Implicit and Explicit Aspects of Scope and Block Structure

Ulrik Pagh Schultz

May 15, 1997

Abstract

Block structure is a fundamental mechanism for expressing the scope of variables in a computation. Scope can be expressed either explicitly with formal parameters or implicitly with free variables. We discuss the transformations between explicit and implicit scope. Making scope explicit is current practice in functional programming: it is called "lambda-lifting."

Our thesis addresses the transformations between explicit and implicit scope. We show lambda-dropping to be a useful transformation that can be applied to clarify the structure of programs and to increase the efficiency of recursive functions. In addition, we demonstrate that lambda-dropping is of practical use as a back-end in a partial evaluator.

Preface

This document describes most of the work that was done in conjunction with the author's "speciale" (master's thesis). One of the primary subjects of this work is a program transformation called "lambda-dropping." It was first conceived by my thesis advisor, Olivier Danvy. Many thanks go to him, both as a source of inspiration but also for the excellent job he has done in guiding my way through the pitfalls of creating something as complicated as a speciale. The article on lambda-dropping that we wrote together served to provide a solid foundation for this work. Thanks are also due to Tue Jakobsen for the many discussions of related topics, and for assistance with a few exotic features of LaTeX. Finally, John-Anker Cornelliusen and Lars Clausen were a source of inspiration also.

Many of the diagrams within these pages were typeset using Kristoffer Rose's excellent XY-pic package. Part of this work deals with block structure in general. It is a general tendency of block-structured programs to grow wider as they mature. For this reason, little space is left for any would-be marginal notes.

The chapters of this thesis are semi-independent. A reader interested only in lambda-dropping can skip to Chapter 4 for a few introductory examples, or even straight to Chapter 3. For a discussion of lambda-dropping in relation to partial evaluation, one can skip directly to Chapter 7.

Contents

List of Figures

Chapter 1

Introduction

1.1 Scope and Block Structure

Block structure and lexical scope stand at the foundation of most modern programming languages. This thesis discusses transformations between block-structured programs with scope and programs consisting of recursive equations. Lambda-lifting maps block-structured programs to recursive equations. Because of the lack of scope the context of each function must be explicitly carried around in the formal parameters. Lambda-dropping is the symmetric transformation. Programs consisting of recursive equations are given block structure and the context thus created is used to reduce the number of formal parameters.

Programs that are lambda-lifted suffer from a chronic inflation of parameters potentially degrading their performance and obfuscating the natural structure of the program. In the words of Alan J. Perlis, "if you have a procedure with ten parameters, you probably missed some." Lambda-dropping makes clear the structure of the program by employing block structure to hide auxiliary declarations and by refraining from passing unchanging parameters.

Lambda-dropping offers a convenient way of undoing the effects of lambda-lifting. Programs are often lambda-lifted in conjunction with other processes. Lambda-dropping can be applied as an independent tool to restore block structure and lexical scope.

1.1.1 Background and conventions

The basic concept of block structure should be known by anyone who has done advanced programming in any block-structured language. To properly understand and appreciate the lambda-lifting and lambda-dropping transformations, familiarity with functional programming languages is a must. Experience with compiler design and implementation will greatly enhance a reader's understanding of this entire thesis. To properly appreciate the chapter discussing lambda-lifting and lambda-dropping in relation partial evaluation the reader should be familiar with partial evaluation, even though we do provide a short introduction to the basic terms and ideas of partial evaluation.

Some languages make a sharp distinction between procedures and functions. Throughout this thesis we consider procedures and functions and even the methods of object-oriented languages to be the same thing. In the context of functional languages we use the term function when discussing ideas pertaining directly to the entity being a function. We use the word procedure as a general term covering both the procedures of imperative languages, the functions of functional languages, and the methods of object-oriented languages. Similarly, we make no distinction between the statement of an imperative language and the expression of a functional language unless doing so leads to an ambiguous situation.

In a language like Haskell the global functions of a program are written as equations, in effect making them recursive equations. In a language like Scheme, the global functions are explicitly constructed as functions, but also they can be regarded as recursive equations. Since the difference is purely syntactical, we will use both terms interchangeably, depending on which is best suited to convey the correct intuition. On a similar note, some languages such as Scheme make a strong distinction between mutually recursive let blocks (letrec blocks) and non-recursive let blocks. In a lazy language like Haskell there is no difference. Throughout most of this thesis letrec blocks are our primary interest (be it in Scheme or in Haskell) so unless otherwise noted a let block has mutually recursive definitions.

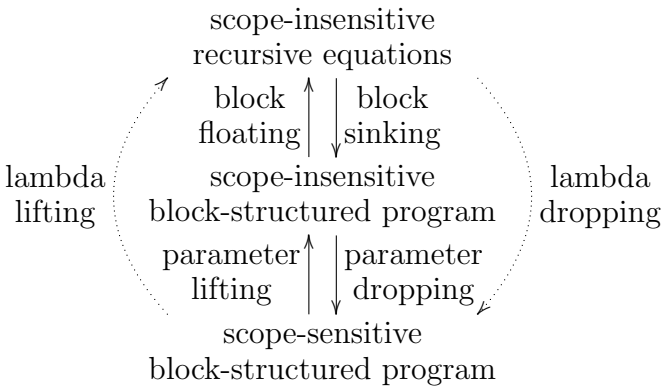
The main points of this thesis and several minor points pertains to program transformations. We work with program transformations that preserve the semantic meaning of the program. A transformation that may change the result produced by a program is not of interest. At this point is should be noted that many of the transformations presented in this thesis are incompatible with direct side-effects on variables. This restriction can be circumvented, but we will not treat this aspect any further, as it is not considered to be of interest in relation to the topics presented in this thesis.

As a final note on vocabulary, we use the term "flow graph" to mean the flow of values through the program by procedure application, not just the flow of control. The flow of values can be more fine-grained than the flow of control.

1.1.2 Overview

The rest of this chapter is devoted to explaining well-known aspects of compiler theory in the context of lambda-lifting and lambda-dropping. The next chapter extends this by describing individual pieces of theory that relate to but are not part of the lambda-dropping program transformation.

Lambda-lifting and lambda-dropping are presented in a symmetric way. These transformations have many aspects in common, and are best understood together. Parameter lifting a program makes it scope insensitive. Subsequent block floating transforms the program into recursive equations. Block sinking re-creates block structure. Parameter dropping restores scope sensitivity.



Chapters 3 and 5 presents these transformations. In between is a chapter that provides an introduction to the basic considerations of lambda-dropping. Chapter 6 provides a discussion of many topics of lambda-lifting and lambda-dropping as symmetric transformations but with a focus on lambda-dropping. Chapter 7 outlines the main application of lambda-dropping: partial evaluation.

1.2 Block Structure

The origins of block-structure can be traced back to ALGOL 60 [5, 36], known as "The Father of Modern Programming Languages." This was the first language with a syntax formally defined in BNF. It spirited many features found in most modern programming languages, such as structured control statements, first-class procedures (akin to those of Pascal), and a powerful support for block-structure. In addition, ALGOL 60 enforced the now common style of programming where each declaration, be it type, variable or procedure, is bound to a name. Chapter 8 concludes.

1.2.1 Basic concepts

A named declaration has a *scope*. The scope of a declaration is the set of statements in which the declaration can be referred to by its name. Declarations that can be referred to throughout the entire program are *global*. Block structure offers a convenient way of expressing the scope of declarations by making them *local* to a block. In the general case, a block contains a set of declarations along with a number of statements (each of which may itself be a block). When the flow of control enters a block, the local declarations become visible. When the execution of a block is completed, the local declarations become invisible. In a block-structured program a restricted set of names are visible to each declaration. A name is visible to a declaration if the scope of the name extends to the declaration. Declarations can include anything identifiable by a name although the types of declarations that can be made locally are restricted in some languages. Type declarations, for instance, can only be made globally in many languages.

Block structure is often associated with lexical scope, since these scoping rules are most closely related to the textual appearance of a block-structured program. However, as certain Lisp dialects demonstrate, dynamic scope is a viable alternative. To find the point of declaration of a variable using lexical scope the enclosing blocks are searched starting with the immediately enclosing block and continuing outwards. The defining point of a variable is static i.e. it can be determined at compile time. This is opposed to dynamic scope where the defining point of a variable is found by traversing the call chain in reverse. The defining point is dynamic i.e. it may depend on the input of the program. In a program with proper variable hygiene lexical scope is equivalent to dynamic scope, although one type of scope may be more efficient than the other in some situations.

Dynamic scope allows for a special style of programming well suited for systems where name lookup needs to be done dynamically. The object-oriented language CLOS is defined using dynamically scoped Lisp [30]. Most features of the Gnu Emacs editor are programmed in the dynamically scoped lisp dialect Elisp. The program transformations described in the later chapters are designed for lexically scoped languages. To avoid confusion, we assume that we only work with such languages.

Procedures are parameterized blocks. The formal parameters are supplied as arguments upon invocation. The semantics for binding an expression that is passed as an argument to the name of a formal parameter are discussed in Section 1.4. The formal parameters of the procedure are declared by the procedure. They are visible within the body (the block) of the procedure only. Blocks declaring local variables may be nested within the procedure. The formal parameters of a procedure are an explicit context for the computation of the body of the procedure.

A variable that is used in a block but not declared locally to the block is a *free variable*. We use lexical scope so the variable is declared in an enclosing block. The free variables of a block form an implicit context for the block. They must be available whenever the block is entered. The declarations of the blocks that enclose a procedure thus forms an implicit context for the procedure. Block structure can thus be used to express the context of a computation. The closure of a block is a structure holding the values of the free variables. The closure of a procedure holds the implicit context of the procedure. Closure representations are discussed in Section 1.3.

1.2.2 Motivation

Block structure extends the expressibility of a language. The implicit context of a computation can be expressed using block structure. This allows the programmer to convey a value to a procedure either through an explicit context or an implicit context. Delimiting the scope of a declaration using block structure offers modularity and makes clear the distinction between public procedures and local procedures. The former should be available to most of the program, the latter only to a limited set of procedures.

The reverse function on lists can be defined by two recursive equations:

```
reverse nil     = []
reverse (x:xs) = (reverse xs)++[x]
```

This definition of the reverse function will use time proportional to the square of the length of the list to reverse the list, rather than linear time. To remedy this we define reverse in terms of an auxiliary function that carries an accumulator:

```
fastrev xs = rev xs []
rev [] ys     = ys
rev (z:zs) ys = rev zs (z:ys)
```

The result is a function that runs in linear time in the length of the list. The definition of the `fastrev` function is made using three recursive equations. Only one of these needs to have global scope. The definition of the `rev` function is used by `fastrev` only. Giving both functions global scope conveys the notion that `rev` should be used by itself. Although this might be sensical in certain cases it was not our intention. Declaring `rev` as a local function more clearly indicates that it is an auxiliary function to `fastrev`:

```
fastrev xs = let rev [] ys     = []
                 rev (z:zs) ys = rev zs (z:ys)
             in rev xs []
```

A local function has access to the formal parameters of the enclosing function. The enclosing function forms an implicit context for the computations of the local function. Consider the standard definition of the function map defined as recursive equations:

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Map traverses the list structure applying the same function `f` to each element of the list. The formal parameter `f` does not change during the list traversal. There is thus no need to provide `f` through an explicit context in every invocation of the procedure. We should implicitly know that the function to apply is `f`. The current list element does change in every invocation so it must be explicitly specified through a formal parameter. We define map in terms of a local function `loop` that traverses the list structure:

```
map f xs = let loop [] = []
               loop (y:ys) = (f y):(loop ys)
           in loop xs
```

It is now clear that the same function `f` is applied to all elements of the list. The formal parameter `f` is implicitly available to the computations of the function `loop`.

These examples are fairly simple, but similar problems can be demonstrated in many functions traversing data structures. This includes folding functions that pass both the

operator used to reduce the structure and the initial value as an argument through the entire traversal of the structure. Interpreters pass the environment as an explicit argument for each syntactic form, even when no new name bindings have taken place. Instead, a local function with an implicit environment could handle constructs that make no new bindings. Recursive equations offer no linguistic support for modularity, and their inflation of parameters is chronic.

1.2.3 Block structure in contemporary languages

Most high-level contemporary computer languages include linguistic support for block structure. Pascal is known for its adaption of many features of ALGOL, and thus forms a classical example. However, the block structure is slightly restricted compared to ALGOL. A block can only be formed as the body of a procedure. Procedures can be nested, but a block containing declarations and a statement cannot be constructed to itself be syntactically equivalent to a statement. It is possible to form nested compound statements, but these cannot contain declarations. As is the case for most other languages, only procedures and variables can be declared local to a block.

The language C has no real block structure. Procedures cannot be nested. It is possible to delimit the scope of a variable within a procedure by forming a local block, but this construction if often regarded as mere syntactic sugar. C has a module system based on textual inclusion of files. The declaration of a function or a global variable can be limited to a file, providing a rudimentary facility for structuring large programs. A notable exception is the Gnu C compiler. In Gnu C, there are no restrictions on block structure. A block containing local declarations of procedures and variables is syntactically equivalent to a statement.

The language Fortran 77 is known for its speed, efficiency, and lack of features generally considered essential for high-level programming languages. It has no support for block structure save variables declared locally to a (non-recursive) procedure.

The lambda calculus has a natural support for block structure. Lambda forms can be nested arbitrarily and lexical scope ensures that changing the order of evaluation will not cause a free variable to refer to a different value during reduction. Most functional languages, including Haskell, Lisp, Miranda, ML, and Scheme support block structure without restrictions. Here, a let (or letrec) block is an expression that can bind any number of local values, including functions. In many functional languages types can only be declared to have global scope. The module system of functional languages varies greatly, ranging from simple textual inclusion to the higher-order functors of SML/NJ.

Object-oriented languages support block structure to varying degrees. The perhaps most widely used object-oriented language, C++, is a very direct descendant of C. Methods are declared within classes. This, however, has very little to with block structure. The name of a method is always visible outside the class. It might not be accessible to other classes depending on the access specifier. Arguably, all entities manipulated by an object-oriented program are objects. Methods and variables are encapsulated within objects. For an object-oriented language to properly support block structure it must be possible to

declare a class locally to some other class. Object-oriented languages like Beta and Java 1.1 provide a uniform support for block structure. Classes and objects can be declared locally to other classes. Methods and variables can be declared locally to these classes. Comparing to an imperative language, this corresponds to local declarations of variables, procedures, and types. There are many approaches to module systems in object-oriented languages. C++ retains textual inclusion from C. Beta allows objects to be fragmented across several files, providing linguistic support for a structured approach. In Java classes are grouped in packages. Package names are integrated the placement of the package within the file system. The hierarchical nature of the file system is not reflected in any notion of scope amongst packages.

1.3 Closure Representations

At the machine level, the implementation of most computer languages, be they imperative, functional or object-oriented, is identical. A program is compiled into a number of state-modifying constructs, intermixed with instructions that modify the flow of control, and memory allocation primitives. The state of a running program is represented by chunks of memory. We refer to these as vectors. Limiting the features of a language, such as higher order procedures and first-class continuations, allows for a more efficient design.

1.3.1 Implementing functional languages

Functional languages are often defined as those languages where functions can be freely passed as arguments to other functions. Many imperative languages allow similar operations, but not without restrictions. In most implementations of imperative languages, a stack is used to allocate data needed to invoke procedures and to store the local variables of procedures. This works perfectly because the invocations of and returns from first-order procedures follows a stack discipline [19]. The data needed by a procedure must be available when it is invoked. The free variables are stored in the stack in the data area of the enclosing block. If the procedure associated with the enclosing block returns, the free variables are no longer available. A higher-order procedure can be invoked at a time when the enclosing block has been popped from the stack. Thus the restrictions on the use of higher-order procedures in imperative languages.

To avoid restrictions on the use of functions as arguments in functional languages, we must ensure that the free variables of a function are available to the function even when the enclosing block has been evaluated (and returned from). A heap can be used to hold some or all of the data associated with each procedure. At the very least the free variables of each function must be available to the function whenever it is invoked. A structure holding the free variables of a function and possibly a reference to the code of the function is called the *closure* of the function. Allocating this structure in a heap allows access to the data of a function after the block that defined the function has been evaluated. A storage manager can be used to reclaim data that cannot be used in a future computation.

$$\begin{array}{l}\text{let } p\,(a_1,a_2,a_3,a_4) = \text{let } q\,(b_1,b_2,b_3) = \text{let } r\,(c_1,c_2) = \quad \text{let } s\,(d_1) = +\,(d_1,c_2,b_3,a_4) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } s\,(c_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } r\,(b_1,b_2) \\ \qquad\qquad\qquad\qquad\qquad\quad \text{in } q\,(a_1,a_2,a_3) \\ \text{in } p\,(\alpha,\beta,\gamma,\eta)\end{array}$$

Figure 1.1: The sum of four numbers, (mis)using nested block structure

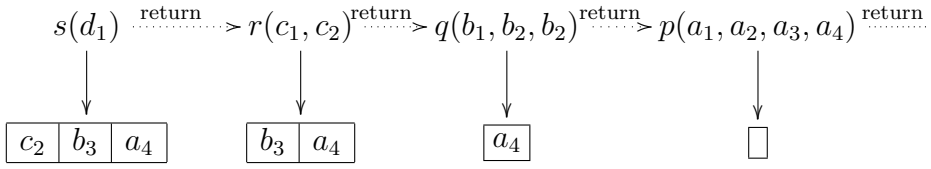


Figure 1.2: Flat closure representation of the program of Figure 1.1

Storage management techniques are described briefly in Section 1.3.5.

A function is invoked with a list of formal parameters and has access to its closure. The formal parameters forms the explicit context of the computation that takes place within the procedure. The closure provides the implicit context (from the enclosing blocks) of the computation. The next three sections describe different ways of representing the implicit context of a computation as a closure. We use the program of Figure 1.1 to illustrate how the closure of a procedure is constructed in each case. The program calculates the sum of four global variables in a complicated way. It allows us to compare the different closure representations.

1.3.2 Flat closures

The flat closure representation favors simplicity of design and quick lookup of free variables at the cost of possible extra overhead when creating closures. When creating a closure for a procedure, any free variables of the procedure are placed in a single vector. All variables that could be referenced from within the procedure are saved in this vector. The advantage of this representation is that looking up a free variable is both simple and quick. A single vector lookup is sufficient. We thus use a number of operations to ensure that the variable can be looked up quickly, even if it is never used. A procedure that is passed as an argument will be associated with a single vector holding its free variables. A free variable may end up being duplicated across several vectors, complicating the implementation of side-effects. To implement side-effects in this setting, pointers to boxed values are passed as

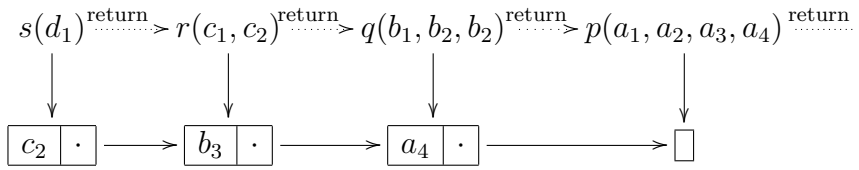


Figure 1.3: Deep closure representation of the program of Figure 1.1

values rather than the values themselves. Figure 1.2 illustrates the flat closure structure of the summation program of Figure 1.1 at the point of execution where the addition function is about to be invoked. To compute the sum of the four variables, the formal parameter d_1 must be passed to the addition function along with the three free variables. The value of each of these free variables is determined by a single vector lookup each.

1.3.3 Deep closures

The deep closure design emphasizes cheap closure creation and tries to minimize space consumption at the cost of potentially expensive lookups of free variables. In this model, only those free variables that are declared in the immediately enclosing block are placed in the closure vector of a procedure. The closure vector contains a pointer to the closure of the enclosing block. The closure of a procedure is thus a linked list of closure vectors. Only variables that occur as free variables are saved in the closure, and each variable is represented only once in the entire structure. Over time, the closure structure may form a tree with the closure of the outermost block as its root. Looking up a variable takes a number of operations proportional to the lexical distance in enclosing blocks from the point of use to the point of declaration. The closure of a procedure holds not only the free variables of the procedure but also the free variables of all enclosing procedures. This may result in a space leak where a disproportionate amount of space is used to represent a procedure with only a few free variables. A clever storage managing algorithm could repair the leak, but not without overhead. Figure 1.3 illustrates the deep closure structure of the summation program. To compute the sum of the four variables, we must traverse the closure structure all the way to the root to look up the values of the free variables.

1.3.4 Other closure representations

There are numerous other ways of representing closures. Some representations will perform better than others in some situations, and usually vice versa. Appel presents no fewer than six alternatives to flat or deep closures, remarking that "Clearly, there is an infinite variety of closure-representation strategies." [2]. Most of these representations try to combine the best of flat closures with the best of deep closures. With flat closures free variables can be

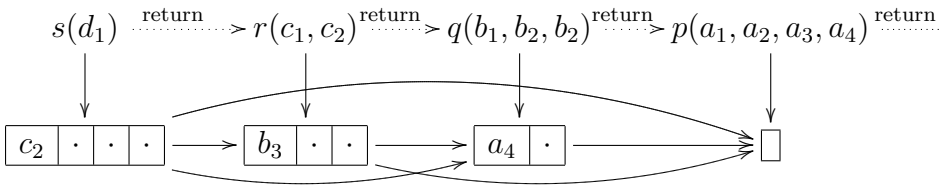


Figure 1.4: Hybrid closure representation of the program of Figure 1.1

looked up quickly. With deep closures each variable is represented only once in the closure structure. Closures that are shared between functions that share free variables is an easy optimization. We present a simple hybrid closure scheme that illustrates the basic idea of combining flat closures and deep closures.

A closure is created by placing the free variables of the immediately enclosing block in the closure vector and also including a pointer to the closures of all enclosing blocks. Looking up a free variable involves at most a single dereferencing operation through a vector. Each variable is represented once, but we still suffer the problem of space leaks. Furthermore, the size of each closure vector grows with the depth of the block structure. Figure 1.4 illustrates the closure structure of the summation program when using hybrid closures. To compute the sum of the four variables, three pointers must be dereferenced to look up the values of three of the variables. As can be seen, the number of pointers used in this closure representation makes it impractical. By using a more clever representation, the number of pointers per closure can be reduced to two on the average while retaining a $\log n$ bound on the number of pointers that must be dereferenced to look up a variable (where n is the depth of the block structure) [2].

1.3.5 Memory-management strategies

A running program that needs to allocate data that may be needed after the flow of control has abandoned the current scope often uses a heap for this purpose. A heap is a collection of vectors. Each vector can contain data and pointers to other heap vectors. A new vector can be allocated in the heap by a request to the storage manager. It is preferable that the space occupied by vectors that are no longer used is recycled to be usable for fresh data. In some implementations the running program must explicitly deallocate data, signaling to the storage manager that a vector is no longer used. If a vector is deallocated and subsequently used, data may be corrupted. If a vector is no longer used but not deallocated the space can never be reclaimed.

Many modern languages extend the storage manager with a garbage collector. The garbage collector automatically reclaims vectors that the program no longer uses. This is essential in functional programming languages where closures have arbitrary lifetimes. As long as there is a reference to a procedure that has been passed as an argument, the closure

must not be deallocated. When the garbage collector detects that a vector is no longer referred to by any part of the program, the space occupied by the vector can be used for future allocations. Detecting whether a vector is unused is usually done in one of two ways. Each vector can be equipped with a reference count indicating how many vectors points to it. This reference count is continuously updated while the program runs. If the reference count reaches zero, the vector can be reclaimed. Reference counting alone cannot be used to reclaim cyclic structures. To handle cyclic structures, a supplemental algorithm must be used. A marking algorithm traverses all data accessible to the program, marking it as "live." Data not marked as live can be reclaimed by the garbage collector. The drawback of this approach is that, unless using a very clever algorithm, the computation must be paused while garbage collection takes place.

The vectors allocated in a heap may have arbitrary size. Allocating vectors and subsequently deallocating some of them may leave the free space of the heap in fragments. A garbage collector can compress the accessible (live) data of the program by copying the live data into a separate heap area. A traditional copying garbage collector uses a heap separated into two semispaces [9]. At any time only one of the semispaces are active. Data is always allocated in the active semispace. When the active semispace becomes full, the live data is marked and copied into the other semispace. Free space is not copied so the live data is compacted at one end of the heap. The other semispace now becomes the active semispace. There are numerous other ways of doing garbage collection, the most prominent of which is perhaps generational garbage collection [2].

Using a stack for allocation when possible is almost always faster then using a garbage collector. But as Appel points out [3], the overhead is not necessarily very big. A copying garbage collector that operates on a large heap (large compared to the amount of live data) is only activated when the heap runs full. Only the live data must be copied into the other semispace. Data that was no longer live was not even manipulated by the garbage collector. After the copying operation the live data is compacted potentially improving cache performance.

1.4 Parameter-passing Semantics

The program transformations described in later chapters manipulate the formal parameters of procedures. The correctness of these transformations depends on the parameter-passing semantics of the language that is being transformed. For this reason we discuss the most common parameter passing semantics in the light of these transformations.

Call by name Under call by name parameter passing semantics an expression is encapsulated in a self-contained thunk (a closure) before passing it as an argument. Each time the value of the formal parameter that the expression became bound to is used the thunk is evaluated. If the formal parameter is not used the argument is never evaluated.

Call by value Under call by value parameter passing semantics an expression is completely evaluated before it is passed as an argument to a procedure. The formal parameter is simply bound to the value of the expression. The expression that was passed as an argument is thus evaluated exactly once. A functional language that uses call by value semantics is said to be strict.

Call by reference Call by reference parameter passing is most useful as supplement for call by value semantics. Call by reference allows a variable to be passed by reference rather than by the value of the variable. The variable is not evaluated (this would create a copy of the value of the variable). Making side-effects to the formal parameter changes the variable that was passed as an argument.

Call by need Under call by need parameter passing semantics an expression is encapsulated in a thunk (closure) before passing it as an argument. However, using the value of the formal parameter that was bound to the thunk causes the thunk to overwrite itself with the result of the computation encapsulated in the thunk.[1] The expression within the thunk is evaluated only if the formal parameter is used, and in this case at most once. A functional language that uses call by need semantics is said to be lazy.

Parameter passing semantics can be defined in many ways. Call by substitution is commonly used for macro expansion. Call by copy/restore is used in conjunction with RPC calls in distributed systems. Both of these are incompatible with the program transformations presented later on, and will not be considered any further.

1.5 Summary

Block structure is a widely used and very useful structuring mechanism for programming languages. Block structure can be used to delimit the scope of a declaration and to create an implicit context for a computation. To implement block structure in languages with higher-order functions each procedure must be associated with a closure structure that can outlive the current scope. The closure provides the implicit context for the procedure. There are many different ways of creating closures in functional languages, all of which require storage management beyond the capabilities of a stack. The process of passing an argument to a procedure and binding it to a formal parameter is described by the parameter passing semantics of the language. Most languages use at least one of four parameter passing semantics, namely call by name, call by value, call by reference, or call by need.

[1] This is a self-modifying model. This is basically the approach used in the Glasgow Haskell Compiler [29]. Call by need parameter passing semantics can be implemented in many ways.

Chapter 2

On Block Structure and Trees

2.1 Introduction

This chapter presents two independent topics both of which are related to block structure. These topics are put to use in later chapters. They each form self-contained topics rather than integral parts of some other chapter.

Ever since the advent of block structured programming languages with the definition of ALGOL, the textual appearance of programs forms trees. We formalize this relationship in Section 2.2, and present a mapping from block structured programs to trees. We provide a graphical way of representing the block structure of a program and present a translation from this graphical representation to textual programs.

The later chapters present several program transformations. Section 2.3 presents a framework for evaluating such program transformations. We discuss the aspects of constructing an abstract model for evaluating the computational cost of a program. Based on this discussion, we present a sufficiently detailed model for evaluating those computational costs that we deem to be of interest.

2.2 Isomorphism of Block-Structure and Trees

The syntactic appearance of a program forms a tree. The program of Figure 2.1 is displayed alongside a graphical representation of the structure of the program. This informal tree representation has been with us since block structure was first conceived. In this section, we demonstrate that restricting the syntactic appearance of a program appropriately yields a program the block structure of which is isomorphic to a tree. We describe a meaning-preserving translation from such programs to trees. The tree of a program has expressions without block structure as its vertices. Formalizing the structure of programs with block structure isomorphic to a tree becomes important later on. The program transformation of Chapter 5 produces programs by mapping trees to programs. Programs produced by this transformation will thus conform to the restrictions outlined in this section.

```
compile = λcharStream.
    let parse = λcharStream.
            let lex = λcharStream. ...
                syntax = λtokStream. ...
            in syntax (lex charStream)
        threeAddrCode = λast. ...
        optimize = λtac.
            let defUse = λtac.
                    let pointsTo = λannTac. ...
                        flowGraph = λtac. ...
                    in ...
                deadCodeRem = λannTac. ...
            in deadCodeRem (defUse tac)
        codeGen = λannTac.
            let assemble = λannTac. ...
            in ...
    in ...
```

```
compile
┌ parse
│ ┌ lex
│ └ syntax
│ threeAddrCode
│ optimize
│ ┌ defUse
│ │ ┌ flowGraph
│ │ └ pointsTo
│ └ deadCodeRem
│ codeGen
└ ┌ assemble
```

Figure 2.1: The block structure forms a tree

2.2.1 Block structure in normal form

In most block-structured languages, such as Gnu C and functional languages in general, a block is treated as a single expression. On the contrary, a language such as Pascal restricts the use of blocks by embedding them in procedure declarations. The former allows blocks to be placed within other constructs, such as conditionals or loops. The latter restricts blocks by only allowing them to appear directly within procedure declarations.

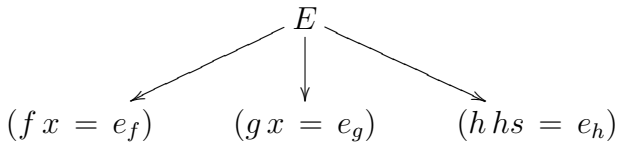
We wish to express the block structure of a program as a graph. The syntactic appearance of a program forms a tree. In compiler terms, this is the abstract syntax tree (AST). Since the block structure of a program is embedded in the program, a graph expressing the block structure of a program is a tree. We refer to the block-structure graph of a program as the scope tree.

The structure of a graph is induces by its edges. Thus, the block structure of a program should be reflected in the edges of the scope tree. The vertices of a graph holds values that are structured by the edges. Thus, the vertices of a scope tree should contain the expressions of the program, and each expression should be without block structure.

The order of declarations in a block of mutually recursive procedures is unimportant. Thus, we can represent a block graphically by a vertex holding the main expression having edges to each locally declared procedure. The edges are read as “declares.” For example, the block

$$
\begin{array}{l}
\text{let } f\,x = e_f \\
\quad\;\; g\,x = e_g \\
\quad\;\; h\,x = e_h \\
\text{in } E
\end{array}
$$

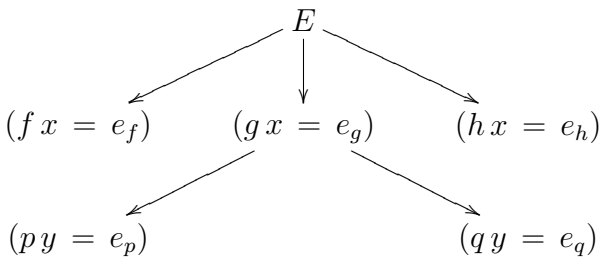
is represented graphically as



The interpretation of the tree is that the block with main expression E declares the functions f, g, and h. This representation naturally extends to functions declared locally to other functions:

$$
\begin{array}{l}
\text{let } f\,x = e_f \\
\quad\;\; g\,x = \text{let } p\,y = e_p \\
\qquad\qquad\;\; q\,y = e_q \\
\qquad\quad \text{in } e_g \\
\quad\;\; h\,x = e_h \\
\text{in } E
\end{array}
$$

is represented graphically as



This representation explicitly models the block structure induced by procedure definitions. The program transformations presented in later chapters manipulate procedure declarations, so it makes sense to explicitly represent the structure of the procedure declarations of a program in the scope tree of the program.

Moving on to other expression types presents us with some difficulties, however. An an example, consider a conditional expression. Both the condition, the consequence, and the alternative could contain blocks declaring local functions. The natural representation would be the typical representation in an AST. To make a dedicated conditional vertex and let the edges relate to it its three sub-expressions. Unfortunately, we would be using the edges to construct an expression, not represent block structure. Insisting that the edges of the graph represents block structure implies that we cannot represent graphically blocks that are nested inside arbitrary expressions. Nesting blocks inside certain expressions types

is not block structure. However, we cannot represent the structure of the such expressions graphically without using edges to construct expression.

Our assumption of variable hygiene makes any let block equivalent to a letrec block. The textual structure of a let block is identical to that of a letrec block. It thus makes sense to represent a let block graphically in the same way as a letrec block. Interpreting the graph as a program will produce letrec blocks only. For a program to be isomorphic to a graph, we must thus insist that the program has letrec blocks only. An alternate solution is to treat let blocks as expressions rather than blocks. Each let block will be part of the expression in a node rather than represented graphically. If used as part of a program transformation that focuses on letrec blocks rather than let blocks, this is a suitable graphical representation.

Nesting of letrec blocks inside one another cannot be represented graphically (i.e. the body of a letrec block is itself a letrec block). An edge between two nodes indicate that an expression is local to a block. The expression of the father node is the body of a block. To represent nested blocks we would thus need to place an entire sub-tree within the father node.

A program consists of a number of global functions and, possibly, a main expression. If the program has a main expression, it is the root node of the scope tree of the program. Otherwise, we use an empty node to represent the root. In both cases, the root node has an edge to each global function. The graphical representation of a program is equivalent to the graphical representation of a letrec block.

Definition 1 *A program is in Block-Structured Normal-Form (BSNF) if neither blocks nor procedure declarations occur inside other expression types, and no block or procedure declaration forms the body of another block.*

Programs in BSNF have procedure declarations and blocks (letrec blocks in the case of functional languages) as their outermost constructs. Inside these, as the body of a block or a procedure declaration, are other expression types. The program of Figure 2.1 is in BSNF. The block structure of the program is depicted graphically in Figure 2.2. Any program can be transformed to BSNF by a number of transformation steps, the basic steps of lambda-lifting. These steps are described in Section 3.2. The transformation to BSNF is described along with lambda-lifting in Section 3.3.

The lambda-dropping transformation presented in Chapter 5 creates block structure using the isomorphism of trees and block structure. Lambda-dropping a program thus produces a program in BSNF. Iterating the lambda-dropping algorithm still produces programs in BSNF.

2.2.2 Block structure and trees

This section describes in detail the isomorphism between block structure and scope trees. We assume that a program has a main expression. Programs consisting of a number of global functions without a main expression are considered to have an empty main expression. We can thus view a program as a letrec block. We describe a mapping from

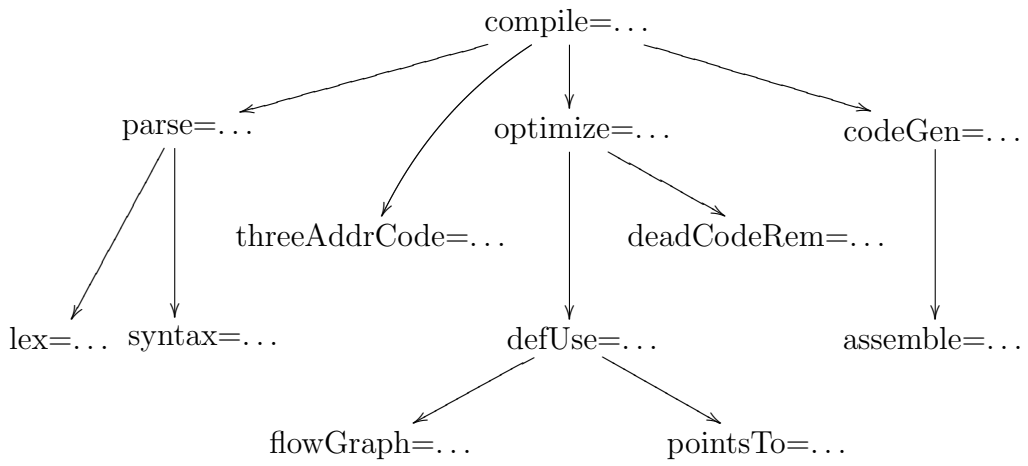


Figure 2.2: The program of Figure 2.1 represented as a graph

programs in BSNF to scope trees and a mapping from scope trees to programs in BSNF. We then demonstrate that this mapping gives rise to an isomorphism between scope trees and programs in BSNF.

A program in BSNF is mapped to a scope tree by using a compositional mapping $\mathcal{T}$ on each global function. The mapping $\mathcal{T}$ on a function f that declares no local functions is a node containing f. The mapping $\mathcal{T}$ on a function f that declares a number of local functions $g_1, \ldots, g_n$ in a letrec block with body b yields a tree with a root node containing f rewritten to have b as its body. The functions g_i are mapped to a tree by $\mathcal{T}$ and the root node of each resulting tree is made a direct successor of the node of f. At the global level a node is created for the main expression and the root nodes of the trees that result from mapping $\mathcal{T}$ onto the global functions are made direct successors of this node.

A scope tree is mapped to a program in BSNF by using a compositional mapping $\mathcal{P}$ on each subtree of the root node. The mapping $\mathcal{P}$ on a node N with no successors that contains a function f is the function f. The mapping $\mathcal{P}$ on a node N containing a function f and with successors $M_1, \ldots, M_n$ yields a rewritten version of f. Let b be the body of f, and $g_1, \ldots, g_n$ the result of applying $\mathcal{P}$ to the nodes $M_1, \ldots, M_n$. The result is the function f where the body has been replaced by a letrec block that declares the functions $g_1, \ldots, g_n$ and has b as body. The extent of the expression that forms the "body" of a function is limited by the body not being a function. To handle the root node, the expression of this node is made the main expression of the program, and the global functions are the results of mapping $\mathcal{P}$ onto the direct successors of the root node.

The scope of a function declared in a letrec block extends to the function itself, to the body of the block, and to any other functions declared in the block. In a scope tree, the scope of a function extends to the function itself, to the direct predecessor (this corresponds to the body of the block), and to any successor of the predecessor (this corresponds to

functions declared in the same block).

The mapping from programs in BSNF to trees is injective modulo the ordering of bindings in each letrec block: There is one and only one way to produce a tree from a program, and vice versa. Since the mapping is compositional, it forms an isomorphism between programs in BSNF and trees.

2.3 Abstract Model of Computation Costs

This section presents an abstract model of computational costs. We discuss how to measure the cost of a computation in reference to the hardware of a traditional computer system. We use this abstract model to describe the costs associated with different implementations of functional programming languages and an imperative programming language. In Chapter 6 we employ this model to evaluate the program transformations presented in chapters 3 and 5.

2.3.1 Considerations

A running program consumes system resources. System resources include CPU cycles, memory, and available external devices. Disregarding Input/Output, a running program uses the CPU to perform computations and uses the memory store results. Most modern computers have a multi-layered memory. This includes the CPU registers, one or more caches, the main memory, and external storage for swapping. A program that manipulates a large amount of data will place a higher load on the memory system than a program that manipulates a small amount of data.

The number of cycles consumed by an instruction depends on the complexity of the instruction and whether the values it operate on are available in the CPU registers. The availability of such values depend on when this data was last manipulated. The older the data the greater the chance that it is only available in a slow layer of the memory. Informally, this is known as register spilling. If a value is spilled from the registers, it is highly unpredictable at which level of the memory the value will be available. This depends on the location of other data that was manipulated and on the caching strategy of the hardware that provides the multi-layered memory. If the program is running under an interpreter, the performance becomes even harder to predict.

In most cases, decreasing the number of instructions and limiting the memory use will result in a faster program. When transforming a program, it thus makes sense to estimate the number of instructions and the amount of data that is used by the program. This estimate can be used to evaluate whether the transformation is likely to cause the program to run slower, at the same speed, or faster. The amount of data that is used should be measured not only in terms of the amount of memory allocated by the program. It is also important to know how much data is read from memory and how much data is written to memory. The amount of data that the program needs to keep track of at the same time (the amount of live data) is also highly relevant when estimating performance, as is

dependencies between different sets of data. The former can be used to predict the amount of register spilling, the latter to predict performance in a pipelined architecture. Here, data should be loaded from memory several CPU cycles ahead of the instruction that needs to manipulate it. Dependencies between data may force us to order the instructions in a sub-optimal way.

In a functional program where the computations performed within a procedure depends on the input values, some of these estimates can be made in a fairly simple way. The number of instructions consumed by a procedure can be computed as a function of its formal parameters. The amount of memory allocated, read, and written is likewise fairly simple to compute. The number of live variables can be determined using a liveness analysis. This is a non-trivial computation, so we leave it out for now to keep the estimate simple. The same holds for data dependencies. Both of these are treated in Section 2.3.5.

2.3.2 The abstract machine

The lambda-dropping program transformation that is the primary subject of this thesis transforms procedure declarations and applications. To study the effect of this transformation in different settings, the abstract machine must be low-level enough for us to be able to specify how a procedure call takes place. On the other hand, the higher the level of the machine the simpler it is to specify programs within its framework.

We have chosen an abstract machine that as its basic instructions has operations manipulating the flow of control (conditionals and jumps), primitive functions (arithmetic and datatype construction), instructions to manipulate a garbage-collected heap of vectors, and instructions to manipulate a stack of frames. Data that does not obey a stack discipline can be stored in the heap. Vectors allocated in the heap must be initialized since they are manipulated by the garbage collector. Stack frames need not be initialized since they are simply reclaimed by popping the stack. The stack can be compressed linearly by eliminating any unreferenced frames (these could crop up during tail calls, for example) Pointers to the current continuation, a closure structure holding free variables, and the return address are maintained automatically by the abstract machine.

Instruction set

We describe the specific instructions in an order given by their type. Some of the instructions have tags that either indicate a generic operation or simply are a shorthand notation for the instruction.

- The basic operations. Instructions such as “if,” “call,” and “return” manipulate the flow of control. The tag `stmt` denotes a generic instruction that manipulates the flow of control. Instructions such as “null?” compute a function on available data. The tag `op` denotes generic functions. Instructions such as “car” and “cdr” indirectly fetch values from memory; these instructions are denoted by the tag `cxr`. The instruction “cons” allocates memory and writes to it; this instruction is denoted by the tag `cons`.

- Allocation operations. Memory can be allocated either as a frame or a vector. The instruction "allocFrame" pushes a stack frame that will be automatically popped by the return instruction. The instruction "allocVector" allocates and initializes a vector in the heap.

- Setting a memory value. There are instructions for setting the values of frames and vectors. Manipulating vectors is considered to be slightly more expensive than manipulating stack frames because of the extra overhead associated with garbage-collection. The instruction "setFrame" sets the value of an entry in a frame. The instruction "setVector" is the vector counterpart.

- Looking up a memory value. These instructions are identical to the instructions for setting memory values, except that they read values from memory.

- Dereferencing a memory value. These instructions dereference a pointer that is stored in a vector or a frame.

- Creating a block of code. The "makeLambda" instruction returns a reference to a block of instructions that can be interpreted by the abstract machine.

We will for now assume that all models based on the abstract machine use the stack area to store continuation frames. The continuation frame holds the formal parameters, control information, and temporary values. The control information includes the stack frame that the procedure should return to, the code pointer, and the closure pointer. Many implementations place the continuation frames in the heap, facilitating an efficient implementation of operations that capture the current continuation, such as call/cc. Placing the continuation frames on a stack implies that if we provide access to the flow of control, the frames on the stack must be preserved whenever the continuation is saved. There are numerous strategies for accomplishing this efficiently, such as the segmented stacks of Hieb *et al.* [25].

Formal parameters and temporary values are stored on the stack. Operations that read these from the stack are tagged `local`. Operations that write to the stack are tagged `save`. We assume that a globally accessible vector holds either the values of or pointers to the global variables of the program, such as global procedures. We also assume that the program has been compiled. Each code segment is thus accessible at run-time. Again, this is available from a globally accessible vector. Figure 2.3 summarize these design decisions.

Each operation of the abstract machine has been assigned a computational cost. The computational cost is expressed in terms of time, space allocated, data read, and data written. The costs are assigned somewhat arbitrarily. Operations on vectors are more expensive than their stack counterparts, reflecting that vectors are more expensive in general (but not much, as discussed in Section 1.3.5). To maintain a consistent heap, vectors must be initialized. This is not necessary with stack frames. Simple operations are cheap, operations that manipulate memory are more expensive.

One could be concerned with the performance of a specific compiler and runtime system when estimating the costs of optimizations. Presumably, it would be possible to calibrate

`local`	=	lookupFrame
`save`	=	setFrame
`global`	=	lookupVector
`makeLambda`	=	lookupVector

Figure 2.3: Design choices for abstract machine

Operation	time	space	read	write
`op` (predefined function)	0.5	0	0	0
`stmt`	0.5	0	0	0
`cxr` (car or cdr)	1.0	0	1	0
`cons`	1.5	2	1	2
allocFrame(n)	1.0	n	1	0
allocVector(n)	1.5	$n+1$	1	$n+1$
lookupFrame	1.0	0	1	0
lookupVector	1.5	0	1	0
setFrame	1.0	0	0	1
setVector	1.5	0	0	1
dereferenceFrame	1.5	0	2	0
dereferenceVector	2.0	0	2	0

Figure 2.4: Basic operation costs of abstract machine

the basic costs of the abstract machine to better match those of the system that is being optimized. For now, some "reasonable" costs will suffice.

Procedure calls

We are now ready to specify a procedure call mechanism and the associated costs. The cost of calling a procedure depends primarily on the number of parameters and their configuration. Free variables must be looked up in a closure. The cost of this lookup is dependent on the closure representation. We prefer for the costs to be specified as generally as possible. For this reason, we use the costs "referenceLocal" and "referenceFree(i)" to specify the cost of looking up a formal parameter or temporary variable, and the cost of looking up a free variable respectively. The argument i indicates the distance in enclosing blocks to the defining point of the variable as a parameter (distance zero is the immediately enclosing block). A block is usually a procedure declaration. In a specific model it could be a block of declarations with a statement.

There are numerous ways of defining the semantics for passing an expression as an argument to a procedure. The most common of these were discussed in Section 1.4. The

$$\begin{aligned}
&\text{callProcedure}(n_0, n_1, \ldots, n_k) \;=\; \texttt{stmt} + \text{allocFrame}(\Sigma_{i=0}^{k} n_i + k_0) + k_0 \cdot \text{setFrame} \\
&\qquad + n_0(\text{referenceLocal} + \text{setFrame}) + \Sigma_{i=1}^{k} n_i(\text{referenceFree}(i) + \text{setFrame}) \\
&\text{tailCallProcedure}(n_0, n_1, \ldots, n_k) \;=\; \texttt{stmt} + \text{allocFrame}(\Sigma_{i=0}^{k} n_i) \\
&\qquad + n_0(\text{referenceLocal} + \text{setFrame}) + \Sigma_{i=1}^{k} n_i(\text{referenceFree}(i) + \text{setFrame}) \\
&\text{returnFromProcedure} \;=\; \texttt{stmt} + k_0 \cdot \text{lookupFrame} + \text{setFrame}
\end{aligned}$$

Figure 2.5: Generic time cost for procedure invocation

models of computation we define in this section are independent of the parameter passing semantics. The program that is evaluated within the framework of the abstract model must explicitly manipulate values as is required by a specific semantics. The compiled programs we evaluate with the abstract model are all compiled with call-by-value parameter passing semantics.

A procedure call is made by saving the control configuration, pushing the arguments onto the stack, and jumping to the address of the procedure. We use the constant k_0 to indicate the number of control configuration values that must be saved on the stack. To copy the arguments into a newly allocated stack frame, we first copy the formal parameters or temporary values that must be passed as arguments, then any free variables. For simplicity, we consider constant arguments to be local variables throughout this section. The "callProcedure" cost is defined in Figure 2.5. The first argument is the number of procedure arguments that are formal parameters. The second argument, if present, is the number of free variables at distance zero that are passed as arguments. The next argument is the number of free variables at distance one, and so on. To return from a procedure, we restore the state and set the value returned by the procedure. This is the "returnFromProcedure" cost of Figure 2.5.

Tail calls are important for functional languages, and can be equally useful in imperative languages. If the return value of a procedure is the result of a call to another procedure, a direct jump can be made this other procedure. When the other procedure returns, it returns directly to the original caller. To allow tail calls, we use the convention that control configuration values are saved in the stack frame of the caller rather than the callee. The caller still allocates space for these values in the stack frame of the callee. This space is used by the callee if it makes procedure calls. Performing tail calls may result in a fragmented stack, since we cannot pop the stack frame of the caller before allocating the stack frame of the callee (it could contain values that must be copied into the new frame). The stack can easily be compressed by traversing it in a compacting pass, starting in the current stack frame and following the return link. The "tailCallProcedure" cost is defined in Figure 2.5, with arguments corresponding to "callProcedure."

```
(let ([f (lambda (x1 x2 x3)
           (let ([g (lambda (y1 y2)
                      (let ([h (lambda (z1)
                                 (+ z1 y2 x3))])
                        (h y1)))])
             (g x1 x2)))])
  (f alpha beta gamma))
```

```
(let ([a1 (λ{} (x1 x2 x3)                                   makeLambda,save
        (let ([b1 (closure [x3])]                           clos(1),save
             [b2 (λ{b1 ← [x3^(0)]} (y1 y2)                  makeLambda,save
                  (let ([c1 (closure [y2, x3])]             clos(1,1),save
                       [c2 (λ{c1 ← [y2^(0), x3^(1)]} (z1)   makeLambda,save
                            (tail + z1 y2 z3))])            tail(1,1,1),global
                    (tail c2 y1)))])                        tail(1),local
          (tail b2 x1 x2)))])                               tail(2),local
  (tail a1 alpha β γ))                                      tail(0,3),local
```

Figure 2.6: Deeply nested program to sum three integers

The program of Figure 2.6 computes the sum of three numbers in a complicated way that illuminates certain aspects of block structure. It is a slightly reduced version of the program of Figure 1.1.[1] It will be used to demonstrate the costs of several computational models. The program is shown both in Scheme syntax and as a symbolically compiled program. The right column next to the compiled program shows the computation cost of the corresponding program line. The abbreviations are explained in Figure 2.8. In the compiled program closures are explicitly constructed and are subsequently attached onto a procedure. A procedure that is either recursive or passed as an argument will be represented by its closure (this is similar to closure conversion which is discussed in Section 3.5.1). The syntax $\lambda\{v \leftarrow [x_1^{(i_1)}, \ldots, x_n^{(i_n)}]\}$ represents a function that receives the values of its free variables from the vector v. The superscript numbers indicates the distance in enclosing blocks.

The computational cost of the instructions of the program of Figure 2.6 can be expressed as the equation of Figure 2.7. Had the program contained conditionals, these would either have to be determined statically based on the input or expressed as part of the equations. Expressing the computations cost of a program that loops will result in an unsolvable equation.

[1] The compiled version of the unreduced program is too complicated to serve as an illustrative example.

$$
\begin{aligned}
P(\alpha,\beta,\gamma) \;=\; & 3 \times \texttt{makeLambda} + 5 \times \texttt{save} + 3 \times \texttt{local} + \texttt{global} \\
& + \text{tailCallProcedure}(0,3) + \text{tailCallProcedure}(2) \\
& + \text{tailCallProcedure}(1) + \text{tailCallProcedure}(1,1,1) \\
& + \text{makeClosure}(1) + \text{makeClosure}(1,1)
\end{aligned}
$$

Figure 2.7: Computational cost of the program of Figure 2.6 as an equation

`call(`i_0`,`i_1`,...)`	=	callProcedure(i_0,i_1,...)
`tail(`i_0`,`i_1`,...)`	=	tailCallProcedure(i_0,i_1,...)
`clos(`i_0`,`i_1`,...)`	=	makeClosure(i_0,i_1,...)
`return`	=	returnFromProcedure
`vset`	=	setVector
`vref`	=	lookupVector

Figure 2.8: Abbreviations

2.3.3 Functional programs

In a functional program procedures can be passed as arguments without any restrictions. The closure of the procedure holds the context of the procedure, the free variables. The closure must be stored in the heap, as it might not obey stack discipline. There are numerous ways of representing closures. Section 1.3 provided an overview of the most common techniques. We build an abstract model for functional programs with flat closures, deep closures, and hybrid closures.

In a model with flat closures, the free variables of a procedure are stored in a single vector. Flat closures were described in Section 1.3.2. Figure 2.9 displays the computational costs for functional programs with flat closures. The costs are fairly simple; all free variables can be found in the closure vector. The cost of "callProcedure" was obtained by substituting the relevant costs into the definition from Figure 2.5. The cost for "tailCallProcedure" changes accordingly and is not shown here.

The computational cost of the summation program of Figure 2.6 can now be determined for a model based on flat closures:

time	space	read	write
53.5	14	27	24

In a model with deep closures, closures form a minimally sized structure of linked vectors. Deep closures were described in Section 1.3.3. Figure 2.10 displays the computation

$$\begin{aligned}
&\text{referenceLocal} = \text{lookupFrame}\\
&\text{referenceFree}(i) = \text{lookupVector}\\
&\text{callProcedure}(n_0, n_1, \ldots, n_k) = \texttt{stmt}\\
&\quad + \text{allocFrame}(\Sigma_{i=0}^{k} n_i + k_0) + k_0 \cdot \text{setFrame} + n_0(\text{lookupFrame} + \text{setFrame})\\
&\quad + (\Sigma_{i=1}^{k} n_i)(\text{lookupVector} + \text{setFrame})\\
&\text{makeClosure}(n_0, n_1, \ldots, n_k) =\\
&\quad \text{allocVector}(1 + \Sigma_{i=0}^{k} n_i) + n_0(\text{lookupFrame} + \text{setVector})\\
&\quad + (\Sigma_{i=1}^{k} n_i)(\text{lookupVector} + \text{setVector})
\end{aligned}$$

Figure 2.9: Basic time costs for flat closures

$$\begin{aligned}
&\text{referenceLocal} = \text{lookupFrame}\\
&\text{referenceFree}(i) = (i-1) \cdot \text{dereferenceVector} + \text{lookupVector}\\
&\text{callProcedure}(n_0, n_1, \ldots, n_k) = \texttt{stmt}\\
&\quad + \text{allocFrame}(\Sigma_{i=0}^{k} n_i + k_0) + k_0 \cdot \text{setFrame} + n_0(\text{lookupFrame} + \text{setFrame})\\
&\quad + \Sigma_{i=1}^{k} n_i((i-1) \cdot \text{dereferenceVector} + \text{lookupVector} + \text{setFrame})\\
&\text{makeClosure}(n_0, n_1, \ldots, n_k) =\\
&\quad \text{allocVector}(1 + n_0) + n_0(\text{lookupFrame} + \text{setVector}) + \text{setVector}
\end{aligned}$$

Figure 2.10: Basic time costs for deep closures

costs for functional programs with deep closures. To find the value of a free variable the closure structure must be traversed by dereferencing the link to the outer closure vector holding the variable.

The computational cost of the summation program of Figure 2.6 can be determined for deep closures:

time	space	read	write
77	15	49	29

Surprisingly, the summation program consumes both more time and more space under the deep closure model than under the flat closure model. Looking closer at Figures 1.2 and 1.3 gives us an indication why this is so. When the function r has been invoked, the deep closure model has consumed slightly more space for pointers than it has saved by

$$
\begin{aligned}
&\text{referenceLocal} = \text{lookupFrame}\\
&\text{referenceFree}(0) = \text{lookupVector}\\
&\text{referenceFree}(i) = \text{dereferenceVector} + \text{lookupVector},\ i > 0\\
&\text{callProcedure}(n_0, n_1, \ldots, n_k) = \texttt{stmt}\\
&\quad + \text{allocFrame}(\Sigma_{i=0}^{k} n_i + k_0) + k_0 \cdot \text{setFrame}\\
&\quad + n_0(\text{lookupFrame} + \text{setFrame}) + n_1(\text{lookupVector} + \text{setFrame})\\
&\quad + \Sigma_{i=2}^{k} n_i(\text{dereferenceVector} + \text{lookupVector} + \text{setFrame})\\
&\text{makeClosure}(n_0, n_1, \ldots, n_k) =\\
&\quad \text{allocVector}(n_0 + (k-1)) + n_0(\text{lookupFrame} + \text{setVector})\\
&\quad + (k-1)(\text{lookupVector} + \text{setVector})
\end{aligned}
$$

Figure 2.11: Basic time costs for hybrid closures

not copying values across closures. The extra work for constructing pointers and the extra work needed to look up a free variable account for the higher time consumption.

There are many other ways of representing closures. The simple hybrid closure model was described in Section 1.3.4. These hybrid closures are almost identical to deep closures, except that each closure is linked to all outer closures. This makes free variable lookup simple at the cost of extra pointers in each closure. The computational costs for functional programs with hybrid closures are shown in Figure 2.11. This closure representation scheme is almost identical to deep closures when only a few blocks are nested. When more than three blocks are nested, free variables can be looked up using fewer operations, but the closures take up more space.

The computational costs of the summation program of Figure 2.6 in a model with hybrid closures are identical to the costs of the program in a model based on deep closures. Had the block structure been deeper (as in the program of Figure 1.1 we would have observed that hybrid closures were slightly more expensive that deep closures. This is because each free variables is referenced only once.

2.3.4 Imperative programs

In an imperative program of the classical ALGOL-style, procedures can be passed as arguments with certain restrictions. The restrictions ensure that the context of a procedure is available on the stack when it is invoked. There is thus no need for a separate closure holding the free variables of a procedure. The closure of a procedure is a pointer to the stack frame of the procedure that created it. To maintain lexical scope a stack frame points to the stack frame of the enclosing block. Similarly to the hybrid closure representation,

$$
\begin{aligned}
&\text{referenceLocal} \;=\; \text{lookupFrame}\\
&\text{referenceFree}(i) \;=\; \text{dereferenceFrame} + \text{lookupFrame}\\
&\text{callProcedure}(n_0, n_1, \ldots, n_k) \;=\\
&\qquad \text{allocFrame}(\Sigma_{i=0}^{k} n_i + k_0) + k_0 \cdot \text{setFrame}\\
&\qquad + n_0(\text{lookupFrame} + \text{setFrame}) + n_1(\text{lookupFrame} + \text{setFrame})\\
&\qquad + \Sigma_{i=2}^{k} n_i(\text{dereferenceFrame} + \text{lookupFrame} + \text{setFrame})\\
&\text{makeClosure}(n_0, n_1, \ldots, n_k) \;=\; \text{allocFrame}(1 + \Sigma_{i=0}^{k} I_1(n_i))\\
&\qquad + (\Sigma_{i=0}^{k} I_1(n_i))(\text{lookupFrame} + \text{setFrame}) + \text{setFrame}\\
&I_1(n) \;=\; \begin{cases} 0, & n = 0 \\ 1, & n > 0 \end{cases}
\end{aligned}
$$

Figure 2.12: Basic time costs for stack-based machine with displays

a stack frame can contain pointers to the stack frames of all the enclosing blocks. Such pointers are called "display pointers," and are useful in implementations of ALGOL-like languages.

Figure 2.12 shows the computational costs for a model based on stack allocation of closures and displays. The costs are identical to hybrid closures except for the cost of the "makeClosure" operation. To create a closure block for a procedure a stack frame is allocated that contains the display pointers. The formal parameters are saved in the closure by a pointer to the stack frame. The pointers in the closure can include a pointer to the stack frame of the procedure, letting us pass the procedure as an argument by passing a pointer to the closure. The free variables of a procedure are either formal parameters or free variables of the enclosing procedure. The stack frame of the procedure already contains display pointers for its free variables. These can be copied into the closure. When the enclosing procedure is popped from the stack the closure frame is automatically disposed of.

The computational cost of the summation program of Figure 2.6 can now be determined for a stack-based model:

time	space	read	write
47.5	12	27	19

Not surprisingly the stack-based model is cheaper than the closure representations. Retaining all values on the stack and only building closure frames containing display pointers is a definite advantage.

2.3.5 Other computational costs

The computational costs described in the previous sections are easy to compute. Once the instructions of the compiled program have been annotated with the associated computational costs, these costs can be expressed as a set of numerical recursive equations (akin to the equation for the summation program, displayed in Figure 2.7). It is desirable to also determine other computational costs such as the number of live variables and data dependencies.

The number of live variables can be determined as a function of the formal parameters by first simplifying the program based on the formal parameters (eliminating conditionals) and then applying a live variable analysis. The live variables of a program are determined by traversing the simplified program in reverse. A reference to a value makes this value live. Defining a value makes it dead (i.e. it is no longer live). For a recursive function, we would be interested in the maximum number of live variables used within the body in the base case and in the recursive case. The higher the maximum number of live variables the greater the chance of register spilling. If the average number of live variables throughout the evaluation of the function is higher than the number of available registers, heavy register spilling is probably unavoidable.

Dependencies between computations can be determined by building a data dependency graph. This graph is a DAG in which an edge between two instructions expresses a dependency of one instruction to the result of the other instruction. The roots of the data dependency graph of a procedure would be the formal parameters and any free variables together with values that do not depend on the context of the procedure invocation. The deeper the DAG compared to the total number of instructions, the greater the dependency between data. Data dependencies are important in a pipelined architecture. Here instructions that read from memory can do so in advance, if the memory location is known. An expensive memory read instruction becomes inexpensive because the data is immediately available to the instruction. For this to work properly, the processor must execute other instructions while fetching the data from memory. Instructions that operate on independent sets of data can be re-scheduled to achieve better performance. A simple estimate of data dependency within a procedure is the depth of the dependency DAG in relation to the total number of instructions. An estimate that conveys more information about whether instructions can be scheduled efficiently is the number of dependencies between instructions that read from memory since these must be scheduled with other instructions between them.

There are numerous other ways of estimating the cost of the computation of a program. The simple costs presented in the previous sections were easy to determine with a minimal analysis of the compiled program. The costs presented in this section require more work to determine. Other costs might be even more complicated to determine, making profiling and benchmarking a viable option.

2.4 Summary

The block structure of a program forms a tree. Restricting the form of the block structure to Block-Structured Normal-Form makes it isomorphic to a tree. Conversely, given a tree that describes the block-structure of a program, a textual representation of the program can be generated.

We have developed an abstract model suitable for reasoning on the computation costs of a program in the presence of procedure calls. This lets us describe the computational cost of a program before and after a transformation. The model allows the computational cost to be determined for several actual models of computation, such as functional languages and imperative languages.

Chapter 3

Lambda-Lifting

3.1 Introduction

We consider Johnsson's algorithm [4, 26, 27]. Johnsson's target is the G-machine, which can run recursive equations efficiently. Lambda-lifting transforms a block-structured program into a set of recursive equations. These recursive equations correspond to local functions in the block-structured program. Each equation is passed variables that would have occurred free in a block-structured program.

This chapter describes the basic principles of lambda-lifting and Johnsson's lambda-lifting algorithm itself. In a block-structured program, the context of a computation may be implicit. The enclosing block structure forms a context. Free variables are defined by this context. Lambda-lifting makes the context of a computation explicit by passing as parameters the free variables. When all contexts are explicit, block structure is no longer needed to form implicit contexts, and all computations can expressed as global equations. This chapter also provides a detailed example of how the lambda-lifting algorithm works. Finally, other algorithms for obtaining explicit contexts are discussed.

3.2 The Basics of Lambda-Lifting

Lambda-lifting is achieved using two transformations that are applied iteratively:

1. *Eta-expansion.* Each lambda-abstraction is eta-expanded with its free variables, both at its definition site and at all its application sites.

2. *Letrec floating.* Each set of local functions that do not refer to local free variables is moved to the enclosing scope level.

Eta-expanding an application can introduce more free variables, which are handled in a later iteration. Function names do not need to be passed as parameters, since they are used as the names of the recursive equations and thus are globally visible.

The two transformations listed above can be freely interleaved. Johnsson's algorithm, however, factorizes the transformations into two stages.

3.3 Lambda-Lifting

In a program with proper variable hygiene, a let expression can be replaced by a letrec expression. Thus, we assume that the program being lambda-lifted uses letrec expressions for both function and value bindings. Johnsson's algorithm proceeds in two stages.

1. *Parameter lifting.*

   Free variables are eliminated by eta-expanding each local function definition. Each time the definition of a function is eta-expanded, all call sites of this function must be correspondingly eta-expanded. In the resulting program, no function has free variables. This process is detailed in Figure 3.1.

2. *Block floating.*

   The floating step is trivial because each function is scope-insensitive after parameter lifting. Definitions can freely be moved outwards through the block structure of the program. In the resulting program, all function definitions are global. This process is detailed in Figure 3.2.

Other styles and implementations of lambda-lifting exist. We review them in Section 3.5.

3.3.1 Block structure in normal form

Section 2.2.2 describes an isomorphism between the block structure of programs in block-structured normal-form (BSNF) and trees. Programs in BSNF have letrec blocks and procedure declarations as their outermost constructs. The transformation of a program to BSNF could be accomplished by simply lambda-lifting the program, thus flattening the block structure of the program. This is of little practical interest, as the purpose of transforming a program to BSNF is to map as much as possible of the existing block structure of the program to a tree. The transformation to BSNF must preserve as much of the original block structure as is possible.

Assuming variable hygiene, transforming a program to BSNF implies lifting any procedure definitions nested inside other expression types to some enclosing block. Nested letrec blocks (i.e. a letrec block that forms the body of some other letrec block) are simply coalesced into a single letrec block. A procedure definition nested inside some expression should be made scope-insensitive to free variables declared by this expression through parameter lifting. Block floating lets us move the procedure definitions to the nearest enclosing block. Entire blocks of functions can be moved in this way.

In the resulting program, the structure of the procedure declarations forms a tree (as expected). Edges indicate local declarations. The nodes are expressions without block structure.

1. Naming anonymous lambda abstractions.

   Each $(\lambda x.e)$ is replaced by (letrec $f = \lambda x.e$ in f), where f is fresh.

2. Lifting the parameters of each function, by traversing the program top-down.

   During the traversal we use a set to describe the information that is gathered. This set associates the name of a function with the set of variables it needs to be eta-expanded with. We call this set the "set of solutions."

   - We process each occurrence of a function name according to the set of solutions. If the function is associated with the empty set, no change is made. If a function is associated with a non-empty set of variables, the function is eta-expanded into an application to these variables.
   - We process each letrec block by computing a new set of solutions describing the functions.
     (a) To describe the occurrences of variables and functions in each local function, we create a group of recursive set equations. We associate a local function g with:
       – A set V_g holding the free variables of g. These variables are needed by the function.
       – A set F_g holding all function names (the callees) occurring in the body of g. The lifted definition of g must provide variables for all the callees.
     (b) First we process the occurrences of functions not declared in the block. In each set F_g, each function h which is described in the set of known solutions S, is removed from F_g. The set V_g is extended with the variables associated with h in S.
     (c) Then we process each function h remaining in F_g by adding all elements of V_h to V_g. We repeat this step until a fixed point is reached.
     (d) The new set of solutions is the union of the old solutions with the set equations.

     We traverse the body and each binding of the letrec block with the newly computed set of solutions.

Figure 3.1: Parameter lifting. Free variables are made parameters

1. Moving each block to the global level.

   We process each letrec block by removing all of its function definitions and making them global. If no local declarations remain, we replace the block by its body.

2. Removing the remaining letrec blocks.

   We remove the remaining (now non-recursive) letrec blocks by converting each one into an application of a global function that has a fresh name. The formal parameters of the global function are the variables that were bound by the letrec block, and the body of the function is the body of the block.

Figure 3.2: Block floating. Flattening of block structure

3.4 A Detailed Example

As an example, we consider an implementation of a Deterministic Finite Automata (DFA), as displayed in Figure 3.3. The DFA accepts the language defined by the regular expression $R = \alpha(\beta(\delta|\gamma\alpha))^*$, given the alphabet $\{\alpha, \beta, \gamma, \delta\}$. The DFA reads symbols from an input stream, and invokes the associated functions `a`, `b`, `c`, and `d`, as long as the input symbols conform to R. If the input is non-conforming, the DFA aborts. The formal parameter `die?` determines whether to signal an error or just return the empty list. The stream is terminated by either the symbol $ or the end of the list.

The next two sections describe the process of lambda-lifting this program according to the algorithm of Figures 3.1 and 3.2. Some of the variables share the same name. This lack of hygiene does not interfere with the algorithm here, so we retain the names for the sake of clarity.

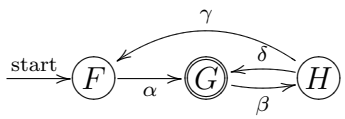
3.4.1 Parameter lifting

We proceed according to the algorithm of Figure 3.1.

1. The anonymous lambda-abstraction within the body of `f` needs to be explicitly named:

```
(f (if die?
       r-error
       (lambda (x) '()))
 xs)
```

   becomes




```
(define (r a b c d die? xs)
  (letrec ([err
            (lambda (x)
              (if (null? x)
                  (error 'r "end of stream~%")
                  (error 'r "token ~a~%" x)))]
           [empty?
            (lambda (s)
              (or (null? s) (eq? (car s) '$)))]
           [h (lambda (reject xs)
                (if (empty? xs)
                    (reject '())
                    (case (car xs)
                      [(gamma) (c (f reject (cdr xs)))]
                      [(delta) (d (g reject (cdr xs)))]
                      [else (reject (car xs))])))]
           [g (lambda (reject xs)
                (if (empty? xs)
                    xs
                    (case (car xs)
                      [(beta) (b (h reject (cdr xs)))]
                      [else (reject (car xs))])))]
           [f (lambda (reject xs)
                (if (empty? xs)
                    (reject '())
                    (case (car xs)
                      [(alpha) (a (g reject (cdr xs)))]
                      [else (reject (car xs))])))])
    (f (if die? err (lambda (x) '())) xs)))
```

Figure 3.3: Our original DFA

```
(f (if die?
       r-error
       (letrec ([fresh0 (lambda (x) '())])
          fresh0))
 xs)
```

where `fresh0` is a fresh variable.

2. The entire program is traversed. At all times S represents the current set of solutions.

   `(define (r ...) (letrec ...))` with $S = \emptyset$:

   A global function contains no free variables. Thus, its parameters do not need any lifting.

   `(letrec ...)` with $S = \emptyset$:

   We process a block by extending the set of solutions. We start by computing the sets of free variables for each function:

$$\begin{aligned} V_{\texttt{err}} &= V_{\texttt{empty?}} = \emptyset \\ V_h &= \{c, d\} \\ V_g &= \{b\} \\ V_f &= \{a\} \end{aligned}$$

   Then we compute the sets of function names describing references to other functions:

$$\begin{aligned} F_{\texttt{err}} &= F_{\texttt{empty?}} = \emptyset \\ F_h &= \{f, g, \texttt{empty?}\} \\ F_g &= \{h, \texttt{empty?}\} \\ F_f &= \{g, \texttt{empty?}\} \end{aligned}$$

   These sets are used to express the set equations:

$$\begin{aligned} V_{\texttt{err}} &:= V_{\texttt{err}} = \emptyset \\ V_{\texttt{empty?}} &:= V_{\texttt{empty?}} = \emptyset \\ V_h &:= V_h \cup V_f \cup V_g \cup V_{\texttt{empty?}} = \{c, d\} \\ V_g &:= V_g \cup V_h \cup V_{\texttt{empty?}} = \{b\} \\ V_f &:= V_f \cup V_g \cup V_{\texttt{empty?}} = \{a\} \end{aligned}$$

   To solve these set equations, we iterate the assignments until a fixed point is reached.

Iteration 1:

$$\begin{aligned} V_{\texttt{err}} &:= \emptyset \\ V_{\texttt{empty?}} &:= \emptyset \\ V_h &:= \{c,d\} \cup \{a\} \cup \{b\} \cup \emptyset = \{a,b,c,d\} \\ V_g &:= \{b\} \cup \{c,d\} \cup \emptyset = \{b,c,d\} \\ V_f &:= \{a\} \cup \{b\} \cup \emptyset = \{a,b\} \end{aligned}$$

Iteration 2:

$$\begin{aligned} V_{\texttt{err}} &:= \emptyset \\ V_{\texttt{empty?}} &:= \emptyset \\ V_h &:= \{a,b,c,d\} \cup \{a,b\} \cup \{b,c,d\} \cup \emptyset \\ &= \{a,b,c,d\} \\ V_g &:= \{b,c,d\} \cup \{a,b,c,d\} \cup \emptyset = \{a,b,c,d\} \\ V_f &:= \{a,b\} \cup \{b,c,d\} \cup \emptyset = \{a,b,c,d\} \end{aligned}$$

Iteration 3:

$$\begin{aligned} V_{\texttt{err}} &:= \emptyset \\ V_{\texttt{empty?}} &:= \emptyset \\ V_h &:= \{a,b,c,d\} \cup \{a,b,c,d\} \cup \{a,b,c,d\} \\ &\quad \cup \emptyset \\ &= \{a,b,c,d\} \\ V_g &:= \{a,b,c,d\} \cup \{a,b,c,d\} \cup \emptyset \\ &= \{a,b,c,d\} \\ V_f &:= \{a,b,c,d\} \cup \{a,b,c,d\} \cup \emptyset \\ &= \{a,b,c,d\} \end{aligned}$$

We have reached a fixed point. The new set equations are:

$$\begin{aligned} V_{\texttt{err}} &= \emptyset \\ V_{\texttt{empty?}} &= \emptyset \\ V_h &= V_g = V_f = \{a,b,c,d\} \end{aligned}$$

We can now process the local functions and the body of the letrec block, using the new set equations.

`(letrec ([err ...] ...) ...)` is handled as follows:

$S = \{r = \emptyset, \ldots, h = \{a,b,c,d\}, \ldots\}$.

S associates `err` with the empty set, so there is no need to eta-expand the definition of `err`. The body contains no functions described in the set of solutions, so no changes need to be made here either.

`(letrec (...[empty? ...]) ...)` is handled like `err`.

`(letrec (...[h ...]) ...)` is handled as follows:

$S = \{r = \emptyset, \ldots, h = \{a, b, c, d\}, \ldots\}$.

The solutions associate `h` with the set $\{a, b, c, d\}$. Thus, we must eta-expand the definition:

```
[h (lambda (reject xs) ...)]
```

is replaced by

```
[h (lambda (a b c d) (lambda (reject xs) ...) ...)]
```

The body of `h` contains references to `empty?` and `g`. The solutions associate `empty?` with the empty set, so it is unchanged. The solutions also associate `g` with the set $\{a, b, c, d\}$. The occurrence of `g` as

```
(g reject (cdr xs))
```

is replaced by

```
((g a b c d) reject (cdr xs))
```

`(letrec (...[g ...] ...) ...)` is handled like `h`.

`(letrec (...[f ...] ...) ...)` is handled like `h`.

`(letrec (...) (f ...))` is handled as follows:

$S = \{r = \emptyset, \ldots, h = \{a, b, c, d\}, \ldots\}$.

We eta-expand the occurrence of `f` with the appropriate variables. The declaration and occurrence of `fresh0` need not be eta-expanded.

3.4.2 Function floating

We proceed according to the algorithm of Figure 3.2.

1. We remove all functions from the letrec block declaring `f` to the outermost (global) one, and we replace this block by its body. Ditto for the block declaring `fresh0`.

2. Since no letrec blocks are left, there is no need to convert them into applications.

Figure 3.4 displays the final result.

3.5 Related Issues

Johnsson's algorithm makes a function scope insensitive by eta-expanding every application site. Functions that are scope insensitive can be let-floated to become global functions. Eta-expansion is a well-known transformation, forming an inverse for the basic eta-conversion step of the lambda calculus. Let-floating is a generally useful program transformation technique, which we discuss further in Section 5.5.1.

```
(define (r a b c d die? xs)
  ((f a b c d) (if die? err fresh0) xs))
(define (fresh0 x)
  x)
(define (err x)
  (if (null? x)
      (error 'r "unexpected end of stream~%")
      (error 'r "unexpected token ~a~%" x)))
(define (empty? s)
  (or (null? s) (eq? (car s) '$)))
(define (h a b c d)
  (lambda (reject xs)
    (if (empty? xs)
        (reject '())
        (case (car xs)
          [(gamma) (c ((f a b c d) reject (cdr xs)))]
          [(delta) (d ((g a b c d) reject (cdr xs)))]
          [else (reject (car xs))]))))
(define (g a b c d)
  (lambda (reject xs)
    (if (empty? xs)
        '()
        (case (car xs)
          [(beta) (b ((h a b c d) reject (cdr xs)))]
          [else (reject (car xs))]))))
(define (f a b c d)
  (lambda (reject xs)
    (if (empty? xs)
        (reject '())
        (case (car xs)
          [(alpha) (a ((g a b c d) reject (cdr xs)))]
          [else (reject (car xs))]))))
```

Figure 3.4: The DFA after lambda-lifting

Several alternatives to Johnsson's algorithm exists. Rather than passing free variables explicitly as parameters, the program can be transformed to explicitly represent closures holding the free variables of each function (Section 3.5.1). A different approach is to compile a program into combinators. These combinators can either be a fixed set that form a base for the all lambda terms (Section 3.5.2) or a set of combinators constructed to represent the program (Section 3.5.3).

3.5.1 Closure conversion

Transforming a program with closure conversion yields a block-structured program that has no free variables. Block structure is formed by non-recursive let blocks. A procedure is represented through its closure. The closure of a procedure is a vector holding a code pointer and the values of the free variables of the procedure. A procedure is invoked by extracting the code pointer and calling the associated code with the closure as the first argument and any arguments to the procedure as the remaining arguments. This representation of a program closely models a low-level abstract machine evaluating a block-structured functional program. Appel presents closure conversion in [2]. We present closure conversion using a slightly different notation.

A non-recursive procedure can be closure converted by the algorithm of Figure 3.5. Closure conversion creates explicit references to the free variables of locally declared functions. It should thus be applied to the innermost procedures first. In a closure converted program, the free variables of a procedure are stored in a closure with a reference to the function at the point where the procedure is declared. At each invocation of the procedure, this closure provides the values of the free variables of the procedure. References to other functions are treated as free variables, and are handled accordingly. Closure converting the program of Figure 3.6 yields the program of Figure 3.7. Each free variable is referenced explicitly as an element of the closure of the procedures of the program.

The closure conversion algorithm of Figure 3.5 handles curried procedures automatically (the inner lambda expressions of a curried procedure will be named by the algorithm). A curried procedure is simply regarded as a procedure returning a closure. A set of mutually recursive functions declared in a letrec block are closure converted by letting these functions share a single closure that references itself. Appel optimizes the closure conversion transformation in the case of first-order functions. These functions are effectively lambda-lifted by passing the free variables as extra arguments in every invocation. While this optimization is not of interest in this section (and will not be considered part of the closure conversion algorithm), it is reflected briefly upon in Section 6.5.

It is worthwhile to compare closure conversion to lambda-lifting. The lambda-lifted version of the program of Figure 3.6 can be found in Figure 3.8. In the lambda-lifted program, we pass the free variables as parameters in every invocation. In the closure converted program, we store the free variables of a procedure in a vector whenever we enter the scope of the declaration of the procedure. In the lambda-lifted program, the free variables of the callees and the callees of the callees of a procedure are passed as arguments to this procedure. In the closure converted program, each closure provides the local free

Closure conversion of a function f with formal parameters $x_1, \ldots, x_n$ and free variables $v_1, \ldots, v_n$, including procedures.

1. Name anonymous lambdas.
2. Rename the declaration of f to a fresh name, F. Extend the list of formal parameters by prepending a fresh name, f'.
3. At the level of the declaration of F, bind f to a vector holding the values $(F, v_1, \ldots, v_n)$.
4. Within the body of F, replace each reference to a free variable v_i by $(\#_i\, f)'$, where $\#_i$ returns the i'th element of a vector.
5. Replace each invocation of a function by a let block:

   $$(g\ (a_1, \ldots, a_n))$$

   is replaced by

   $$\text{let } g'' = (\#_0\ g) \text{ in } (g''\ (g, a_1, \ldots, a_n))$$

   where g'' is fresh.

Figure 3.5: Closure conversion of a non-recursive function

$$\begin{array}{rcl} \text{tripleSum}\,(i,j,k) & = & \text{let } h\,(x,y) = (+\ i\ x\ y) \\ & & \quad\ g\ z \qquad\ = (h\ (j,z)) \\ & & \text{in } (g\ k) \end{array}$$

Figure 3.6: A complicated summation function

$$
\begin{array}{rl}
\text{tripleSumClosConv}\,(\text{clos}, i, j, k) \;=\; \text{let}\, H\,(c_H, x, y) = & \text{let}\, i' = (\#1\; c_H) \\
& \quad \text{in}\;\; (+\; i'\; x\; y) \\
G\,(c_G, z) \quad = & \text{let}\, h' = (\#1\; c_G) \\
& \quad\;\; h'' = (\#0\; h') \\
& \quad\;\; j' = (\#2\; c_G) \\
& \quad \text{in}\;\; (h''\,(h', j', z)) \\
\text{in}\;\; \text{let}\, h = \#(H, i) & \\
\quad g = \#(G, h, j) & \\
\text{in}\;\; \text{let}\, g' = (\#0\; g) & \\
\quad \text{in}\;\; (g'\,(g, k)) &
\end{array}
$$

Figure 3.7: The function of Figure 3.6, closure converted

$$
\begin{array}{l}
\text{tripleSumLifted}\,(i, j, k) \;=\; (g\,(i, j, k)) \\
h\,(i, x, y) \;=\; (+\; i\; x\; y) \\
g\,(i, j, z) \;=\; (h\,(i, j, z))
\end{array}
$$

Figure 3.8: The function of Figure 3.6, lambda-lifted

variables of a procedure. Lambda-lifting can thus be viewed as the transitive closure of closure conversion.

3.5.2 IKBCS combinators

A combinator is a lambda term with no free variables. Disregarding constants (such as numbers and predefined functions), most programs have no free variables. The set of all combinators is denoted Λ^0. A set B of combinators forms a basis for Λ^0 if for any $t \in \Lambda^0$ a lambda term can be constructed from the elements of B that is equivalent[1] to t. Given a basis B for Λ^0, any program can be compiled into a lambda term without variables consisting of elements of B and a number of constants.

The five combinators of Figure 3.9 form a basis for Λ^0. To compile a program into a term consisting of a fixed set of combinators and constants, we first transform the program into a simple lambda term. Conditionals can be represented as combinators using Church booleans: conditional values are combinators that when applied to a consequence and an alternative return the appropriate term. Let blocks are transformed into applications. Letrec blocks and recursive functions are transformed to applications of appropriately defined

[1]Modulo α, β and η reduction.

$$
\begin{aligned}
\mathbf{I} &= \lambda x.x \\
\mathbf{K} &= \lambda xy.x \\
\mathbf{B} &= \lambda xyz.x\,(y\,z) \\
\mathbf{C} &= \lambda xyz.x\,z\,y \\
\mathbf{S} &= \lambda xyz.(x\,z)\,(y\,z)
\end{aligned}
$$

Figure 3.9: A set of combinators forming a basis for Λ^0

$$
\begin{array}{lcll}
[\![x]\!] & = & x, & x \in \{\mathbf{I},\mathbf{K},\mathbf{B},\mathbf{C},\mathbf{S}\} \cup \mathsf{Symbols} \\
[\![\lambda x.x]\!] & = & \mathbf{I} & \\
[\![\lambda x.M']\!] & = & (\mathbf{K}\,[\![M']\!]), & x \notin \mathsf{FreeVar}(M') \\
[\![\lambda x.(M'M''_x)]\!] & = & (\mathbf{B}\,M'\,[\![\lambda x.M''_x]\!]), & x \notin \mathsf{FreeVar}(M'), x \in \mathsf{FreeVar}(M'') \\
[\![\lambda x.(M'_xM'')]\!] & = & (\mathbf{C}\,[\![\lambda x.M'_x]\!]\,M''), & x \in \mathsf{FreeVar}(M'), x \notin \mathsf{FreeVar}(M'') \\
[\![\lambda x.(M'_xM''_x)]\!] & = & (\mathbf{S}\,[\![\lambda x.M'_x]\!]\,[\![\lambda x.M''_x]\!]), & x \in \mathsf{FreeVar}(M'), x \in \mathsf{FreeVar}(M'')
\end{array}
$$

Figure 3.10: Translation from lambda term to combinators

fixpoint combinators. The translation scheme of Figure 3.10 will transform a simplified lambda term into a term consisting of the combinators $\{\mathbf{I},\mathbf{K},\mathbf{B},\mathbf{C}\,\mathbf{S}\}$ and constants. The translation should be applied to the innermost lambda expressions first. Variables and constants are members of the set $\mathsf{Symbols}$.

A lambda term can by evaluated by compiling it into IKBCS combinators, and evaluating the resulting term using a set of simple rewriting rules. Alternatively, the set of combinators could be expressed as equations and the resulting lambda term would thus be the main equations of a set of (non-recursive) equations.

The proper combinator ω can be translated into IKBCS combinators:

$$[\![\omega]\!] = [\![\lambda x.(x\,x)]\!] = \mathbf{S}\,[\![\lambda x.x]\!]\,[\![\lambda x.x]\!] = \mathbf{S}\,\mathbf{I}\,\mathbf{I}$$

The $\mathbf{S}$ combinator expresses the use of the variable x in both sub-expressions of the application of the body of ω. The $\mathbf{I}$ combinator expresses that the variable is simply returned without modification. On a larger scale, the Figure 3.11 displays a translation of a small program computing the absolute value -2 using two predefined functions. The let block and the conditional are eliminated in the first step of the translation. Subsequently, combinators are used to express the flow of data through the lambda term. The result is a single term without variables.

The IKBCS translation is simple and produces in an intuitive fashion terms that are roughly the same size as the original lambda term. The combinators $\mathbf{I}$, $\mathbf{B}$ and $\mathbf{C}$ can be

$$
\begin{aligned}
&[\![\text{let } \mathsf{abs} = (\text{if } \mathsf{negative?}\ n \text{ then } \mathsf{negate}\ n \text{ else } n) \text{ in } \mathsf{abs}\ \text{-}2]\!] \\
&\quad = \quad [\![(\lambda \mathsf{abs}.\mathsf{abs}\ \text{-}2)\ (\lambda n.((\mathsf{negative?}\ n)(\mathsf{negate}\ n))n)]\!] \\
&\quad = \quad (\mathbf{C}\ [\![\lambda \mathsf{abs}.\mathsf{abs}]\!]\ \text{-}2)\ (\mathbf{S}\ [\![\lambda n.(\mathsf{negative?}\ n)\ (\mathsf{negate}\ n)]\!]\ [\![\lambda n.n]\!]) \\
&\quad = \quad (\mathbf{C}\ \mathbf{I}\ \text{-}2)\ (\mathbf{S}\ (\mathbf{S}\ [\![\lambda n.\mathsf{negative?}\ n]\!]\ [\![\lambda n.\mathsf{negate}\ n]\!])\ \mathbf{I}) \\
&\quad = \quad (\mathbf{C}\ \mathbf{I}\ \text{-}2)\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{B}\ \mathsf{negative?}\ [\![\lambda n.n]\!])\ (\mathbf{B}\ \mathsf{negate}\ [\![\lambda n.n]\!]))\ \mathbf{I}) \\
&\quad = \quad (\mathbf{C}\ \mathbf{I}\ \text{-}2)\ (\mathbf{S}\ (\mathbf{S}\ (\mathbf{B}\ \mathsf{negative?}\ \mathbf{I})\ (\mathbf{B}\ \mathsf{negate}\ \mathbf{I}))\ \mathbf{I})
\end{aligned}
$$

Figure 3.11: IKBCS translation of a simple lambda term

$$
\begin{aligned}
\$\text{tripleSum}\ i\ j\ k &= \$g\ i\ j\ k \\
\$h\ i\ x\ y &= +\ i\ x\ y \\
\$g\ i\ j\ z &= \$h\ i\ j\ z
\end{aligned}
$$

Figure 3.12: The function of Figure 3.6, as supercombinators

expressed using **K** and **S**. Thus, the set $\{\mathbf{K}, \mathbf{S}\}$ forms a basis for Λ^0. Fewer rewrite rules are needed to evaluate a term expressed using the two combinators **K** and **S** than for a set of five combinators. However, a term expressed using **K** and **S** is usually longer and thus takes more steps to fully reduce. The combinators **K** and **S** can be expressed using a single combinator only. We refer the interested reader to [22].

The IKBCS translation has been treated in many different texts using different notations. The presentation in this sections is due to Goldberg [22], with some minor modifications.

3.5.3 Supercombinators

Rather than expressing a program in terms of a fixed set of combinators, the program can be expressed in terms of a dedicated set of combinators. These named combinators form a set of recursive equations. In each instance, they are defined by the functions of a program. Peyton Jones describes how these "super-combinators" can be used to efficiently compile and execute functional programs [38]. In this work, the translation to supercombinators is called lambda-lifting. To avoid confusion, we refer to Johnsson's algorithm as "lambda-lifting," and to the supercombinator translation algorithm as "supercombinator conversion."

Non-recursive functions and anonymous lambdas are treated analogously to lambda-

lifting. The innermost functions are processed first. The name of each function is prepended with a "$" sign and lifted to become a global equation. Any free variables are passed explicitly as parameters. Any applications of the function are changed to reflect this. Figure 3.12 illustrates the conversion of the program of Figure 3.6 to supercombinators. An uncurried notation is used reflecting the ability of the abstract machine evaluating the supercombinator program to handle underapplied functions. As we shall see later, this is essential for handling recursion. Disregarding the difference in notation, the result is equivalent to lambda-lifting (shown in Figure 3.8).

As was the case for Johnsson's lambda-lifting algorithm, the target of the supercombinator conversion algorithm is the G-machine. The G-machine is a graph reducing abstract machine that runs recursive equations efficiently. In the world of graph reducers, mutually recursive structures are expressed concisely as graph. Thus, it makes sense to express mutually recursive functions using graphs. Analogous to the scope graph that will be described in Section 5.3.1, Peyton Jones points out the correspondence between a strongly connected graph and a letrec block that has been simplified using dependency analysis (described in Section 5.5.2). Put simply, a letrec block containing mutually recursive functions can be interpreted as a strongly connected graph with functions as its nodes, and vice versa. A letrec block can thus be used to express a graph that describes a set of mutually recursive functions.

Applying this to the supercombinator conversion algorithm, we can construct supercombinators with *graphical bodies*. A supercombinator with a graphical body is directly interpreted by the G-machine as a graph. Combined with underapplication of functions, this allows us to convert a set of mutually recursive functions with free variables into a graph of functions defined by supercombinator applications. To convert a set of mutually recursive functions, the free variables of each function are parameter lifted. This includes the names of other functions, yielding the familiar supercombinators. These supercombinators are applied within a letrec block to the arguments that were free variables. This mutually recursive application relies on the intrinsic lazy evaluation of graph reduction. Figure 3.13 displays the DFA of Figure 3.3 converted to supercombinators. Comparing this to the lambda-lifted DFA (found in Figure 3.4), we can see that the supercombinators `$F`, `$G`, and `$H` receive all of their free variables as arguments. These arguments include the underapplied recursively called functions, each of which has already been applied to its free variables in the body of `$R`. In effect, the graph constructed in `$R` is a closure shared by the three mutually recursive functions.

The supercombinator conversion algorithm can be viewed as mix of the other translation schemes from block structured programs to global equations discussed in this chapter. The algorithm is inspired by the IKBCS translation scheme, converting a block-structured program into application of combinators. Non-recursive functions are translated similarly to lambda-lifting, yielding recursive equations. Mutually recursive functions receive a graphically constructed shared closure, similarly to the shared closures of mutually recursive functions translated by closure conversion.

```
$R a b c d die? xs =
   (letrec ([h ($H f g c d)]
            [g ($G h b)]
            [f ($F g a)])
     (f (if die? err $FRESH0) xs)
$FRESH0 x = '()
$ERR x = (if (null? x)
             (error 'r "end of stream~%")
             (error 'r "token ~a~%" x))
$EMPTY? s = (or (null? s) (eq? (car s) '$))
$H f g c d reject xs =
   (if (empty? xs)
       (reject '())
       (case (car xs)
         [(gamma) (c (f reject (cdr xs)))]
         [(delta) (d (g reject (cdr xs)))]
         [else (reject (car xs))]))
$G h b reject xs =
   (if (empty? xs)
       xs
       (case (car xs)
         [(beta) (b (h reject (cdr xs)))]
         [else (reject (car xs))])))]
$F g a reject xs =
   (if (empty? xs)
       (reject '())
       (case (car xs)
         [(alpha) (a (g reject (cdr xs)))]
         [else (reject (car xs))]))
```

Figure 3.13: The DFA expressed as supercombinators

3.6 Summary

There are numerous strategies for translating block-structured programs into a simpler representation. This chapter has provided an overview strategies for transforming block-structured programs into recursive equations. The common goal of these strategies is to eliminate the complications associated with complex expression types and free variables. A common overhead of these strategies is an increase in size.

In a lambda-lifted program, the free variables of each callee of a procedure must by passed as argument to this procedure. The resulting equations are simple, and can be implemented efficiently. In a closure-converted program, the closure is explicitly constructed and manipulated. This closely models the actions of an abstract machine evaluating a block-structured program. A program translated into IKBCS must explicitly manipulate values that are used in different places in the program. A program in IKBCS form can be evaluated by a very simple abstract machine. A program converted to supercombinators relies on lazy evaluation (achieved by efficient graph reduction) and captures many advantageous aspects of the other translation strategies. Non-recursive function receive their arguments directly. Recursive functions use graphical structures that serve as closures to handle free variables.

Each of the strategies presented in this chapter make explicit the context of the computations of each procedure. It is our opinion that the transformation that accomplishes this task while producing something that most closely resembles a program written by a human programmer is Johnsson's algorithm. As a consequence, this thesis focuses on Johnsson's algorithm.

Chapter 4

From Equations To Blocks

4.1 Introduction

This chapter informally describes the basic program transformations needed to convert a set of recursive equations into a block-structured program. We start by illustrating how functions defined as recursive equations can be transformed into equivalent block-structured functions. We then proceed with a more complicated task, namely undoing the lambda-lifting transformation of the DFA program performed in Section 3.4, thereby obtaining a block-structured program.

Localizing a function in a letrec block moves it into the context where it is used. Once a function is localized, it is no longer visible outside the letrec block. This localization often makes the program easier to understand for a human reader, and simplifies compilation. Localization, however, is not always possible. Functions used in different parts of the program might not be localizable to all these places, unless one is willing to duplicate code.

The scope introduced by localization can make a formal parameter redundant. Removing a redundant formal parameter simplifies the flow of data through the program, and often speeds up compilation. A formal parameter of a function can only be removed if it can be determined that it is bound to the same parameter in every invocation of the function.

4.2 Creating Block Structure

Consider the `foldr1` function, defined by

$$\texttt{foldr1}\, f\, [x_1, x_2, \ldots, x_{n-1}, x_n] \;=\; f(x_1, f(x_2, \ldots, f(x_{n-1}, x_n)\ldots))$$

We define it in terms of a dedicated declaration of `foldr`, with recursive equations:

```
fun foldr1 f (x::xs) = doFoldr f x xs
fun doFoldr g a nil     = a
  | doFoldr g a (y::ys) = g y (doFoldr g a ys)
```

The formal parameters are local to the body of each function, but the auxiliary function `doFoldr` is globally visible, even though it is only called by `foldr1`. It does not need to be visible to any functions that call `foldr1`. Introducing a local binding for the function expresses that `doFoldr` is an auxiliary function of `foldr1`.

```
fun foldr1 f (x::xs) =
    let fun doFoldr g a nil = a
          | doFoldr g a (y::ys) = g y (doFoldr g a ys)
     in
       doFoldr f x xs
    end
```

Defining `foldr1` in this way enforces that only the intended entry point for the `foldr1` function is visible. The variables `f`, `x`, and `xs` have become visible to the definition of `doFoldr` as free variables within in this definition of `foldr1`.

Further observation reveals that in all application sites of `doFoldr`, the first argument is either `f` or `g`, and that `g` will always be bound to the same value as `f`. The same holds for the variables `x` and `a`. Thus, some of the formal parameters of `doFoldr` are redundant, and `foldr1` can be rewritten in a more concise way, as follows:

```
fun foldr1 f (x::xs) =
    let fun doFoldr nil     = x
          | doFoldr (y::ys) = f y (doFoldr ys)
     in
       doFoldr xs
    end
```

It is now more clear that `doFoldr` traverses a list, always applying the same function, and only using the initial element at the base case of the recursion. In the original definition, the value to use in the base case of `doFoldr` (the formal parameter `a`), was passed as an argument during the traversal of the entire list, which was obviously undesirable.

Unfortunately, it seems as if the merits of block structure are applied nowhere as often as they could be in everyday programming. Consider the standard append-function, defined as a recursive equation:

```
fun append nil ys     = ys
  | append (x::xs) ys = x::(append xs ys)
```

Note how the second argument is passed without modification at each recursive call, and only used in the base case. Using block structure, append could have been defined as follows:

```
fun append xs ys = let fun app nil     = ys
                       | app (x::xs) = x::(app xs)
                   in
                     app xs
                   end
```

Assuming that creating closures and referencing free variables is relatively fast compared to the extra overhead of passing two parameters rather than one, the block-structured version of append is likely to be faster than its recursive-equation counterpart. Section 2.3 provides a discussion of the amount of computation performed in each instance.

4.3 Reversing Lambda-Lifting

As described in Section 3, lambda-lifting starts by making functions scope-insensitive through eta-expansion and then proceeds to make all functions global through let-floating. To reverse lambda-lifting, we could make the appropriate global functions local, and then make them scope-sensitive through eta-reduction. To simplify the process, we always generate letrec blocks, even if a let block would suffice.

4.3.1 Block sinking

To reverse the effect of lambda-lifting, let us examine the program of Figure 3.4, which was lambda-lifted in Section 3.4. The main function of the program is `r`. All other functions are used by `r`, and are thus localizable to `r`. We replace the body of `r` with a block declaring these functions and having the original body of `r` as its body.

```
define r = letrec f = ...
                  g = ...
                  h = ...
                  err = ...
                  empty? = ...
                  fresh0 = ...
           in ...
```

We can see that the body of `r` refers to the functions `f`, `err`, and `fresh0` only. The functions `g`, `h`, and `empty?`, however, are used only by `f`. Therefore it makes sense to localize them to `f`.

```
define r = letrec f = letrec g = ...
                             h = ...
                             empty? = ...
                         in ...
                  err = ...
                  fresh0 = ...
           in ...
```

Examining the newly localized functions more closely reveals that `h` is only called by `g`. It is therefore possible to localize `h` in `g`. The function `empty?` is used by `f`, `g`, and `h`, but we cannot localize it any further, since it needs to be accessible to `f`. The result of our function localization is:

```
define r = letrec f = letrec g = letrec h = ...
                                     in ...
                                empty? = ...
                           in ...
                     err = ...
                     fresh0 = ...
           in ...
```

The functions of the program cannot be localized any further. This lambda-dropped version has more block structure than the original version. It is our experience that one tends to write such incompletely lambda-dropped programs.

4.3.2 Parameter dropping

To reverse the parameter lifting performed during lambda-lifting, we need to determine the origins of each formal parameter. The functions `f`, `g`, and `h` all pass the variables `a`, `b`, `c`, and `d` to each other. These formal parameters always correspond to the variables of the same name defined by the function `r`. Since all these parameters are now visible where the three functions are declared, there is no need to pass them around as parameters. We can simply remove the four formal parameters from the declaration of each function, and refrain from passing them as arguments at each application site.

The function `err` is passed as an argument to `f`. We would need to perform a control-flow analysis to determine which arguments are passed to `err` (in this case a control-flow analysis is very simple). If parameter lifting had been performed on `err`, `err` would have been applied directly to any such variables. Since `err` is not applied, we can safely assume that it has not been parameter lifted. The function `empty?` is never passed as an argument, but it is passed arguments that are not themselves formal variables. These arguments could not have been the produced by the lambda-lifter, so we need not consider dropping the parameters of this function either.

We have thus removed all the formal parameters of the functions `f`, `g`, and `h`. What remains in each case is a function of no parameters which returns a function of two parameters — the two parameters of the function declaration from the original program. Since these thunks are always applied, it makes sense to eliminate them, leaving only the part of the function corresponding to the original program.

Figure 4.1 displays the final result of our reversal process. Comparing with the original program, and taking into account that we generate more block structure, a single difference remains: the anonymous lambda-abstraction from the body of `r` is still explicitly named. Lambda-lifting the lambda-dropped program would yield the same program as we obtained at first, that of Figure 3.4.

```
(define (r a b c d die? xs)
  (letrec
      ([f (lambda (reject xs)
            (letrec
                ([g (lambda (reject xs)
                      (letrec
                          ([h (lambda (reject xs)
                                (if (empty? xs)
                                    (reject '())
                                    (case (car xs)
                                      [(gamma)
                                       (c (f reject (cdr xs)))]
                                      [(delta)
                                       (d (g reject (cdr xs)))]
                                      [else
                                       (reject (car xs))])))])
                        (if (empty? xs)
                            xs
                            (case (car xs)
                              [(beta) (b (h reject (cdr xs)))]
                              [else (reject (car xs))])))))]
                 [empty?
                  (lambda (s)
                    (or (null? s) (eq? (car s) '$)))])
              (if (empty? xs)
                  (reject '())
                  (case (car xs)
                    [(alpha) (a (g reject (cdr xs)))]
                    [else (reject (car xs))])))))]
       [fresh0
        (lambda (x) x)]
       [err
        (lambda (x)
          (if (null? x)
              (error 'r "unexpected end of stream~%")
              (error 'r "unexpected token ~a~%" x)))])
    (f (if die? err (lambda (x) '())) xs)))
```

Figure 4.1: The DFA program after lambda-dropping

4.4 Summary

A lambda-lifted program can be transformed into a semantically equivalent block structured program. This can be employed to increase the locality of function definitions and reduce the number of formal parameters of each function definition. It also provides an inverse for lambda-lifting. However, there is no guarantee that a program that has been lambda-lifted is mapped back to its original form by this process. This unpreciseness is unavoidable since a lambda-lifter may generate the same set of recursive equations from different block-structured programs.

Chapter 5

Lambda-Dropping

5.1 Introduction

Lambda-dropping a program minimizes parameter passing, and serves as an inverse of Johnsson's lambda-lifting algorithm. Function definitions are localized maximally using lexically scoped block structure. Parameters made redundant by the newly created scope are eliminated.

This chapter describes the basic principles of lambda-dropping and the algorithm for lambda-dropping. In a program consisting of recursive equations, the context of a computation is explicit. Formal parameters provide all variables information needed for a computation. Block-structure can be formed, enclosing some computations in a context. The computation can make use of this implicit context rather than the explicit formal parameters. This chapter also provides a detailed example of how the lambda-dropping algorithm works. Finally, other algorithms for working with implicit contexts are discussed.

5.2 The Basics of Lambda-Dropping

Lambda-dropping is achieved using two transformations that are applied iteratively:

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

2. *Eta reduction.* 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 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.

Letrec sinking of functions moves functions that can be localized into the scope of the formal parameters of other functions. Formal parameters made redundant by this new scope are removed through eta reduction.

The two transformations listed above can be freely interleaved. Our algorithm, however, factorizes the transformations into two stages.

5.3 Lambda-Dropping

To ensure scope insensitivity of functions, we start by lambda-lifting the source program. If the program contains global function declarations only, the lambda-lifting step is not necessary. The lambda-dropping algorithm then works in two stages. It handles other binding constructs (such as let or the letType construct of Schism [12]), but we have made no attempt to drop parameters that are bound to variables defined by these constructs (see also Section 5.3.6).

1. *Block sinking.*

   Each reference to a function introduces restrictions on where it can be declared. Expressing these restrictions as a tree gives guidelines for translating from recursive equations to block structure. In the resulting program, function definitions are declared locally wherever possible. This process is detailed in Figure 5.1.

2. *Parameter dropping.*

   Two things determine whether some of the parameters of a function can be dropped: the scope in which the function is declared and the set of variables it is passed as arguments. In the resulting program, a formal parameter is never passed as an argument to a function if it instead could have been substituted throughout the body of the function. This process is detailed in Figure 5.2.

The block-sinking step of the lambda-dropping process starts by representing the program as a graph. To obtain a tree describing how block-structure should be generated, we perform a number of graph transformations. The correctness of these transformations rests on the scope of each function within graph representation. In sections 5.3.1, 5.3.2 and 5.3.3 we provide a discussion of the graph representation of programs and how we transform it.

The parameter-dropping step starts by representing the flow of data through the program as a graph. To obtain Use/Def chains describing parameters that remain constant through all invocations of a procedure (and thus can be removed) we propagate variable identities though the flow graph. In sections 5.3.4 and 5.3.5 we provide a more detailed explanation of the flow graph and the information we obtain from it.

5.3.1 The scope graph

In programs consisting solely of recursive equations, concerns relating to scope and program structure are greatly simplified. Each equation has a name that is visible to all other

1. Creating a scope graph.

   We construct a graph describing the scope constraints imposed by references to global functions:

   - For each function definition, we create a node.
   - If a function f refers to some other function g by its name, we introduce an edge from the node of f to the node of g.
   - We create an empty node, the root node. This root node will point to the node of any function that should remain as a global definition.

   During this pass, we trivially create Def/Use chains for functions. We also tag any function passed as an argument as "higher order."

2. Transforming the scope graph into a DAG.

   We coalesce the strongly connected components of the scope graph. We coalesce components that contain at most a single cycle into a node. We associate the resulting node with the set of functions represented by the nodes of the component. If the component only has a single entry point and contains more than one node, we try to reduce the size of the component. We remove edges from nodes in the component to the node that is the entry point of the component. We recurse this step on the nodes of the component.

3. Transforming the DAG into a tree.

   - Two disjoint paths p_1 and p_2 from a node N to some other node N' indicate that a common successor exists in the graph. We must detach the common successor N' from at least one of its direct predecessors, and re-attach it without violating the scope restrictions imposed by the graph. Detaching N' from those of its predecessors that lie on p_1 and p_2, and making it a direct successor of N preserve scope restrictions. We coalesce all successors of N' into a single node, which we place at the previous location of N'.
   - A node associated with two or more functions that use functions from a child node is merged with this child node. If only a single function uses the functions of a child node, these functions are localized to the function. We remove the child node from the graph re-attaching any subtree onto the father node after processing it.

4. Re-constructing the program from the tree.

   The tree representation is isomorphic to a program. We make the direct successors of the root node global functions. Then we traverse the tree, generating a function containing a letrec block at each branch. The functions associated with the direct successors of this node we declare locally in this block. If the node has no successors, we generate no letrec block.

Figure 5.1: Block sinking. Re-creation of block structure

1. Determining Use/Def information for each formal parameter.

   We annotate each use of a variable declared as a formal parameter with a reference to the function that declared it.

2. Constructing a flow graph.

   We process the formal parameters of each function definition separately, except when the function is marked as "higher order." In this case, we leave it unprocessed.

   - We create a definition node for the function that declares the formal parameter.
   - We create an application node for each application site of the function where the argument corresponding to the formal parameter is itself a formal parameter.
   - If the argument corresponding to the formal parameter is not a formal parameter, we discard the definition node and all corresponding application nodes, and we create no more nodes for this parameter.

   We coalesce the strongly connected components of the graph, using the standard algorithm to coalesce strongly connected components [1]. Reversing the resulting DAG yields the flow graph.

3. Propagating variable identities through the flow graph.

   We assign a unique ID to all nodes without predecessors. All other nodes have uninitialized IDs. If all the predecessors of a node have the same ID, we assign this ID to the node. If two predecessors of the node have different IDs, we assign a new, unique ID to the node.

4. Removing redundant formal parameters.

   We process the parameters of each function definition separately. If the U/D chain of a formal parameter p of a function definition D points to another formal parameter p', and D is declared in the scope of p', we remove p. We remove the formal parameter p from the parameter list of the declaration of D, and we substitute p' for p throughout the body of D. If all formal parameters are removed from D, a possibly redundant thunk encapsulates the body of D. If the body is a lambda form, we substitute this lambda form for D. If the body is a letrec block with a lambda form as its body, we move the block into the lambda form, and substitute the lambda form for D. In both cases, the function is "de-thunked."

5. Removing redundant arguments.

   We process each application site by removing the arguments corresponding to formal parameters that were removed. If the function being applied was de-thunked, we replace the application by a reference to the function.

Figure 5.2: Parameter dropping. Removing redundant formal parameters

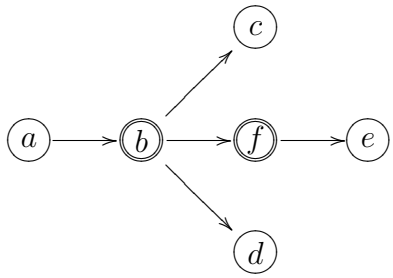


Figure 5.3: The scope of the definition of f in a scope graph

equations and declares a number of local formal parameters. The body of the equation is an expression without variable declarations. The only free variables allowed in the body of the equation are the formal parameters together with names of other recursive equations. We regard each recursive equation as a global function.

The scope graph of a program describes relations between the functions of the program together with the main expression. The nodes of the scope graph are the global functions of the program. A reference from a function to the name of another function is represented as an edge. In the first-order case, the scope graph coincides with the call graph of the program. The call graph of a program has nodes for each function, like the scope graph. An invocation of a function by some other function is represented by an edge from the caller to the callee. In a first-order program functions are always called directly, i.e. they are never passed as arguments to other functions. The only way to refer to another function is by calling it. Thus, each edge in the scope graph (representing a reference to a name) corresponds to an edge in the call graph (representing a call to the function of this name).

The scope graph must properly represent references to function names in a higher-order program where functions may be passed as arguments to other functions. For this reason, every occurrence of the name of a function introduces an edge. This includes not only functions that are directly applied, but all function names occuring in the body of a function. These function names are the free variables of each function. The only other variables occuring in the body of a function are the formal parameters of the function. We never create multiple edges between nodes, nor do we create edges from a node to itself. The scope graph is a directed graph, so there can be edges in both directions between two nodes.

Instead of allowing each function to reference any global function, we can now restrict the set of accessible function names. The successors of a function f in the scope graph are the only functions f needs to refer to (excluding, possibly, f itself). In other words: the scope of a function declaration is the function itself and any direct predecessors in the scope graph. The scope a function f in a scope graph is illustrated in Figure 5.3. The scope of the definition of f extends to all nodes are doubly circled.

The main expression of the program is the root node of the scope graph. The evaluation

of the program starts in this expression. If the program is a set of global functions without a main expression, an empty root node can be created. Edges are introduced from the root node to any global function that should be retained as global during function localization.

The scope graph describes an ideal block structure form for a program. It is ideal in the sense that a function definition is visible only to those other functions that make use of it. Unfortunately, the structure of the program is described as an arbitrary graph. Since programs form trees rather than arbitrary graphs, the scope graph cannot be directly used to describe block structure. The purpose of the block sinking stage is to most closely approximate the structure of the scope graph with a tree. As described in Section 2.2, a tree with functions as nodes is equivalent to a block-structured program.

It is possible to define and make use of other definitions of the ideal block structure of a program. We have taken the approach that the ideal block structure of a program is realized by declaring locally as many function as is possible. Section 6.2 provides a discussion of this point.

5.3.2 From scope graph to scope DAG

The structure of the scope graph of a program is unrestricted. Transforming this graph into a tree is a non-trivial operation. The graph may contain any number of cycles. A cycle represents a recursive dependency between the functions of the cycle. To simplify the graph, we isolate these recursive dependencies, and treat each of them separately. The strongly connected component (SCC) coalescence algorithm of The Dragon Book [1] does exactly this. A strongly connected component is a set of nodes with paths from each node to all the other nodes of the component. Thus, any cycle is a strongly connected component, and a strongly connected component can hold several interlinked cycles. The algorithm transforms an arbitrary graph into a graph where each strongly connected component has been coalesced into a single node. The nodes of the resulting graph are sets holding the nodes of the original graph. A node that is part of no cycle is placed in a singleton node. Nodes from the same strongly connected component are placed in the same node.

Using the strongly connected component algorithm to coalesce the nodes of the scope graph results in a directed acyclic graph (DAG). In this scope DAG, the scope of a function may be larger than its scope in scope graph. Since we have assumed variable hygiene, we need not worry about name clashes. The scope of a function f belonging to a set F is: all the functions of F together with the functions of all direct predecessors of F in the scope DAG.

While this transformation is certainly correct given the extended scope, it is too coarse for our purposes. Using this algorithm, the scope of a function declaration may be extended too far. Consider the program of Figure 5.4, displayed with the scope graph of the program. Coalescing the strongly connected components of the graph yields a graph with only two nodes:

$$\{\mathsf{body}\} \longrightarrow \{\mathsf{odd?}, \mathsf{even?}\}$$

```
let odd? x  =  if zero? x
               then false
               else even? (pred x)
    even? x =  if zero? x
               then false
               else odd? (pred x)
in  odd? n
```

Figure 5.4: A simple program and its scope graph

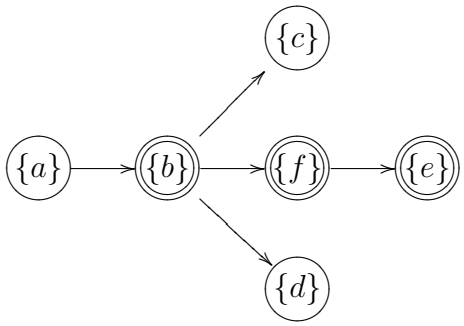


Figure 5.5: The scope of the definition of f in a scope DAG

However, as can be seen from the original scope graph, the function even? is not called directly by the body of the let block. Thus, the scope of the declaration of even? does not need to extend into the body of the let block.

The solution is to recurse the SCC coalescence algorithm on each SCC of the graph, after removing certain redundant edges. We also need to extend the definition of scope in the graph, in accordance with how scope is defined for the tree representation of a program (see Section 2.2.2). We thus extend the scope of a function to include not only the functions in the direct predecessors of its node, but also the functions of any successors of the node. In the textual representation of a program, any enclosing function is visible to a local function. This corresponds to the extension of scope to include all successors in the graph. The extended scope is illustrated in Figure 5.5.

In the example of even? and odd?, the component holding even? and odd? has a single entry-point, odd? (this can be seen from the scope graph). The edge from even? to odd? is redundant because the scope of a function now extends to its successors. We can thus remove it, yielding a more detailed graph:

$$\{\mathsf{body}\} \longrightarrow \{\mathsf{odd?}\} \longrightarrow \{\mathsf{even?}\}$$

The algorithm proceeds as follows. If a SCC has a single entry point, we can remove all edges from nodes in the SCC to this entry point. In this case the entry point function is the outermost, enclosing definition of the functions in the SCC since the other functions can be localized to it. It is thereby visible to all functions in the SCC. Removing the edges reduces the number of cycles in the SCC. Using the SCC detection algorithm recursively on the nodes of the component will refine the structure of the result. Any SCC detected in this pass is processed recursively, by applying the algorithm to the subgraph holding the nodes of the component.

The recursion stops when there are no SCC left in the subgraph, and a SCC with more than one entry point is never processed recursively. The result is a scope DAG where the scope of function extends to a larger number of nodes. It extends to the other functions of its node, to represent a set of mutually recursive function with multiple entry points. The scope also extends to any direct predecessor in the DAG, allowing non-recursive functions to call each other. Finally, to compensate for edges that were removed by the recursive SCC coalescence algorithm, the scope of a function also extends to any successor in the scope DAG.

5.3.3 From scope DAG to scope tree

Applying the recursive SCC coalescence algorithm to the scope graph results in a scope DAG. In order to utilize the isomorphism of trees and block structure to create a block-structured program, the DAG must be transformed into a tree. Any part of the graph where a node has more than one predecessor must be transformed in a scope-preserving way, so that every node has a single predecessor.

The scope of each function declaration is extended in accordance with the definition of scope in a scope tree (see Section 2.2.2). The scope of a function f belonging to a set F extends to the other functions of F, the direct predecessor F' of F, any successors of F and any successors of F'. This definition of scope corresponds to the representation of a letrec block in Section 2.2.2: each local function is a successor of the node representing the body of the block. This scope extension is illustrated in Figure 5.6.

From DAG to tree

A node that has two or more direct predecessors must be moved to the earliest common predecessor of these direct predecessors. More precisely, a node N that is a direct successor of two nodes N_1 and N_2, is detached from N_1 and N_2. Let the closest common predecessor of N_1 and N_2 be the node M. If the nodes M' and M'' are equally close in terms of the length of the shortest path, the DAG is locally symmetric and we choose either one. The node N is made a direct successor of M. The successors of N are not changed — they remain as successors of N. This transformation is iterated on the graph until a fixed point is reached. At this time, no nodes will have more than one predecessor. The result is thus a tree structure.

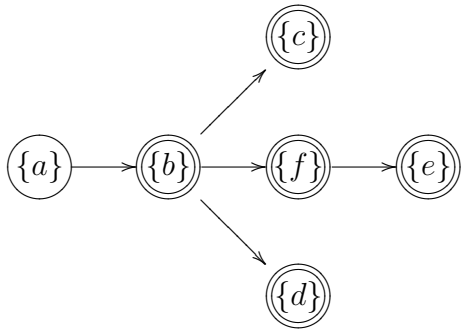


Figure 5.6: The scope of the definition of f in a scope tree

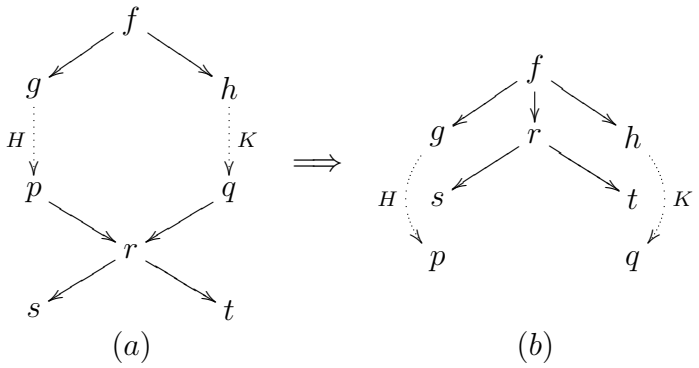


Figure 5.7: Graph transformation from DAG to tree

The correctness of this transformation on parts of the graph where edges were removed by the recursive SCC coalescence algorithm may appear dubious, but rests on the fact that edges were only removed within SCCs with single entry points. Consider the graph transformation illustrated in Figure 5.7. Here, the arcs labeled H and K represent arbitrary, disjoint DAG structures holding functions that are reachable from f, and all of which can reach r.

In this figure, the scope of r extends to the predecessors p and q and the successors s and t. Moving r to become a successor of f does not restrict the scope of the definition of r. The functions s and t can still refer to r, since their position in the graph is unchanged. The scope of r now extends to f and any successor of f, which includes p and q. On the other hand, the functions f, g, s, t, p and q were visible to r before the transformation, along with any functions in H and K. After the transformation, the function p and q are no longer visible to r, and neither are any functions in H and K. For the transformation to be correct, it is thus necessary that r did not refer to these functions.

We must consider the correctness of the transformation step in the presence of two transformations that may have removed edges from the scope graph. The first of these is the SCC elimination algorithm, which removes edges from strongly connected components that have a single entry point only. The second is the transformation step itself, which may remove edges that can be determined to be redundant after the insertion of edge.

If no edges were removed before the transformation from the subgraph (a) of Figure 5.7, the correctness of the transformation is trivial. In this case, the function r does not need to refer to the functions p and q, nor to any of the functions of H and K. If r had contained a reference to any of these functions, this reference would have been explicitly represented by an edge.

Assume that edges had been removed from (a) by the SCC elimination algorithm, and that these were edges from r to either p, q or some function in H or K. If this was the case, r would have been in the same SCC as this function. The SCC would have had multiple entry points, since both branches of the subgraph (a) could not have been contained in the component unless the component had the entire subgraph as its nodes. This contradicts the condition that edges are only removed if the SCC has a single entry point. No SCC could have included both branches of (a), since H and K are disjoint. The disjointness effectively limits our choice of f since edges from H to K would have forced us to pick a smaller subgraph of (a) for the transformation, and thereby a different node f.

The edges that are removed by the transformation step itself have been determined to be redundant after the node has been moved, so they can safely be removed. Since the transformation always extends the scope of a function definition when moving it and conversely limits the set names it can refer to, the edges that were removed will not cause subsequent uses of the transformation to be incorrect.

From sets of functions to singleton nodes

The result of transforming the scope DAG to a tree structure is a tree with nodes that are sets of functions. To complete the transformation from scope graph to scope tree, each node must represent a single function.

A node M that is the successor of a singleton node S is represented as a tree by making each function of M a direct successor of S. A node N that is the successor of a node N' that holds several functions must be represented differently. The node N' together with N represents a set of functions with multiple entry points that call any number of local functions through multiple entry points. If the functions of N are called by a single function f in N', the functions of N can be localized to f. The node N' is processed as if N was not part of the graph, and N is made the direct successor of f.

If multiple functions from N' call the function of N, the sets N and N' must be merged. The merging reflects that a set of mutually recursive functions that are used by several functions cannot be localized to all of them. Any successors of N are made successors of N'. Each merging operation extends the scope of the functions involved. Once again, variable hygiene ensures that this is a safe operation.

Result

The result of the last graph transformation is a scope tree where each node holds a single function. The original scope graph has been transformed through a series of scope-extending transformations to a tree. This tree can be mapped back to a program by utilizing the isomorphism of trees and block structure, described in Section 2.2.2.

5.3.4 Flow graph

The formal parameters of a procedure can be divided into two categories: the parameters that are bound to the same expression in every invocation of the procedure, and those that are not. Parameters that are always bound to the same expression can be further divided into two subcategories: those expressions that simply are formal parameters of some other procedure, and those that are not. Only those formal parameters that are bound to the same parameters of some other procedure in every invocation can be parameter dropped (and this only in some cases — this is discussed in the next section).

Considerations

Parameters that vary from invocation to invocation can obviously not be removed. This reflects the nature of procedures: a procedure can be considered a parameterized expression. The expression (the body of the procedure) usually evaluates to different values under different bindings of these parameters.

Procedures may be passed parameters that are arbitrary expressions. Even when the expression being passed as parameter is the same in every invocation of the procedure, the binding to a formal parameter through the procedure call mechanism is important in most languages. In a language with call-by-value semantics, a non-terminating program could be made to terminate by textually substituting this unchanging expression for the formal parameter throughout the body of the procedure. (Name clashes and the scope of the free variables of an expression usually makes this an invalid transformation). Parameter passing strategies were discussed in Section 1.4.

Concentrating on the most common parameter passing semantics that allow passing as argument arbitrary expressions, namely call-by-name, call-by-value and call-by-need, enables us to state an important property:

Property 1 *Passing an expression as an argument to a procedure is an idempotent operation. With call-by-name semantics, the argument is encapsulated in a thunk. With call-by-value semantics, the argument is fully evaluated. With call-by-need semantics, the argument is encapsulated in a thunk that is evaluated at most one time. Subsequently passing any of these values as an argument does not change their representation.*

This property is essential for the correctness of parameter-dropping, as it enables us to drop parameters that are bound to the same formal parameters of some other procedure in every invocation. The lambda-dropping algorithm could be extended to handle variables bound by other constructs. This is discussed in Section 5.3.6.

The identity of the callee

The flow graph of a program describes the flow of arguments through the formal parameters of each procedure. A formal parameter is bound to an argument through application. Thus, the flow of values through a program is from the application of a procedure to the definition of a procedure, and from the passing of formal parameters as arguments within this procedure to other procedures. The flow graph has nodes for procedure definitions and nodes for procedure applications. Edges model the flow of data through the program by connecting application nodes to a definition nodes.

Procedures may be defined for any number of arguments. For this reason, the flow graph represents each procedure definition by separate nodes for each parameter. Similarly, the application of a procedure is represented by a node for each argument. As described earlier, our only interest is formal parameters that are passed as arguments to procedures. Thus, there is no need to represent in the flow graph procedures that receive anything but formal parameters as argument in a specific parameter position.

To create the flow graph of a program, we must know the identity of each procedure that is invoked. This is trivial in a first-order program. In a higher-order program where procedures may be passed as arguments, we might not be able to determine the identity of a procedure that is invoked in an application. Edges could be introduced in the flow graph from the application node to any procedure that is passed as a an argument. We have taken the alternative approach of removing from the flow graph the definition nodes for any function that is passed as an argument.

A control-flow analysis [41] can be used to limit the set of procedures that must be considered in an application of a higher-order procedure, making the former alternative of introducing extra edges a feasible solution since this would occur on a smaller scale. This would allow parameter dropping in certain cases where using the second alternative of removing nodes would not allow parameter dropping.

Limiting the lambda-dropping algorithm to providing an inverse for lambda-lifting relieves us from having to perform a control-flow analysis. A procedure with free variables that was passed as an argument in the original (before lambda-lifting) program is not passed directly as an argument in the lambda-lifted program. Every occurrence of such a procedure is eta-expanded with its free variables. Thus, any procedure that has been eta-expended by the lambda-lifter is always applied directly to the variables it was eta-expanded with. These parameters are easily represented in the flow graph.

The identity of the parameters

The flow graph of a program is constructed by introducing edges for each application of a function and for each use of a formal parameter as an argument to a function. The application of a known function onto a variable declared as a formal parameter gives rise to two edges. Let the function that is applied be f and the argument it receives in the i'th position be x. Let x be declared as the j'th formal parameter of the function g (this is illustrated in Figure 5.8). Edges are introduced in the flow graph modeling the flow of

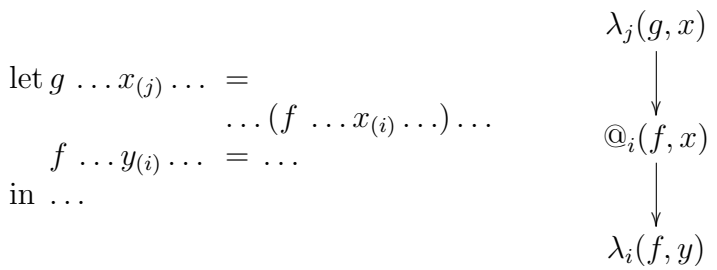


Figure 5.8: The flow graph of a small program

data:

- An edge is introduced from the j'th definition node of g to the i'th application node of f. This reflects the binding of x in the declaration of g.
- An edge is introduced from the i'th application node of f to the i'th definition node of f. This reflects passing the argument into the body of f.

The construction of a small flow graph is illustrated in Figure 5.8. The algorithm we use to construct the flow graph, described in Figure 5.1, constructs the graph in reverse.

The flow graph may contain recursive dependencies. A recursive dependency reflects a formal parameter that is passed directly as an argument in a recursive call. The set of parameters associated with the nodes of a strongly connected component in the flow graph are bound to the same value in every recursive invocation. They can thus be treated as a single node. Coalescing the strongly connected components of the flow graph will place such nodes in the same set. It is non-sensical to use the recursive SCC elimination algorithm in this case, since the removal of edges by the recursive SCC elimination algorithm is legal only in a scope graph with extended scope.

Given the flow graph in DAG form, the identity of each formal parameter can be determined by propagating the names of the formal parameters through the flow graph. Each node without a predecessor is associated with the name of the formal parameter of the node. The associated names are propagated through the graph. If all the predecessors of a node are associated with the same name, the node becomes associated with that name. If two predecessors of a node are associated with different names, the node is associated with the formal parameter of the node (i.e. a new name is associated to the node).

The Use/Def chain for the formal parameter of a function is determined by traversing the graph in reverse. The last node that is associated with the same name as the node of the function is the defining point of the formal parameter. The Use/Def chain thus points from the node of the function to this last node.

Figure 5.9 displays an example program. It serves to illustrate some important points of flow graph construction and application in a simple way. The flow graph of the program,

$$
\begin{array}{lll}
\alpha\,(\text{succ}, \text{pred}, n) = \text{let}\, t\, x = \text{let}\, p\, y & = r\; y\; \text{pred} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\; q\; z & = r\; z\; \text{succ} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\; r\; i\; f = & \text{if}\; (\mathsf{zero?}\; i) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad & \text{then}\; f \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad & \text{else}\; (r\; (i-1)\; f) \\
\qquad\qquad\qquad\qquad\quad \text{in}\;\; (p\; x)((q\; x)\; x) & \\
\qquad\qquad\quad \text{in}\;\; t\; n &
\end{array}
$$

Figure 5.9: A complicated identity function

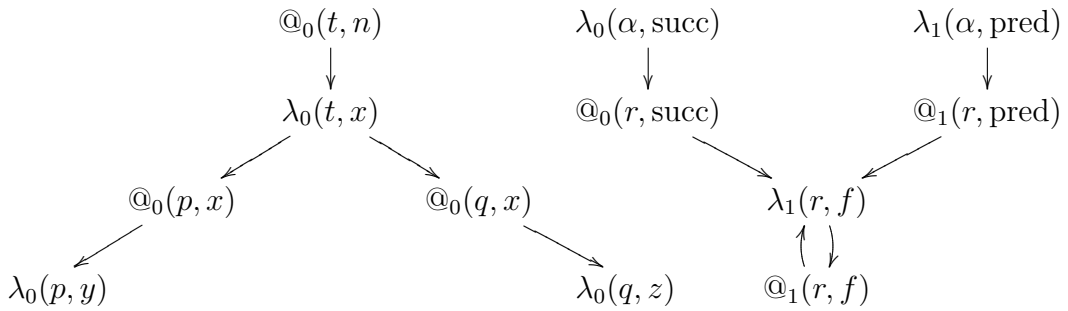


Figure 5.10: Flow graph of the program of Figure 5.9

shown in Figure 5.10, indicates the flow of data through the program. The definition node for the i'th parameter x of the function f is depicted as $\lambda_i(f, x)$. The application of the function f onto the formal parameter x in the i'th argument is depicted as $@_i(f, x)$. The nodes $\lambda_0(r, i-1)$ and $@_0(r, i-1)$ were discarded since the argument $(i-1)$ to r isn't a formal parameter. As a consequence, the nodes $@(r, a_y)_0 0$ and $@(r, a_z)_0 0$ were also discarded.

The third parameter of α, the integer n, is passed to the function t, and onwards to the functions p and q. The first and second parameters of α are both passed as arguments to the function r. The function r thus receives two different formal parameters as its second argument. The parameter f of the function r can be seen to depend upon itself. Such recursive dependencies are eliminated by isolating the strongly connected components of the flow graph. The result of the SCC elimination can be seen in Figure 5.11.

We proceed to propagate variable identities through the flow graph. The result can be seen in Figure 5.12. The nodes are annotated with the propagated variable identities. The formal parameter n of the function α has been propagated through the graph, indicating that the functions p and q are always passed this variable as their argument. The arguments "succ" and "pred" to the function r are different so the node of r is annotated with a new

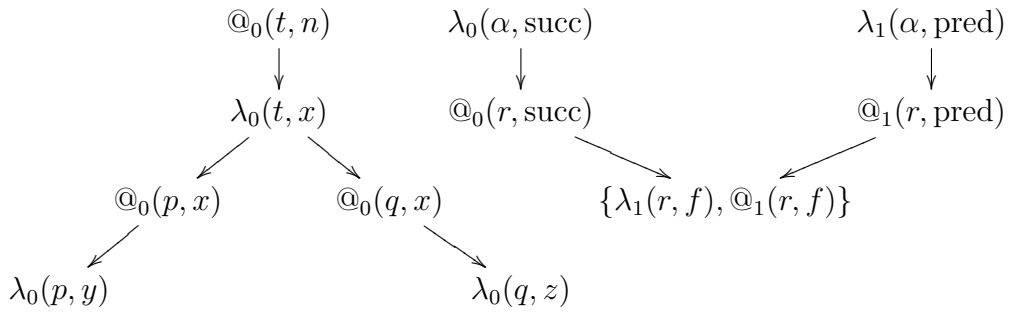


Figure 5.11: The flow graph of Figure 5.10 after SCC elimination

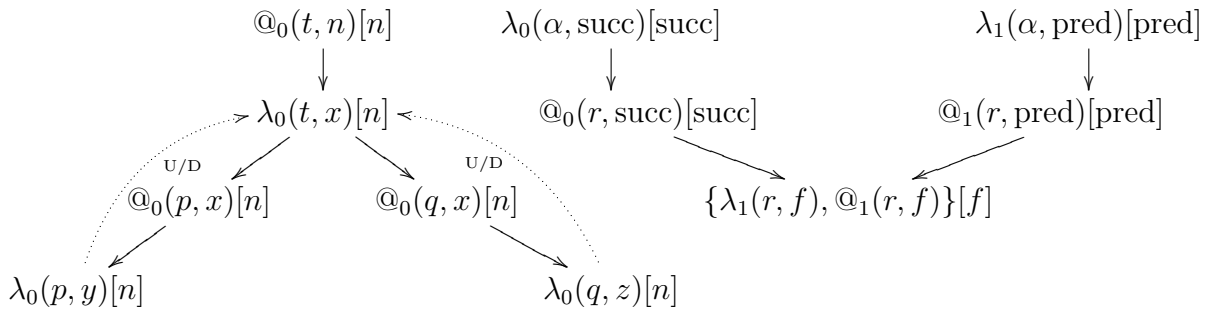


Figure 5.12: The flow graph of Figure 5.11 annotated with Use/Def chains

name (the name of the parameter: f). The Use/Def chains of the functions t and r point to themselves, and are not shown in the graph.

5.3.5 Removing formal parameters

Given the Use/Def chains for the formal parameters of each function, the removal of redundant formal parameters is easily accomplished. The block structure may have been locally collapsed during the block sinking stage, so a function that receives the same formal parameter as an argument in every invocation may not be declared in the scope of the formal parameter. Naturally, a formal parameter is dropped only if it is made redundant by some other formal parameter that is visible to the function declaration.

A formal parameter of a function is dropped if its Use/Def chain points to some other formal parameter that is visible in the scope of the declaration of the function. The name of the parameter at the end of the Use/Def chain is substituted for the name of the

formal parameter of the function throughout the body of the function. When all function definitions have been processed, each application is processed. Arguments corresponding to a formal parameter that was removed are themselves removed.

Removing the redundant parameters of a function may result in function of no parameters (i.e. a thunk) that encapsulates a function. If this is the case, we remove the empty declaration and the corresponding empty applications. This can be done while removing parameters and arguments.

5.3.6 Sugared or unsugared

A lambda-lifting algorithm moves procedure declarations out of letrec blocks, making each a recursive equation. Other constructs, such as let blocks and destructors (letType in Schism and case in ML), can be desugared into constructs that are subsequently processed by the lambda-lifter. A lambda-lifter designed to produce a set of recursive equations suitable for interpretation or compilation would perform this desugaring. A lambda-lifter designed to make each procedure scope-insensitive and simplify the structure of each procedure declaration by removing local functions does not need to perform this desugaring step.

A lambda-dropper designed to provide an inverse for a lambda-lifter that that performs desugaring could "re-sugar" the program. Let blocks, for example, could be generated in lieu of every non-recursive function that is applied once only. As was the case for lambda-dropping without considering syntactic sugar, we cannot provide a unique inverse. In some situations, the result may be very different from the original.

A lambda-dropper designed to provide an inverse for a lambda-lifter that leaves behind name-binding syntactic constructs can make use of the extra scope introduced by these constructs. Alternatively, the lambda-dropper can tolerate such constructs without making use of the extra scope.

Employing let-floating to actually move non-functional let bindings inwards into a more restricted scope or outwards to a wider scope is discussed in Section 5.5.1.

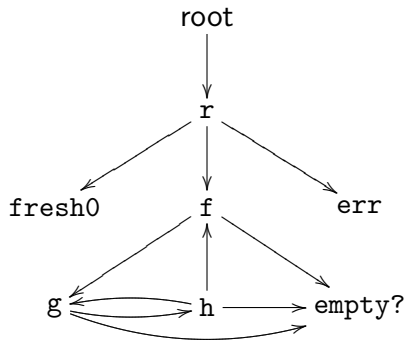
5.4 A Detailed Example

In Section 3.4, we demonstrated how the program of Figure 3.3 was lambda-lifted, resulting in the program displayed in Figure 3.4. Lambda-dropping can be applied to this program, as described in Figures 5.1 and 5.2. In this program, many variables have the same name. For the sake of clarity, we retain the original names, and annotate the variables to tell them apart when necessary.

5.4.1 Function sinking

We proceed according to the algorithm of Figure 5.1.

1. We start by creating the scope graph of the program. The edges correspond to references to function names.



During the creation of the scope graph, we also construct Def/Use chains for functions. Reversing the edges of the scope graph yields the Def/Use chains.

2. We recurse the SCC coalescence algorithm on the scope graph, by removing edges and coalescing nodes into sets.

   (a) The nodes $\{\texttt{h}, \texttt{g}, \texttt{f}\}$ are in the same SCC.

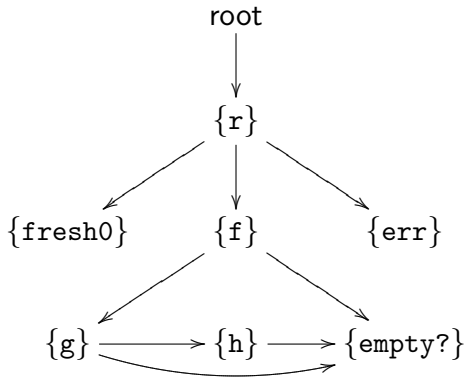
   (b) $\texttt{f}$ is the entry point of this SCC (by the edge $\langle \texttt{r} \to \texttt{f} \rangle$), so $\langle \texttt{h} \to \texttt{f} \rangle$ is removed.

   (c) Now $\{\texttt{h}, \texttt{g}\}$ are in the same SCC.

   (d) $\texttt{g}$ is the entry point of this SCC (by the edge $\langle \texttt{f} \to \texttt{g} \rangle$), so $\langle \texttt{h} \to \texttt{g} \rangle$ is removed.

   (e) There are no more SCC.

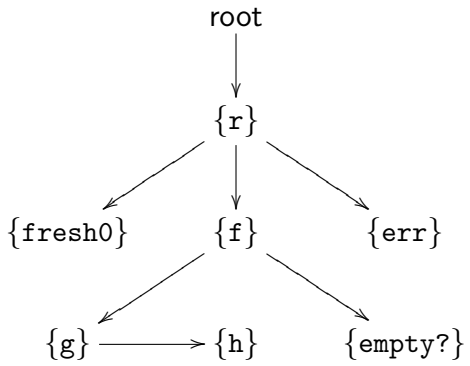
   The resulting DAG has the same structure as the original graph. Each node is a singleton, since recursive dependencies could be resolved by removing edges rather than coalescing nodes.



3. We transform the DAG into a tree:

- There are two disjoint paths from {f} to {empty?}: one direct from {f} itself, and one going through {g}. To remove the common successor, we remove the edges ⟨{f} → {empty?}⟩ and ⟨{g} → {empty?}⟩, and we add the edge ⟨{f} → {empty?}⟩.
  There are still two disjoint paths from f to empty?. One is direct from f and one goes through h. We remove the edges ⟨{f} → {empty?}⟩ and ⟨{h} → {empty?}⟩, and we add the edge ⟨{f} → {empty?}⟩.
- No node is associated with more than one function, so there is no need to collapse the tree.

The result of the transformation is the following tree:



4. We re-construct the program from the tree:

   - There are edges from {r} to {fresh0}, {err} and {f}. So we place the functions from these nodes in a block in r.[1]
   - There are edges from {f} to {g} and {empty?}, so we place g and empty? in a block in f.
   - There is an edge from {g} to {h}, so we place h in a block in g.

The block structure of the resulting program can be outlined as:

```
(define (r ...)
  (letrec ([f (lambda (...)
                (letrec ([g (lambda (...)
                              (letrec ([h ...])
                                 ...))]
                            [empty? ...])
                   ...))]
           [fresh0 ...]
           [err ...])
    ...))
```

[1] If r had not been a singleton node, the structure of the tree would have been locally collapsed in the previous step.

5.4.2 Dropping parameters

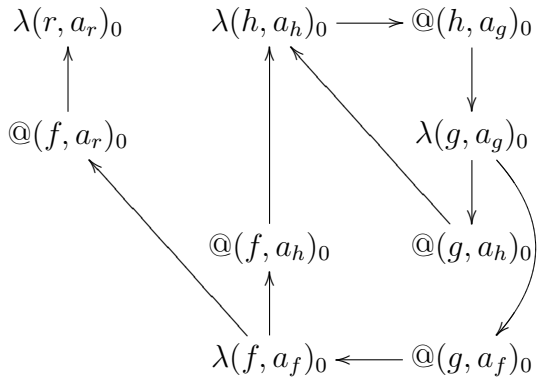
We proceed according to the algorithm of Figure 5.2.

1. There are only function names and formal parameters in the program. For each function we let the formal parameters occurring in the body point to the definition of the function.

2. Constructing the flow graph:

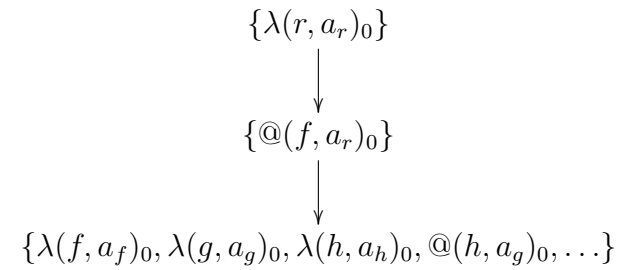
   - We create appropriate nodes for all functions and all applications, with the following exceptions:
     - The function `fresh0` is passed as an argument, so we create no nodes for it.
     - The function `err` is passed as an argument, so we create no nodes for it.
     - The function `empty?` receives an argument that is not a formal parameter. Thus we discard all nodes associated with this first parameter.

     In the rest of this section, we index uncurried parameters starting from zero. The resulting graph consists of four components, one for each of `a`, `b`, `c`, and `d`. These components are all identical except for variable positions and parameter names. The graph for variable position zero and the parameter `a` looks like this:



     For clarity, the formal parameter `a` has been annotated with the name of the function defining it, in each node. The three other components of the graph detail parameter positions one to three, with the variables `b`, `c`, and `d`. In the general case, the graph can have connections between nodes with different variable positions.

   - All nodes for the parameter `a` of `f`, `g`, and `h` are in the same SCC, and likewise for `b`, `c`, and `d`. Thus, in the case of variable position zero and the parameter `a`, coalescing the SCC of each component of the flow graph yields:

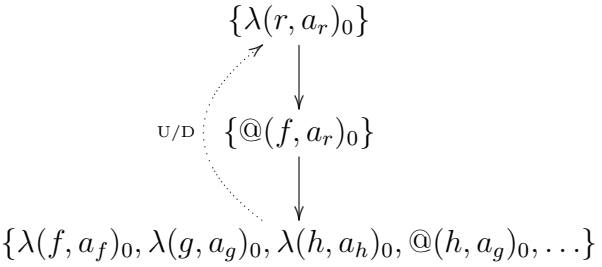


The three other components also have the same structure.

- Reversing the graph yields the flow graph.

3. The propagation of identities through the flow graph is trivial: The roots are $\lambda(r, v)_i$, where v is `a`, `b`, `c` or `d`, and i is 0, 1, 2 or 3, respectively. Each root is assigned a unique ID. All nodes for `a` receive the same ID, and similarly for `b`, `c`, and `d`.

   This yields a flow graph annotated with U/D chains for variables:



4. Finally, we traverse the program:

   `(define (r ...) ...)` : The parameters of a top-level definition cannot be parameter dropped.

   `(letrec (...[f (lambda (a b c d) ...)] ...) ...)` :

   The U/D chain of `a` points to the variable `a` in the definition of `r`. This variable is visible in the current scope, so `a` can be parameter-dropped. Similarly for the parameters `b`, `c`, and `d`.

   Dropping these four parameters leaves none behind. The body of `f` is a letrec block, the body of which is a lambda form (with the formal parameters `reject` and `xs`). We replace the declaration of `f` with this lambda form, and place the letrec block inside.

```
[f (lambda (a b c d)
     (letrec (...)
       (lambda (reject xs) ...)))]
```

   is replaced by

```
[f (lambda (reject xs) (letrec (...) ...))]
```

The definitions of `g` and `h` are handled similarly.

The functions `fresh0`, `err` and `empty?` have no U/D chains, so they cannot be dropped.

5. Every application is processed. If it is an application of `f`, `g` or `h`, the application is replaced by the function itself.

The resulting program is the same as the program we lambda-dropped by hand in Section 4.3. It can be found in Figure 4.1.

5.5 Related Issues

Lambda-dropping creates block structure from recursive equation by sinking definitions inside letrec blocks. The newly created scope is used to reduce parameter passing by eliminating redundant formal parameters.

Block structure can be created in many ways. Section 5.5.2 describes an alternate way of generating block structure from a set of recursive equations. Moving let bindings inwards and outwards through the block structure of a program forms a group of generally useful program transformations, and are described in Section 5.5.1. Monadic programming can remove the need to explicitly thread variables representing state through the entire computation. This is appears to be similar to parameter dropping, and is discussed in Section 5.5.3.

5.5.1 Let floating

Peyton Jones *et al.* describe a number of transformations all classified as "let-floating" [37]. These transformations move let-block value bindings either inwards into let blocks or outwards outside procedure declarations. A number of additional transformations serve to fine-tune the let-block structure. These transformations are simplest in a lazy language (contrary to a strict language), but if the optimizations were augmented by a strictness analysis they would be equally valid in a strict language. In this section we concentrate on lazy languages.

Moving a binding inwards can be beneficial. Even in a lazy language where the computation is only performed if the value is needed, constructing a thunk holding the computation still uses time and space. A binding with a wide scope can be moved to have a more local scope if it is only referenced within this more restricted scope. If the more restricted scope is entered on certain conditions only, the thunk for the localized binding will be constructed under these conditions only. This justifies floating let bindings inwards into other expressions. Consider the inwards let-floating transformation shown in Figure 5.13. After the transformation, the let binding of x has been moved into a branch of the

$$\begin{array}{l} \text{let } x = y + 1 \\ \text{in } \quad \text{if } y < 0 \\ \qquad \text{then } 0 \\ \qquad \text{else } x * x \end{array} \quad \Longrightarrow \quad \begin{array}{l} \text{if } y < 0 \\ \text{then } 0 \\ \text{else let } x = y + 1 \\ \qquad \text{in } \; x * x \end{array}$$

Figure 5.13: Let-floating inwards for optimization

$$\begin{array}{l} \text{let } f\, x = \text{let } g\, y = \text{let } z = x + 1 \\ \qquad\qquad\qquad\qquad \text{in } \; z * z * y \\ \qquad\qquad \text{in } \; \ldots g \ldots g \ldots \\ \text{in } \; \ldots \end{array} \quad \Longrightarrow \quad \begin{array}{l} \text{let } f\, x = \text{let } z = x + 1 \\ \qquad\qquad \text{in let } g\, y = z * z * y \\ \qquad\qquad\quad \text{in } \; \ldots g \ldots g \ldots \\ \text{in } \; \ldots \end{array}$$

Figure 5.14: Let-floating outwards for optimization

conditional. The thunk holding the computation $y + 1$ is thus only created in a situation where it is subsequently used.

Moving a binding outwards outside a procedure can be beneficial. This transformation is called "full laziness." A binding that is nested inside a procedure but that depends only on values that are free variables of the procedure can be floated outside the procedure. The value of the binding is made once every time a closure for the procedure is allocated rather than every time the procedure is invoked. After floating a binding outwards outside a procedure it may be a candidate for inwards let-floating. Outwards let-floating is illustrated in Figure 5.14. Here, the binding for the value z is floated outside the definition of g. It is thus computed once only for every invocation of f, rather than once for every invocation of g.

Outwards let-floating of bindings corresponds exactly to the block-floating transformation employed during lambda-lifting. Inwards let-floating of bindings corresponds precisely to the block-sinking transformation employed during lambda-dropping. Outwards and inwards let-floating moves value bindings, the block-floating and block-sinking transformations used during lambda-lifting move function bindings. Outwards and inwards let-floating as presented here is more complicated than block-floating and block-sinking as presented in relation to the corresponding algorithms because the bindings that are moved have not been made scope insensitive. Since outwards and inwards let-floating are optimization strategies designed to manipulate non-functional bindings (although they can be applied to functions as well), and lambda-lifting/lambda-dropping are transformations that move function declarations, the two transformations can be viewed as orthogonal even though they are almost identical.

5.5.2 Dependency analysis

In his textbook, Peyton Jones uses an analysis to generate block structure [38]. In several ways, the algorithm for this dependency analysis is similar to the algorithm we employ for function localization. Its goal, however, differs. The goal of the dependency analysis is to decrease the size of each letrec block and simplify type checking. The dependency analysis only works for a program that has a main expression.

The dependency analysis constructs a dependence graph, a graph that is identical to the scope graph used for lambda dropping. The strongly connected component coalescence algorithm of The Dragon Book [1] is used to transform the dependency graph into a DAG. If a function f depends on (uses) a function g, either f and g are in the same node or there is an edge from the node of f to the node of g. A topological sorting of the DAG orders the nodes in dependency order: a function is ordered before or together with any functions it depends upon. Block structure can be generated sequentially. We start with the main expression. The functions of the least node are placed in a block, the body of which is the current expression. If the node is associated with a single function, a let block is generated. Otherwise, a letrec block is generated. The remaining nodes are processed recursively by generating enclosing let blocks.

Transforming a set of recursive equations using the dependency analysis results in a program where each set of mutually recursive functions is declared in separate blocks. Each function declared in a letrec block can refer to any other function of the same block. The compiled code is simplified by the dependency analysis because only those function that are used recursively are placed in the same block. Most polymorphic type-checking algorithms restrict the types of mutually recursive functions while determining their type. This does restrict polymorphism during recursion, but it also makes the type-checking algorithm much simpler. Each use of these functions inside the block is polymorphic without restrictions. The type-checker may arrive at more detailed type information after a dependency analysis, since each set of functions that share types is reduced in size. Mycroft provides a more detailed discussion of the type-checking problems that may crop up without a dependency analysis [35].

Assuming the lambda-lifted version of the DFA program (see Figure 3.4) was extended with a main expression calling the function `r` with appropriate arguments, Peyton Jones's dependency analysis could be applied to this program. Coalescing the strongly connected components of the dependency graph (which is identical to the scope graph, presented at the beginning of 5.4.1) yields the DAG of Figure 5.15.

Topologically sorting the graph and generating block structure accordingly yields the skeleton of Figure 5.16. Comparing this result to the result obtained from lambda-dropping, we can see that functions that were localized by the lambda-dropper are are the declared in the outermost blocks. The lambda-dropping algorithm generates block structure starting with the outermost definitions, according to a topological ordering on the scope tree. The dependency analysis generates block structure starting with the innermost definitions, according to a topological ordering on the scope graph.

The transformation from scope graph to scope tree may extend the scope of the defi-

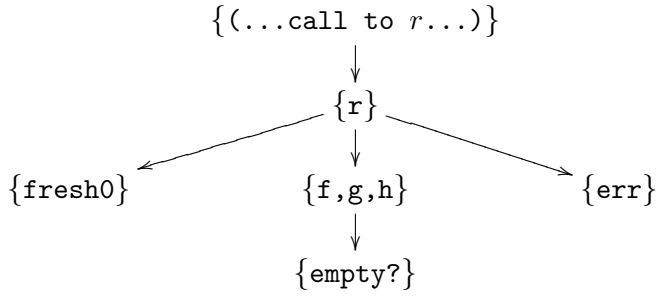


Figure 5.15: Coalesced dependency graph for the DFA example

```
let empty? = ...
in letrec f = ...
          g = ...
          h = ...
   in let err = ...
      in let fresh0 = ...
         in let r = ...
            in (...call to r...)
```

Figure 5.16: Skeleton of the DFA example as generated by dependency analysis

nition of each function. The dependency analysis generates block structure based on the scope graph, so the resulting block structure can be more detailed than the block structure generated by the lambda-dropping algorithm. On the other hand, the lambda-dropping algorithm uses the recursive SCC coalescence algorithm, which serves to refine the block structure. In the DFA example, the block structure of mutually recursive functions generated by the lambda-dropping algorithm is more refined than that of the dependency analysis. The functions f, g, and h are nested inside one another. The block structure of non-recursive functions generated by the dependency analysis is more detailed than that of the lambda-dropping algorithm. The functions `empty?`, `err`, and `fresh0` are placed in separate let blocks. This suggests to use the dependency analysis on each letrec generated by the lambda dropper.

The lambda-dropping algorithm generates block structure where local functions are declared in a block nested inside some other function. This places the local functions in the scope of the formal parameters of this function. The dependency analysis, on the other hand, does not place local functions in the scope of the formal parameters of other functions. The names of the functions declared in an enclosing block are visible to local functions, but their formal parameters are not. The block structure generated by the dependency analysis is thus incompatible with parameter dropping.

The dependency analysis can be adapted to post-process lambda-dropped programs. This post-processing will have no effect on parameter dropping, but will bestow the usual

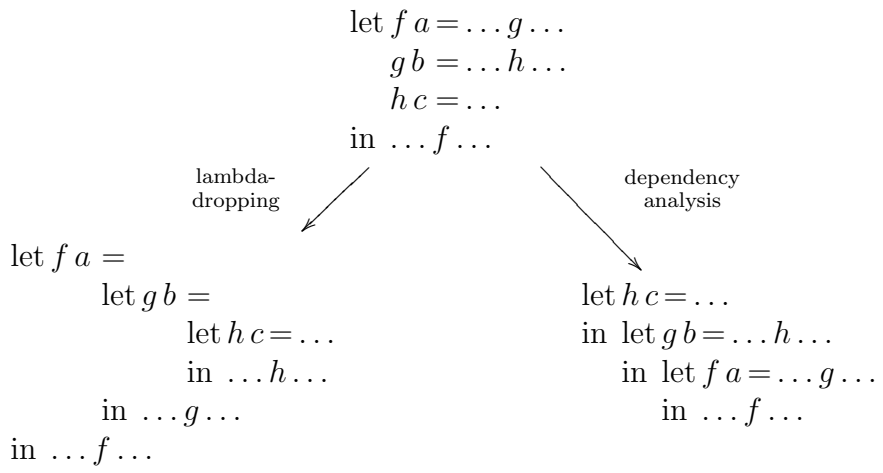


Figure 5.17: Block structure generated by lambda dropping and by dependency analysis

benefits of dependency analysis upon the lambda-dropped program, namely decreasing the size of letrec blocks and simplifying type checking.

Both the lambda-dropping algorithm and the dependency analysis use a topological sorting of the scope graph to order the way block structure is generated (in the case of lambda-dropping, after transforming it). The lambda-dropping algorithm generates block structure starting with the outermost definitions, while the dependency analysis generates block structure starting with the innermost definitions. This is reflected by the ordering of the definitions of the resulting program, from the outermost definition to the innermost definition: they will in many cases be ordered approximately in reverse. If the scope graph of the program is a tree, this is even more apparent. If the scope graph is a path (i.e. each node has a single successor), the order of the definitions in a program transformed by the lambda-dropping algorithm is the reverse of the order of the definitions in a program transformed by the dependency analysis, as is illustrated by Figure 5.17.

5.5.3 Monadic programming

In an impure language, certain features that are used locally affect the global state of the program. This includes modifiable state, exception handling (and other access to the control flow of the program), and Input/Output. This appears to be contrary to the philosophy of functional programming, since it seems to impose a sequential ordering of computation and the ability of a computation to modify the result of some other expression. Many function programming languages, such as SML and Scheme, incorporate these impure features into the language. References in SML and global variables in Scheme are modifiable. Call with current continuation (call/cc) provides access to the control flow of

```
type Tree x = Node x | Branch (Tree x) (Tree x)

processTree :: Tree Bool -> (Tree Bool, Int)
processTree tree =
   let counter = ref 0
       traverse :: Tree Bool -> Tree Bool
       traverse (Node true) = i:=i+1;Node false
       traverse (Node false) = Node true
       traverse Branch left right = Branch (traverse left)
                                           (traverse right)
   in let res = traverse tree
      in (res,!i)
```

Figure 5.18: Traversal function programmed using side effects

the program. Input and output are performed by operations that independently interact directly with the operating system.

In a pure language, state can be represented by threading a data structure through every part of the program that needs access to the state. The sequential relationship between state modification and access is expressed by interdependence of values. The program of Figure 5.18 shows a simple traversal function, programmed in an informal mix of Haskell and SML.[2] It traverses a tree of boolean values, returning a tree with negated leaves. Additionally, it counts the number of **true** boolean values in the tree using an integer stored in a reference cell. The program of Figure 5.19 computes the same result using an integer value that is passed to the traversal function and returned in each invocation.

Access to the flow of control can be implemented in a pure language by transforming the program to Continuation-Passing Style (CPS, [20] provides an introduction to CPS). After CPS transformation, the current continuation (the flow of control) can be explicitly manipulated. The result of evaluating an expression is a computation (the continuation) rather than a value. This allows implementation of error handling (including exception handling) and control operators, such as call/cc. I/O is inherently impure, as it causes side-effects in the operating system. Implementing I/O by a set of functions that side-effect the environment works well in a strict language, such as SML. The same approach could be made to work in a lazy language, but could easily yield unexpected results. Each of the return values of a sequence of I/O operations would have to be explicitly evaluated for the operations to take place. Some versions of Haskell implement I/O by dialogues. Interaction with the operating system is viewed as a lazy request transformer. Lists of requests are transformed to lists of responses.

[2]The local type declaration would not be allowed in Haskell, the references are written in SML style.

```
type Tree x = Node x | Branch (Tree x) (Tree x)

processTree :: Tree Bool -> (Tree Bool, Int)
processTree tree =
   let traverse :: (Tree Bool, Int) -> (Tree Bool, Int)
       traverse (Node true, i) = (Node false, i+1)
       traverse (Node false, i) = (Node true, i)
       traverse (Branch left right, i) =
          let (res0, i0) = traverse (left, i)
              (res1, i1) = traverse (right, i0)
          in (Branch res0 res1, i1)
   in let (res, i) = traverse (tree,0)
      in (res, i)
```

Figure 5.19: The function of Figure 5.18 programmed without side effects

Monads provide a uniform style of programming for introducing features such as state, exceptions, and I/O in a pure language. Monadic programming provides a convenient notation for threading extra values through the computation. CPS and monadic programming are for most purposes equivalent; one can be used to simulate the other. However, monadic programming allows a more restrictive style of programming, where the continuation is manipulated within the monad only. Monads provide a convenient syntax for evaluating and sequencing I/O operations. Our primary interest in monads is that transforming a pure program that threads state through the computation (such as the program of Figure 5.19) to use monads results in a program where the state appears to have been parameter dropped. Figure 5.20 displays the transformed version of the program. The program is written in a pure language and the state is no longer passed as an explicit argument during the traversal of the tree. The program is described in detail later.

A monad is a triple `(M, unitM, bindM)` consisting of a type constructor `M`, a polymorphic function `unitM` that coerces a value into a computation, and a polymorphic function `bindM` that evaluates a computation. The types of the two functions are:

```
unitM :: a -> M a
bindM :: M a -> (a -> M b) -> M b
```

A program is converted to monadic form by changing the types to return computations rather than values. A function of type `a -> b` is changed to have type `a -> M b`. In the traversal program of Figure 5.20, the type of the local function `traverse` has been changed accordingly, to work with the monad `S`. Functions that return values are made to return computations using the `unitM` function. In the traveral program, `traverse` on a node with a **false** value uses `unitS` to accomplish this. A computation is evaluated by using `bindM`,

```
type Tree x = Node x | Branch (Tree x) (Tree x)

processTree :: Tree Bool -> (Tree Bool, Int)
processTree tree
   let traverse :: Tree Bool -> S (Tree Bool)
       traverse (Node True) = tickS `bindS` \() ->
                              unitS (Node false)
       traverse (Node False) = unitS (Node true)
       traverse (Branch left right) =
          (traverse left) `bindS` \res0 ->
          (traverse right) `bindS` \res1 ->
          unitS (Branch res0 res1)
   in showS ( (traverse tree) `bindS` \res ->
              getS `bindS` \i ->
              unitS (res, i) )
```

Figure 5.20: The function of Figure 5.18 programmed using monads

allowing the result to be used. The traveral program uses `bindS` to evaluate the result of the recursive calls for a `Branch`. Here, `` `bindS` `` is the Haskell notation for the application of a prefix function as an infix function. Figure 5.21 shows the state monad that was used in the traversal program.

The state monad of Figure 5.21 contains three additional declarations that were been used in the traversal program. The function `tickS` changes the state, by incrementing the counter. It is used in the traversal program to count every occurence of a **true** value. To reflect the change in state, as opposed to the computation of a value, an empty value is returned. The function `getS` provides access to the state. The current value of the counter is simply returned as the value of computation. The function `showS` lets us get rid of the monad by evaluating the computation in an initial, empty store. It could have been defined to return a printable representation of the value, or something else entirely. In some cases, such as an I/O monad that commmunicates with the operating system, this operation might be non-sensical.

The transformation of the traversal program from explicit threading of the state through the computations to monadic style is similar to parameter dropping. The formal parameter `i` that was passed as a parameter becomes embedded within the monad. However, looking closely at the definition of `bindS` reveals an important difference. The state, that was represented by `i`, is explicitly manipulated within the `bindS` operation. The same holds for the `unitS` operation. Thus, the parameter `i` is manipulated by the monad in every invocation of `traverse`. This is hardly surprising, since the state changes during the traversal of the tree. The parameter `i` could never have been parameter dropped.

```
type State = Int
type S x = State -> (x,State)

unitS :: a -> S a
unitS h = \s0 -> (h, s0)

bindS :: S a -> (a -> S b) -> S b
f `bindS` g = \s0 -> let (v1,s1) = f s0
                         (v2,s2) = g v1 s1
                     in (v2,s2)
tickS :: S ()
tickS h = \s -> ((),s+1)

getS :: S State
getS h = \s -> (s,s)

showS :: S a -> a
showS h = let (v,s) = h 0
          in v
```

Figure 5.21: Simple state transforming monad

Parameter dropping allows formal parameters that do not change between invocations to be accessible implicitly rather than explicitly. Such formal parameters can be removed and accessed as free variables. Monads allow (amongst other things) a separation of formal parameters and return values into the values that are computed as the result of a request and the values that are used to provide auxiliary features.

The notation and some of the naming conventions used in this section are due to Wadler [43].

5.6 Summary

Lambda-dropping is achieved by let floating of function definitions into other functions definitions and eta-reduction. The lambda-dropping algorithm starts by sinking the functions of the program, and proceeds to parameter-drop any formal parameters made redundant by the newly created scope. Function sinking is accomplished by expressing the program as a graph, and transforming it into a tree. A flow graph provides the information necessary for parameter dropping.

The lambda-dropping algorithm is applicable to any program, and generates maximally nested block structure in a first-order program. The parameter dropping step removes all redundant parameters in a first-order program. A control-flow analysis could be used to improve the algorithm for higher-order programs. The lambda-dropping algorithm provides

an inverse for Johnsson's lambda-lifting algorithm.

Lambda-dropping a program may significantly reduce parameter passing, perhaps yielding a faster program. Lambda-dropping a program will reveal information about scope and parameter passing. An implicit context is created for computations when suitable, and employed to reduce the number of explicitly passed arguments when possible. Along with lambda-lifting, lambda-dropping has its place on the programmer's toolbench. Both transformations are generally applicable to programs written in a block-structured language, and both reveal information about the structure of the program.

Chapter 6

Lambda-Lifting vs. Lambda-Dropping

6.1 Introduction

This chapter evaluates lambda-lifting and lambda-dropping. Not only in terms of properties and the implementation of the transformations themselves but also covering applications of the transformations to other programs. Sections 6.2 and 6.3 describe formal properties of the transformations. In Section 6.4 the abstract model of Section 2.3 is applied to evaluate the effect of the transformations on the computational cost of a few example programs. This is backed by a practical study in many different languages, all of which focus on the possible disadvantages of lambda-lifting, in Section 6.5. Algorithmical and implementational aspects are covered in sections 6.6 and 6.7. Section 6.8 concludes by surveying work related to lambda-dropping.

6.2 Global Equations and Block Structure

This section discusses the correctness of the lambda-lifting and lambda-dropping transformations. A formal proof would be in order here but this was not possible within the given time constraints. Instead, we informally sketch the outlines of a proof by relating each algorithm to the basic, enabling transformations. A proof of correctness for these transformations may need to take into account that they modify the storage properties of the program that is transformed. Sullivan and Wand present a proof of correctness for the incremental lambda-lifting process used in the optimizing Scheme compiler TwoBit [24]. TwoBit uses lambda-lifting to improve the register allocation algorithm [10].

Throughout this section we present a number of program transformations. We use a syntax similar to that of rewrite systems [39]. A square bracket after an expression is used to denote the free variables of this expression. A lambda-bound variable must appear within the brackets of an expression for it to occur in the expression. We generalize this to specifying free variables appearing in a context, such as an application, in the brackes. We

use a double arrow "⇒" to indicate a transformation step. An expression within a bracket that is different after the transformation indicates that all expressions that were included in the bracket expression before the transformation should be changed accordingly after the transformation. Relying on our assumption of variable hygiene we can thus specify ordinary β-reduction within this framework (to illustrate the syntax):

$$(\lambda x.E[x])\ E' \ = \ E[E']$$

6.2.1 Correctness of lambda-lifting

Lambda-lifting is performed by iterating two simple, well-known transformations. Parameter lifting binds free variables as formal parameters by eta-expansion. Block floating moves procedure definitions towards the global level by outwards let-floating. These steps were presented in Section 3.2. A lambda-lifting algorithm could work by iterating these steps. Parameter lifting would be applied until all function definition could have been block floated to the global level. There are other, faster approaches.

The approach taken by Johnsson (the approach that was the main subject of Chapter 3) factorizes the two transformations into two stages. Rather than applying the transformations directly a number of macro steps are used. Each macro step is equivalent to one or more ordinary steps. Consider first the parameter lifting phase presented in Figure 3.1. The correctness of the first program transformation step is trivial by the definition of let blocks and will not be considered further. The second step is essentially applied to each letrec block. In a single macro step the function definitions are parameter lifted. This is equivalent to multiple applications of the parameter lifting transformation to a single function. The parameter lifting transformation is shown in a simplified[1] context in Figure 6.1. The macro step performed by the lambda-lifter is equivalent to several applications of this transformation. If this transformation is correct the parameter-lifting stage of the lambda-lifting algorithm will thus be correct. The correctness of this transformation step rests on the correctness of eta-expansion:

$$E \ = \ \lambda x.E\,x,\ x \notin \mathsf{FreeVar}(E)$$

We proceed to consider the block floating phase presented in Figure 3.2. Only the first program transformation step is of interest here. The correctness of the second transformation step follows from Landin's correspondence principle [32]:

$$(\mathrm{let}\, x = e_1\ \mathrm{in}\, e_2) \ = \ (\lambda x.e_2)\, e_1$$

The second step is simultaneously applied to all procedure definitions of the program. In a single macro step all function definitions are made global. This is equivalent to multiple applications of the block floating transformation to each function. The block floating transformation is shown in a simplified context in Figure 6.2. The macro step

[1] The let block could have contained any number of procedure definitions, for example.

$$\begin{array}{l}\text{let } f = \lambda(x_1, \ldots, x_n).\text{let } g = \lambda(y_1, \ldots, y_m).E_g[f, g] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{in } E_f[f, g] \\ \text{in } \ldots\end{array}$$

$$\Downarrow$$

$$\begin{array}{l}\text{let } f = \lambda(x_1, \ldots, x_n).\text{let } g = \lambda(x_i, \ldots, x_j).\lambda(y_1, \ldots, y_m).E_g[f, (g\ (x_i, \ldots, x_j))] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{in } E_f[f, (g\ (x_i, \ldots, x_j))] \\ \text{in } \ldots\end{array}$$

Figure 6.1: Parameter lifting transformation: eta expansion

$$\begin{array}{l}\text{let } f = \lambda(x_1, \ldots, x_m).\text{let } g = \lambda(y_1, \ldots, y_n).E_g[f, g] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{in } E_f[f, g] \\ \text{in } \ldots\end{array}$$

$$\Downarrow$$

$$\begin{array}{l}\text{let } f = \lambda(x_1, \ldots, x_m).E_f[f, g] \\ \quad\ \ g = \lambda(y_1, \ldots, y_n).E_g[f, g] \\ \text{in } \ldots\end{array}$$

Figure 6.2: Block floating transformation: let floating

performed by the lambda-lifter is equivalent to several applications of this transformation. After parameter lifting the only free variables of each function are the names of other functions, as required by the transformation. The correctness of the floating stage of the lambda-lifting algorithm thus rests on the correctness of the block-floating step. This transformation step is similar to let-floating of independent let bindings:

$$\lambda x.(\text{let } y = E_y \text{ in } E) \ = \ (\text{let } y = E_y \text{ in } \lambda x.E),\ x \notin \mathsf{FreeVar}(E_y)$$

The correctness of the lambda-lifting algorithm follows from being able to determine precisely the set of free variables that each function should be parameter lifted with and that block floating is trivial in a completely parameter lifted program.

6.2.2 Correctness of lambda-dropping

As was the case for lambda-lifting, lambda-dropping is performed by iterating two simple, well-known transformations. Parameter dropping removes redundant formal parameters by eta-reduction. Block sinking makes procedures definitions local to other procedures by inwards let-floating. These steps were presented in Section 5.2. A lambda-dropping algorithm could work by iterating these steps. Block sinking would be applied until all redundant formal parameters could have been parameter dropped. There are other, faster approaches. Similarly to lambda-lifting, the approach of Chapter 5 factorizes the two transformations into two stages. Each stage consists of a number of macro steps.

Consider first the block sinking phase presented in Figure 5.2. It localizes all procedure definitions in a single macro step. Using a graphical representation of the program, each procedure definition is first localized so that it is visible only to the procedures that need to refer to it. The graphical representation is then simplified until it resembles a textual representation of a program all the while extending the scope of the definition of each procedure. The block structure of the resulting program is thus a textual approximation of the graphical representation of a maximally localized block structure. Sections 5.3.1, 5.3.2 and 5.3.3 seek to justify that this is the closest possible approximation. The block sinking step is illustrated in a simplified context in Figure 6.3. Iterating the block sinking step would approximate maximally localized block structure from the other side. It would start with the textual representation of a program without block structure and then approximate maximally localized block structure by applying the block sinking step. The result of the macro step can thus be constructed by using a number of block sinking steps. The correctness of the block sinking phase thus relies on the correctness of the basic block sinking step. This transformation step is once again similar to let-floating of independent let bindings:

$$\begin{pmatrix} \text{let } f = \lambda x.E_f \\ \quad\; v = E_v \\ \text{in } \; E \end{pmatrix} \;=\; \begin{pmatrix} \text{let } f = \lambda x.\text{let } v = E_v \text{ in } E_f \\ \text{in } \; E \end{pmatrix} \; v \notin \mathsf{FreeVar}(E)$$

We proceed to the parameter dropping phase presented in Figure 5.2. It removes redundant formal parameters in a single phase consisting of several passes of the program. The last step of the parameter dropping phase removes empty applications. This is eta-reduction of an application to an empty value so the correctness is considered trivial. A flow graph is constructed that describes the flow of data through the program. This graph is simplified so that it can be used to determine the origin of each formal parameter of each procedure. A formal parameter that has a unique origin is always bound to the same expression. If this expression is a formal parameter it is not incorrect to substitute this parameter throughout the body of the procedure. Traversing the program determines whether the scope of such potentially droppable formal parameters extends to the procedure that always receives them as an argument. The basic parameter dropping step is shown in a simplified context in Figure 6.4. This step removes the formal parameter of a procedure if it can be determined that some other formal parameter with scope extending

$$
\begin{array}{l}
\text{let } f = \lambda(x_1, \ldots, x_m).E_f[f, g, h] \\
\quad\; g = \lambda(y_1, \ldots, y_n).E_g[f, g, h] \\
\quad\; h = \lambda(z_1, \ldots, z_\ell).E_h[f, g, h] \\
\quad\; p = \lambda(a_1, \ldots, a_q).E_p[f] \\
\text{in } \ldots
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let } f = \lambda(x_1, \ldots, x_m).\text{let } g = \lambda(y_1, \ldots, y_n).E_g[f, g, h] \\
\qquad\qquad\qquad\qquad\qquad\quad\; h = \lambda(z_1, \ldots, z_\ell).E_h[f, g, h] \\
\qquad\qquad\qquad\qquad\quad \text{in } E_f[f, g, h] \\
\quad\; p = \lambda(a_1, \ldots, a_q).E_p[f] \\
\text{in } \ldots
\end{array}
$$

Figure 6.3: Simplified block sinking transformation: let sinking

$$
\begin{array}{l}
\text{let } f = \lambda(x_1, \ldots, x_i, \ldots, x_n).\text{let } g = \lambda(y_1, \ldots, y_j, \ldots, y_m).E_g[B] \\
\qquad\qquad\qquad\qquad\qquad\quad\; h = \lambda(z_1, \ldots, z_k, \ldots, z_\ell).E_h[B] \\
\qquad\qquad\qquad\qquad\quad \text{in } E_f[B] \\
\text{in } \ldots \\
\qquad \text{where } B \;=\; \left[\begin{array}{l}
(f\ (e_1, \ldots, e_i \in \{x_j, y_i, z_k\}, \ldots, e_n)) \\
(g\ (e'_1, \ldots, e'_j \in \{x_j, y_i, z_k\}, \ldots, e'_m)) \\
(h\ (e''_1, \ldots, e''_k \in \{x_j, y_i, z_k\}, \ldots, e''_\ell))
\end{array} \right]
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{l}
\text{let } f = \lambda(x_1, \ldots, x_i, \ldots, x_n).\text{let } g = \lambda(y_1, \ldots, y_{j-1}, y_{j+1}, \ldots, y_m).E_g[B'] \\
\qquad\qquad\qquad\qquad\qquad\quad\; h = \lambda(z_1, \ldots, z_{k-1}, z_{k+1}, \ldots, z_\ell).E_h[B'] \\
\qquad\qquad\qquad\qquad\quad \text{in } E_f[B'] \\
\text{in } \ldots \\
\qquad\qquad \text{where } B' \;=\; \left[\begin{array}{l}
(f\ (e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n)) \\
(g\ (e'_1, \ldots, e'_{i-1}, e'_{i+1}, \ldots, e'_m)) \\
(h\ (e''_1, \ldots, e''_{i-1}, e''_{i+1}, \ldots, e''_\ell))
\end{array} \right]
\end{array}
$$

Figure 6.4: Simplified parameter dropping transformation: eta reduction

into the procedure is always (perhaps indirectly) passed as the argument corresponding to this formal parameter. During the parameter dropping stage the preceeding flow analysis ensures that this is always the case. The parameter dropping stage will thus only drop parameters that could have been dropped by the parameter dropping step. The correctness of the parameter dropping phase thus relies on the correctness of the basic parameter dropping step. This transformation step is essentially eta-reduction:

$$\lambda x.E\, x \;=\; E,\; x \notin \mathsf{FreeVar}(E)$$

The correctness of the lambda-dropping algorithm follows from the approximating nature of the graph transformations used to construct block structure and from the invariants ensured by the flow analysis.

6.3 Assessment of Lambda-Dropping

This section presents a short discussion of two important properties of lambda-lifting and lambda-dropping. It is of interest to know whether each of these transformations are idempotent. Idempotence is a desirable property ensuring that independently of the application, the transformation need only be done once. Lambda-dropping was designed to provide an inverse transformation for lambda-lifting but this only holds under certain restrictions. We would have preferred to present formal proofs of the properties presented in this section but this turned out to be unrealistic to accomplish within the alloted time.

Within this section we consider syntactic equivalence of programs modulo the ordering of mutually recursive declarations and modulo naming of variables.

6.3.1 Idempotence

Lambda-lifting is trivially idempotent. Lambda-lifting a program creates a set of globally visible recursive equations with no free variables. There remain no parameters that can be lifted and no local procedures can be made global by block floating.

Property 2 *Lambda-lifting is idempotent.*

Lambda-dropping as presented in Chapter 5 is not idempotent. This is due to the last step of parameter dropping as described in Figure 5.2. The last step, informally called "dethunkification," removes thunks that encapsulate procedures if these thunks are redundant. A thunk is considered redundant if it the function bound to the thunk is always applied to an empty argument to force evaluation of the thunk. Removing the thunk reveals a new set of parameters that are no longer considered to be the formal parameters of a high-order function. It is thus possible that they can be parameter dropped. In Section 5.4 the lambda-dropping algorithm was demonstrated on a DFA. Figure 6.5 shows the lambda-dropped DFA program of Figure 4.1 after a second application of the lambda-dropping algorithm. Iterating the lambda-dropping algorithm in this way results in a new

```
(define (r a b c d die? xs)
  (letrec
      ([f (lambda (reject xs)
            (letrec
                ([g (lambda (xs)
                      (letrec
                          ([h (lambda (xs)
                                (if (empty? xs)
                                    (reject '())
                                    (case (car xs)
                                      [(gamma)
                                       (c (f reject (cdr xs)))]
                                      [(delta)
                                       (d (g (cdr xs)))]
                                      [else
                                       (reject (car xs))])))])
                            (if (empty? xs)
                                xs
                                (case (car xs)
                                  [(beta) (b (h (cdr xs)))]
                                  [else (reject (car xs))])))))]
                 [empty?
                  (lambda (s)
                    (or (null? s) (eq? (car s) '$)))])
              (if (empty? xs)
                  (reject '())
                  (case (car xs)
                    [(alpha) (a (g (cdr xs)))]
                    [else (reject (car xs))])))))]
       [fresh0
        (lambda (x) x)]
       [err
        (lambda (x)
          (if (null? x)
              (error 'r "unexpected end of stream~%")
              (error 'r "unexpected token ~a~%" x)))])
    (f (if die? err (lambda (x) '())) xs)))
```

Figure 6.5: The DFA program after iterated lambda-dropping

program because a curried function was simplified in the previous iteration. The number of simplifications is limited by the depth of the currying. Iterating the lambda-dropping algorithm will eventually reach a fix point.

Property 3 *The process of applying lambda-dropping iteratively reaches a fixed point.*

The problem is that the lambda-dropping algorithm considers any curried function to be a higher-order function even if it is always fully applied. Extending the lambda-dropping with a simplified control-flow analysis would allow it to properly process curried functions. Parameter dropping would be extended to include not only the immediate parameters but also the curried parameters.

6.3.2 Inverseness

Lambda-lifting maps block structured programs to recursive equations. Lambda-lifting is idempotent so we can never provide a unique inverse for lambda-lifting. Two different block-structured programs can be mapped to the same set of recursive equations. Lambda-dropping a set of recursive equations generates a maximally block-structured program. The program is in Block-Structured Normal-Form and has at least as much block structure as any block-structured version would have had before lambda-lifting. If applied several times lambda-dropping becomes idempotent, so we face similar difficulties here.

We consider the inverseness properties on a restricted set of programs. If restricted to be transformations between recursive equations and programs in BSNF with maximal block structure then lambda-lifting and lambda-dropping are inverse mappings. Lambda-lifting and lambda-dropping can themselves be used to express a program as recursive equations or in BSNF with maximal block structure.

Property 4 *Lambda-lifting is the inverse of lambda-dropping on programs that have been completely lambda-lifted.*

A lambda-lifted program consists of recursive equations. Lambda-dropping the program introduces block structure by block sinking of functions into other procedures and creates local free variables by parameter dropping formal parameters into free variables. Parameter lifting will remove the free variables by making them formal parameters. Block floating will move all procedures from their local definition site to once again become global equations.

Property 5 *Lambda-dropping is the inverse of lambda-lifting on programs that have been completely lambda-dropped.*

A lambda-dropped program has maximally localized block structure and there are no redundant formal parameters. Lambda-lifting the program first parameter-lifts all free variables to become formal parameters. Then each procedure is made global by block floating. Block sinking localizes procedures maximally using block structure, restoring

block structure. Parameter dropping drops redundant formal parameters making them free variables.

The previous section discussed using a simplified control flow analysis to improve the idempotence properties of lambda-dropping. To retain the inverseness properties when extending lambda-dropping with a control-flow analysis we must distinguish between formal parameters that are bound to a known procedure and other formal parameters. Formal parameters that are bound to a procedure should not be parameter dropped since this could imply replacing the formal parameter with the name of the procedure throughout the body of the procedure that is being parameter dropped. Variables that are procedure names are not considered free variables during lambda-lifting and will not be parameter lifted. The formal parameter that was dropped will thus not be lifted back into place.

6.4 Applying the Abstract Model

In Section 2.3 we presented an abstract model of computation costs. We now use this model to evaluate the computational cost of lambda-lifted programs compared to the computational cost of lambda-dropped programs using a few simple examples.

6.4.1 Evaluating lambda-dropping

We evaluate lambda-dropping by first evaluating the computational cost of a program written as recursive-equations and then evaluating the computational cost of the equivalent lambda-dropped program. The effect of lambda-dropping will be measured on the functions append and map. Neither of these can be transformed from their recursive equations version to a version that is programmed with block structure using plain lambda-dropping since they each consist of a single function only. However, lambda-dropping extended to optimize recursive functions, as described in Section 6.5.4, will produce the well-known block structured versions of these functions. In Section 2.3 we used a program with deep block structure to illustrate the use of the abstract model with different closure representation. For completeness we also use the abstract model to evaluate the effect of lambda-lifting this program.

The append function

The append function takes two lists as its arguments. If the first argument is the empty list the second argument is returned. Otherwise the function is called recursively on the tail of the first argument and the head is concatenated onto the return value. Append runs in linear time in the size of the first argument, as we would expect. The recursive-equation version of append is displayed in Figure 6.6 in the style of Section 2.3.

By enclosing the recursive-equation version of append in an enclosing definition and then lambda-dropping the block-structured version of append is constructed (this process is described in Section 6.5.4). Here we let the second list argument reside in the implicit

```
(define lifted-append
  (lambda (xs ys)
    (if (null? xs)
        ys
        (cons (car xs) (lifted-append (cdr xs) ys)))))
```

```
(define lifted-append
  (lambda{} (x_s y_s)
    (let* ([v_1 (null? x_s)])                          op,local,save
      (if v_1                                          stmt,local
          (return y_s)                                 stmt,local
          (let* ([v_2 (car x_s)]                       cxr,local,save
                 [v_3 (cdr x_s)]                       cxr,local,save
                 [v_4 (call lifted-append v_3 y_s)]    call(2),global,save
                 [v_5 (cons v_2 v_4)])                 cons,2×local,save
            (return v_5))))))                          stmt,local
```

Figure 6.6: Append programmed as a recursive equation

context of the loop, as shown in Figure 6.7. Each recursive invocation will thus neither need to pass it as an explicit argument nor need to refer to it in the implicit context. To simplify the notation we do not annotate free variables in closures with the relative nesting level. Since there is only one local block the nesting level is always zero. In the block structured version of append the second argument is only used in the base case.

Counting the instruction costs and analyzing the program shows that the computational costs can be expressed as a function of the length of the first list argument, which is hardly surprising. The equations are shown as an example in Figure 6.8.

The map function

The map function takes a function and a list as its arguments. If the list is empty the empty list is returned. Otherwise the result of applying the function to the head of the list is concatenated with the result of calling recursively of the rest of the list. Map runs in linear time in the size of its second argument. We do not count the time consumed by the body of the function argument, only the instructions used to invoke the function. The recursive-equation version of map is displayed in Figure 6.9.

The block-structured version of append can be constructed in the same way we constructed the block-structured version of map. The function that is applied to the elements now resides in the implicit context of the inner loop, as shown in Figure 6.7. The function argument is used in every recursive invocation of the map function. We refer to it through

```
(define dropped-append
  (lambda (xs ys)
    (letrec ([a (lambda (x)
                  (if (null? x)
                      ys
                      (cons (car x) (a (cdr x)))))])
      (a xs))))
```

```
(define dropped-append
  (lambda{} (x_s y_s)
    (let* ([v_1 (closure ['(), y_s])]                    clos(2),save
           [v_2 (lambda{v_1 ← [a, y_s]} (x)              makeLambda,save
                  (let* ([w_1 (null? x)])                op,local,save
                    (if w_1                              stmt,local
                        (return y_s)                     stmt,free(1)
                        (let* ([w_2 (car x)]             cxr,local,save
                               [w_3 (cdr x)]             cxr,local,save
                               [w_4 (call a w_3)]        call(1),free(1),
                                                         local,save
                               [w_5 (cons w_2 w_4)])     op,2×local,save
                          (return w_5))))]               stmt,local
           [() (vector-set! v_1 0 v_2)])                 vset,2×local
      (call v_2 x_s))))                                  call(1),local
```

Figure 6.7: Append programmed using block structure

```
liftedAppend 0 = 2*stmt + op + 3*local + save
liftedAppend n = 2*stmt + op + 2*cxr + cons + call(2) + 7*local
                 + global + 5*save
droppedAppend n = vset + call(1) + clos(2) + makeLambda + 3*local
                  + 2*save + dA(n)
dA 0 = 2*stmt + op + 2*local + free(1) + save
dA n = 2*stmt + op + 2*cxr + cons + call(1) + 7*local + free(1)
       + 5*save + dA(n-1)
```

Figure 6.8: The costs of the append functions expressed as equations

```
(define lifted-map
  (lambda (f xs)
    (if (null? xs)
        '()
        (cons (f (car xs)) (lifted-map f (cdr xs))))))
```

```
(define lifted-map
  (lambda{} (f x_s)
    (let* ([v_1 (null? x_s)])                          op,local,save
      (if v_1                                          stmt,local
          (return nil)                                 stmt
          (let* ([v_2 (car x_s)]                       op,local,save
                 [v_3 (call f v_2)]                    call(1),local,save
                 [v_4 (cdr x_s)]                       op,local,save
                 [v_5 (call lifted-map f v_4)]         call(2),global,save
                 [v_6 (cons v_3 v_5)])                 op,2×local,save
            (return v_6))))))                          stmt,local
```

Figure 6.9: Map programmed as a recursive equation

```
(define dropped-map
  (lambda (f xs)
    (letrec ([m (lambda (x)
                  (if (null? x)
                      '()
                      (cons (f (car x)) (m (cdr x))))))])
      (m xs))))
```

```
(define dropped-map
  (lambda{} (f x_s)
    (let* ([v_1 (closure ['(), f])]                          clos(2),save
           [v_2 (lambda{v_1 ← [m, f]} (x)                    makeLambda
                  (let* ([w_1 (null? x)])                    op,local,save
                    (if w_1                                  stmt,local
                        (return nil)                         stmt
                        (let* ([w_2 (car x)]                 op,local,save
                               [w_3 (call f x)]              call(1),free(1),save
                               [w_4 (cdr x)]                 op,local,save
                               [w_5 (call m w_4)]            call(1),local,save
                               [w_6 (cons w_3 w_5)])         op,2×local,save
                          (return w_6)))))]                  stmt,local
           [() (vector-set! v_1 0 v_2)])                     vset,2×local
      (call v_2 x_s))))                                      call(1),local
```

Figure 6.10: Map programmed using block structure

the implicit context in every invocation rather than explicitly passing it as an argument in every invocation. The computational costs can be specified in similarly to the append function.

The deeply nested function

In Section 2.3 we used the abstract model to evaluate different closure representations using the deeply block-structured summation program of Figure 6.11. Lambda-lifting this program results in a program where each procedure carries the same number of parameters holding the values of α, β, and γ. On the other hand the final call to the addition function can be made with only formal parameters as arguments.

```
(let ([f (lambda (x1 x2 x3) (g x3 x1 x2))]
      [g (lambda (x3 y1 y2) (h x3 y2 y1))]
      [h (lambda (x3 y2 z1) (+ z1 y2 x3))])
  (f alpha beta gamma))
```

```
(let ([a1 (λ{} (x1 x2 x3)                    makeLambda,save
           (tail + z1 y2 z3))]               tail(3),global
      [b1 (λ{} (x3 y1 y2)                    makeLambda,save
           (tail c1 x3 y2 y1))]              tail(3),local
      [c1 (λ{} (x3 y2 z1)                    makeLambda,save
           (tail b1 x3 x1 x2))])             tail(3),local
  (tail a1 α β γ))                           tail(0,3),local
```

Figure 6.11: Lambda-lifted version of the program of Figure 2.6

Function	time	space	read	write
lifted-append(x, y)	$31.5\|x\| + 5.5$	$7\|x\|$	$16\|x\| + 3$	$12\|x\| + 1$
dropped-append(x, y)	$27\|x\| + 26$	$6\|x\| + 7$	$14\|x\| + 12$	$10\|x\| + 13$
lifted-map(f, x)	$42\|x\| + 4.5$	$11\|x\|$	$20\|x\| + 2$	$18\|x\| + 1$
dropped-map(f, x)	$38\|x\| + 20$	$10\|x\| + 4$	$18\|x\| + 9$	$16\|x\| + 8$

Figure 6.12: Costs for append and map, using flat closures

Comparisons

The computational costs for both the append function and the map function are linear in the size of the list argument that is being traversed. Figure 6.12 displays the computation costs of both functions. We use flat closures. Using deep closures would yield the same result since there is but a single level of block structure in the block-structured version of both functions. In both cases the recursive-equation version of the function is less expensive for smaller problem sizes than the block-structured version. When the problem size increases the block-structured version becomes less expensive than the recursive-equation version. The breaking point is at around list length 5 for append and at list length 4 for map.

Figure 6.13 shows the computational costs for map and append calculated using the stack based computational model of Section 2.3.4. The results are similar to those achieved using flat closures, although the change in cost from recursive-equation to block-structure is slightly smaller.

Finally we turn to the computation costs the lambda-lifted version of the deeply block-structured summation program shown in Figure 6.11. The costs are constant here and are

Function	time	space	read	write
lifted-append(x, y)	$26.5\|x\| + 5.5$	$7\|x\|$	$14\|x\| + 3$	$10\|x\| + 1$
dropped-append(x, y)	$26\|x\| + 25.5$	$6\|x\| + 5$	$15\|x\| + 13$	$9\|x\| + 10$
lifted-map(f, x)	$34.5\|x\| + 4.5$	$11\|x\|$	$17\|x\| + 2$	$15\|x\| + 1$
dropped-map(f, x)	$36\|x\| + 17.5$	$10\|x\| + 4$	$20\|x\| + 8$	$14\|x\| + 7$

Figure 6.13: Costs for append and map, using the stack-based machine

easily determined (in any of the computational models we have defined):

time	space	read	write
40	12	23	15

The lambda-lifted program explicitly passes every variable saving the cost of closure creation. And since each procedure is called only once, there is no extra overhead associated with passing the same variables several times.a

6.4.2 Live variables

In Section 2.3.5 we described alternative ways of expressing computation costs (alternative to those computed in the previous section). This included the number of live variables and the interdependencies of values. The formal parameters of a procedure are available when the computation within the procedure takes place. Parameter dropping can place such formal parameters in the closure of the procedure, where they will be readily available as well. It would thus seem as if estimating computation costs by building a Data Dependence Graph will not reveal anything interesting about lambda-dropping. The DDG's for the recursive call of both functions turn out to be identical for our purposes.

Taking a closer look at the number live variables throughout the recursive call reveals very little in the case of the map function. But as can be seen from comparing Figures 6.14 and 6.15, lambda-dropping append reduces the maximum number of simultaneous live variables by one, from four to three. If only three registers are available to the compiler when generating code (not unlikely in the presence of a runtime system) this can make a difference in terms of cache load.

In general, when a formal parameter that is used in some cases only is parameter dropped the maximum number of live variables may decrease. If the procedure is recursive and it is invoked many times using the same closure, not passing the formal parameter in every invocation is bound to make the computation less expensive.

```
(define flat-append
  (lambda{} (x_s y_s)
    (let* ([v_1 (null? x_s)])                          {x_s, y_s, f}
      (if v_1                                          {v_1, x_s, y_s, f}
          (return y_s)                                 (base case only)
          (let* ([v_2 (car x_s)]                       {x_s, y_s, f}
                 [v_3 (cdr x_s)]                       {v_2, x_s, y_s, f}
                 [v_4 (call flat-append v_3 y_s)]      {v_2, v_3, y_s, f}
                 [v_5 (cons v_2 v_4)])                 {v_2, v_4}
            (return v_5))))))                          {v_5}
```

Figure 6.14: Live variables in recursive-equation version of append

```
(define block-append
  (lambda{} (x_s y_s)
    (let* ([v_1 (closure ['(), y_s])]                  (initialization)
           [v_2 (lambda{v_1 ← [a, y_s]} (x)            (initialization)
                  (let* ([w_1 (null? x)])              {x, a}
                    (if w_1                            {w_1, x, a}
                        (return y_s)                   (base case only)
                        (let* ([w_2 (car x)]           {x, a}
                               [w_3 (cdr x)]           {w_2, x, a}
                               [w_4 (call a w_3)]      {w_2, w_3, a}
                               [w_5 (cons w_2 w_4)])   {w_2, w_4}
                          (return w_5))))))]           {w_5}
           [() (vector-set! v_1 0 v_2)])               (initialization)
      (call v_2 x_s))))                                (initialization)
```

Figure 6.15: Live variables in block-structured version of append

6.5 An Empirical Study

This section performs a brief, empirical study of the effects of lambda-lifting on a limited set of recursive programs. We demonstrate that in some cases lambda-lifting can deplete the efficiency of a program. This holds for many implementations, but a wide range of results are obtained and discussed.

6.5.1 Hypothesis

It is our hypothesis that certain recursive programs are more efficient as block-structured programs than as recursive equations. This implies that lambda-lifting can increase the cost of computation and that lambda-dropping can make programs more efficient. We aim to show that using most standard implementations of programming languages, the block-structured version of certain programs have better running times than the equivalent recursive-equation programs. We want for our experiments to illuminate any advantages of block-structured programming; not only in a wide range of functional languages, but also in imperative and object-oriented languages.

We do not try to show that lambda-dropping can be used as a general optimization technique for block-structured languages. A non-recursive function will most likely not benefit from lambda-dropping. The abstract model was used in Section 6.4 to evaluate the computational cost of two recursive functions and a non-recursive function in a recursive-equation version and in a block-structured version. In all three cases it became apparent that lambda-dropping a non-recursive function is probably not a good idea.

Meijer's unpublished note "Down with Lambda-Lifting" (April 1992) supports our hypothesis. Here, Meijer argues that lambda-lifting can result in suboptimal programs. Appel, on the other hand, promotes passing variables as parameters rather than through a closure when performing closure conversion [2]. This seems to indicate that lambda-lifting is good in some situations.

6.5.2 Experiments

To investigate our hypothesis we have implemented a set of simple test programs in a wide range of languages. These serve offer a comparison of any advantages in speed gained from using block structure across different implementations. There is also a single more complicated test program that could be implemented in a few languages only.

The simple test programs were implemented in a number of languages:

Functional Haskell programs are compiled with the Glasgow Haskell Compiler (GHC) in versions 2.01 and 0.29, and Gofer. ML programs are compiled with Moscow ML (MosML) and Standard ML of New Jersey (SML/NJ). Scheme programs are compiled with Chez Scheme and Scheme48 (the latter to byte-code), and interpreted are by SCM.

Imperative Pascal programs are compiled by the Borland Delphi system. Gnu C programs are (not surprisingly) compiled with GCC.

Object-oriented Java 1.1.1 programs are compiled with the Sun javac compiler included in the standard Java Development Kit (version 1.1.1).

Our primary interest is functional programming languages so much care was taken to include as many relevant implementations as possible. Pascal, Gnu C, and Java were included for completeness.

The test programs

A number of simple test programs were compiled and evaluated using the different implementations. These programs are:

Append #1 Computation of the sum of the numbers from 1 to n, representing numbers by lists. The length of a list is the value of the number and *append* is the addition operator.

Append #2 Test of *append*'ing a very long list onto the empty list.

RGB function A traversal of a binary tree with leaves that are colored either **R**ed, **G**reen or **B**lue. The RGB function is shown in Figure 6.16, using block structure. The value at a branch node depends on the result of a pre-processing function applied to the node. The value of the pre-processing function depends on a traversal of each subtree of the node. Thus, the number of recursive calls on a given path from the root to a leaf are n^2. In the lambda-lifted version, the functions needed to process the leaves (one for each color) and two functions for combining results from sub-trees (one for the traversal function and one for the pre-processing function) are passed as arguments. The current node and a "base value" that is used for the computation are passed as arguments as well.

The more complicated test program was compiled and tested with a limited set of implementations. Higher-order functions are needed for this program, so it was most practical to only use the functional languages. Scheme was excluded because of lack of time, as was testing in Gofer, leaving only MosML, SML/NJ, and Haskell. The program is:

Fold function A fold function across the grammar of a small programming language. The fold function is shown in Figure 6.17. The first eight arguments are the functions that are to be applied to each basic syntactic form of the grammar (also shown in the figure). The two last arguments are the initial value and the program to fold over. The fold function is used for two program different tasks, namely that of determining the lexical size of the program (shown in Figure 6.18) and performing a single-pass CPS transformation of the program (can be found in Appendix A).

These programs offer a selection of different opportunities for lambda-dropping.

```
-- Data type
data Rgb  = Red Integer | Green Integer | Blue Integer
data Node = Branch Node Node | Leaf Rgb

-- Lambda-dropped version of the RGB function
traverse tree combine1 combine2 red green blue base =
   let trav (Leaf (Red n)) base = red n base
       trav (Leaf (Green n)) base = green n base
       trav (Leaf (Blue n)) base = blue n base
       trav (Branch left right) base =
          let newbase = pp (Branch left right)
              leftval = trav left newbase
              rightval = trav right newbase
          in  combine1 leftval rightval
       pp (Branch left right) =
          let leftval = trav left 0
              rightval = trav right 0
          in  combine2 leftval rightval
       in trav tree base
```

Figure 6.16: The RGB benchmark program

```
-- Data type
data Block = Blk ([Define], Exp)
data Define = Def (Variable, [Variable], Exp)
data Exp = App (Exp, [Exp]) | If (Exp, Exp, Exp)
         | Lam ([Variable], Exp) | Let Block
         | Var Variable | Con Constant

-- Lambda-dropped version of the fold function
foldProgram pBlk pDef pApp pIf pLam pLet pVar pCon init prog =

    let foldBlock (Blk (defs, e)) =
           pBlk (map foldDefine defs, foldExp e)

        foldDefine (Def (v, vs, e)) = pDef (v, vs, foldExp e)

        foldExp (App (e, es)) =
           pApp ( foldExp e, map foldExp es )
        foldExp (If (e1, e2, e3)) =
           pIf ( foldExp e1, foldExp e2, foldExp e3 )
        foldExp (Lam (vs,e)) = pLam (vs, foldExp e)
        foldExp (Let b) = pLet (foldBlock b)
        foldExp (Var v) = pVar v
        foldExp (Con c) = pCon c

    in  init (foldBlock prog)
```

Figure 6.17: Definition of the fold function

```
-- The size of a program (in tokens)
lexSizeOfBlock (defsResLs, bodyRes) = (listSum defsResLs) + bodyRes
lexSizeOfDefine (v, vs, bodyRes) = 2 + (length vs) + bodyRes
lexSizeOfApply (appRes, argResLs) = appRes + (listSum argResLs)
lexSizeOfIf (e1Res, e2Res, e3Res) = 1 + e1Res + e2Res + e3Res
lexSizeOfLam (vs, bodyRes) = 1 + (length vs) + bodyRes
lexSizeOfLet (b) = 1 + b
lexSizeOfVar v = 1
lexSizeOfCon c = 1

lexSizeOf = foldProgram lexSizeOfBlock lexSizeOfDefine lexSizeOfApply
               lexSizeOfIf lexSizeOfLam lexSizeOfLet lexSizeOfVar
               lexSizeOfCon id
```

Figure 6.18: Lexical size of a program defined with fold

Testing

The speedup achieved by going from the recursive-equation (lambda-lifted) version of the program to the block-structured (lambda-dropped) version is calculated as:

$$S = \frac{T_L}{T_D}$$

Where T_L is the execution time for the lambda-lifted version, and T_D is the execution time for the lambda-dropped version. Thus, a value higher than 1.0 indicates a speedup, a value below 1.0 indicates a slowdown. A speedup of 2.0 indicates that the execution time is reduced to half of the original. Similarly, a "speedup" of 0.5 indicates that the execution time has doubled.

A number of things should be kept in mind when evaluating the results:

- The test programs are very limited in the sense that they only make use of a limited set of language features, and thus will probably not hold for real-life programs. They might, however, give an indication as to the results that could be achieved by lambda-dropping such programs.
- Benchmarking lazy languages like Haskell is inherently tricky, as it is necessary to force the runtime-system to perform all the desired computation. This extra work (i.e. a close inspection of the computed result) may have effected the results of the benchmarks.
- Both versions of GHC are included because they are very different. GHC 0.29 is in general more optimized, and it specifically has better code optimizer. GHC 2.01 is

less optimized, but is perhaps less optimized for certain programming styles, such a recursive equations vs. block-structured programs.

6.5.3 Results

In general a speedup was observed across most implementations. In a few cases a slowdown could be observed. A few of the speedups were rather extreme, probably reflecting how restricted the test programs are. Because of a few last-minute technical difficulties, the Gofer and Scheme48 results for the append test programs are not included. The results are listed with information on the machine and operating system that was used to obtain them, as well as an approximation of the problem size. The function α that is used to describe the problem size in Figure 6.22 reflects the size of the program that is being transformed. The program is generated semi-randomly with a fixed seed, and is the same across all implementations. The function α is approximately linear in size.

Figures 6.19 and 6.20 show the results of running the append test program. In general, moderate speedups were observed, if at all. The speedup for Moscow ML was surprisingly high, perhaps reflecting improved register allocation. The Java benchmark clearly shows that lambda-dropping is a bad idea for simple, recursive Java methods. The observed slowdown for SML/NJ does seem to indicate that closure creation is fairly expensive compared to the advantage of passing one rather than two parameters. A different, more detailed examination reveals that the lambda-lifted and the lambda-dropped versions run at approximately the same speeds for any given list size. These results are clearly contradictory, and reflect that the part of the test program that is used to construct and measure the problem can have a strong effect on the result. We leave the question of whether lambda-dropping degrades the performance of append in SML/NJ open (and hope to resolve it in a future, more detailed study).

Figure 6.21 shows the result for the RGB benchmark. Most of the programs run significantly faster after lambda-dropping. In particular for the SML/NJ implementation. We return to this later. In Moscow ML a speedup that was more moderate than for append but still significantly greater than for most other implementations could be observed. The result for the Java test program shows that the overhead of creating extra objects and referencing free variables through the outer class link has been overcome by the reduced number of parameters. The result for Chez Scheme is rather disappointing. Lambda-dropping is probably not beneficial for Chez Scheme, although a closer examination revealed that the lambda-dropped version consumes slightly less space. Chez Scheme is highly optimized and has an advanced storage manager. It is possible that these are optimized for lambda-lifted programs, thus resulting in a slowdown after lambda-dropping.

The result of running the RGB program in its lifted and dropped versions under SML/NJ were very surprising. A closer examination revealed that the running time is linearly dependent on the number of formal parameters. An adapted version of the RGB program that passed a single parameter only ran in approximately 3 seconds only. Adding a second parameter increased the running time to approximately 4 seconds. From then on there was a linear dependence in the running time on the number of formal parameters,

resulting in the running time of 53 seconds with 7 formal parameters. Adding more formal parameters increased the running time, but the speedup from the lambda-lifted version to the lambda-dropped version (approximately 7.5 times) remained the same. We credit the unexpected speedup to a highly optimized register allocator.

The result of the folding function used for CPS transformation can be seen for a very limited number of implementations in Figure 6.22. The SML/NJ test program shows that even though a fair amount of computation and data manipulation takes place, reducing the formal parameters still gives a very significant speedup. The results for Moscow ML are much more moderate, and probably more realistic than the previous results. The results for GHC are a bit disappointing, suggesting that the optimizer for some reason can do a better job with the lambda-lifted program than with the lambda-dropped program. Without optimization, a small speedup is observed.

Initial measurements on different architectures (Sparc, MIPS, and UltraSparc) appear to give results that are similar to those observed. On a processor such as the UltraSparc the results were more consistent. The advantage of lambda-dropping on the test programs was better on the average, and the results were less extreme for implementations such as SML/NJ. In particular, a consistent speedup was observed after lambda-dropping for programs compiled with GHC.

6.5.4 Further work

The results of the previous section indicate that lambda-dropping is applicable as an optimization technique in certain situations. This is highly dependent on the implementation. In some cases lambda-dropping may only be an advantage in extreme situations. It is probable that lambda-dropping might improve not only the running time of the resulting program but also the compilation time. In a program that has been lambda-dropped the scope of some functions may have been delimited to a more restricted set of declarations. This and the reduction of the number of parameters potentially simplifies type-checking. For example, initial experiments seem to indicate that lambda-dropping can greatly increase the compile time for SML/NJ programs.

To apply lambda-dropping as an optimization we must first consider why lambda-dropping can speed up programs. The evaluation of lambda-dropping by the abstract model revealed that less data had to be manipulated during recursive calls after parameter dropping. Fewer values are pushed on the stack when a function is invoked so potentially, less data is manipulated in every invocation of the function. If the number of simultaneous live variables is reduced, register allocation techniques can more easily be applied to prevent register spilling.

There are two prerequisites for applying lambda-dropping as an independent optimization in a compiler. To properly optimize recursive functions we must be able to transform a function like map from its recursive equation form to its block structured form. We must also be able to tell recursive functions with deep recursion from other functions.

To transform the recursive-equation version of map to the block-structured version, all that needs to be done is to enclose a renamed version of the original definition in a

Language		Measurement			Problem Size	Time (sec)		
Name	Type	Time	Arch	OS		T_L	T_D	$S_{L\to D}$
MosmML	ML	CPU	Pentium	Linux	$50 \times 100^2 = 500{,}000$	62.2	35.1	1.774
Delphi	Pascal	CPU	Pentium	Win95	$1000 \times 50^2 = 2{,}500{,}000$	17.0	15.6	1.089
GHC 2.01	Haskell	CPU	Pentium	Linux	$50 \times 100^2 = 500{,}000$	15.8	15.5	1.022
GHC 0.29	Haskell	CPU	Pentium	Linux	$50 \times 100^2 = 500{,}000$	15.8	15.5	1.022
Scm	Scheme	CPU	Pentium	Linux	$10 \times 100^2 = 100{,}000$	24.9	24.8	1.001
Chez v4	Scheme	CPU	SPARC	SunOS	$10 \times 100^2 = 100{,}000$	17.9	17.9	1.000
Chez v5	Scheme	CPU	MIPS	IRIX	$100 \times 100^2 = 1{,}000{,}000$	32.7	32.7	1.000
GCC	"C"	Real	Pentium	Linux	$100 \times 100^2 = 1{,}000{,}000$	19.3	19.3	1.000
SML/NJ	ML	CPU	Pentium	Linux	$100 \times 100^2 = 1{,}000{,}000$	11.5	13.2	0.875
Java	v. 1.1.1	Real	Pentium	Win95	$20 \times 100^2 = 200{,}000$	34.8	88.0	0.396

Figure 6.19: First append benchmark (many invocations on small lists)

Language		Measurement			Problem Size	Time (sec)		
Name	Type	Time	Arch	OS		T_L	T_D	$S_{L\to D}$
MosML	ML	CPU	Pentium	Linux	$20 \times 100{,}000 = 2{,}000{,}000$	55.2	28.2	1.957
Delphi	Pascal	CPU	Pentium	Win95	$1500 \times 30{,}000 = 4{,}500{,}000$	36.5	33.1	1.103
GHC 2.01	Haskell	CPU	Pentium	Linux	$50 \times 100{,}000 = 5{,}000{,}000$	30.9	30.5	1.012
GHC 0.29	Haskell	CPU	Pentium	Linux	$50 \times 100{,}000 = 5{,}000{,}000$	30.9	30.5	1.012
Scm	Scheme	CPU	Pentium	Linux	$3 \times 200{,}000 = 600{,}000$	14.3	14.2	1.002
GCC	"C"	Real	Pentium	Linux	$100 \times 200{,}000 = 20{,}000{,}000$	24.0	24.0	1.000
Chez v5	Scheme	CPU	MIPS	IRIX	$20 \times 300{,}000 = 6{,}000{,}000$	24.6	25.2	0.976
Chez v4	Scheme	CPU	SPARC	SunOS	$3 \times 200{,}000 = 600{,}000$	21.3	22.1	0.964
Java	v. 1.1.1	Real	Pentium	Win95	$3 \times 25{,}000 = 75{,}000$	32.0	35.7	0.895
SML/NJ	ML	CPU	Pentium	Linux	$100 \times 200{,}000 = 20{,}000{,}000$	14.6	18.3	0.795

Figure 6.20: Second append benchmark (few invocations on long lists).

Language		Measurement			Problem Size	Time (sec)		
Name	Type	Time	Arch	OS		T_L	T_D	$S_{L\to D}$
SML/NJ	ML	CPU	Pentium	Linux	$3 \times 2^{10} = 3072$	53.3	7.1	7.466
MosML	ML	CPU	Pentium	Linux	$3 \times 2^{10} = 3072$	47.6	33.5	1.420
GHC 0.29 -O	Haskell	CPU*	Pentium	Linux	$3 \times 2^{17} = 393216$	22.16	17.98	1.232
Scm	Scheme	CPU	Pentium	Linux	$3 \times 2^9 = 1536$	129.8	112.0	1.159
Java	v. 1.1.1	Real	Pentium	Win95	$3 \times 2^{10} = 3072$	58.4	51.8	1.127
Delphi	Pascal	CPU	Pentium	Win95	$5 \times 2^{11} = 10240$	50.2	44.7	1.124
Gofer	Haskell	CPU	Pentium	Linux	$4 \times 2^15 = 131072$	20.67	19.03	1.086
GHC 2.01	Haskell	CPU	Pentium	Linux	$3 \times 2^{17} = 393216$	26.1	24.55	1.063
Scheme48	Scheme	CPU	Pentium	Linux	$3 \times 2^9 = 1536$	67.09	63.49	1.057
Chez v4	Scheme	CPU*	SPARC	SunOS	$3 \times 2^9 = 1536$	65.9	64.1	1.028
GCC	"C"	Real	Pentium	Linux	$5 \times 2^{12} = 20480$	84.0	83.0	1.012
Chez v5	Scheme	CPU*	MIPS	IRIX	$3 \times 2^{10} = 3072$	31.9	33.7	0.946

Figure 6.21: The RGB benchmark (mutually recursive functions).

Language		Measurement			Problem Size	Time (sec)		
Name	Type	Time	Arch	OS		T_L	T_D	$S_{L\to D}$
SML/NJ	ML	CPU†	Pentium	Linux	$20 \times \alpha(25)$	20.3	7.7	2.636
GHC 2.01	Haskell	CPU*	Pentium	Linux	$10 \times \alpha(21)$	16.4	14.5	1.131
MosML	ML	CPU	Pentium	Linux	$20 \times \alpha(25)$	56.3	51.7	1.089
GHC 0.29 -O	Haskell	CPU*	Pentium	Linux	$10 \times \alpha(21)$	9.7	10.8	0.898

Figure 6.22: The fold benchmark (CPS and lexical size).

$$
\begin{array}{lcl}
\mathsf{map}\ f\ [\,] & = & [\,] \\
\mathsf{map}\ f\ (x : x_s) & = & (f\ x) : (\mathsf{map}\ f\ x_s)
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lcl}
\mathsf{map}\ g\ y_s & = & \mathrm{let}\ \mathsf{map}'\ f\ [\,] = [\,] \\
 & & \quad\ \mathsf{map}'\ f\ (x : xs) = (f\ x) : (\mathsf{map}'\ f\ x_s) \\
 & & \mathrm{in}\ \ \mathsf{map}'\ g\ y_s
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lcl}
\mathsf{map}\ g\ y_s & = & \mathrm{let}\ \mathsf{map}'\ [\,] = [\,] \\
 & & \quad\ \mathsf{map}'\ (x : xs) = (g\ x) : (\mathsf{map}'\ xs) \\
 & & \mathrm{in}\ \ \mathsf{map}'\ g\ y_s
\end{array}
$$

Figure 6.23: Optimization of the map function

non-recursive function of the original name that accepts the same parameters. This is shown for map in Figure 6.23. If it turns out that this does not improve the result of the lambda-dropping transformation, the enclosing function can be removed.

Lambda-dropping of non-recursive functions is probably not an advantage, and a recursive function that calls itself only once or twice will probably be slowed down by the extra overhead of closure creation. A compiler may attempt to unfold a recursive application of a function as part of a constant propagation transformation. At some level the compiler usually aborts (most compilers are not allowed to loop). At this point lambda-dropping could be applied. Depending on the implementation a threshold value for the depth of the recursion and the number of parameters that can be lambda-dropped could be used to control when lambda-dropping is applied. An abstract model tuned to the implementation could perhaps be applied to determine the threshold value for each recursive function.

6.6 Time complexity

This section briefly discusses the time complexity of the lambda-lifting algorithm of Chapter 3 and the lambda-dropping algorithm of Chapter 5. The time complexities are specified assuming we have at our disposal an associate data structure that implement its operations in time $\mathcal{O}(\log n)$ where n is the amount of data stored in the structure (a splay tree would do just fine, for example).

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 letrec 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 [26]. The $m \log m$ component is derived from traversing the program while gathering and making use of variable names during the parameter lifting stage.

The lambda-dropping algorithm has a time complexity of $\mathcal{O}(n^2 + m \log m)$, where n is the number of definitions and m is the size of the program. The n^2 component is derived from Step 3 of the block sinking stage. In a worst-case situation, we may need to make n traversals of the graph (which has size n) during this step. The other parts of the block sinking stage take time $m \log m$ since they consist of passes of the program structure while updating a data structure containing information about the use of variable names. The same holds for the parameter dropping stage.

6.7 Implementation Issues

This section discusses various practical aspects of implementing lambda-lifting and lambda-dropping. It also describes the current state of the lambda-dropper used to obtain example program throughout this work.

6.7.1 Lambda-lifting

Lambda-lifting is easy to implement in a functional language by following Johnsson's description [26]. His paper describing lambda-lifting includes a complete, working lambda-lifter written in lazy-ML. A complete lambda-lifter for higher-order programs (i.e. a standard lambda-lifter) was implemented in Scheme as part of the work of the author's MS.

6.7.2 Lambda-dropping

Hopefully it should be possible to implement a lambda-dropper based on the description of Chapter 5. To implement a lambda-dropper according to this description a set data structure and a graph representation are necessary. A complete lambda-lifter for higher-order programs (i.e. a standard lambda-dropper) was implemented in Scheme as part of the work of the author's MS. The implementation handles the language of Schism [11] and has been fitted as a back-end for Schism, as described in Chapter 7.

The current implementation does not generate maximally localized block structure, although it does come very close. In the second part of item 3 of Figure 5.1 it is described how to convert the scope tree consisting of nodes that are sets of functions into a scope tree with functions as its node. The description states that "if only a single function uses the function of a child node, these functions are localized to the function." The current implementation does not check for the use of a specific function of the father node within the functions of the child node. For this reason, the functions are only localized to a function in the father node if the father node is a singleton set. This limitation will be removed in a future revision, and has had no effect on any of the examples presented throughout this work.

6.8 Related Work

This section contains short descriptions of and references to work related to lambda-dropping. It is based on part of the corresponding outline found in Danvy and Schultz' paper [18].

6.8.1 Continuation-based programming

Shivers optimizes a tail-recursive function by "promoting" its CPS counterpart from being a function to being a continuation [41]. For example, consider the function returning the last element of a non-empty list.

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

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

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

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

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

Shivers promotes last′ from the status of function to the status of continuation as follows:

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

The type of last′ now reads:

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

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.

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

[2] For example.

6.8.2 Stackability

Recently, Tofte and Talpin have suggested to implement the λ-calculus with a stack of regions and no garbage collector [42]. 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.

6.9 Summary

Throughout this chapter many diverse and important aspects of lambda-lifting and primarily lambda-dropping have been discussed. The correctness of both of these transformations has been reviewed and the skeleton of a proof of correctness was outlined. Properties such as idempotence and inverseness have been specified. The transformations are either idempotent or reach a fixed point when iterated. On a specific subset of programs lambda-lifting and lambda-dropping are inverses. The abstract model that was developed in an earlier chapter was applied demonstrating that lambda-dropping could be beneficial for recursive functions. This was backed by the results of an empirical study the purpose of which was to show that lambda-lifting block structured programs can degrade the performance of recursive programs. Implementations of a lambda-lifter and a lambda-dropper were reviewed, and finally related work was summarized.

Chapter 7

Partial Evaluation

7.1 Introduction

Certain partial evaluators pre-process source programs by lambda-lifting them. The residual programs produced by the partial evaluator consist of recursive equations. It thus makes sense to employ lambda-dropping to reverse the effects of the earlier lambda-lifting transformation.

We provide a very short introduction to partial evaluation in Section 7.2. Readers who are familiar with partial evaluation can skip this section. Section 7.3 describes why and how lambda-dropping is useful with partial evaluation, and provides a few examples. Section 7.4 surveys related work. This chapter is based on part of Danvy and Schultz' paper [18] that discuss partial evaluation in relation to lambda-dropping.

7.2 A Quick Introduction to Partial Evaluation

This section is intended as an introduction to partial evaluation for someone not familiar with this field of study. It is taken with a few modifications from Consel and Danvy's "Tutorial Notes on Partial Evaluation" [14].

Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input. In essence, partial evaluation removes layers of interpretation. In the most general sense, an interpreter can be defined as a program whose control flow is determined by its input data. As Abelson points out, [21, Foreword], even programs that are not themselves interpreters have important interpreter-like pieces. These pieces contain both compile-time and run-timei.e.constructs. Partial evaluation identifies and eliminates the compile-time constructs.

Partial evaluation: what

Given a general program and part of its input, we want to specialize this program with respect to this known information.

Consider a program p and its input i, and say that somehow we can split i into a *static* (i.e. known) part s and a *dynamic* (i.e. unknown) part d. For example, p might take two arguments, one of which is a static value, and the other one of which is a dynamic value. Given a specializing function $\mathcal{S}$, we can specialize p with respect to s:

$$\mathcal{S}(p, \langle s, \_ \rangle) \;=\; p_s$$

By definition, running the residual program p_s must yield the same result as the general program would yield, provided both terminate:

$$\text{run}\, p\, \langle s, d \rangle \;=\; \text{run}\, p_s\, \langle \_, d \rangle$$

Operations may have been removed from p_s by performing them during the specialization process, so p_s can run faster then p.

In fact, $\mathcal{S}$ is Kleene's S_n^m-function [31]. This function is computable and thus it can be implemented: the result is what is called a partial evaluator, denoted *PE*.

$$\text{run}\, PE\, \langle p, \langle s, \_ \rangle \rangle \;=\; \mathcal{S}(p, \langle s, \_ \rangle)$$

Partial evaluation: how

Specializing a program with respect to all of its input amounts to running this program and producing a constant residual program, i.e.a program that takes an empty input and produces an already computed value. Therefore, a partial evaluator must include an interpreter to perform reduction at specialization-time.

Dually, specializing a program with respect to none of its input amounts to produce a (possibly simplified) version of this program. Therefore, a partial evaluator must include a compiler to construct the residual program. In the common case where the source and the target language coincides this compiler essentially mimics the identity function. Otherwise, it is a simple translator.

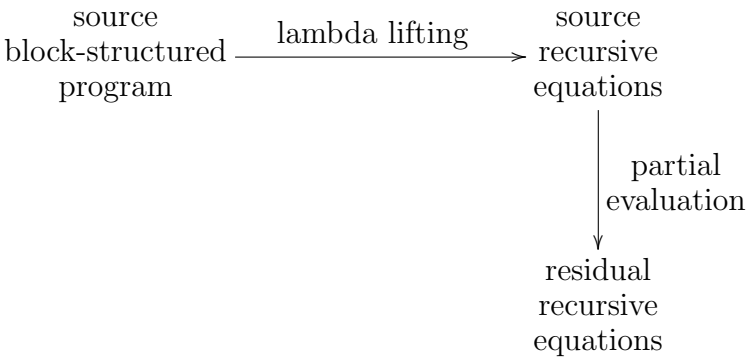
When specializing a program with respect to some known parts of its input, a partial evaluator corresponds to a non-standard interpreter: it evaluates the expressions depending on static data (i.e., data available at specialization-time) and it reproduces the expressions depending on dynamic data (i.e., data available only at run-time). A partial evaluator propagates constant values and folds constant expressions. It also inlines functions by unfolding calls, and produces specialized functions by residualizing calls. A *monovariant* specializer produces at most one specialized function for every source function. A *polyvariant* specializer can produce many specialized versions of a source function [8].

7.3 Lambda-Dropping

Our compelling motivation to sort out lambda-lifting and lambda-dropping is partial evaluation [28]. Both program transformations are highly useful within the field of partial evaluation.

As analyzed by Malmkjær and Ørbæk [34], 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.

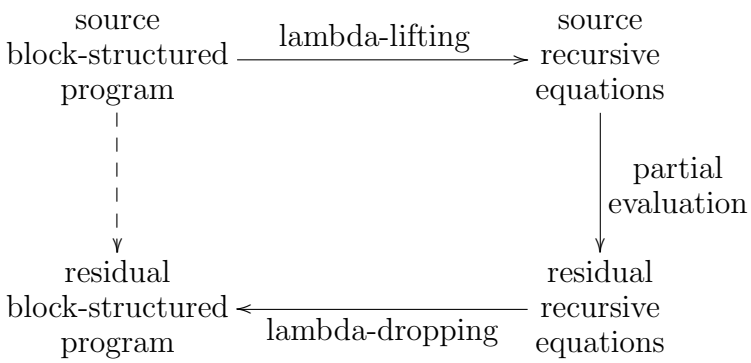
Specialization of programs without block structure never exhibits this problem. All specialization points are globally visible, so a higher-order function is never returned past the scope of of a specialization points that it refers to. Thus, recursive equations offer a convenient format for a partial evaluator. Similix and Schism, for example [7, 12], lambda-lift source programs before specialization and produce residual programs in the form of recursive equations.



Since using this approach implies that the block structure of the source program is flattened during specialization, the structure of the residual program is unlikely to resemble the structure of the dynamic part of the source program, unless the source program was written as recursive equations to begin with. The residual program suffers all the drawbacks of lambda-lifting outlined in the previous chapters, including unsafe declarations of auxiliary functions and an increased number of formal parameters for recursive functions.

The formal parameters of a recursive equation may be extended when statically propagating and reducing abstractions with dynamic parameters. Very often, because of this arity raising, the recursive equations produced by a specializer that lambda-lifts the source program are afflicted with a huge number of parameters. This increases the compilation time enormously, sometimes to the point of making the whole process of partial evaluation impractical [17].

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. Block structure in the source program will be reflected in the residual program. Recursive functions need only pass only the formal parameters that change through the recursion. Parameter dropping can potentially reduce arity raising.



Our lambda-dropper handles the output language of Schism. As this thesis is being submitted, we have integrated lambda-dropping into Schism, but have not yet had feedback from other Schism users.

7.3.1 Examples

A fold function

Figure 7.1 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 7.2. 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 7.3 displays the corresponding lambda-dropped program, which was obtained automatically.

As partial-evaluation users, we find it more clear to compare Figure 7.1 and Figure 7.3 rather than Figure 7.1 and Figure 7.2. (N.B. A monovariant specializer would have directly produced the program of Figure 7.3, using mere unfolding and no memoization).

The first Futamura projection

Let us consider a while-loop language as is traditional in partial evaluation and semantics-based compiling [13]. Figure 7.4 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 7.5. Each source while loop has given rise to a recursive equation. Figure 7.6 displays the corresponding lambda-dropped program, which was obtained automatically.

Again, we find it more clear to compare Figure 7.4 and Figure 7.6 rather than Figure 7.4 and Figure 7.5. The relative positions of the residual recursive functions now match

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

```
(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 7.1: Source program

```
(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 7.2: Specialized (lambda-lifted) version of Figure 7.1

```
(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 7.3: Lambda-dropped version of Figure 7.2

the relative positions of the source while loops. (N.B. Again, a monovariant specializer would have directly produced the program of Figure 7.6).

7.4 Related work

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 previous section, 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 [34], no partial evaluator for procedural programs handles block structure today.

The problem with higher-order functions that refer to a specialization point but are returned past the scope of this specialization point only occurs for a specific type of spe-

```
{
  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 7.4: 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 7.5: Specialized (lambda-lifted) version of the definitional interpreter with respect to Figure 7.4

```
(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 7.6: Lambda-dropped version of Figure 7.5

cializers. Only 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 exhibit the problem. Of these, there are few: Lambda-Mix [23] and type-directed partial evaluation [15] are monovariant; Schism [12] and Similix [7] lambda-lift before binding-time analysis; Pell-Mell [33] lambda-lifts after binding-time analysis; ML-Mix [6] does not specialize with respect to higher-order values; and Fuse [40] does not allow upwards funargs.

7.5 Summary

The lambda-dropping algorithm can be fitted as a back-end to a partial evaluator. This allows a polyvariant specializer for higher-order programs to produce block-structured residual programs. The block structure of the source program will be reflected in the block structure of the residual program. The increase in dynamic parameters will be reduced by parameter dropping in the context created by block sinking.

Chapter 8

Conclusion

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.

In this work, we have outlined the basic principles of block structure. We have defined and developed the notion of an explicit and an implicit context for a computation. To properly analyze lambda-lifting and lambda-dropping we have developed an abstract model of computation costs. Johnsson's lambda-lifting algorithm has been explained in detail relating it to similar program transformations. Our lambda-dropping algorithm was described in a similar framework thus realizing the automatic translation from recursive equations to block-structured programs that employ an implicit context to reduce parameter passing. We have described relations to other areas and formalized many aspects of lambda-lifting and lambda-dropping. At the end, we have provided a major application for lambda-dropping: as a back end in a partial evaluator. We have implemented lambda-dropping both as an independent Scheme to Scheme translator and also as a back-end for Schism.

This work could be extended in a number of directions. For it to be complete a formal proof of the correctness of both lambda-lifting and lambda-dropping should be developed. There is to the author's knowledge no formal proof of the correctness of lambda-lifting. The primary application of lambda-dropping is partial evaluation. For this reason it would be interesting to further investigate the influence of lambda-dropping on residual programs. Arity raising is a major concern, but we do not yet know to what degree lambda-dropping can be successfully applied to reduce arity raising. In an empirical study a small set of programs were examined and the effect of lambda-dropping in terms of execution time was approximated. The results seem to indicate that lambda-dropping could be used to improve the output of an advanced optimizing compiler. Whether this actually holds for real-world problems remains to be tested in practice.

Lambda-lifting and lambda-dropping are useful transformations that map higher-order programs between sets of recursive equations and block-structured programs. The lambda-

dropping algorithm is simple to implement and can be reduced to a series of simple transformations. Lambda-dropping has a wide range of applications, the most prominent of which is partial evaluation.

Appendix A

CPS Transformation by Fold

This appendix shows a simple implementation of the single-pass CPS transformation of [16], without tail-call optimization. The second, simplifying pass of other CPS transformations is eliminated by recognizing static redexes and reducing them during the transformation, using higher-order functions.

```
-- Single-pass CBV CPS transformer without tail recursion
cpsBlock (defsResLs', bodyRes') =
    (foldS defsResLs') `bindS` \defsResLs -> bodyRes' `bindS` \bodyRes ->
    unitS (\kappa -> (Blk (defsResLs, bodyRes kappa)))
cpsDefine (v, vs, bodyRes') =
    gensymS `bindS` \k ->
    bodyRes' `bindS` \bodyRes ->
    unitS (Def (v, vs++[k], bodyRes (\x -> App (Var k,[x]))))
cpsApply (appRes', argResLs') =
    gensymS `bindS` \x ->
    appRes' `bindS` \appRes ->
    (foldS argResLs') `bindS` \argResLs ->
    let make [] = (\kappa -> kappa [])
        make (e:es) = (\kappa -> e (\h -> (make es) (\t -> kappa (h:t))))
    in  unitS (\kappa -> appRes (\f ->
                (make argResLs) (\l -> App (f,l++[Lam ([x], kappa (Var x))]))))
cpsIf (e1Res', e2Res', e3Res') =
    gensymS `bindS` \k ->
    gensymS `bindS` \a ->
    e1Res' `bindS` \e1Res -> e2Res' `bindS` \e2Res -> e3Res' `bindS` \e3Res ->
    unitS (\kappa -> App (Lam ([k],
                              e1Res (\p -> If (p,
                                              e2Res (\m -> App (Var k,
                                                                [m])),
                                              e3Res (\n -> App (Var k,
                                                                [n])))),
                          [Lam ([a], kappa (Var a))]))
```

```
cpsLam (vs, bodyRes') =
    gensymS `bindS` \k ->
    bodyRes' `bindS` \bodyRes ->
    unitS (\kappa -> kappa (Lam (vs++[k], bodyRes (\x -> App (Var k,[x])))))
cpsLet bRes' = bRes' `bindS` \bRes -> unitS (\kappa -> Let (bRes kappa))
cpsVar v = unitS (\kappa -> kappa (Var v))
cpsCon c = unitS (\kappa -> kappa (Con c))

cpsProgram p' = p' `bindS` \p -> unitS (p (\m -> m))

cps = (showS 0) . (foldProgram cpsBlock cpsDefine cpsApply cpsIf
                     cpsLam cpsLet cpsVar cpsCon cpsProgram)
```

Bibliography

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

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

[3] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for language with closures. Technical report CS-TR-450-94, Department of Computer Science, Princeton University, Princeton, New Jersey, March 1994.

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

[5] J.W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132, 1959.

[6] 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.

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

[8] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[9] C. J. Cheeney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):667–678, 1984.

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

[11] Charles Consel. *The Schism Manual*. Yale University, New Haven, Connecticut, December 1990. Version 1.0.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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*, Amsterdam, The Netherlands, June 1997. To appear.

[19] Edsger W. Dijkstra. Recursive programming. In Saul Rosen, editor, *Programming Systems and Languages*, chapter 3C, pages 221–227. McGraw-Hill, New York, 1960.

[20] Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1989. DIKU Rapport 89/11.

[21] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

[22] Mayer Goldberg. An introduction to the $\lambda\mathbf{K}\beta\eta$-Calculus. Course notes, DAIMI, April 1996.

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

[24] Mitchell Wand Gregory T. Sullivan. Incremental lambda lifting: An exercise in almost-denotational semantics. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, November 1996.

[25] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[26] 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.

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

[28] 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.

[29] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine version 2.5, March 1993.

[30] Sonya E. Keene. *Object-Oriented Programming in* COMMON LISP. Addison-Wesley, 1989.

[31] Stephen C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

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

[33] 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.

[34] 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.

[35] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming*, Lecture Notes in Computer Science, pages 217–239. Springer Verlag, 1984.

[36] P. Naur. Revised report on the algorithmic language ALGOL 60. In *Comm. ACM, 6*, pages 1–17, 1963.

[37] 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.

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

[39] Kristoffer H. Rose. Explicit substitution *tutorial & survey*. Technical Report LS-96-3, DAIMI, Computer Science Department, Aarhus University, March 1996.

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

[41] 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.

[42] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994. ACM Press.

[43] Philip Wadler. The essence of functional programming (tutorial). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.