the local definition of `loop`. Parameter lifting replaces this free variable by a new formal parameter to `loop` named `n0` (to preserve variable hygiene), and passes `n` or `n0` as appropriate in each call to `loop`. The function `loop` no longer has any free variables. Block floating moves the `loop` function outwards, making it a global function. In Figure 2(b), the two resulting global functions have no block structure: they are recursive equations.

2.1.2   The basics of lambda-dropping

Symmetrically to lambda-lifting, lambda-dropping is achieved by applying the following two transformations to a set of recursive equations:

1. *Block sinking.* Any set of functions that is referenced by a single function only is made local to this function. This is achieved by moving the set inside the definition of the function.

$$\frac{\vdash_{\mathsf{PL}}^{\mathsf{Def}} d_1 \quad ... \quad \vdash_{\mathsf{PL}}^{\mathsf{Def}} d_m}{\vdash_{\mathsf{PL}} \{d_1, ..., d_m\}} \qquad\qquad \frac{\{v_1, ..., v_n\} \vdash_{\mathsf{PL}}^{\mathsf{Exp}} e}{\vdash_{\mathsf{PL}}^{\mathsf{Def}} f \equiv \lambda(v_1, ..., v_n).e}$$

$$\frac{}{P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} \ell} \qquad \frac{v \in P}{P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} v} \qquad \frac{P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} e_1 \quad ... \quad P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} e_n}{P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} f\,(e_1, ..., e_n)}$$

$$\frac{\vdash_{\mathsf{PL}}^{\mathsf{Def}} d_1 \quad ... \quad \vdash_{\mathsf{PL}}^{\mathsf{Def}} d_k \quad P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} e_0}{P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} \mathrm{LetRec}\,\{d_1, ..., d_k\}\,e_0}$$

Figure 3: Specification of a parameter-lifted program.
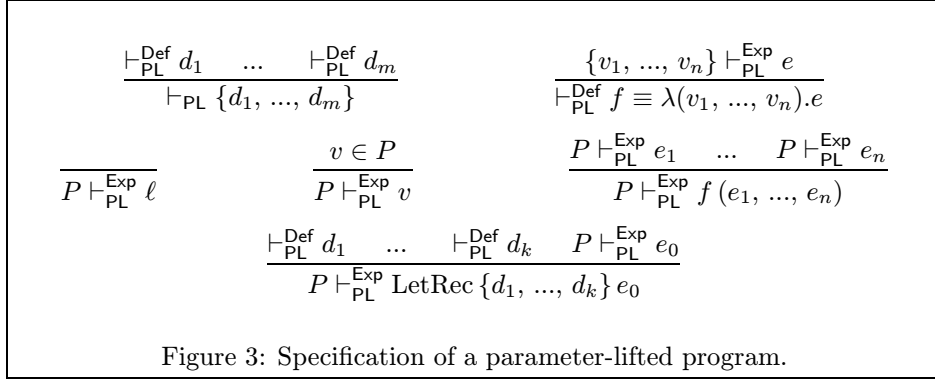
2. *Parameter dropping.* A function with a formal parameter that is bound to the same variable in every invocation can potentially be parameter-dropped. If the variable is lexically visible at the definition site of the function, the formal parameter can actually be dropped. A formal parameter is dropped from a function by removing the formal parameter from the parameter list, removing the corresponding argument from all invocations of the function, and substituting the name of the variable for the name of the formal parameter throughout the body of the function.

Example: the power function, revisited. To illustrate lambda-dropping, we use the recursive equations of the power function shown in Figure 2(b). The function loop is used only by the function power. Block sinking inserts a block defining loop into the body of power, removing the global definition of loop. The now local function loop has an unchanging formal parameter n0 that is bound in every invocation to the value of the formal parameter n of the enclosing function power. Parameter dropping removes n0 as a formal parameter and replaces each use of it with n. The result is the program of Figure 2(a).

2.2 Lambda-lifting

We now provide a detailed description of lambda-lifting. Lambda-lifting simplifies program structure first by lifting free variables into formal parameters and then by floating local functions outwards to make them global.

2.2.1 Characterization of lambda-lifted programs

We define a lambda-lifted program as a program that has been both "parameter lifted" and "block floated."

$$\frac{\vdash_{\mathsf{BF}}^{\mathsf{Def}} d_1 \quad ... \quad \vdash_{\mathsf{BF}}^{\mathsf{Def}} d_m}{\vdash_{\mathsf{BF}} \{d_1, ..., d_m\}} \qquad \frac{\{v_1, ..., v_n\} \vdash_{\mathsf{BF}}^{\mathsf{Exp}} e}{\vdash_{\mathsf{BF}}^{\mathsf{Def}} f \equiv \lambda(v_1, ..., v_n).e}$$

$$\overline{P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} \ell} \qquad\qquad \overline{P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} v} \qquad \frac{P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} e_1 \quad ... \quad P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} e_n}{P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} f(e_1, ..., e_n)}$$

$$\frac{\vdash_{\mathsf{BF}}^{\mathsf{Def}} d_1 \quad ... \quad \vdash_{\mathsf{BF}}^{\mathsf{Def}} d_k \quad P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} e_0}{P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} \mathrm{LetRec}\,\{d_1, ..., d_k\}\,e_0}$$

if for each strongly connected component C in the call graph of $\{d_1, ..., d_m\}$, $\mathrm{FV}(C) \cap P \neq \emptyset$ or C dominates a strongly connected component C' such that $\mathrm{FV}(C') \cap P \neq \emptyset$.

Figure 4: Specification of a block-floated program.
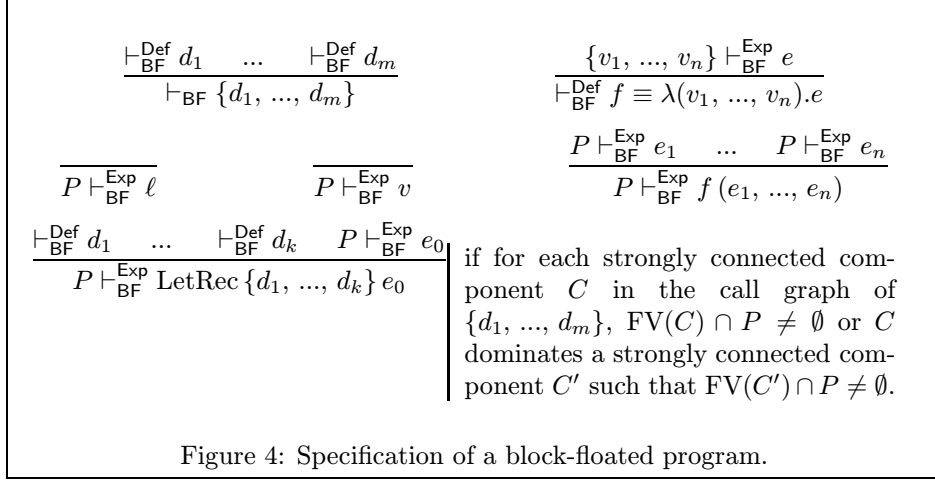
Parameter-lifted programs: A block-structured program is completely parameter-lifted if none of its functions has any free variables. Thus, all free variables occurring in the body of any function must be declared as formal parameters of this function. Figure 3 formally defines parameter-lifted programs. A program p is parameter-lifted whenever the judgment

$$\vdash_{\mathsf{PL}} p$$

is satisfied. A declaration d is parameter-lifted whenever the judgment

$$\vdash_{\mathsf{PL}}^{\mathsf{Def}} d$$

is satisfied. And an expression e whose closest enclosing formal parameters are denoted by P is parameter-lifted whenever the judgment

$$P \vdash_{\mathsf{PL}}^{\mathsf{Exp}} e$$

is satisfied.

Block-floated programs: A block-structured program is completely block-floated when each function f either is global or contains free variables (defined by an enclosing function) or free functions (defined in the same block) such that it cannot float to an enclosing block. Figure 4 formally defines block-floated programs. A program p is block-floated whenever the judgment

$$\vdash_{\mathsf{BF}} p$$

is satisfied. A declaration d is block-floated whenever the judgment

$$\vdash_{\mathsf{BF}}^{\mathsf{Def}} d$$

13

is satisfied. And an expression e whose closest enclosing formal parameters are denoted by P is block-floated whenever the judgment

$$P \vdash_{\mathsf{BF}}^{\mathsf{Exp}} e$$

is satisfied. The key to Figure 4 is the side condition for blocks: based on the observation that mutually recursive functions should float together, we partition the definition $\{d_1, ..., d_k\}$ into strongly connected components of their call graph. Such strongly connected components cannot float above the enclosing abstraction $f \equiv \lambda(v_1, ..., v_n).e$ if they dominate a strongly connected component (possibly themselves) where at least one v_i occurs free. (Reminder: in a graph with root r, a node a dominates a node b if all paths from r to b go through a.) For example, in the lambda-abstraction

$$\lambda(x). \text{ letrec } \begin{array}{rcl} f & = & \lambda().x \\ g & = & \lambda().f() \end{array}$$
$$\text{in } ...$$

f cannot float because x occurs free in its definition and g cannot float because it refers to f. (g dominates f in the call graph.)

If the source program is parameter-lifted, then a block-floated program degenerates to recursive equations.

2.2.2 Lambda-lifting algorithm

We consider Johnsson's algorithm, which first performs parameter lifting and then performs block floating [4, 23, 24].

Figure 5 describes the parameter-lifting part of Johnsson's algorithm. To parameter-lift a program, all free variables of a function must be explicitly passed as parameters to this function. Thus, all callers of the function must be provided with these variables as additional arguments. The algorithm traverses the program while building a set of solutions. A solution associates each function with the minimal set of additional variables it needs to be passed as arguments. Each block gives rise to a collection of set equations that describe which variables should be passed as arguments to the functions defined by the block. The equations are mutually recursive for mutually recursive functions, and they are thus solved by fixed-point iteration. The set of solutions is extended with the solution of the set equations, and is then used to analyze the body of the block and the body of each local function.

Figure 6 describes the block-floating part of Johnsson's algorithm. In the general case, the floating of functions outwards through the block-structure is constrained by the scope of formal parameters and function names. However, a parameter-lifted program is scope-insensitive, meaning that no function depends on being defined within the scope of the formal parameters of some other function. Furthermore, a global function is visible to all other global functions,

parameterLiftProgram :: Program → Program
parameterLiftProgram p = map (parameterLiftDef $\emptyset$) p

parameterLiftDef :: Set(FunName,Set(Variable)) → Def → Def
parameterLiftDef S $(f \equiv \lambda(v_1, ..., v_n).e)$ =
   applySolutionToDef S $(f \equiv \lambda(v_1, ..., v_n).(\text{parameterLiftExp } S\ e))$

parameterLiftExp :: Set(FunName,Set(Variable)) → Exp → Exp
parameterLiftExp S $(\ell) = \ell$
parameterLiftExp S $(v) = v$
parameterLiftExp S $(f\,(e_1, ..., e_n))$ =
   applySolutionToExp S $(f\,(\text{map parameterLiftExp } (e_1, ..., e_n)))$
parameterLiftExp S (LetRec $\{d_1, ..., d_k\}\, e_0$) =
   foreach $(f_i \equiv l_i) \in \{d_1, ..., d_k\}$ **do**
      $V_{f_i} := \text{FV}(f_i \equiv l_i)$;
      $F_{f_i} := \text{FF}(f_i \equiv l_i)$
   foreach $F_{f_i} \in \{F_{f_1}, \ldots, F_{f_k}\}$ **do**
      foreach $(g, V_g) \in S$ such that $g \in F_{f_i}$ **do**
         $V_{f_i} := V_{f_i} \cup V_g$;
         $F_{f_i} := F_{f_i} \backslash \{g\}$
   fixpoint over $\{V_{f_1}, \ldots, V_{f_k}\}$ **by**
      foreach $F_{f_i} \in \{F_{f_1}, \ldots, F_{f_k}\}$ **do**
         foreach $g \in F_{f_i}$ **do**
            $V_{f_i} := V_{f_i} \cup V_g$
   let $S' = S \cup \{(f_1, V_{f_1}), \ldots, (f_k, V_{f_k})\}$
      $f_s = \text{map (parameterLiftDef } S')\ \{d_1, \ldots, d_k\}$
      $e_0' = \text{parameterLiftExp } S'\ e_0$
   in (LetRec $f_s\, e_0'$)

applySolutionToDef :: Set(FunName,Set(Variable)) → Def → Def
applySolutionToDef $\{\ldots, (f, \{v_1, ..., v_n\}), \ldots\}$ $(f \equiv \lambda(v_1', ..., v_n').e)$ =
   $(f \equiv \lambda(v_1, ..., v_n, v_1', ..., v_n').e)$
applySolutionToDef S $d = d$

applySolutionToExp :: Set(FunName,Set(Variable)) → Exp → Exp
applySolutionToExp $\{\ldots, (f, \{v_1, ..., v_n\}), \ldots\}$ $(f\,(e_1, ..., e_n))$ =
   $(f\,(v_1, ..., v_n, e_1, ..., e_n))$
applySolutionToExp S $e = e$

Figure 5: Parameter lifting – free variables are made parameters.

```
    blockFloatProgram :: Program → Program
    blockFloatProgram p = foldr makeUnion ∅ (map blockFloatDef p)

    blockFloatDef :: Def → (Set(Def),Def)
    blockFloatDef (f ≡ λ(v₁, ..., vₙ).e) = let (F_new,e′) = blockFloatExp e
                                            in (F_new, f ≡ λ(v₁, ..., vₙ).e′)

    blockFloatExp :: Exp → (Set(Def),Exp)
    blockFloatExp (ℓ) = (∅,ℓ)
    blockFloatExp (v) = (∅,v)
    blockFloatExp (f (e₁, ..., eₙ)) = let x = map blockFloatExp (e₁, ..., eₙ)
                                         F_new= foldr (∪) ∅ (map fst x)
                                         (e′₁, ..., e′ₙ) = map snd x
                                      in (F_new,f (e′₁, ..., e′ₙ))
    blockFloatExp (LetRec {d₁, ..., dₖ} e₀) =
                let x = map blockFloatDef {d₁, ..., dₖ}
                    b = blockFloatExp e₀
                    F_new = foldr makeUnion ∅ x
                in (F_new ∪ (fst b), snd b)

    makeUnion :: (Set(Def),Def) → Set(Def) → Set(Def)
    makeUnion (F_new, d) S = F_new ∪ {d} ∪ S
```

Figure 6: Block floating – flattening of block structure.

so references to function names are trivially resolved. Programs are parameter-lifted before block-floating, and all functions can be declared as global, so a very simple algorithm can be used for block-floating: 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 functions. The resulting program can simply be characterized as having no local blocks.

Other styles and implementations of lambda-lifting exist. We review them in Section 6.2.

2.2.3 Example

Figure 7 displays a textbook example that Andrew Appel borrowed from the Static Single Assignment (SSA) community to make his point that SSA is functional programming [3, Chapter 19]. In this example, two mutually recursive

```
    (define main
      (lambda (i1 j1 k1)
        (letrec ([f2 (lambda (j2 k2)
                       (if (< k2 100)
                           (letrec ([f7 (lambda (j4 k4)
                                          (f2 j4 k4))])
                             (if (< j2 20)
                                 (f7 i1 (+ k2 1))
                                 (f7 k2 (+ k2 1))))
                           j2))])
          (f2 j1 k1))))
  ;(main 1 1 0)
```

Figure 7: A textbook example.

functions, f2 and f7, are located in a global function, main. The formal param-
eter i1 is free in the bodies of f2 and f7.

   In this section, we describe the process of lambda-lifting this program ac-
cording to the algorithms of Figures 5 and 6.

Parameter lifting: The parameter-lifting algorithm of Figure 5 is used on the
example program of Figure 7 by applying the function "parameterLiftProgram"
to the function main:

parameterLiftProgram {(define main ...)}: We apply "parameterLiftDef" to
      main, with an empty set of solutions.

parameterLiftDef $\emptyset$ (define main ...): We descend recursively into the body
      of the function.

parameterLiftExp $\emptyset$ (letrec ([f2 ...]) ...): We create two sets $V_{f2} = \{i1\}$
      and $F_{f2} = \emptyset$, which remain unchanged through the first fixed-point it-
      eration. We extend the empty set of solutions with $(f2, \{i1\})$ and then
      continue recursively on f2 and the body of the letrec.

parameterLiftDef S [f2 (lambda (j2 k2) ...)]: The set of solutions $S = \{(f2,
      \{i1\})\}$ directs us to extend the list of formal parameters of f2 to (i1 j2
      k2). We continue recursively on the body of the function.

parameterLiftExp S (letrec ([f7 ...]) ...): We create two sets $V_{f7} = \emptyset$ and
      $F_{f7} = \{f2\}$, and since f2 is described in the set of solutions $S = (f2, \{i1\})$,
      we extend V_{f7} accordingly to $V_{f7} = \{i1\}$. The set remains unchanged

17

```
    (define main
      (lambda (i1 j1 k1)
        (letrec ([f2 (lambda (x1 j2 k2)
                       (if (< k2 100)
                           (letrec ([f7 (lambda (y1 j4 k4)
                                          (f2 y1 j4 k4))])
                             (if (< j2 20)
                                 (f7 x1 x1 (+ k2 1))
                                 (f7 x1 k2 (+ k2 1))))
                           j2))])
          (f2 i1 j1 k1))))
    ;(main 1 1 0)
```

Figure 8: The program of Figure 7, after parameter lifting.

through the first fixed-point iteration. We extend the set of solutions with $(\texttt{f7}, \{\texttt{i1}\})$. We continue recursively on $\texttt{f7}$ and the body of the letrec.

parameterLiftDef S [f7 (lambda (j4 k4) ...)]: The set of solutions $S = \{(\texttt{f7}, \{\texttt{i1}\}), ...\}$ directs us to extend the list of formal parameters of $\texttt{f7}$ to (i1 j4 k4). We continue recursively on the body of the function.

parameterLiftExp S (f2 j4 k4): The set of solutions $S = \{(\texttt{f2}, \{\texttt{i1}\}), ...\}$ directs us to insert $\texttt{i1}$ as the first argument of $\texttt{f2}$.

parameterLiftExp S (f7 i1 (...)): The set of solutions $S = \{(\texttt{f7}, \{\texttt{i1}\}), ...\}$ directs us to insert $\texttt{i1}$ as the first argument of $\texttt{f7}$.

parameterLiftExp S (f7 k2 (...)): The set of solutions $S = \{(\texttt{f7}, \{\texttt{i1}\}), ...\}$ directs us to insert $\texttt{i1}$ as the first argument of $\texttt{f7}$.

parameterLiftExp S (f2 j1 k1): The set of solutions $S = \{(\texttt{f2}, \{\texttt{i1}\}), ...\}$ directs us to insert $\texttt{i1}$ as the first argument of $\texttt{f2}$.

The result, after alpha-renaming to ensure variable hygiene, is displayed in Figure 8.

Block floating: The block-floating algorithm of Figure 6 is used on the scope-insensitive program of Figure 8 by applying the function "blockFloatProgram" to the function main. This function traverses the program, gathering local definitions and removing blocks. The result is displayed in Figure 9.

18

```
    (define main
      (lambda (i1 j1 k1)
        (f2 i1 j1 k1)))

    (define f2
      (lambda (x1 j2 k2)
        (if (< k2 100)
            (if (< j2 20)
                (f7 x1 x1 (+ k2 1))
                (f7 x1 k2 (+ k2 1)))
            j2)))

    (define f7
      (lambda (y1 j4 k4)
        (f2 y1 j4 k4)))

    ;(main 1 1 0)
```

Figure 9: The program of Figure 8, after block floating.

2.3 Reversing lambda-lifting

As described in Section 2.2, lambda-lifting first makes functions scope-insensitive, by expanding their list of formal parameters, and then proceeds to make all functions global through block-floating. To reverse lambda-lifting, we can make the appropriate global functions local, and then make them scope-sensitive by reducing their list of formal parameters. (Reminder: In our simplified syntax, we always generate letrec blocks, even when a let block would suffice.)

Localizing a function in a block moves it into the context where it is used. Once a function is localized, it is no longer visible outside the block. In the abstract-syntax tree, localization thus stops at the closest common context of all the uses. Going any further would entail code duplication.

2.3.1 Block sinking

To reverse the effect of lambda-lifting, let us examine the program of Figure 9, which was lambda-lifted in Section 2.2.3. The main function of the program is `main`. The two other functions are used only by `main`, and are thus localizable to `main`. We replace the body of `main` with a block declaring these functions and having the original body of `main` as its body.

```
    define main = letrec f2 = ...
                         f7 = ...
                  in ...
```

19

We can see that the body of `f2` refers to the function `f7`. The function `main`, however, does not use `f7`. Therefore it makes sense to localize `f7` to `f2`.

```
define r = letrec f2 = letrec f7 = ...
                            in ...
              in ...
```

The functions of the program cannot be localized any further. The block structure of this program is identical to that of the original (of Figure 7). In Section 1.2, we made the point that one tends to write partly lifted / partly dropped programs in practice. In such cases, lambda-dropping a program would create more block structure than was in the original program. We discuss efficiency issues in Section 4.6.
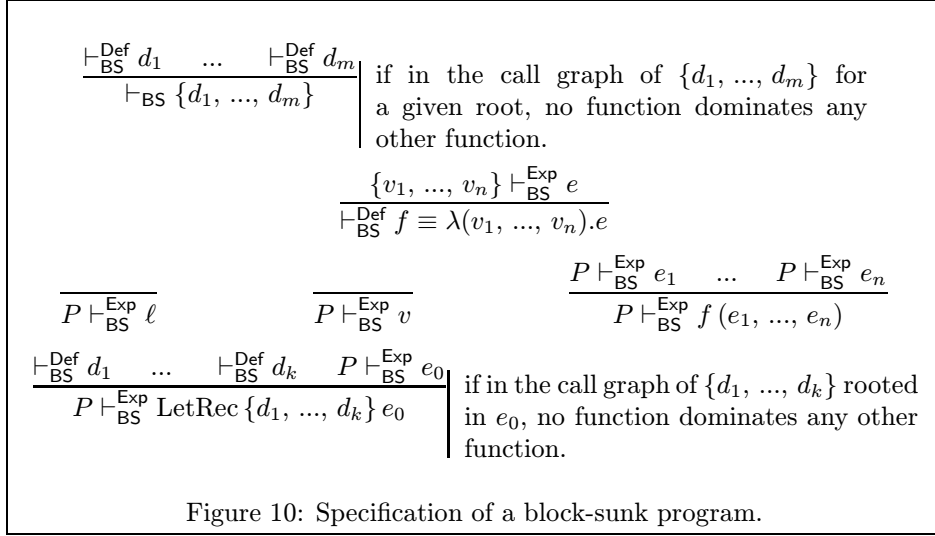
2.3.2 Parameter dropping

To reverse the parameter lifting performed during lambda-lifting, we need to determine the origin of each formal parameter. The mutually recursive functions `f2` and `f7` both pass the variables `x1` and `y1` to each other, as their first formal parameter. These formal parameters always correspond to the variable `i1` of `main`. Since `i1` is now visible where the two functions are declared, there is no need to pass it around as a parameter. We can simply remove the first formal parameter from the declaration of both functions and refrain from passing it as argument at each application site. As for the other formal parameters, they are bound to different arguments at certain application sites, and thus they are not candidates for parameter dropping.

Figure 7 displays the final result of our reversal process, which is also the program we started with to illustrate lambda-lifting. If this program had contained parameter-droppable formal parameters to start with, then the functions of our final program would have had fewer formal parameters.

2.4 Lambda-dropping

We now specify lambda-dropping more formally. Lambda-dropping a program minimizes parameter passing, and serves in principle as an inverse of lambda-lifting. Function definitions are localized maximally using lexically scoped block structure. Parameters made redundant by the newly created scope are eliminated.

This section makes extensive use of graphs. The graph functions are described in the appendix, in Figure 24. In particular, the dominator tree of a graph is computed for both stages. (Reminder: in the dominator tree of a graph, a node a precedes a node b if a dominates b in the graph.)

$$\frac{\vdash^{\mathsf{Def}}_{\mathsf{BS}} d_1 \quad ... \quad \vdash^{\mathsf{Def}}_{\mathsf{BS}} d_m}{\vdash_{\mathsf{BS}} \{d_1, ..., d_m\}}$$ if in the call graph of $\{d_1, ..., d_m\}$ for a given root, no function dominates any other function.

$$\frac{\{v_1, ..., v_n\} \vdash^{\mathsf{Exp}}_{\mathsf{BS}} e}{\vdash^{\mathsf{Def}}_{\mathsf{BS}} f \equiv \lambda(v_1, ..., v_n).e}$$

$$\frac{}{P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} \ell} \qquad \frac{}{P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} v} \qquad \frac{P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} e_1 \quad ... \quad P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} e_n}{P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} f(e_1, ..., e_n)}$$

$$\frac{\vdash^{\mathsf{Def}}_{\mathsf{BS}} d_1 \quad ... \quad \vdash^{\mathsf{Def}}_{\mathsf{BS}} d_k \quad P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} e_0}{P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} \mathrm{LetRec} \{d_1, ..., d_k\} e_0}$$ if in the call graph of $\{d_1, ..., d_k\}$ rooted in e_0, no function dominates any other function.

Figure 10: Specification of a block-sunk program.

2.4.1 Characterization of lambda-dropped programs

We define a lambda-dropped program as a program that has been both "block sunk" and "parameter dropped."

Block-sunk programs: A block-structured program is completely block sunk when no function can be sunk into some definition in which it is used. A function that is used by at least two other functions from the same block cannot by itself be sunk into either function. Likewise, a group of mutually recursive functions that are used by at least two other functions from the same block, cannot be sunk into either function. In addition, no function that is used in the body of a block can be sunk into a function defined in this block.

Figure 10 formally defines block-sunk programs. A program p is block-sunk whenever the judgment
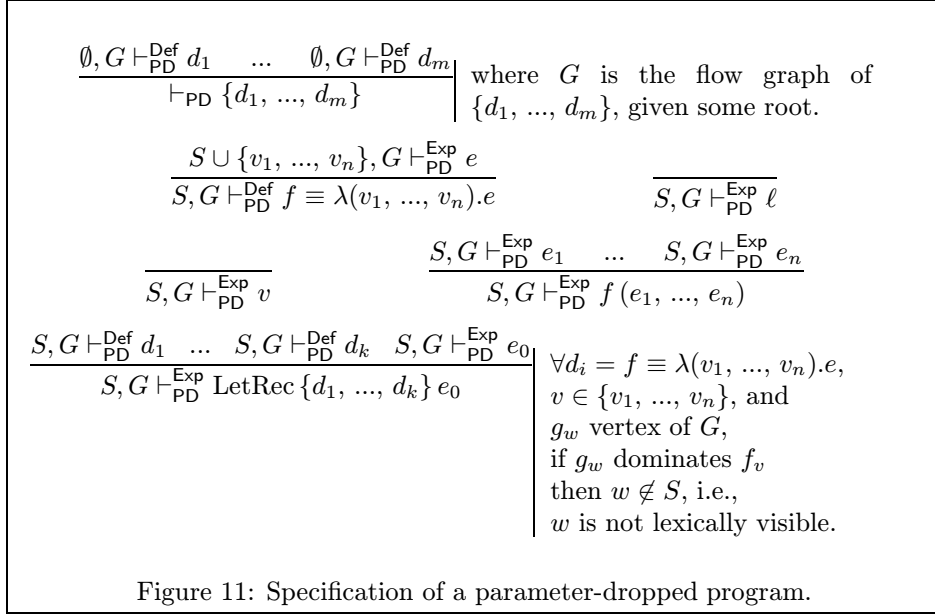
$$\vdash_{\mathsf{BS}} p$$

is satisfied. A declaration d is block-sunk whenever the judgment

$$\vdash^{\mathsf{Def}}_{\mathsf{BS}} d$$

is satisfied. And an expression e whose closest enclosing formal parameters are denoted by P is block-sunk whenever the judgment

$$P \vdash^{\mathsf{Exp}}_{\mathsf{BS}} e$$

is satisfied.

21

$$\frac{\emptyset, G \vdash_{\mathsf{PD}}^{\mathsf{Def}} d_1 \quad ... \quad \emptyset, G \vdash_{\mathsf{PD}}^{\mathsf{Def}} d_m}{\vdash_{\mathsf{PD}} \{d_1, ..., d_m\}} \Bigg| \quad \text{where } G \text{ is the flow graph of } \{d_1, ..., d_m\}, \text{ given some root.}$$

$$\frac{S \cup \{v_1, ..., v_n\}, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} e}{S, G \vdash_{\mathsf{PD}}^{\mathsf{Def}} f \equiv \lambda(v_1, ..., v_n).e} \qquad\qquad \overline{S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} \ell}$$

$$\overline{S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} v} \qquad\qquad \frac{S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} e_1 \quad ... \quad S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} e_n}{S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} f(e_1, ..., e_n)}$$

$$\frac{S, G \vdash_{\mathsf{PD}}^{\mathsf{Def}} d_1 \quad ... \quad S, G \vdash_{\mathsf{PD}}^{\mathsf{Def}} d_k \quad S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} e_0}{S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} \mathrm{LetRec}\, \{d_1, ..., d_k\}\, e_0} \Bigg| \begin{array}{l} \forall d_i = f \equiv \lambda(v_1, ..., v_n).e, \\ v \in \{v_1, ..., v_n\}, \text{ and} \\ g_w \text{ vertex of } G, \\ \text{if } g_w \text{ dominates } f_v \\ \text{then } w \notin S, \text{ i.e.,} \\ w \text{ is not lexically visible.} \end{array}$$

Figure 11: Specification of a parameter-dropped program.

Parameter-dropped programs: A block-structured program is completely parameter-dropped when no variable whose scope extends into a function definition is passed as a parameter as well in all invocations of this function. In the flow graph of the program, a formal parameter that dominates some other formal parameter occurs in every path from the root to this formal parameter. Thus, a program is completely parameter-dropped when no formal parameter that dominates a formal parameter of some other function is visible to this function.

Figure 11 formally defines parameter-dropped programs. A program p is parameter-dropped whenever the judgment

$$\vdash_{\mathsf{PD}} p$$

is satisfied. Let G denote its flow graph for some given root. A declaration d occurring in the scope of the variables contained in the set S is parameter-dropped whenever the judgment

$$S, G \vdash_{\mathsf{PD}}^{\mathsf{Def}} d$$

is satisfied. And an expression e occurring in the scope of the variables contained in the set S is parameter-dropped whenever the judgment

$$S, G \vdash_{\mathsf{PD}}^{\mathsf{Exp}} e$$

is satisfied.

2.4.2 Lambda-dropping algorithm

The lambda-dropping algorithm works on any kind of program, but is perhaps most easily understood as operating on lambda-lifted programs. It is composed of two stages, block sinking and parameter dropping.

Figure 12 describes the block-sinking part of lambda-dropping. Two entry points are provided. The first retains the "main" function and any unused functions as global functions, and the other allows the caller to specify a set of functions to retain as global rather than the default "main" function. To block-sink a program, any set of functions that are used solely within some other function must be moved into a block local to this function. We use a call graph to describe the dependencies between the functions of the program. A function f that calls some other function g depends on g. A function g that is dominated by some other function f in the call graph can only be called from the definition of f.

The dominator tree of the call graph is thus a tree that induces a new block structure into the program. In the new program, each function is declared locally to its predecessor.

Figure 13 describes the parameter-dropping part of lambda-dropping. We use the notation f_x to indicate the formal parameter x of the function f, as explained in the appendix. To parameter-drop a program, any formal parameter of a function that in every invocation of the function is bound to a variable whose scope extends to the declaration of the function, is considered redundant and can be removed. If in the flow graph of the program a formal parameter w of a function g is dominated by some other formal parameter v of a function f, then w will always denote the value of v. If the scope of v extends to the declaration of g, then w can be removed and v can be substituted for w throughout the body of g.

Several formal parameters $\{v_1, ..., v_n\}$ (where v_i of f_i dominates v_{i+1} of f_{i+1}) may dominate the parameter-droppable formal parameter w of g. Let v_j of the function f_j be the first node in this list whose scope extends to the declaration of g, meaning that g is declared within f_j. Every function whose parameters are dominated by v_j is declared within f_j, since they could not otherwise invoke the function g. The scope of v_j thus extends to all the declarations it dominates, and thereby makes redundant the formal parameters that it dominates. Thus, the algorithm need only parameter-drop the first parameter v_j, since this will have the same effect as iteratively dropping all of the variables $\{v_j, \ldots, v_n\}$.

For simplicity, we have defined lambda-dropping over recursive equations. In general, however, lambda-dropping and thus block-sinking can be given block-structured programs to increase their nesting.

```
blockSinkProgram :: Program → Program
blockSinkProgram p = blockSinkProgram2 p {main}

blockSinkProgram2 :: Program → Set(FunName) → Program
blockSinkProgram2 p globalFns =
   let buildCallGraph :: () → (Graph(Def),Def)
       buildCallGraph () =
          let (G as (V, E)) = ref (∅, ∅)
              root = Graph.addNode G "root"
          in foreach ((d as (f ≡ l)) ∈ p) do
                 foreach f' ∈ (FF(d)\{f}) do
                    let (d' as (f' ≡ l')) ∈ p in Graph.addEdge G (d, d')
                 foreach f ∈ globalFns do
                    let (d as (f ≡ l)) ∈ p in Graph.addEdge G (root, d)
                 foreach d ∈ V do
                    if (∀d' ∈ V : (d', d) ∉ E) then Graph.addEdge G (root, d)
                 (G, root)
       buildProgram :: (Graph(Def),Def) → Program
       buildProgram (G as (V, E), root) =
          let succ :: Def → Set(Def)
              succ d = {d' ∈ V|(d, d') ∈ E}
              build :: Def → Def
              build (d as f ≡ λ(v₁, ..., vₙ).e) =
                 let S = map build (succ d)
                 in if S = ∅
                    then d
                    else (f ≡ λ(v₁, ..., vₙ).(LetRec S e))
          in map build (succ root)
   in buildProgram (Graph.findDominators (buildCallGraph ()))
```

Figure 12: Block sinking – re-creation of block structure.

parameterDropProgram :: Program → Program
parameterDropProgram p =
 let $(G$ **as** $(V, E), r) =$ Graph.findDominators (Graph.flowGraph p)
  processGlobalDef :: Def → Def
  processGlobalDef = removeMarkedPartsDef
         ∘ (markArgumentsDef ∅)
         ∘ (markFormalsDef ∅)
  markFormalsDef :: Set(Variable) → Def → Def
  markFormalsDef S $(f \equiv \lambda(v_1, ..., v_n).e) =$
   let markFormalsExp :: Set(Variable) → Exp → Exp
    markFormalsExp S $e =$
     . . . *descend recursively, calling* markFormalsDef *on definitions*. . .
    mark :: Variable → (Exp, List(Variable)) → (Exp, List(Variable))
    mark v $(e, s) =$
     if $\exists g_w \in V :$ (Graph.isPath G g_w $f_v) \wedge w \in S$
           $\wedge((r, g_w) \in E \vee \exists h_x \in V : (h_x, g_w) \in E \wedge x \notin S)$
     then $(e[v/w], \cancel{v} :: s)$
     else $(e, v :: s)$
    $(e', s') =$ foldr mark $(e, [\,])$ $(v_1, ..., v_n)$
   in $(f \equiv \lambda s'.($markFormalsExp $(S \cup \{v_1, ..., v_n\})$ $e'))$
  markArgumentsDef :: Set(Def) → Def → Def
  markArgumentsDef S $(f \equiv \lambda(v_1, ..., v_n).e) =$
   let markArgumentsExp :: Set(Def) → Exp → Exp
    markArgumentsExp S $(\ell) = \ell$
    markArgumentsExp S $(v) = v$
    markArgumentsExp S $(f (e_1, ..., e_n)) =$
     let $(f \equiv \lambda(v_1, ..., v_n).e) \in S$
      mark :: (Variable, Exp) → Exp
      mark $(\cancel{v}, e) = \cancel{e}$
      mark $(v, e) =$ markArgumentsExp S e
     in $(f$ (map mark (zip $(v_1, ..., v_n)$ $(e_1, ..., e_n))))$
    markArgumentsExp S (LetRec B $e) =$
     LetRec $\{$map (markArgumentsDef $(S \cup B)) $ $B\}$
       (markArgumentsExp $(S \cup B)$ $e)$
    body $=$ markArgumentsExp $(S \cup \{f \equiv \lambda(v_1, ..., v_n).e\})$ e
   in $(f \equiv \lambda(v_1, ..., v_n).$body$)$
  removeMarkedPartsDef :: Def→Def
  removeMarkedPartsDef $d =$
   . . . *descend recursively, removing all marked parts of the definition* . . .
 **in** map processGlobalDef p

Figure 13: Parameter dropping – removing parameters.

2.4.3 Example (revisited)

In Section 2.2.3, we demonstrated how the program of Figure 7 was lambda-lifted, resulting in the program displayed in Figure 9. Let us lambda-drop this program according to the algorithms of Figures 12 and 13.

Block sinking

The block-sinking algorithm of Figure 12 is used on the program of Figure 9 by applying the function "blockSinkProgram" to the set of global functions. Applying "buildCallGraph" builds the call graph of the program:

$$\text{``root''} \longrightarrow \texttt{main} \longrightarrow \texttt{f2} \rightleftharpoons \texttt{f7}$$
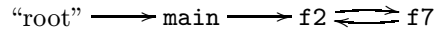
It is straightforward to compute the dominator tree of this graph:

$$\text{``root''} \longrightarrow \texttt{main} \longrightarrow \texttt{f2} \longrightarrow \texttt{f7}$$
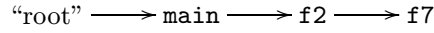
The function "buildProgram" builds a new program by invoking "build" on `main`. This creates the definition of `main` with its successor `f2` as a local function. Likewise, the function `f2` is created with `f7` as a local function. The result is the program of Figure 8.

Parameter dropping

The parameter-dropping algorithm of Figure 13 is used on the scope-insensitive program of Figure 8 by applying the function "parameterDropProgram" to the function `main`. The function first builds the flow graph of the program:
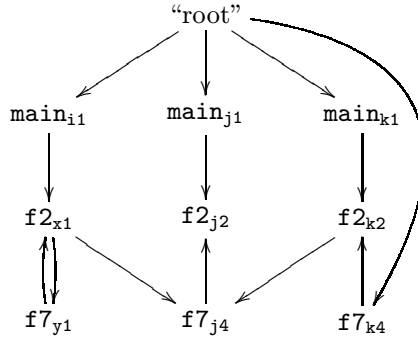
(Reminder: the notation f_x indicates the formal parameter x of the function f.) The dominator tree of this graph is:
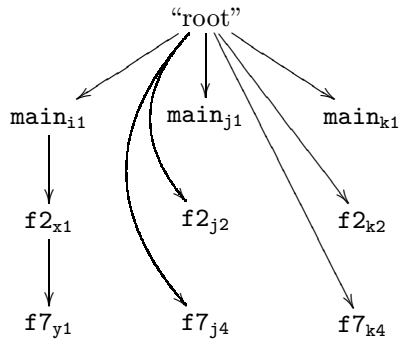
The dominator tree reveals that the only parameter that is passed to other functions on every invocation is i1, which may be bound to x1 and y1. All other parameters are direct successors of the root node, meaning that they are bound to different variables at their application sites.

The function "processGlobalDef" is invoked on a global function, and proceeds in three stages, first marking formals for removal, then the corresponding arguments, and finally removing them. The function "markFormalsDef" is used to traverse the program and mark those formal parameters that can be parameter dropped. The traversal over expressions is trivial, serving only to process all function definitions. Invoking "markFormalsDef" on a function whose formal parameters are dominated only by the root node always works as an identity function, since these variables cannot have been made redundant by other variables. For this reason, we only describe the invocations of this function on those functions whose formal parameters have a predecessor that is not the root node:

markFormalsDef [f2 (lambda (x1 j2 k2) ...)]: Each formal parameter is processed in turn, using the "mark" function. The formal parameter x1 is dominated by the formal parameter i1, which is in the set S of visible formal parameters. Since (i1, x1) is an edge in the dominator tree, x1 is replaced by ~~x1~~ and i1 is substituted for x1 throughout the body of f2. The other two variables are dominated only by the root node and are thus of no interest.

markFormalsDef [f7 (lambda (y1 j2 k2) ...)]: Again, each formal parameter is processed in turn, using the "mark" function. The formal parameter y1 is dominated by the formal parameters i1 and x1, both of which are in the set S of visible formal parameters. However, the variable i1 is a direct successor of the root, so it will be used as a replacement. The variable y1 is replaced by ~~y1~~ and i1 is substituted for y1 throughout the body of f7. Again, the other two variables are dominated only by the root node, and are thus of no interest.

The function "markArgumentsDef" is used to traverse the program and mark those arguments of functions that correspond to formal parameters that were marked as removable. Thus, the first argument of `f2` and `f7` is removed in all invocations. Finally, the function "removeMarkedPartsDef" is used to remove all marked parts of the program, both as formal parameters and as actual parameters. The result is the program of Figure 7.

2.5 Summary

Lambda-lifting of first-order programs is composed of two stages, parameter-lifting and block-floating. Parameter-lifting extends the formal parameters of each function definition, binding all local variables through applications to these variables. The resulting program is scope insensitive. Block-floating then moves its local functions to the outermost level, making them global recursive equations.

Symmetrically, lambda-dropping of first-order programs is also composed of two stages, parameter-dropping and block-sinking. Block-sinking moves global functions that only are used within the body of some other function into this function. Parameter-dropping then removes redundant formal parameters from each function definition and the corresponding actual parameters from each function application.

3 Higher-Order Programs

This section provides a simple generalization of lambda-lifting and lambda-dropping to higher-order programs, along the same lines as Section 2.

3.1 Introduction

Higher-order programs can be lambda-lifted without considering the applications of higher-order functions that have been passed as parameters. Similarly, higher-order programs can be lambda-dropped while only considering the first-order function applications. As illustrated in Section 5.3, going further would require a control-flow analysis [40, 43]. Introducing higher-order functions into our language thus only entails slight changes to the specifications and algorithms for lambda-lifting and lambda-dropping.

For higher-order programs, we generalize the syntax of Figure 1 by allowing functions to appear as arguments and variables to be applied as functions. We consider curried functions as a whole, rather than as a sequence of nested functions. Anonymous functions can be explicitly named and thus uniformly represented in the generalized syntax.

3.2 Lambda-lifting

The specification of lambda-lifting (Figures 3 and 4) requires only trivial changes.
A higher-order program is completely parameter-lifted when no functions have
any free variables. A higher-order program is completely block-floated when its
functions cannot be moved further out through the block structure.

To parameter-lift a higher-order program, an algorithm almost identical to
that for first-order programs can be used. In first-order and higher-order pro-
grams alike, functions may have free variables. However, in a higher-order pro-
gram, a function may be passed as an argument, and applied elsewhere under
a different name. While this seemingly poses problems because we cannot ex-
pand the application of each function with its free variables, having higher-order
functions does enable us to create curried functions. The approach is thus to
curry each function that has free variables, making the free variables the formal
parameters on the newly introduced lambda. We thus rewrite a function f with
free variables $\{v'_1, ..., v'_n\}$ as follows:

$$f \equiv \lambda(v_1, ..., v_n).e \Rightarrow f \equiv \lambda(v'_1, ..., v'_n).\lambda(v_1, ..., v_n).e$$

Similarly, each occurrence of the function name is replaced by an application to
its free variables. For the function f:

$$f \Rightarrow (f\,(v'_1, ..., v'_n))$$

Using this technique, parameter lifting can be performed with a slightly modified
version of the existing algorithm. The only difference lies in the way that the
function definition is expanded with its free variables and that the occurrences
of the function application to be expanded with its free variables may be in any
expression.

To block-float a scope-insensitive higher-order program, exactly the same al-
gorithm can be used as for first-order programs. A curried function is considered
as a whole, and is thus kept together as a global function.

Example

Figure 14 shows a program that maps a higher-order function onto a list. The
function `filter`, which filters out values that do not satisfy some predicate, is
mapped onto a list of numbers.

To parameter-lift the program of Figure 14, the free variables `pred` and `i`
must be passed to `filter`. Similarly, the free variable `f` must be passed to the
function `loop`. In both cases, we bind these variables by extending the function
declaration with a curried parameter list. All occurrences of these functions are
replaced by applications of the functions onto the variables its declaration was
extended with. This makes the program scope insensitive, as shown in Figure
15, enabling the usual block-floating pass. The result is displayed in Figure 16.

```
(define main
  (lambda (pred i ls)
    (letrec ([filter (lambda (j)
                       (if (pred j) j i))]
             [map (lambda (f xs)
                    (letrec ([loop (lambda (s)
                                     (if (null? s)
                                         '()
                                         (cons (f (car s))
                                               (loop (cdr s)))))])
                      (loop xs)))])
      (map filter ls))))
```

Figure 14: Example involving a higher-order function.

```
(define main
  (lambda (pred i ls)
    (letrec ([filter (lambda (pred i)
                       (lambda (j)
                         (if (pred j) j i)))]
             [map (lambda (f xs)
                    (letrec ([loop (lambda (f)
                                     (lambda (s)
                                       (if (null? s)
                                           '()
                                           (cons (f (car s))
                                                 ((loop f)
                                                  (cdr s))))))])
                      ((loop f) xs)))])
      (map (filter pred i) ls))))
```

Figure 15: The program of Figure 14, after parameter-lifting.

```
(define main
  (lambda (pred i ls)
    (map (filter pred i) ls)))

(define filter
  (lambda (pred i)
    (lambda (j)
      (if (pred j) j i))))

(define map
  (lambda (f xs)
    ((loop f) xs)))

(define loop
  (lambda (f)
    (lambda (s)
      (if (null? s)
          '()
          (cons (f (car s))
                ((loop f) (cdr s)))))))
```

Figure 16: The program of Figure 15, after block-floating.

3.3 Reversing lambda-lifting

As was the case for first-order programs, reversing lambda-lifting must be done by first re-creating block-structure and then removing redundant formal parameters. In the first-order case, all occurrences of functions were in the form of applications. It was these occurrences that constrained the creation of block structure. In the higher-order case, functions may appear at any place, but they still constrain the creation of block structure in the same way as before.

As was the case for first-order programs, variables that were free in a function in the original program are given directly as arguments to the function. However, lambda-lifting introduced an extra level of currying into the program and thus lambda-dropping should remove this extra level of currying. Thus, a curried function without any arguments, a "thunk," that is always applied directly to obtain its curried value, should de "thawed" by removing this extra level of currying.

3.4 Lambda-dropping

Like lambda-lifting, the specification of lambda-dropping (Figures 10 and 11) requires only trivial changes. A higher-order program is completely block-sunk when references to function names between functions prevent further localization. A higher-order program is completely parameter-dropped when no parameters that are given as direct arguments to a function are redundant.

To block-sink a higher-order program, the first-order block-sinking algorithm can be used. The only difference is conceptual: free functions now represent scope dependencies between functions, not necessarily calls. So rather than beginning by constructing the call graph of the program, we proceed in exactly the same fashion, and construct the "dependence graph" of the program.

To parameter-drop a higher-order program, an algorithm almost identical to the one for first-order programs can be used. In first-order and higher-order programs alike, the only variables that we consider as being redundant are those that are given as direct arguments to a function. However, in a higher-order program, we may be left with a parameterless function after having dropped the redundant formal parameters. A curried function of no arguments that only occurs as an application should be uncurried. For a function f that is always applied, this uncurrying reads as follows:

$$f \equiv \lambda().\lambda(v_1, ..., v_n).e \ \Rightarrow \ f \equiv \lambda(v_1, ..., v_n).e$$

Similarly, each occurrence of the function is in an application, which is replaced by the function itself. For the function f above, this reads:

$$(f\,()) \ \Rightarrow \ f$$

This "thawing" transformation should be performed as the last stage of parameter-dropping. The ordinary first-order flow graph of the program is still

sufficient for our purposes. Functions that are passed as arguments are simply ignored, since they cannot have redundant parameters that are passed to them directly.

Example (revisited)

To block-float the program of Figure 16, we construct the function dependence graph of the program. Re-constructing the program accordingly yields the block structure of the original program after parameter lifting shown in Figure 15. Parameter dropping proceeds as in the first-order case, ignoring any curried arguments of a function. This eliminates the formal parameters `pred` and `i` of `filter`, and `f` of `loop`. However, `filter` and `loop` are still curried functions without arguments. Both functions had arguments removed, so none of them are passed as arguments to other parts of the program. Thus we are sure that in every occurrence of these functions, they are applied to zero arguments. We can therefore remove the empty currying in the definition and in the applications of each of these functions, yielding the program of Figure 14.

3.5 Summary

The algorithms for lambda-lifting and lambda-dropping higher-order programs are simple generalizations of those that operate on first-order programs. Lambda-lifting must create curried functions rather than expanding the list of formal parameters, and change each occurrence of such functions into an application. Lambda-dropping must thaw curried functions of zero arguments, changing each application of these functions correspondingly.

4 Perspectives on Lambda-Lifting and Lambda-Dropping

Lambda-dropping has been indirectly addressed in other contexts, in relation to SSA form (Section 4.1), in relation to optimizing compilers (Section 4.2) and in relation to a similar block-structure transformation (Section 4.3). We have already provided precise definitions of programs that are completely lambda-lifted and lambda-dropped, as well as formal descriptions of each algorithm. We further discuss correctness and properties in Section 4.4, as well as time complexity in Section 4.5. Finally, in Section 4.6, we address efficiency issues through an empirical study.

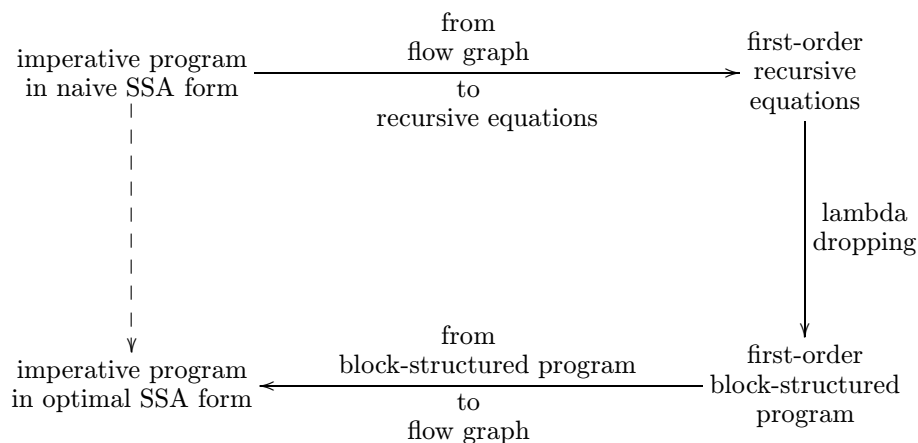## 4.1 Optimal Static Single Assignment (SSA) form

In an imperative program in SSA form, each variable is assigned only at a single point in the program. Values from different branches in the flow graph

of program are merged using "ϕ-nodes." A naive algorithm for transforming an imperative program into SSA form creates ϕ-nodes for all variables, at all merge points in the program. Minimizing the number of ϕ-nodes yields a program in optimal SSA form.

Several algorithms exist for transforming a program into optimal SSA form. They work by creating a dominator tree for the flow graph of the program.

In his textbook on compiler construction [3], Appel made a significant connection between SSA and functional programming: converting an imperative program in SSA form (represented by its flow graph) to functional form creates a block-structured functional program, where each function corresponds to a basic block.

What struck us in Appel's connection is that converting from naive SSA form to optimal SSA form corresponds to parameter-dropping. A program in naive SSA form that is translated into its functional counterpart, parameter dropped, and then translated back into SSA form is in optimal SSA form. Similarly, the translation from SSA form to a block-structured program is done using the dominator tree of the flow graph, which corresponds exactly to block-sinking.

```
imperative program        from              first-order
in naive SSA form      flow graph            recursive
             ─────────────────────────────>  equations
                           to
                   recursive equations
      │                                           │
      │                                           │
      │                                           │  lambda
      │                                           │  dropping
      │                                           │
      │                                           ▼
      ▼                    from              first-order
imperative program   block-structured program  block-structured
in optimal SSA form  <─────────────────────────  program
                           to
                       flow graph
```

Once we had realized this coincidence, it became clear how we could simplify our initial presentation of lambda-dropping [17]. Specifically, using dominator graphs simplified our overall presentation. It also yields a substantial improvement of lambda-dropping in time complexity.

4.2 Optimizing compilers

Peyton Jones, Partain, and Santos optimize programs using block floating, block sinking and parameter dropping in the Glasgow Haskell compiler [32]. In particular, blocks are not necessarily floated all the way to the top level. As for source programs, they are no longer systematically lambda-lifted [34, 35].

Parameter dropping has been specifically addressed as an optimization in two very similar studies. Appel performs loop-invariant removal after loop-header introduction in Standard ML of New Jersey [2]. Santos performs Static-Argument Removal in the Glasgow Haskell Compiler [37]. In each case, the primary concern is to optimize single (as opposed to mutually) recursive functions. In neither case is block sinking performed. Rather, in both cases an enclosing definition is introduced that creates the scope which enables the compiler to perform parameter dropping. In the Glasgow Haskell Compiler, the optimization was later judged to be of very little effect, and subsequently disabled. In Standard ML of New Jersey, the optimization gave consistent improvements in compilation time, running time and space consumption. It is our experience that both compilers significantly benefit from source lambda-dropping in the case of mutually recursive functions.

4.3 Peyton Jones's dependency analysis

In his textbook on implementing functional languages [34], Peyton Jones uses an analysis to generate block structure from recursive equations. In several ways, the algorithm for this dependency analysis is similar to the algorithm we employ for block sinking. Its goal, however, differs. Assume that the lambda-lifted version of the example program of Figure 9 was extended with a main expression calling the function `main` with appropriate arguments. Peyton Jones's dependency analysis would then yield the following skeleton:

```
(letrec ([f2 (lambda (x1 j2 k2) ...)]
         [f7 (lambda (y1 j4 k4) ...)])
   (letrec ([main (lambda (i1 j1 k1) ...)])
     (main 1 1 0)))
```

In this skeleton, any function is visible to other functions that refer to it. In contrast to the program in Figure 8, though, their formal parameters are not visible to these other functions *and thus they cannot be dropped.* Peyton Jones's analysis places no function in the scope of any formal parameters and thus it inhibits parameter dropping.

More detail on Peyton Jones's dependency analysis, its purpose and properties, and its relation to our transformation can be found in the second author's MS thesis [39].

4.4 Correctness issues

Lambda-lifting has never been proven correct formally.

In his MS thesis, the second author informally addresses the correctness of lambda-lifting and lambda-dropping by relating each algorithm to basic steps consisting of let-floating/sinking and formal parameter list expansion/reduction [39]. He argues that the correctness of the transformations would follow from

proving the correctness of the basic steps and expressing lambda-lifting and lambda-dropping in terms of these basic steps, while correctness of the algorithms presented in Section 2 would follow from showing these to be equivalent to the fixed point reached by the stepwise algorithms.

In an alternative attempt, the first author has recast the two transformations extensionally, expressing them as transformations on functionals prior to taking their fixed point [15].

As already pointed out, several algorithms exist for lambda-lifting. Our favorite one is Johnsson's [23], and we designed lambda-dropping as its inverse for maximally nested programs. (We take equality of programs as syntactic equality, modulo renaming of variables and modulo the ordering of mutually recursive declarations.) Indeed, since source programs are often partly lifted / partly dropped, we cannot hope to provide a unique inverse for lambda-lifting. Given several versions of the same program with varied nesting, lambda-lifting maps them into the same set of mutually recursive equations. We thus conjecture the following two properties, which are reminiscent of a Galois connection [31]:

Property 1 *Lambda-dropping is the inverse of lambda-lifting on all programs that have been lambda-dropped.*

Property 2 *Lambda-lifting is the inverse of lambda-dropping on all programs that have been lambda-lifted.*

Property 1 is arguably the most complex of the two. Lambda-dropping a program requires re-constructing of the block structure that was flattened by lambda-lifting and omitting the formal parameters that were lifted by the lambda-lifter.

Examining the lambda-dropping algorithm reveals that a function that is passed as argument never has its parameters dropped. Dropping the parameters of such functions is certainly possible, but is non-trivial since it requires a control-flow analysis to determine the set of variables being passed as arguments (Section 5.3 provides an example).

Restricting ourselves to providing a left inverse for lambda-lifting eliminates the need for this analysis. If a function has no free variables before lambda-lifting, no additional parameters are added, and we need not drop any parameters to provide a proper inverse. If the function did have free variables, these variables are applied as arguments to the function *at the point where it is passed as an argument.* Thus, the extra parameters are easily dropped, since they are unambiguously associated to the function.

In languages such as Scheme where currying is explicit, a lambda-lifter may need to construct a function as a curried, higher-order function when lifting parameters. A lambda-dropper can easily detect such declarations (the currying performed by the lambda-lifter is redundant after lambda-dropping), and remove them.

Johnsson's lambda-lifting algorithm explicitly names anonymous lambda forms with let expressions, and eliminates let expressions by converting them into applications. A lambda-dropper can recognize the resulting constructs and reverse the transformations, thereby satisfying the inverseness properties.

4.5 Time complexity

The lambda-lifting algorithm has a time complexity of $\mathcal{O}(n^3 + m \log m)$, where n is the maximal number of functions declared in a block and m is the size of the program. The n^3 component is derived from solving the set equations during the parameter-lifting stage [23]. As the present article is going to press, we have actually reduced this complexity to be quadratic, which we believe is an optimal bound.

The lambda-dropping algorithm has a time complexity of $\mathcal{O}(n \log n)$, where n is the size of the program. This complexity assumes the use of a linear-time algorithm for finding dominator graphs, such as Harel's [3].

4.6 Empirical study

Lambda-dropping a program removes formal parameters from functions, making locally bound variables free to the function. An implementation must handle these free variables. In higher-order languages, it is usually necessary to store the bindings of these free variables when passing a function as a value. Most implementations use closures for this purpose [27].

4.6.1 Issues

There is a natural trade-off between passing arguments as formal parameters and accessing them via the closure. Passing formal parameters on the stack is simple but must be done on every function invocation. Creating a closure usually incurs extra overhead. Values must be copied into each closure, taking either stack or heap space. Looking up values in a closure often takes more instructions than referencing a formal parameter. The closure of a function is created upon entry into its definitional block. Thus, the function can be called many times with the same closure. This is relevant in the case of recursive functions, since the surrounding block has already been entered when the function calls itself and thus the same closure is used in every recursive invocation of the function.

A recursive function in a program written without block structure must explicitly manipulate everything it needs from the environment at every recursive call. By contrast, if the function instead uses free variables, e.g., after lambda-dropping, the performance of the program may be improved:

- Fewer values need to be pushed onto the stack at each recursive call. This reduces the number of machine instructions spent on each function

invocation. However, if accessing a variable in a closure is more expensive, then more instructions may be spent during the execution of the function.

- If a free variable is used in special cases only, it might not be manipulated during the execution of the body of the function. This reduces the amount of data manipulation, potentially reducing register spilling and improving cache performance.

A compiler unfolding recursive functions up to a threshold could lambda-drop locally before unfolding, thereby globalizing constant parameters, for example.

4.6.2 Experiments

Initial experiments suggest that lambda-dropping can improve the performance of recursive functions, most typically for programs performing recursive descents. The improvement depends on the implementation, the number of parameters removed, the resulting number of parameters, and the depth of the recursion. It is our experience that lambda-dropping increases performance for the following implementations of block-structured functional languages:

- Scheme: Chez Scheme and SCM;

- ML: Standard ML of New Jersey and Moscow ML;

- Haskell: the Glasgow Haskell Compiler;

- Miranda,

and also for implementations of block-structured imperative languages such as Delphi Pascal, Gnu C and Java 1.1.

The results of our experiments are shown in Figure 17. Except for Miranda, they were all performed on a Pentium machine. The experiments with Java and Pascal were done with Windows and all the others with Linux. The Miranda experiments were carried out on an HPPA RISC machine with HPUX. The reported speedup $S_{L \to D}$ is the time taken by the lambda-lifted version of the program divided by the time taken by the lambda-dropped version.

The first two experiments evaluate the efficiency of `append` in different situations. The third, slightly contrived, experiment evaluates the efficiency of mutually recursive functions that perform simple arithmetic computations, where five out of seven parameters are redundant. It represents an ideal case for optimization by lambda-dropping. The last experiment evaluates the performance of a generic fold function over an abstract-syntax tree, instantiated to CPS-transform a generated program in one pass and to compute the lexical size of the result.

In some implementations, such as Standard ML of New Jersey, dropping five out of seven parameters can yield a program which is 2.5 times faster than the

Language	$S_{L\to D}$
ML (Moscow ML)	1.77
Miranda*	1.50
Pascal (Delphi)	1.09
Haskell (GHC 2.01)	1.02
"C" (GCC)	1.00
ML (SML/NJ 110)	1.00
Scheme (SCM)	1.00
Scheme (Chez v5)	0.98
Java (Sun JDK 1.1.1)	0.40

Append
(many invocations on small lists)

Language	$S_{L\to D}$
ML (Moscow ML)	1.96
Miranda*	1.52
Pascal (Delphi)	1.10
Haskell (GHC 2.01)	1.01
"C" (GCC)	1.00
ML (SML/NJ 110)	1.00
Scheme (SCM)	1.00
Scheme (Chez v5)	0.95
Java (Sun JDK 1.1.1)	0.90

Append
(few invocations on long lists)

Language	$S_{L\to D}$
ML (SML/NJ 110)	2.74
Miranda*	2.02
ML (Moscow ML)	1.42
Haskell (GHC 2.01)	1.23
Scheme (SCM)	1.16
Java (Sun JDK 1.1.1)	1.13
Pascal (Delphi)	1.12
Scheme (Chez v5)	1.02
"C" (GCC)	1.01

Mutually recursive functions
(many parameters)

Language	$S_{L\to D}$
Scheme (SCM)	1.21
Miranda*	1.19
ML (SML/NJ 110)	1.17
Scheme (Chez v5)	1.17
Haskell (GHC 2.01)	1.13
ML (Moscow ML)	1.09

Fold function
(CPS and lexical size)

(*): Performed on a different
architecture

$S_{L\to D}$ is the time taken by the lambda-lifted version of the program divided by the time taken by the lambda-dropped version.

Figure 17: Experiments with lambda-dropping.

original. In Chez Scheme, however, lambda-dropping entails a slight slowdown in some cases, for reasons we could not fathom. Limiting our tests to a few programs stressing recursion and parameter passing gave speedups ranging from 1.05 to 2.0 in most cases. This was observed on all implementations but Chez Scheme.

The results indicate that applying lambda-dropping locally can optimize recursive functions. We have done this experimentally, on a very limited scale, using two mature compilers: the Glasgow Haskell Compiler and Standard ML of New Jersey (see Section 4.2). Standard ML of New Jersey in particular does benefit from lambda-dropping mutually recursive functions.

4.6.3 Analysis

The second author's MS thesis presents an abstract model describing the costs of procedure invocation and closure creation. The model is parameterized by the costs of these basic operations of a low-level abstract machine. We have found this abstract model useful for reasoning about the effect of lambda-lifting and lambda-dropping on programs, even though it does not account for unpredictable factors later on at compile time (e.g., register allocation) and at run time (e.g., cache behavior). These make the model unreliable for predicting actual run-time behaviors.

More detailed information on the experiments, the abstract model, and the results can be found in the second author's MS thesis [39].

5 Applications and Synergy

5.1 Partial evaluation

Our compelling motivation to sort out lambda-lifting and lambda-dropping was partial evaluation [13, 25]. As mentioned in Section 1, recursive equations offer a convenient format for a partial evaluator. Similix and Schism, for example [9, 11], lambda-lift source programs before specialization and they produce residual programs in the form of recursive equations. Very often, however, these recursive equations are plagued with a huge number of parameters, which increases their compilation time enormously, sometimes to the point of making the whole process of partial evaluation impractical [16]. We thus lambda-drop residual programs to reduce the arities of their residual functions. As a side benefit, lambda-dropping also re-creates a block structure which is often similar to the nesting of the corresponding source program, thereby increasing readability.

Our lambda-dropper handles the output language of Schism.

```
(define-type binary-tree
  (leaf alpha)
  (node left right))

(define binary-tree-fold
  (lambda (process-leaf process-node init)
    (letrec ([traverse (lambda (t)
                         (case-type t
                           [(leaf n)
                            (process-leaf n)]
                           [(node left right)
                            (process-node (traverse left)
                                          (traverse right))]))])
      (lambda (t) (init (traverse t))))))

(define main
  (lambda (t x y)
    ((binary-tree-fold (lambda (n) (leaf (* (+ x n) y)))
                       (lambda (r1 r2) (node r1 r2))
                       (lambda (x) x)) t)))
```

Figure 18: Source program.

41

```
(define (main-1 t x y)
  (traverse:1-1 t y x))

(define (traverse:1-1 t y x)
  (casetype t
    [(leaf n)
     (leaf (* (+ x n) y))]
    [(node left right)
     (node (traverse:1-1 left y x)
           (traverse:1-1 right y x))]))
```

Figure 19: Specialized (lambda-lifted) version of Figure 18.

5.1.1 Example: a fold function

Figure 18 displays a source program, which uses a standard fold function over a binary tree. Without any static input, Schism propagates the two static abstractions from the main function into the fold function. The raw residual program appears in Figure 19. It is composed of two recursive equations. The static abstractions have been propagated and statically reduced. The dynamic parameters x and y have been retained and occur as residual parameters.[1] They make the traversal function an obvious candidate for lambda-dropping. Figure 20 displays the corresponding lambda-dropped program, which was obtained automatically.

---

[1]That is how partially static values and higher-order functions inflate (raise) the arity of recursive equations.

```
(define (main-1 t x y)
  (letrec ([traverse:1-1 (lambda (t)
                           (case-type t
                             [(leaf n)
                              (leaf (* (+ x n) y))]
                             [(node left right)
                              (node (traverse:1-1 left)
                                    (traverse:1-1 right))])))])
    (traverse:1-1 t)))
```

Figure 20: Lambda-dropped version of Figure 19.

```
{
  int res=1; int n=4; int cnt=1;
  while (cnt > 0) { res = 1;
                    n = 4;
                    while (n > 0) { res = n * res;
                                    n = n - 1;
                    }
                    cnt = cnt - 1;
  }
}
```

Figure 21: Example imperative program.

```
    (define (evprogram-1 s)
      (evwhile-1
        (intupdate 2 1 (intupdate 1 4 (intupdate 0 1 s)))))

    (define (evwhile-1 s)
      (if (gtint (fetchint 2 s) 0)
          (evwhile-2 (intupdate 1 4 (intupdate 0 1 s)))
          s))

    (define (evwhile-2 s)
      (if (gtint (fetchint 1 s) 0)
          (let ([s-1 (intupdate 0
                                (mulint (fetchint 1 s) (fetchint 0 s))
                                s)])
            (evwhile-2 (intupdate 1 (subint (fetchint 1 s-1) 1) s-1)))
          (evwhile-1 (intupdate 2 (subint (fetchint 2 s) 1) s)))))
```

Figure 22: Specialized (lambda-lifted) version of the definitional interpreter with respect to Figure 21.

Figures 19 and 20 directly tell us that the dynamic parameters x and y are the sole survivors of the two static abstractions in the source program. As partial-evaluation users, however, we find it clearer to compare Figure 18 and Figure 20 rather than Figure 18 and Figure 19. Indeed Figure 20 shows more clearly that the letrec block of Figure 18 has been inlined and specialized into the main function. In contrast, the letrec block is more disconnected in Figure 19 and its spurious parameters get in the way of readability.

5.1.2   Example: the first Futamura projection

Let us consider a while-loop language as is traditional in partial evaluation and semantics-based compiling [12]. Figure 21 displays a source program with several while loops. Specializing the corresponding definitional interpreter (not shown here) using Schism with respect to this source program yields the residual program of Figure 22. Each source while loop has given rise to a recursive equation. Figure 23 displays the corresponding lambda-dropped program, which was obtained automatically.

Again, we find it clearer to compare Figure 21 and Figure 23 rather than Figure 21 and Figure 22. The relative positions of the residual recursive functions now match the relative positions of the source while loops. (N.B. A monovariant specializer would have directly produced the lambda-dropped program [14, 20].)

```
(define (evprogram-1 s)
  (letrec ([evwhile-1
             (lambda (s)
               (letrec ([evwhile-2
                          (lambda (s)
                            (if (gtint (fetchint 1 s) 0)
                                (let ([s-1 (intupdate 0
                                                       (mulint
                                                         (fetchint 1 s)
                                                         (fetchint 0 s))
                                                       s)])
                                  (evwhile-2
                                    (intupdate 1
                                               (subint (fetchint 1 s-1)
                                                       1)
                                               s-1)))
                                (evwhile-1
                                  (intupdate 2
                                             (subint (fetchint 2 s) 1)
                                             s))))])
                 (if (gtint (fetchint 2 s) 0)
                     (evwhile-2 (intupdate 1 4 (intupdate 0 1 s)))
                     s)))])
    (evwhile-1 (intupdate 2 1 (intupdate 1 4 (intupdate 0 1 s))))))
```

Figure 23: Lambda-dropped version of Figure 22.

5.2 Programming environment

It is our programming experience that lambda-lifting and lambda-dropping go beyond a mere phase in a compiler for functional programs. They can offer truly useful (and often unexpected) views of one's programs. In the context of teaching, in particular, these unexpected views often help students to improve their understanding of lexical scope and block structure, and to use them more effectively in programming. For example, lambda-dropping tells us that in Figure 18, the fold functional could have been defined locally to the main function. Sections 5.3 to 5.5 present more examples.

5.3 From Curry to Turing

Together, lambda-lifting, lambda-dropping and control-flow analysis allow one to convert Curry's fixpoint operator into Turing's fixpoint operator.

Here is Curry's fixpoint operator [6]:

$$\lambda f.\text{let } g = \lambda x.f\ (x\ x)$$
$$\text{in } g\ g$$

f occurs free in g. Lambda-lifting this block yields the following λ-term:

$$\lambda f.\text{let } g = \lambda f.\lambda x.f\ (x\ x)$$
$$\text{in } g\ f\ (g\ f)$$

Control-flow analysis tells us that x can only denote $g\ f$ and that all the occurrences of f denote the same value. Thus we can safely relocate the second occurrence of f, in the let body, into the let header:

$$\lambda f.\text{let } g = \lambda f.\lambda x.f\ (x\ f\ x)$$
$$\text{in } g\ f\ g$$

Now x only denotes g. Again, control-flow analysis reveals the only application sites of the λ-abstraction denoted by g. Thus we can safely swap its two parameters:

$$\lambda f.\text{let } g = \lambda x.\lambda f.f\ (x\ x\ f)$$
$$\text{in } g\ g\ f$$

Eta-reducing this term yields Turing's fixpoint operator [6].

5.4   Detecting global variables

Following Schmidt's initial impetus on single-threading [38], Sestoft has investigated the detection of global variables in recursive equations [41]. Likewise, Fradet has investigated the detection of single-threaded variables using continuations [19]. Such variables come in two flavors: global, read-only variables, and updatable, single-threaded variables.

Lambda-dropping reveals global read-only variables by *localizing* blocks. (N.B. Many of these global variables are not global to a whole program, only for parts of it. These parts are localized.) Conversely, transforming a program into continuation-passing style (CPS) reveals single-threaded variables: their value is passed to the continuation. This last point, of course, indicates that we should lambda-drop after CPS transforming a program.

5.5   Continuation-based programming

Shivers optimizes a tail-recursive function by "promoting" its CPS counterpart from being a function to being a continuation [43]. For example, consider the function returning the last element of a non-empty list.

$$\text{letrec last} = \lambda x.\text{let } t = \text{tl } x$$
$$\text{in if } t = \text{nil}$$
$$\text{then hd } x$$
$$\text{else last } t$$
$$\text{in last } l$$

Its (call-by-name[2]) CPS counterpart can be written as follows.

$$\lambda k.\text{letrec last}' = \lambda x.\lambda k.\text{tl}' \ x \ \lambda t.\text{if } t = \text{nil}$$
$$\text{then hd}' \ x \ k$$
$$\text{else last}' \ t \ k$$
$$\text{in last}' \ l \ k$$

where hd' and tl' are the CPS versions of hd and tl, respectively. The type of last' reads:

$$\text{Value} \rightarrow (\text{Value} \rightarrow \text{Answer}) \rightarrow \text{Answer}.$$

Shivers promotes last' from the status of function to the status of continuation. He rewrites it as follows:

$$\lambda k.\text{letrec last}' = \lambda x.\text{tl}' \ x \ \lambda t.\text{if } t = \text{nil}$$
$$\text{then hd}' \ x \ k$$
$$\text{else last}' \ t$$
$$\text{in last}' \ l$$

The type of last' now reads:

$$\text{Value} \rightarrow \text{Answer}.$$

It coincides with the type of a continuation, since last' does not pass continuations anymore. Promoting a function into a continuation amounts to parameter-dropping its continuation parameter.

Lambda-dropping the CPS counterpart of a program that uses call/cc also offers a convenient alternative to dragging around escape functions at each function call.

6   Related Work

Aside from SSA-related transformations (Section 4.1), parameter-dropping single recursive functions (Section 4.2), Peyton Jones's localization of blocks (Section 4.3), Sestoft's detection of read-only variables (Section 5.4) and Erik Meijer's unpublished note "Down with Lambda-Lifting" (April 1992) — none of which directly addresses lambda-dropping as such — we do not know of any work about lambda-dropping. There is, however, plenty of work related to lambda-lifting.

---

[2]For example.

6.1 Enabling principles

The enabling principles of lambda-lifting are worth pointing out: Landin's correspondence principle [28], which has been formalized as categorical exponentiation [5], makes it possible for Johnsson's original algorithm to remove let statements.

$$\text{let } x = a \text{ in } e \quad \equiv \quad (\lambda x.e)\, a$$

Expansion makes it possible to remove free variables. Let associativity enables let-floating, which makes it possible to globalize function definitions that have no free variables.

6.2 Curried and lazy vs. uncurried and eager programs

Johnsson concentrated on lambda-lifting towards mutually recursive equations [23], but alternative approaches exist. The first seems to be Hughes's supercombinator abstraction, where recursion is handled through self-application and full laziness is a point of concern [21]. Peyton Jones provides a broad overview of fully lazy supercombinators [33, 34, 35]. Essentially, instead of lifting only free variables, one lifts maximally free expressions. Fully lazy lambda-dropping would amount to keeping maximally free expressions instead of identifiers in the initial calls to local functions.

In their Scheme compiler Twobit, Clinger and Hansen also use lambda-lifting [10]. They, however, modify the flow equations to reduce the arity of lambda-lifted procedures. Lambda-lifting is also stopped when its cost outweighs its benefits, regarding tail-recursion and allocation of closures in the heap. Lambda-lifting helps register allocation by indicating unchanging arguments across procedure calls.

6.3 Closure conversion

To compile Standard ML programs, Appel represents a closure as a vector [1]. The first element of the vector points to a code address. The rest of the vector contains the values of the free variables. Applying a closure to actual parameters is done by passing the closure itself and the actual parameters to the code address. Thus calls are compiled independently of the number of free variables of the called function. This situation is obtained by "closure conversion." Once a program is closure-converted, it is insensitive to lexical scope and thus it can be turned into recursive equations.

Closure conversion, however, differs from lambda-lifting for the following two reasons:

- In both closure-converted and lambda-lifted programs, lambda abstractions are named. In a closure-converted program, free variables are passed

only when the name is defined. In a lambda-lifted program, free variables are passed each time the name is used.

- Closure conversion only considers the free variables of a lambda-abstraction. Lambda-lifting also considers those of the callees of this lambda-abstraction.

In the latter sense, lambda-lifting can be seen as the transitive closure of closure conversion.

Steckler and Wand consider a mix between lambda-dropping and closure conversion: so-called "lightweight closures" [44]. Such closures do not hold the free variables that are in scope at the application sites of this closure. A similar concern leads Shao and Appel to consider whether to implement closures in a deep or in a flat way [42].

6.4   Analogy with the CPS transformation

An analogy can be drawn between lambda-dropping and continuation-based compilation. As observed by Sabry, Felleisen, *et al.* [18], CPS compilers proceed in two steps: first, source programs are transformed into continuation-passing style, but eventually they are mapped back to direct style.

One is left with the conjecture that both transformations (lambda-dropping and CPS transformation) expose, in a simpler way, more information about the structure of a program during its journey through a compiler. The CPS transformation reveals control-flow information, while lambda-dropping reveals scope information. As pointed out in Section 6.2, this information is useful for lambda-lifting proper. We believe that it is also useful for stackability detection by region inference.

6.5   Stackability

Recently, Tofte and Talpin have proposed to implement the λ-calculus with a stack of regions and no garbage collector [46]. Their basic idea is to associate a region for each lexical block, and to collect the region on block exit. While this scheme is very much allergic to CPS (which "never returns"), it may very well benefit from preliminary lambda-dropping, since the more lexical blocks, the better for the region inferencer. We leave this issue for future work.

6.6   Partial evaluation

Instead of lambda-lifting source programs and lambda-dropping residual programs, a partial evaluator could process block-structured programs directly. In the diagram of the abstract, we have depicted such a partial evaluator with a dashed arrow. To the best of our knowledge, however, except for Malmkjær

and Ørbæk's case study presented at PEPM'95 [30] and for Hughes's type specializer [22], no polyvariant partial evaluator for procedural programs handles block structure today.

As analyzed by Malmkjær and Ørbæk, polyvariant specialization of higher-order, block-structured programs faces a problem similar to Lisp's "upward funarg." An upward funarg is a closure that is returned beyond the point of definition of its free variables, thus defeating stackability. The partial-evaluation analogue of an upward funarg is a higher-order function that refers to a specialization point but is returned past the scope of this specialization point. What should the specializer do? Ideally it should move the specialization point outwards to its closest common ancestor together with the point of use for the higher-order function. Lambda-dropping residual recursive equations achieves precisely that, merely by sinking blocks as low as possible.

The problem only occurs for polyvariant specializers for higher-order, block-structured programs where source programs are not lambda-lifted and program points are specialized with respect to higher-order values. Most partial evaluators do not face that problem: Lambda-Mix [20] and type-directed partial evaluation [14] are monovariant; Schism [11] and Similix [9] lambda-lift before binding-time analysis; Pell-Mell [29] lambda-lifts after binding-time analysis; ML-Mix [7] does not specialize with respect to higher-order values; and Fuse [36] does not allow upwards funargs.

Recently, in his type specializer [22], Hughes introduced "first-class polyvariance," i.e., in effect, specialization with respect to higher-order values without any prior lambda-lifting. The problem mentioned above is avoided at the cost of duplicating residual code.

6.7 Other program transformations

Other program transformations can also benefit from lambda-dropping: in Wadler's work on deforestation [47], for example, the "macro" style amounts to lambda-dropping by hand.

7 Conclusion and Issues

In the mid 80's, Hughes, Johnsson and Peyton Jones presented lambda-lifting: the transformation of functional programs into recursive equations. Since then, lambda-lifting seems to have been mostly considered as an intermediate phase in compilers. It is our contention that lambda-lifting is also interesting as a source-to-source program transformation, together with its inverse: lambda-dropping. For example, we observe that Appel's characterization of SSA as functional programming relies on lambda-dropping.

In this article, we have introduced lambda-dropping, outlined some of its properties, and mentioned some other applications than our main one: as a

back end in a partial evaluator. We have implemented a lambda-dropper in Scheme for the target language of Schism and we plan to port it in ML for Pell-Mell. We are currently developing a faster lambda-lifter, and we are still working on the formal semantics of lambda-lifting and lambda-dropping.

Acknowledgements

A   Graph algorithms

The descriptions of the algorithms for lambda-lifting and lambda-dropping (Sections 2.2 and 2.4) make use of a set of standard functions for manipulating graphs. The functions for adding nodes and edges are trivial. Creating the dominator tree can be done in linear time using Harel's algorithm [3]. Creating the first-order flow graph of a program in the syntax of Figure 1 can be done using a simple traversal of the program.

Graph.addNode :: $\mathsf{Graph}(\alpha) \to \alpha \to \alpha$
Graph.addNode $(G \textbf{ as } (V, E))$ $a = V := V \cup \{a\}$; a

Graph.addEdge :: $\mathsf{Graph}(\alpha) \to (\alpha, \alpha) \to (\alpha, \alpha)$
Graph.addEdge $(G \textbf{ as } (V, E))$ $(a, b) =$
    if $a \notin V$ **then** $V := V \cup \{a\}$;
    if $b \notin V$ **then** $V := V \cup \{b\}$;
    if $(a, b) \notin E$ **then** $E := E \cup \{(a, b)\}$;
    (a, b)

Graph.findDominators :: $(\mathsf{Graph}(\alpha), \alpha) \to (\mathsf{Graph}(\alpha), \alpha)$
Graph.findDominators (G, r) : Returns the dominator tree of G [3]. In the dominator tree, node b is a successor of node a if and only if all paths from r to b go through a.

Graph.flowGraph :: $\mathsf{Program} \to (\mathsf{Graph}(\mathsf{DefNode}), \mathsf{DefNode})$
Graph.flowGraph P : Returns the flowgraph of P [3]. The flowgraph has an edge from node f_a to node g_b iff g is invoked from within f, binding the formal argument b of g to a, where a is a formal parameter of f.

Graph.isPath :: $\mathsf{Graph}(\alpha) \to \alpha \to \alpha \to \mathsf{Bool}$
Graph.ispath G a b : **true** if there is a path from a to b in G.

Figure 24: Graph functions.

References

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

[2] Andrew W. Appel. Loop headers in lambda-calculus or CPS. *Lisp and Symbolic Computation*, 7(7):337–343, 1994.

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

[4] Lennart Augustsson. *Compiling Lazy Functional Languages, part II*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1988.

[5] Anindya Banerjee and David A. Schmidt. A categorical interpretation of Landin's correspondence principle. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 587–602, New Orleans, Louisiana, April 1993. Springer-Verlag.

[6] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[7] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1993. DIKU Rapport 93/22.

[8] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[9] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic offline partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

[10] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In Talcott [45], 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] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[13] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[14] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[15] Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. Technical Report BRICS RS-98-2, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 1998.

[16] Olivier Danvy, Nevin C. Heintze, and Karoline Malmkjær. Resource-bounded partial evaluation. *ACM Computing Surveys*, 28(2):329–332, June 1996.

[17] 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.

[18] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[19] Pascal Fradet. Syntactic detection of single-threading using continuations. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 241–258, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[20] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[21] 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.

[22] John Hughes. Type specialisation for the lambda calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. Springer-Verlag.

[23] 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.

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

[25] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[26] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

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

[28] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[29] 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° 2265, INRIA*, pages 112–119, Orlando, Florida, June 1994. Also appears as Technical report CMU-CS-94-129.

[30] Karoline Malmkjær and Peter Ørbæk. Polyvariant specialization for higher-order, block-structured languages. In William L. Scherlis, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–76, La Jolla, California, June 1995. ACM Press.

[31] Austin Melton, David A. Schmidt, and George Strecker. Galois connections and computer science applications. In David H. Pitt et al., editors, *Category Theory and Computer Programming*, number 240 in Lecture Notes in Computer Science, pages 299–312, Guildford, UK, September 1986. Springer-Verlag.

[32] 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.

[33] 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.

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

[35] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1992.

[36] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.

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

[38] David A. Schmidt. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[39] 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.

[40] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, October 1988.

[41] Peter Sestoft. Replacing function parameters by global variables. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, September 1989. ACM Press.

[42] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In Talcott [45], pages 150–161.

[43] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

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

[45] 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.

[46] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In Boehm [8], pages 188–201.

[47] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1989. Special issue on ESOP'88, the Second European Symposium on Programming, Nancy, France, March 21-24, 1988.

Recent BRICS Report Series Publications

RS-99-27 Olivier Danvy and Ulrik P. Schultz. *Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure*. September 1999. 57 pp. To appear in the November 2000 issue of *Theoretical Computer Science*. This revised report supersedes the earlier report BRICS RS-98-54.

RS-99-26 Jesper G. Henriksen. *An Expressive Extension of TLC*. September 1999. To appear in Thiagarajan and Yap, editors, *Fifth Asian Computing Science Conference*, ASIAN '99 Proceedings, LNCS, 1998.

RS-99-25 Gerth Stølting Brodal and Christian N. S. Pedersen. *Finding Maximal Quasiperiodicities in Strings*. September 1999. 20 pp.

RS-99-24 Luca Aceto, Willem Jan Fokkink, and Chris Verhoef. *Conservative Extension in Structural Operational Semantics*. September 1999. 23 pp. To appear in the *Bulletin of the EATCS*.

RS-99-23 Olivier Danvy, Belmina Dzafic, and Frank Pfenning. *On proving syntactic properties of CPS programs*. August 1999. 14 pp. To appear in Gordon and Pitts, editors, *3rd Workshop on Higher Order Operational Techniques in Semantics*, HOOTS '99 Proceedings, ENTCS, 1999.

RS-99-22 Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir. *On the Two-Variable Fragment of the Equational Theory of the Max-Sum Algebra of the Natural Numbers*. August 1999. 22 pp.

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.