

# Implementing Closures using Run-time Code Generation

Martin Grabmüller

**Bericht Nr. 2006-02**

February 2006

ISSN: 1436-9915



Fachgebiet Übersetzerbau und Programmiersprachen  
Institut für Softwaretechnik und Theoretische Informatik  
Fakultät IV – Elektrotechnik und Informatik  
Technische Universität Berlin

Research Report

# **Implementing Closures using Run-time Code Generation**

Martin Grabmüller

February 2006



# Abstract

This report describes an implementation of a purely functional strict programming language which relies heavily on run-time code generation. Closures are not implemented as data structures containing code pointers and bindings for free variables, but instead by generating the machine code for a closure each time it is constructed. The actual values of free variables are embedded into the machine code instead of using references to a closure record. The goal of this experimental implementation is to examine the possibilities of run-time code generation – and just-in-time compilation in general – in the context of purely functional languages. The implementation details are described and the performance of several example programs is measured and compared to other functional language implementations.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Problem . . . . .	1
1.2	Run-time Code Generation . . . . .	2
1.3	Structure of this Report . . . . .	3
1.4	Acknowledgements . . . . .	3
<b>2</b>	<b>Closure Implementation Techniques</b>	<b>5</b>
2.1	Example . . . . .	5
2.2	Flat Closures . . . . .	6
2.3	Nested Closures . . . . .	6
2.4	Run-time Generation of Wrapper Functions . . . . .	7
2.5	Lambda Lifting . . . . .	8
<b>3</b>	<b>Implementing Closures with Run-time Code Generation</b>	<b>10</b>
3.1	Target Machine . . . . .	10
3.2	Architecture . . . . .	11
3.3	Design Decisions . . . . .	11
3.4	Source Language . . . . .	12
3.5	Code Generation Outline . . . . .	13
3.6	Compilation Scheme . . . . .	14
3.7	Code Generation for Source Expressions . . . . .	16
3.8	Optimizing Tail Calls . . . . .	19
3.9	Compilation of Mutually Recursive Functions . . . . .	19
3.10	Optimizations . . . . .	26
3.11	Limitations . . . . .	26
<b>4</b>	<b>Evaluation</b>	<b>27</b>
4.1	Benchmark Programs . . . . .	27
4.2	Platform . . . . .	28
4.3	Performance Results . . . . .	28
4.4	Comparison with other Implementations . . . . .	29
4.4.1	Results . . . . .	30
4.4.2	Language and Implementation Differences . . . . .	33
4.4.3	Comment . . . . .	35
4.5	Evaluation Conclusion . . . . .	35

*Contents*

<b>5</b>	<b>Conclusion and Future Work</b>	<b>36</b>
5.1	Related Work . . . . .	36
5.2	Future Work . . . . .	37
5.3	Conclusion . . . . .	38
<b>A</b>	<b>Example Program Source Code</b>	<b>39</b>
A.1	Benchmark Programs . . . . .	39
A.2	Opal Programs . . . . .	41
A.3	Standard ML Programs . . . . .	43
A.4	Haskell Programs . . . . .	44
A.5	Scheme Programs . . . . .	46
A.6	C Programs . . . . .	47

# 1 Introduction

This report describes a prototype implementation of a simple functional programming language and presents experimental results of its performance. Since the system uses run-time code generation as a central implementation concept, the run-time measurements include compilation times, and the implementation has been tuned to have very short compile times while producing reasonably efficient machine code.

The main goal of this implementation effort was to investigate the design space concerning dynamic compilation in the setting of purely functional programs. Therefore, we have chosen to adapt a closure implementation technique which relies on dynamic compilation, but which has not yet been implemented because of expected poor performance. As we will show in this report, this expectation is false in many cases but true in some others. How to overcome these problems will also be discussed. Dynamic compilation (also called just-in-time compilation) has been used very successfully in the context of object-oriented languages. Since many language concepts in functional and object-oriented languages are implemented in similar ways, we expect that just-in-time compilation is a suitable implementation techniques for functional languages.

The implementation described in this report was built for the purpose of performing experiments, not as a practically usable software development system. Therefore, we have concentrated on the basic functionality necessary for executing programs written in a simple functional language using run-time code generation techniques. Our focus is on getting experimental evidence supporting our closure implementation, which in turn supports dynamic compilation as a technique for implementing full-scale functional programming languages in future work.

The remainder of this chapter introduces the problem we attack using run-time code generation: the implementation of closures in functional languages.

## 1.1 The Problem

In this report, we concentrate on a particular implementation aspect of functional (higher-order) programming languages: The need to close functions ( $\lambda$ -abstractions) over their free occurrences of variables. Consider the following example function  $f$ , which takes a number  $y$  and a list  $l$  and returns a list in which  $y$  is added to each element of  $l$ :<sup>1</sup>

```
f y l = map (\x -> x + y) l
```

---

<sup>1</sup>In this report, Haskell-like syntax [26] is used whenever examples are given that are not concerned with a particular programming language.

## 1 Introduction

The function  $\lambda x \rightarrow x + y$  has one free variable,  $y$ , and the rules of static scoping dictate that when the function is applied (at some unknown point in the future in the function definition of *map*), the value of  $y$  will be the value which  $y$  had when the function was created. The problem is that the text of the function (and the machine code created by a compiler as well) has no indication of which value is to be assigned to  $y$  at which point in the program. Since the function  $f$  can be called many times, with many different values for parameter  $y$ , it is not possible to associate this information with a single representation (text or machine code) of the function.

The problem is traditionally solved by pairing the set of bindings for the free variables of a code fragment with the function text (or machine code) and to store them together in a data structure. This data structure is called *closure*, because the bindings for the free variables together with the program text result in a closed expression. How this data structure is internally organized is conceptually irrelevant: it is only necessary to extract the values of free variables and the program text from closures. Nevertheless, since the point of this report is the implementation of closures, we will have a closer look at closure implementations in Chapter 2.

The implementation of closures using closure records has been used in functional language compilers for a long time. In the work described here, the focus is on another implementation method: run-time code generation of functions whenever a closure is created.

## 1.2 Run-time Code Generation

The idea behind dynamic compilation or run-time code generation is to shift some of the tasks traditionally performed by off-line compilers (at *compile time*) to the time when a program is executed (*run time*). This approach has several advantages as well as disadvantages.

The advantages of run-time (or dynamic) code generation are well-known from popular implementations of object-oriented programming languages, such as Java [14] or C# [?]. Dynamic analysis and transformation techniques have possibilities to exploit information present at run-time, which static compilers have not and are forced to approximate (either using some static heuristics or profiling data from earlier program runs). Especially the possibility to react to dynamically varying program behavior is a clear advantage of the dynamic compilation approach.

The problem with this method is that code generation is costly, and one of the original goals of compilers is to perform as much work as possible *before* the program is run. It is therefore important to balance the increases in performance due to additional optimization possibilities with the cost of doing them while an application program executes.

Dynamic compilation techniques have many purposes, and in the present text we concentrate on the topic of closure representation in functional programs.

Feeley and Lapalme [9] have invented a closure implementation technique which avoids the creation of closures as data structures. The drawback of treating closures as data structures is that the structure has to be accessed each time a function is called or a variable is accessed. This adds some overhead to program execution, and Feeley and Lapalme's technique avoids (most of) this overhead. Our work directly builds on their method. The basic idea of Feeley and Lapalme was to compile *complete* functions whenever a closure is created, substituting the values of variables for their occurrences, thus making the function body a closed expression which can be compiled without any references to free variables, except for global variables, which can be addressed directly. Since in 1992 compilation of whole functions at run time was not practical, they invented the more limited scheme described in the next chapter. This technique reduces code generation time but results in slower execution of the generated functions, because of additional parameter manipulation and function calls.

The contribution of our work is that we extend the closure implementation technique of Feeley and Lapalme to the code generation of complete closures, exploiting the dynamic information gained when closing over variables in order to optimize the resulting machine code. The hope is to reduce execution times despite the additional overhead incurred by run-time code generation of complete functions. Since modern processors are about 50 times as fast as they were in 1992, it can be expected that the approach is feasible.

## 1.3 Structure of this Report

Chapter 2 summarizes the traditional techniques used for implementing closures. Both classic closure record methods as well as Feeley and Lapalme's method is described. Chapter 3 describes the implementation of a simple functional programming language. It is based on the closure implementation technique described above and performs some simple optimizations using dynamically available information. The chapter also contains some details about the difficulties in implementing the dynamic compiler efficiently. Chapter 4 measures the performance of the implementation using some simple benchmark programs and compares it to other popular functional programming language implementations. Chapter 5 relates this report to other work, points out some future work and draws a conclusion.

## 1.4 Acknowledgements

I wish to thank Dirk Reckmann for commenting on a draft of this report. Peter Pepper made many suggestions improving the final version.

The implementation of the run-time code generator uses a set of C macros for generating x86 machine code. These macros are defined in the file `gen-x86.h`,

## 1 Introduction

which was taken from the `libjit` library<sup>2</sup>, which took it (according to a comment in the file) from the Mono Project<sup>3</sup>, which in turn derived it from the ORP project<sup>4</sup> originating at Intel. The file may be redistributed under the Lesser General Public License<sup>5</sup>.

---

<sup>2</sup><http://www.southern-storm.com.au/libjit.html>

<sup>3</sup><http://www.go-mono.com>

<sup>4</sup><http://orp.sourceforge.net/>

<sup>5</sup><http://www.gnu.org/licenses/licenses.html#LGPL>

## 2 Closure Implementation Techniques

Implementing closures using closure records, as briefly mentioned in the introduction, requires three different operations involving these closure records: first, closures must be created every time a  $\lambda$ -abstraction is evaluated. This requires allocating enough memory to hold a function pointer and the values of all the free variables of the function, followed by storing the function pointer and the variable values into the closure record. When a function is called, its address is extracted from the closure record and an indirect jump to this address is made. The closure record is passed to the called function in addition to all the normal function arguments, so that the values contained in the closure can be accessed by the called function. Last, accesses to free variables of the function need to fetch values indirectly from the closure record.

Often, function definitions are nested within each other. Closure implementations differ in how closures for such nested functions are constructed. When using *nested closures*, closures for the inner functions simply contain pointers to the closures of outer functions. These closures are treated like any other heap-allocated data structure. *Flat closures*, on the other hand, are implemented by copying the values of all free variables of lexically surrounding functions into a single closure record. This chapter describes these techniques in more detail and illustrates some points addressed by our closure implementation technique in the next chapter.

Other detailed descriptions of closure implementation techniques can be found in [2, 3, 29] as well as in most compiler text books.

### 2.1 Example

As an example, consider the expression

$$\text{add} = \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow x + y + z$$

which defines a curried addition function. The outer  $\lambda$ -expression has no free variables (if we consider the operation ‘+’ a primitive operation, and not a function), whereas the nested  $\lambda$ -expression has the free variable  $x$ . The innermost  $\lambda$ -expression has the free variables  $x$  and  $y$ . The following sections will describe how the evaluation of the following partial application of the example expression above, which results in a function of one argument, is handled in various implementation techniques for closures:

$$\text{add } 4 \ 3$$

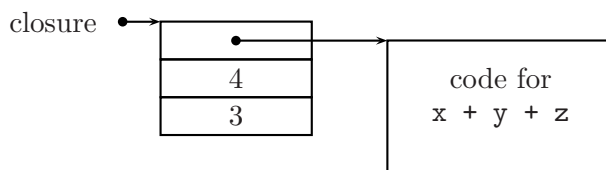


Figure 2.1: Flat closure implementation.

## 2.2 Flat Closures

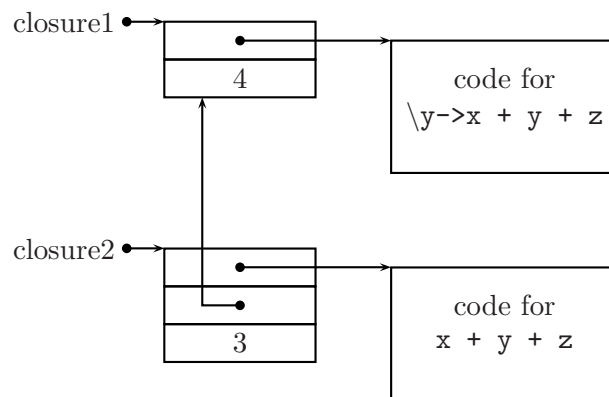
A flat closure is a data structure that contains a pointer to the code of a function, and the value of each free variable used in the function’s body. This situation is depicted in Figure 2.1 for the result of the expression `(add 4 3)`. The result of this expression is a pointer to a three-word record (called *closure* in the figure). The first field of the record points to the code of the function body, and the second and third word hold the values of the free variable  $x$  and  $y$ , which are 4 and 3, respectively. When this closure is to be called, the first word must be extracted and called, and additionally, the closure pointer must be passed as an extra parameter. Inside of the function’s code, all references to the variable  $z$  are made as normal parameter accesses, whereas references to the variables  $x$  or  $y$  must go indirectly through the closure pointer by loading it into a register and fetching the second or third word of the record.

The advantage of flat closures is fast access to free variables, but closures tend to become larger than for the nested closure technique, thereby possibly increasing the memory footprint of the application.

## 2.3 Nested Closures

Nested closures are similar to flat closures, with the difference that each closure record only contains the variable bindings of free variables which are free in the function, but not free in any surrounding  $\lambda$ -abstractions. For our example, this means that the closure returned for the expression `(add 4 3)` only contains the binding for variable  $y$ , because  $y$  is free in the body of the innermost  $\lambda$ -expression, but not in any enclosing abstraction. Instead of the binding for variable  $x$  (as in the case for flat closures), the closure contains a pointer to the closure of the function that created the result closure. This is illustrated in Figure 2.2. The closure referred to as *closure1* is the result of the application `(add 4)`, *closure2* is the result of applying *closure1* to the value 3. Accesses to variable  $z$  now address a local variable, accesses to variable  $y$  refer to the closure record, and accesses to variable  $x$  follow the link to the enclosing (“parent”) closure, and take its value from there.

Note how the nested closure scheme corresponds to how nested functions in



**Figure 2.2:** Nested closure implementation.

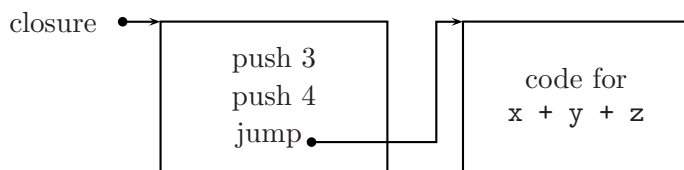
languages like Pascal are implemented. What is called a closure record here is referred to as an *activation record* in imperative languages, and the parent pointer is often called a *static link*.

The advantage is that this implementation technique is very easy to implement and results in very small closures, since only the free variables of the innermost function are contained in the closure. The disadvantage is that accesses to free variables can result in multiple-indirect references when free variables of outer functions are used. Additionally, closures can prevent the deallocation of large closure records of enclosing functions, even when they only contain a single reference to one of its fields. In some cases, this can result in a different space complexity than the flat closure method for the same program.

It is also possible to mix implementation techniques and to use flat closures for some functions and nested closures for others. As long as the function pointer is stored at the same offset for both schemes, the decision can be made locally for each abstraction: only the creation of the closure record and the accesses to free variables inside the abstraction need to agree on the representation. The next implementation method is not compatible with the two previously discussed techniques.

## 2.4 Run-time Generation of Wrapper Functions

A very different technique, pioneered by Feeley and Lapalme [9], is to avoid the indirection at call and variable access time caused by using closure records. They proposed to generate specific code each time a closure is created. This code is specialized to the values of the free variables of the abstraction, which means that the values of free variables can be treated like constants in the body of the function. Since compiling complete functions each time a closure is created seemed



**Figure 2.3:** Wrapper code closure implementation.

too costly, Feeley and Lapalme proposed to trade off execution time for compilation time by reducing the work required for each closure creation. Figure 2.3 shows how the approach works. When the closure for the innermost  $\lambda$ -abstraction in the example is created, no closure record is created but instead a piece of machine code is generated. This code simply pushes the values of the free variables (4 and 3 in the example) onto the run-time stack and transfers control to the code implementing the function body. Since now the values of the free variables are located on the run-time stack, they can be accessed like function parameters (in fact, they have been converted to function parameters by the wrapper function). A little care has to be taken, because the parameters pushed by the wrapper function are pushed *on top* of the return address, whereas the original parameters are below. The epilogue of a function therefore needs to pop the free variable values from the stack before performing a return instruction.

## 2.5 Lambda Lifting

Lambda lifting [16] is a complementary implementation method to the closure representation techniques described above. The idea is to transform the program into a form where all nested functions are lifted to the top level and all free variables of expressions are turned into additional parameters of these functions. Since no free variables appear in abstractions, only some method for representing partial applications is needed. For this, any of the methods described above can be used.

The lambda lifting transformation consists of finding the sets of free variables of each expression and abstracting over them, effectively adding them to the parameters of the enclosing abstractions. At each call site of such an abstraction, the values of the additional parameters (the free variables) are added to the actual parameter list. The transformation process is rather straightforward, with the only exception that finding the sets of variables to abstract over for mutually recursive functions requires a fix point iteration.

The transformation can be demonstrated as follows. Beginning with the example function definition of `add` given above, the following definition is achieved by naming the  $\lambda$ -expressions using `let`-expressions:

```

add = \x -> let l1 = \y -> let l2 = \z -> x + y + z
                in l2
        in l1

```

In a next step, all function definitions are modified to include the free variables of their bodies in their parameter lists. Additionally, all uses of these functions need to supply the free variable values as additional parameters.

```

add = \x -> let l1 = \x y -> let l2 = \x y z -> x + y + z
                in l2 x y
        in l1 x

```

Now there are no free variables in the function bodies anymore (except for global definitions like '+'), and the functions can be lifted to the top level:

```

l2 = \x y z -> x + y + z
l1 = \x y -> l2 x y
add = \x -> l1 x

```

At this point, no closures need to be created and only a mechanism for handling partial applications is needed.

The code generation method described in Section 2.4 is an optimization of the basic idea of Feeley and Lapalme: the method described avoids the code generation for whole functions at the expense of treating free variables like parameters. This does not permit most optimizations which rely on certain expressions to be constants. The goal of our experimental implementation, which is presented in the next chapter, is to make these optimizations possible by implementing their original idea of compiling whole functions, with constants substituted for free variable references.

## 3 Implementing Closures with Run-time Code Generation

The implementation of a functional programming languages requires several distinct components. A compiler translates the source program into a machine-executable binary code (or the byte code of a virtual machine). When the resulting program is executed, it requires several services, such as memory management and facilities for input/output which are not expressible in the language itself. These services are provided by the run-time system. Traditionally, these two components are separated, and the compiler is free to perform any time-consuming analyses and transformations, since it is expected that this cost is only seldomly paid, compared to the number of times the program is actually run. In a dynamic setting, where the compiler is part of the run-time system, it has much less freedom, because compilation time is added to the overall execution time of the program which is to be executed. If the compiler is sufficiently fast, and the performance improvements due to exploiting dynamic information are large, a dynamic system may even outperform a static compiler. But since most dynamic optimizations are based on heuristics, there is no guarantee that this is the case.

The prototype described in this report is in its early stages, and lacks many components one expects from a production-quality programming systems. There is no parser for the language processed, so currently the system can only execute some pre-defined programs (these programs are described in detail in the next chapter). The system also lacks a garbage collector. So basically the prototype concentrates on the main topic of this work: run-time code generation. When started, several example programs are converted from abstract syntax trees to executable x86 machine code and executed. During evaluation, whenever a closure is created, additional code is generated. This interplay between the code generator which (besides other code) creates calls to itself and the application code, which calls the code generator whenever a closure is needed, results in an interleaving between the application program and the run-time code generator.

### 3.1 Target Machine

The dynamic compiler generates machine code for the IA-32 architecture from Intel. Since our system currently only supports basic arithmetic and control flow, it does not depend on any special processor version and does not take advantage of any recent additions to the instruction set. The only exception is the instruction `rdtsc`, which allows access to an internal counter. We use it for performing precise timings when benchmarking the implementation (see the next chapter).

## 3.2 Architecture

The system consists of the following components. The *code cache* is a memory region used for storing dynamically generated machine code. Of course, it is necessary to check when this memory area is full and to react appropriately. A garbage collector could trace functions which cannot be called anymore, because all references to their entry points are not reachable, and reclaim the space occupied by these functions. Another possibility is to simply throw away all compiled functions and to recompile them when they are used again. Care has then to be taken not to throw away any functions which are still executing. Currently, the system simply halts when the code cache overflows. This is done by marking the last page of the code cache as non-writable using the system's memory management unit. Whenever the code generator tries to write code into this protected area, a segmentation fault occurs which halts the program. A real implementation would catch this fault, perform a garbage collection and restart the program at the point it was interrupted.

The second component is the dynamic code generator. The code generator is responsible for generating the machine code from a suitable representation of the program. Currently, this representation consists of simple abstract syntax trees corresponding to the language implemented, which will be described in Section 3.4 below. The functions created by the code generator are compatible with the C calling conventions and can therefore be called through function pointers by ordinary C code.

## 3.3 Design Decisions

A practical closure implementation using run-time code generation must meet the following contradicting goals:

1. Code generation must be fast. This is because the time needed for generating code is contributing to the overall execution time of an application.
2. The generated code must be efficient. Otherwise, all the advantages of our approach, such as efficient function calls and eliminated variable accesses are negated by overall low performance.

The first goal is addressed by restricting the run-time code generator to be a fast one-pass compiler. Since using a one-pass compiler alone would miss many important optimizations, we additionally add an analysis pre-pass that is run once when the code to be executed is loaded. On-demand invocations of the code-generator reuse the information gathered by the analysis pass and need only traverse the abstract syntax trees of the function they compile in the current invocation.

Another possibility would be to use pre-compiled code templates for each function into which the values of free variables are simply patched. We have decided

$E ::= x$	Variable
$c$	Constant
$\lambda x.E$	Abstraction
$E_1 E_2$	Application
$\text{let } x_1 = E_1 \dots x_n = E_n \text{ in } E$	Let
$\text{letrec } x_1 = E_1 \dots x_n = E_n \text{ in } E$	Recursive let
$\text{if } E_1 \text{ then } E_2 \text{ else } E_3$	Conditional
$p E_1 \dots E_n$	Primitive

Figure 3.1: Source language

against this method since it precludes too many important optimizations: when a constant appears in specific positions in the code, it can result in drastically different code, depending on the constant's value. This cannot be expressed as easily with code templates.

Since a code generation pass is restricted to a single function's code and does not traverse nested function definitions (these are compiled when the function is called), the pause times induced by the run-time compiler are expected to be short. Further experimental results are needed to confirm this claim.

### 3.4 Source Language

Since the compiler takes an abstract syntax tree as its input, the actual external representation is not important for its implementation. But for the sake of completeness and in order to present examples, this section defines a surface syntax for the language.

The language is an enriched  $\lambda$ -calculus with variables, abstractions, applications, constants, conditionals, recursive and non-recursive let and primitive operations such as addition, subtraction, comparison, etc. Figure 3.1 summarizes the syntax. The semantics corresponds to a call-by-value  $\lambda$ -calculus comparable to languages like Scheme, Standard ML or Opal.

A variable refers to a variable bound by a  $\lambda$ , **let** or **letrec** expression. A constant is a 32-bit integer in two's-complement. A  $\lambda$ -abstraction creates an anonymous function which, when applied to a value, binds its variable to that value and evaluates the body expression. An application evaluates both subexpression and applies the first result (which must be an abstraction) to the second. A **let**-expression evaluates all expressions  $E_i$  and binds the variables  $x_i$  to the results. The variables are in scope when the body expression is evaluated. **Letrec**-expressions are similar, but the variables  $x_i$  are in scope not only for the body expressions but also for all bound expressions  $E_i$ . Since the language is eager, uses of the variables  $x_i$  in the expressions  $E_i$  must be protected by one or

more  $\lambda$ -abstractions. The conditional expression first evaluates expression  $E_1$ , and when it evaluates to a true value (a value not equal to 0 in the current implementation), then  $E_2$  is evaluated, otherwise  $E_3$  is evaluated. A primitive application evaluates its arguments and performs some basic operation, such as the addition or comparison of two numbers.

We will use the convention that expressions may be parenthesized in order to avoid ambiguities.

Several invariants on valid programs are expected to hold: `letrec` expressions are only allowed to bind abstractions and applications of primitive operations are always fully saturated. The primitive functions currently supported are addition (+), subtraction (-), multiplication (\*) and three comparison operators (<, = and >).

Note that the language does not support data structures (except for the encoding in the  $\lambda$ -calculus, which is not suitable for performance comparisons) and has no means for performing input/output. This restricts the possible benchmark programs (see Chapter 4) to simple arithmetic and recursion.

### 3.5 Code Generation Outline

Code is generated on the basis of a function. Whenever execution reaches a point in the program where a closure is constructed, the run-time code generator is invoked. It is passed the abstract syntax tree of the function and the values of all its free variables. Then the expression forming the body of the function is compiled in a mostly standard way (recursively descending the abstract syntax tree), but avoiding expensive analysis and transformations with little gain, in order to reduce compilation time. In order to speed up certain operations, such as looking up the set of free variables of a function, an expression to be evaluated is analyzed in a pre-pass that collects this essential information. In a production system, the task of collecting syntactic information such as free variable sets, should be performed prior to running the program, because the analysis impose a certain cost without the need to be done at run-time.

One particular problem arises with the compilation of mutually recursive functions defined using a `letrec` form. Since all functions in a set of such functions can refer to each other, and so each function possibly belongs to the free variables of all other functions in the set, it is not possible to compile the functions one after the other. Two solutions to this problem have been considered. One is to allocate the space in the instruction cache for all functions before starting the actual compilation, thereby fixing the addresses of their entry points. The order of compilation for the function is then unimportant, because the entry points are the only information needed for compilation. The disadvantage of this approach is that it is normally not known how large the machine code for each function will be before compiling it. The allocation has to be done conservatively, possibly wasting a lot of space (especially because optimization can often cause a function

to shrink by a large amount when removing dead code). The other solution is to compile one function after the other, but remembering at which positions in the machine code one of the other functions is mentioned. When all functions are compiled, the addresses can be patched into the code at the remembered points. This is similar to the technique used in assemblers to resolve forward references [28] and is the technique we have actually adopted. Details are given in Section 3.9 below.

In the next sections, the compilation process is described in more detail. After describing the fundamental design of the compilation scheme and how the various language constructs are actually compiled, the problems of tail call optimization and mutually recursive functions is shown in more detail. The chapter closes with a discussion of optimization possibilities and the limitations of the current implementation and the general compilation approach.

## 3.6 Compilation Scheme

The compilation of an expression is expressed by a function  $\mathcal{C}$ , which takes six argument:

1. The abstract syntax tree of the expression;
2. a compilation environment which maps variable names to variable descriptors (explained below);
3. the depth of the evaluation stack;
4. information about register usage;
5. a linkage descriptor, which describes how to continue when the current expression is evaluated; and
6. the machine code generated so far.

The function returns three results:

1. The location of the expression result in an operand descriptor;
2. the updated register usage information; and
3. the machine code resulting from appending the input machine code and the code generated for the expression.

All these parameters and return values are explained in this section.

The abstract syntax tree is a minimally annotated syntax tree. At the moment, the implementation only requires that each subexpression is annotated with the set of its free variables.

The compilation environment maps variable names to either constant values, when the variable is bound to a constant; to a stack offset, when the variable

has been allocated to the stack, or to a special value which specifies that the value of the variable is not yet known. This special indicator is necessary for the compilation of mutually recursive functions, as described in Section 3.9. Whenever a variable reference is compiled, its descriptor is looked up in the compilation environment. When the variable is bound a constant (as will always be the case for free variables, except when their value is unknown), it can be treated like any constant value written in the source code. This handling of free variables allows optimizations based on the values of free variables, such as constant or conditional folding. Variables allocated on the stack are either loaded into registers when necessary, or directly used from their stack locations.

The depth of the current stack frame (which serves as the evaluation stack) for complex expressions is passed as a parameter to subexpressions in order to properly address local variables.

The register usage descriptor contains a map of allocated registers, so that register allocation can pick the next available register. It is also used to find out which registers have to be saved across function calls.

A linkage descriptor is a data structure which tells how control will proceed when the evaluation of an expression is completed. This information is passed top-down through the abstract syntax tree and allows to generate efficient code for control structures. For example, when the body of an abstraction expression is compiled, a descriptor of kind **return** is passed as the linkage descriptor. When the subexpression which determines the result of the function is compiled, the **return** descriptor causes the generation of code which destroys the current stack frame and performs a return instruction. Other possible descriptors include **next**, which causes no code to be emitted, so that evaluation simply falls through to the next instruction, **jump**, which causes the code to jump to some location specified in the descriptor, and the similar descriptor **branch**, which jumps if the expression evaluates to a false value, and acts like **next** otherwise. This form of passing the control destination to the compilation function for an expression is called *destination-driven code generation* by Dybvig, Hieb and Butler [7]. For the results of expressions, however, we chose another approach than the one they described, as will be explained below.

The machine code is passed into the compilation function to enable patching forward jumps, and to append newly generated code to it. In practice, the machine code is not explicitly passed to the compilation function, but rather appended to using side-effects, for efficiency reasons.

Evaluating expressions results in values, and we have chosen a compilation scheme for handling expression results in which loading of constants or memory locations is delayed as long as possible. This reduces register usage and allows us to make use of the many addressing modes supported by the CISC architecture which is targeted by the code generator. The scheme works as follows: whenever a simple expression is evaluated (such as a constant or a variable reference), no code is generated. Instead, an operand descriptor is constructed which describes *how* to get the value, should it eventually be needed. When a value is needed,

for example, when performing an arithmetic operation on two such operands, the necessary code is generated for loading the value into a register, or, if the instruction set allows, to embed the operand directly into the machine instruction. This technique not only saves registers, because it is not necessary to load trivial expressions into a register, but also allows some simple targeted register allocation. Function results, for example, are expected to be loaded into the `eax` register, so the code for a function ends in a move instruction which loads the register. Whenever the value is already in the `eax` register (for example, from a nested function call), no move instruction is necessary at all.

The second output from the compilation function is an updated register descriptor which tracks when new registers have been allocated for generating code for the expression.

The last output (conceptually, as it is passed as a global data structure in reality) is the updated machine code containing all the code generated so far.

## 3.7 Code Generation for Source Expressions

In this section, the code generation for the different syntactic constructs listed in Figure 3.1 is described. If not otherwise mentioned, the code generated for each subexpression is followed by code implementing the linkage descriptor passed in for the compilation. For example, when a variable expression is compiled with a linkage descriptor `return`, its value is loaded into register `eax`, the current stack frame is destroyed and a return-from-function instruction is emitted.

**Variables** Variables are looked up in the compilation environment. When they are bound to constants, an operand descriptor containing that value is created and returned. When they are bound to stack locations, an operand descriptor indicating this fact is created. When the value and location of a variable is unknown, this is also mentioned in the result operand descriptor. In none of these cases any code is generated. When the operand is later needed by an operation, the machine code actually generated depends on the use of the variable: when used in an arithmetic or move instruction, the appropriate addressing mode for accessing local variables relative to the current function's frame pointer is used. When the variable is passed to a function, the variable is directly pushed to the stack (for non-tail calls) or loaded into a register and written to the parameter stack location (for tail calls).

**Constants** Constants are treated similarly by packaging them in operand descriptors and not generating any code. The code emitted when the constant is later used depends on the use, similar to the case for variables described above.

**Abstractions** Abstractions are handled by generating code which calls the run-time code generator. The abstract syntax tree of the abstraction is passed to

```

1  push  %ebp
2  mov   %esp,%ebp
3  push  $0x8055030
4  call  0x804ff37
5  add   $0x4,%esp
6  movl  $0x2a,0x8(%ebp)
7  leave
8  jmp   *%eax

```

**Figure 3.2:** Assembler code for identity function example (part 1).

```

1  push  %ebp
2  mov   %esp,%ebp
3  mov   0x8(%ebp),%eax
4  leave
5  ret

```

**Figure 3.3:** Assembler code for identity function example (part 2).

the generator, together with the values of its free variables. Sometimes the value of one or more of the free variables is not yet known, in which this fact is also communicated to the code generator. This happens in the case of mutually recursive functions defined using a `letrec` expression. For these abstractions, the code generation of the abstraction is followed by a patching phase, as described in Section 3.9 below. Since the result of function calls is always in the register `eax`, the returned operand descriptor refers to that register.

As an example, consider the following expression, which is essentially an application of the identity function to the number 42:

```
(\ x -> x) 42
```

The code generated for this example is shown in Figure 3.2. The lines 1 and 2 set up the stack frame for the expression. This is necessary for allocating local variables. Then, in lines 3–5, the run-time code generation function is called, passing the abstract syntax tree as a parameter. Since the function has no free variables in this case, there are no other parameters. When the code generator returns it has left a pointer to the newly generated function in register `eax`. This code is shown in Figure 3.3. It simply moves the incoming parameter from the stack into the result register and returns. The function returned by the code generator is then called by passing the parameter and calling the new function in lines 6–7. (The parameter passing is done by overwriting the parameter of the function and jumping to the function, since it is a tail call.)

Note that the initial compilation of the expression only generates the code in Figure 3.2, and that the generation of the second function is delayed until the

expression is evaluated.

**Applications** Code generation for application consists of generating code for the function and the argument of the application. The value of the argument is pushed onto the run-time stack, followed by a call instruction to the value returned by compiling the function expression. When the call appears in tail position, the parameter of the current function is overwritten with the argument and instead of a call, a jump to the destination function is performed after cleaning up the current stack frame. This optimization saves stack space for tail-recursive functions and is described in more detail in Section 3.8 below.

**Let and Letrec Expressions** The code for both `let` and `letrec` expressions begin with allocating stack slots for the variables declared in their binding clauses, followed by evaluating the bound expressions and storing their values into the variables. Then, the code for the body of the expression is compiled with the linkage descriptor of the whole expression. The only difference between the two expression types is that for `letrec`-expressions the compilation of the bound expressions is followed by a patching phase which inserts the addresses of the values into the other values (which must be abstractions), so that they can refer to each other.

**Conditionals** Conditionals are compiled by first compiling the condition part with a `branch` linkage descriptor, which branches to the alternative expression of the conditional. Since a `branch` linkage descriptor falls through if the calculated value is true, control will reach the consequent expression in this case. Both the consequent and the alternative expression are compiled with the linkage descriptor of the whole conditional, which allows to move return or jump instructions into both arms of a conditional.

The following expression will be used to illustrate the compilation of conditionals:

```
if (2 > 0) then 23 else 42 fi
```

(The code generator has been instructed not to fold conditionals, otherwise the conditional expression would get optimized away. In order to save space, we use such a simple expression anyway.)

The machine code compares the two operands of the `<`-operator and jumps to the else part if the test does not succeed. Note how the function epilogue (`leave; ret`) has been moved into both branches of the conditional by our use of destination-driven code generation.

**Primitives** Primitive expressions, such as addition or comparisons, are compiled by generating code for all operands, loading them into registers if necessary and emitting the machine instruction which implements the operation. For comparison operations in the context of `branch` linkage descriptors, code is generated that

```

1  push  %ebp
2  mov   %esp,%ebp
3  mov   $0x2,%eax
4  cmp   $0x0,%eax
5  jle   0x4015d03a
6  mov   $0x17,%eax
7  leave
8  ret
9  mov   $0x2a,%eax
10 leave
11 ret

```

**Figure 3.4:** Assembler code for conditional expression.

branches directly based on the contents of the condition code register, without transferring condition codes to a general-purpose register and branching on its value.

### 3.8 Optimizing Tail Calls

Tail calls are function calls whose return value determines the value of the expression in which the call site is located. Since there is no further processing of the result of the call, a tail call can be optimized so that when the called function returns, it returns to the caller of the call site directly. This optimization allows the compiler to destroy the current stack frame before performing the call. This results in an implementation that uses constant stack space when iteration is expressed using tail-recursive function calls.

The optimization of tail calls is done by generating jump instructions instead of call instructions whenever a call appears in tail position. So instead of pushing the argument on top of the call stack and performing a call, the argument of the current function is overwritten with the actual parameter and a jump to the target function is generated. In order to manage the stack properly, it is necessary to destroy the current stack frame after evaluating the function and argument expressions of the call, and before jumping to the function denoted by the function expression.

With the compilation scheme described in the previous section, it is easy to discover possibilities for this optimization: it is applicable whenever a call expression is generated with a control linkage descriptor `return`.

### 3.9 Compilation of Mutually Recursive Functions

The compilation of recursive functions requires solving a problem we have not encountered when compiling simpler expressions: the compilation of forward ref-

```

1   letrec
2     odd = \ x -> if x = 0
3         then 0
4         else even (x - 1)
5         fi
6     even = \ x -> if x = 0
7         then 1
8         else odd (x - 1)
9         fi
10    start = \ y -> even y
11  in
12    start 42
13  end

```

Figure 3.5: Odd/even example

erences, which requires generating code using addresses of code that has not yet been compiled. This problem is known from assemblers, which need to solve a similar problem for handling forward jumps. We use a similar technique as employed in assemblers: creating a list of forward references during code generation and fixing up these references in a simple second pass [28].

Recall that recursive (self-recursive or mutually recursive) functions are defined using `letrec`-expressions in the source language. Consider the `letrec`-expression in Figure 3.5, which will be used as a running example in the explanation of the compilation of recursive functions. This expression defines two mutually recursive functions `odd` and `even`, which test whether their argument value is odd or even, respectively. On success, the functions return 1 and 0 otherwise. Both functions compute their results by testing for zero and either returning 0 or 1, or by calling the other function with a decremented parameter. The third function, `start`, will illustrate how multiple references to functions are handled and simply calls the function `even` with its parameter. (In a fully optimizing compiler, the function `start` would disappear because of inlining.) The body of the `letrec`-expression calls `start` with the argument 42.

In the following, the steps involved in compiling recursive functions will be described by studying the machine code generated. First, the code for the `letrec`-expression is discussed and then we will have a closer look at the generated code for the recursive functions. Using these as examples, the technique for handling forward references will be discussed in more detail. Please note that for the example in this section the tail-call optimization was switched off in order to make the generated code more readable.

Compilation of the example `letrec`-expression results in the assembler code shown in Figure 3.6. For better readability, lines which perform a particular task are grouped in shaded boxes. The numbers in the upper-right corners of the boxes

### 3.9 Compilation of Mutually Recursive Functions

1	push %ebp	1
2	mov %esp,%ebp	
3	sub \$0x18,%esp	
4	xor %eax,%eax	2
5	mov %eax,0xffffffff0(%ebp)	
6	mov %eax,0xfffffec(%ebp)	
7	mov %eax,0xfffffe8(%ebp)	
8	lea 0xfffffec(%ebp),%eax	3
9	push %eax	
10	push \$0x1	
11	push \$0x8051700	
12	call 0x804e600	
13	add \$0xc,%esp	
14	mov %eax,0xffffffc(%ebp)	
15	lea 0xffffffff0(%ebp),%eax	4
16	push %eax	
17	push \$0x1	
18	push \$0x8051598	
19	call 0x804e600	
20	add \$0xc,%esp	
21	mov %eax,0xffffff8(%ebp)	
22	lea 0xfffffec(%ebp),%eax	5
23	push %eax	
24	push \$0x1	
25	push \$0x8051408	
26	call 0x804e600	
27	add \$0xc,%esp	
28	mov %eax,0xffffff4(%ebp)	
29	pushl 0xffffffc(%ebp)	6
30	pushl 0xffffffff0(%ebp)	
31	call 0x804ea72	
32	add \$0x8,%esp	
33	pushl 0xffffff8(%ebp)	7
34	pushl 0xfffffec(%ebp)	
35	call 0x804ea72	
36	add \$0x8,%esp	8
37	pushl 0xffffff4(%ebp)	
38	pushl 0xfffffe8(%ebp)	
39	call 0x804ea72	
40	add \$0x8,%esp	
41	add \$0xc,%esp	
42	push \$0x2a	9
43	mov 0xffffff4(%ebp),%eax	
44	call %eax	
45	add \$0x4,%esp	10
46	leave	
47	ret	

**Figure 3.6:** Assembler code for the example letrec-expression.

will be used as references.

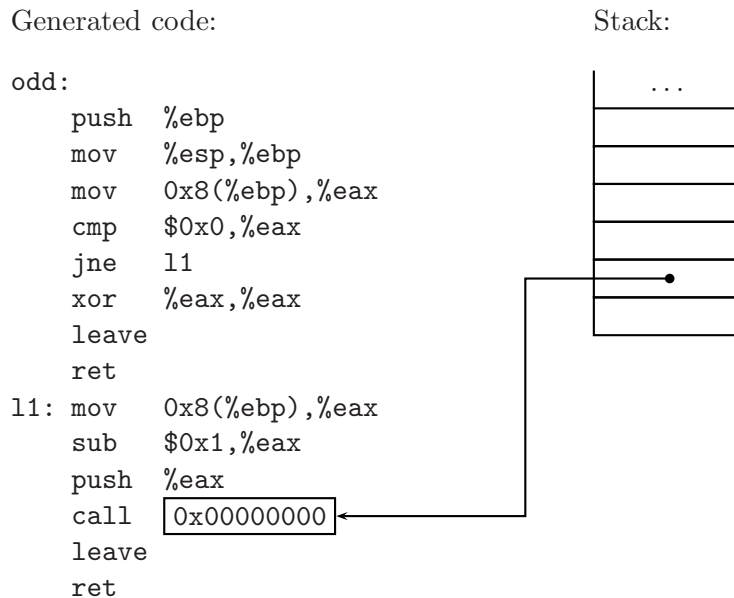
Boxes 1 and 10 set up and remove a stack frame for the expression, respectively. This is necessary because the expression allocates local variables. Box 1 additionally reserves space for six words on the run-time stack: three words are reserved for the local variables which will hold the addresses of the functions `odd`, `even` and `start`, and three words for storing information required for compiling mutually recursive functions, as will be explained below. Box 2 initializes the information variables by setting them to zero.

The next three boxes (3, 4 and 5) are responsible for compiling the functions `odd`, `even` and `start`. For each function, compilation starts by loading the addresses of the auxiliary variables mentioned above (the variables correspond to the free variables of the functions) and pushing them onto the stack, followed by an indicator that the address is not the value of the free variable, but instead a pointer to a location where information about recursive definitions is stored. Then a pointer to the abstract syntax tree of the function to be compiled is pushed and the code generator is called, which leaves a pointer to the generated code in register `eax`. From there, the function addresses are stored into their respective local variable locations.

At this time, all functions have been compiled and their addresses are located in local variables. Additionally, all uses of the functions in the machine code are put on linked lists. For each function, there exists one linked list where the head of the list is stored in one of the additional local variables whose addresses have been passed to the code generation function in boxes 3 and 4. In order to complete code generation, the addresses now need to be patched into the locations linked by the lists. This is done by calling a function in the run-time system (written in C), which receives a function address and the head of the linked list of uses of this function as parameters (boxes 6, 7 and 8). The function now patches the real address into the machine code at each instruction on the list, adjusting it when necessary (for example for `call` or `jump` instructions relative to the program counter).

The last step to be considered is evaluating the `letrec`'s body expression. This is done in box 9, where the argument is pushed onto the stack, followed by fetching the address of function `start` from its stack location and indirectly calling it.

The steps taken by the compilation process are illustrated more detailed in Figures 3.7 to 3.9. Figure 3.7 shows the machine code and run-time stack after the first function is compiled. The function `odd` contains a call to the function `even`, which has not yet been compiled and whose address is unknown (as we do not know at the time of generating the `call` instruction how long the machine code of the function `odd` will be eventually). Therefore, no real address is compiled into the `call` instruction, but instead the address contained in the list head for function `even` is placed in the instruction stream, and the address of this instruction is stored into the list head. At this point, this address is zero, because the reference to the function which is currently compiled is the first one. By repeating this process, the list of all uses of `even`'s address will be threaded through the machine



**Figure 3.7:** After compiling `odd`. The single reference to `even` is pointed to by the stack slot for these references.

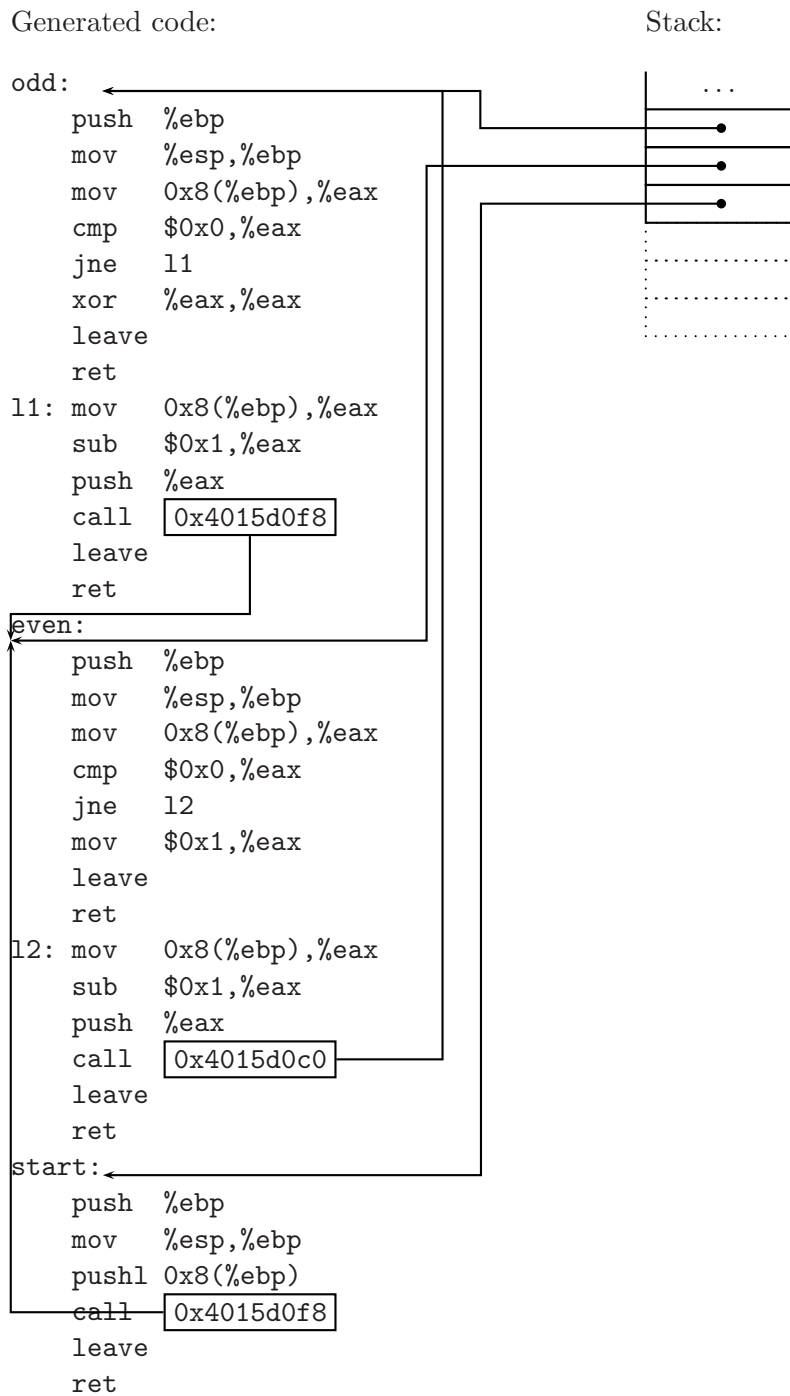
code.

Figure 3.8 shows the state after all three functions have been compiled and their addresses (which are now known, of course) have been stored into the two local variables. Since all functions in the mutually recursive set defined by the `letrec`-expression have been compiled, their addresses can be patched into the machine code. This is done by traversing, for each defined function, the list of its uses and storing the function address (possibly adjusted) into the machine code. Whether an adjustment is needed, and which, can be determined by examining the instruction's operation code which is located immediately before the stored address. As mentioned above, adjustments are needed for PC-relative addressing, such as for calls. Other uses of the address, like passing it to a function or storing it into a data structure, does not require any adjustment.

Figure 3.9 shows the final state after code generation, just before evaluation of the `letrec`-body begins. All references have been resolved and the function addresses are available in local variables, so that the body of the `letrec`-expressions may reference them. The auxiliary variables used for storing the relocation list heads are not needed anymore and can be deallocated (as indicated by dotting their outlines).

The main advantage of our approach to compiling mutually recursive functions is that the resulting code is quite efficient. The compilation technique used is also fast, since it is linear in the maximum of the number of functions and the number of mutually recursive function calls.





**Figure 3.9:** After compiling all functions and patching the call instructions. Note that at this point, all function calls are efficient direct calls. The dotted stack slots are no longer needed.

### 3.10 Optimizations

The only optimizations currently implemented are constant folding of arithmetic expressions, compile-time evaluation of conditional expressions, deferred stack pointer restoring and tail call optimization. Constant folding performs the addition, subtraction or multiplication of compile-time constants. Folding of conditionals means that when a function is compiled and a conditional depends on the value of one of its free variables or other constants, the value of the variable is substituted in the condition and a constant folder will simplify it. If the result is a constant, the conditional is replaced by one of its branches, depending on the value of the constant. Since this optimization reduces the size of the function, it both reduces compilation time and code size in addition to the removed conditional test and branch instructions.

Normally, the stack pointer needs to be adjusted after each call. It is possible to combine two or more of these adjustments into one by deferring them until necessary. This incurs a bit waste of stack space, but this can be bounded by a constant.

The optimization of tail calls has already been discussed in Section 3.8 above.

### 3.11 Limitations

As already shown, the code generator has a lot of limitations. Some are due to the fact that it is still in an early development stage, such as missing primitives, missing optimizations or lack of a garbage collector. One important improvement would be the introduction of multi-argument abstractions and applications. Simulating them with currying introduces unacceptable overhead because of repeated code generation. Other limitations are of general nature: expensive interprocedural analyses and transformations require a more global view of the program than the simple one-function-at-a-time model currently implemented. In future work, we plan to lift this limitation.

The simple structure of the compiler, together with the input format (abstract syntax trees without any annotations) puts other limitations on the resulting code. It is planned to add a more sophisticated pre-compiler which transforms the program into some suitable canonical representation (such as continuation-passing style (CPS) [2] or  $\lambda$ -normal form (ANF) [10]) and adds static annotations, thereby shifting work from the on-line compiler to the pre-compiler.

Nevertheless, in spite of these omissions and limitations, the performance of the compiler as well as the generated code is quite competitive, as will be shown by presenting experimental evidence for simple benchmark programs in the next chapter.

## 4 Evaluation

In order to judge the practicality and usefulness of the closure implementation approach described in the last chapter, a series of experiments was conducted. The example programs chosen were written for testing the implementation technique and are designed to test the time-critical parts of the implementation, mainly closure creation. The test programs have then additionally been translated into several other functional programming languages compiled using modern state-of-the-art compilers. For comparison, the benchmarks have also been translated to C and measured. The results of these tests are compared to our dynamic compilation approach.

### 4.1 Benchmark Programs

Table 4.1 summarizes the benchmark programs. The full source code of the examples appears in Appendix A. `Iter` is a tail-recursive loop which is executed  $10^9$  times. The program `citer` is also a simple loop, expressed tail-recursively, but with two arguments: the loop count and an accumulating parameter which counts the number of iterations. The loop is performed  $10^6$  times. Because our implementation does not support multiple-argument functions or data structures, this loop is expressed using a curried function. This results, of course, in the construction of one million closures, which are all generated at run-time in our implementation. This program was written to stress-test the run-time code generator. `Odd-even` calculates whether the number  $10^9$  is odd or even. This is expressed with two mutually recursive functions, `odd` and `even`, which call themselves tail-recursively. The fourth benchmark, `fib`, calculates the Fibonacci function for the value 42. The benchmark `nested` consists of two nested loops, expressed as nested tail-recursive functions where the inner function uses the parameter of the outer function and therefore needs to build a closure. This test was made to test repeated closure creation in a more moderate way than the `citer` benchmark.

Of course, these benchmark programs are not realistic applications, so the overall performance of the language implementation cannot be determined from the numbers presented in the following. Nevertheless, the measurements should give an indication of how efficiently run-time-generated code can perform, even if created by a relatively simple dynamic compiler.

In order to get fair results, each benchmark was run at least 3 times, taking the lowest execution time for the results presented. This was made to make sure that all relevant code was swapped in.

**Table 4.1:** Summary of benchmark programs

Name	Description
iter	Loops $10^9$ times
citer	Counts from 0 to $10^6$
odd-even	Calculates whether $10^9$ is odd or even
fib	Calculates the Fibonacci number for 42
nested	Two nested loops, inner loop depending on outer variable

## 4.2 Platform

All experiments have been performed on an Intel Pentium 4 CPU, running at 2.4GHz with 512 KB Cache and 500 MByte main memory. The machine ran using the Debian GNU/Linux operating system version 3.1 and was unloaded.

For the dynamic compiler, run times have been measured by using the machine instruction `rdtsc`, which delivers the number of clock cycles since the processor was powered up as an unsigned 64-bit quantity. This instruction allows very precise measurements, which is necessary because the compilation times for most of the benchmark programs are very short.

The results reported for other language implementations in Section 4.4 have been measured using the standard `time` command, which has much lower accuracy. Therefore, we have tried to write the benchmarks so that they run at least half a second, where possible (see below for details).

## 4.3 Performance Results

This section presents the results of the implementation outlined in the previous chapter on the benchmark programs described above. The results are shown in Table 4.2. For each benchmark program, the total time is reported in the second column and the time spent in the run-time code generator in the third column. The last column contains the size of the generated machine code (not including administrative overhead like maintaining a list of compiled functions). Times are reported in seconds, code size in bytes.

The figures show that code generation time is negligible for programs that do not create many closures, like `odd-even` or `fib`. On the other hand, programs that require many closures to be built, like the curried loop in benchmark `citer`, generate specialized functions for each loop iteration! The running time is then nearly completely dominated by the time required for code generation. Since our system does not yet include a garbage collector, the system runs out of space for much larger numbers than the tested million iterations. For the `nested` benchmark, the times suggest that run-time generation of closures is realistic if the frequency of compilation is not too high. In this benchmark, 1000 closures (one for each itera-

**Table 4.2:** Results of benchmark programs.

Name	Total (s)	Codegen (s)	Code size (bytes)
iter	2.516	0.00001	109
citer	3.353	3.06646	44000163
odd-even	2.544	0.00002	244
fib	6.842	0.00001	132
nested	2.744	0.00255	41193

tion of the outer loop) are created, and each of these is run one million times. We expect that in cases where the inner loop depends on some outer loop variables and the loop is run sufficiently often, optimizations resulting from treating the outer variables as constants can give substantial performance improvement. Code like this occurs, for example, in matrix calculations.

The table indicates that performance is reasonable, and the next section shows how these result relate to other, mature, implementations of functional languages. Note that the code size is quite small for all benchmarks (except `citer`) so that the memory traffic for writing the generated code into the code cache is minimal.

## 4.4 Comparison with other Implementations

Since we wish to compare the performance of our implementation with existing functional language compilers, which use traditional closure implementation techniques, we have translated the benchmark programs into Standard ML [22], Haskell [26], Scheme [17] and Opal [25]. Haskell has been tested in three versions: without (Haskell-a) and with type annotations (Haskell-b), which affects the representation of integers, and with type annotations and manual overflow checks for addition and subtraction (Haskell-c). Haskell-a uses unlimited precision integers whereas Haskell-b and Haskell-c use machine integers.

Additionally, measurements have been done for C versions of the benchmark programs. The results for the C programs are not really comparable to the functional language implementations, but they provide a bottom line for how fast these functions can execute with a state-of-the-art imperative compiler. The source code of the benchmark programs is contained in Appendix A.

The languages and language implementations used for the comparison are shown in Table 4.4. For each of the languages, a recent version of an efficient implementation has been chosen. According to claims made by the authors of the implementations or in programmer folklore, the compilers for Scheme and Standard ML are amongst the most efficient implementations of these languages.

The command line options used for compiling the benchmarks are summarized in Table 4.3. Please note that only basic optimization switches have been added

**Table 4.3:** Command line parameters.

Compiler	Options
gcc	-O3 -fomit-frame-pointer -o <i>outfile file</i>
ghc	-O2 -o <i>outfile --make file</i>
mlton	-output <i>outfile file</i>
stalin	-copt -O3 -On <i>file</i>
ocs	-top <i>structure main opt=full</i>

**Table 4.4:** Languages and language implementations

Language	Compiler	Version
C	GNU Compiler Collection [11, 33]	4.0.2
Scheme	Stalin [31, 30]	0.10
Haskell-a	Glasgow Haskell Compiler [12, 13] (no annotations)	6.4.1
Haskell-b	Glasgow Haskell Compiler (annotations)	6.4.1
Haskell-c	Glasgow Haskell Compiler (annotations+checks)	6.4.1
Standard ML	MLton [23]	20041109
Opal	OCS [24]	2.3i

to the command line, we have not taken the effort to squeeze the last bit of performance out of each implementation. The reason is that the numbers presented below are only meant as an orientation, and not as a definite performance assessment.

Note that the languages and implementations used for the tests cannot be compared directly because of semantic differences between the various languages. Below in Section 4.4.2 we will comment on several of the differences and how they affect the outcome of the measurements.

#### 4.4.1 Results

The results of the measurements for the other implementations are presented in Tables 4.5 to 4.9. Each table contains the result for one benchmark program, one line per implementation language. The first column contains the language, the second column the total run time and the third the ratio compared to the results of our implementation (Table 4.2). A ratio less than one means that the language of that row was faster, a ratio greater than one means that our implementation was faster. The first line of each table (labeled *Reference*) repeats the results of Table 4.2 for convenience.

Note that the times reported for these languages do not include any compilation times, since all systems used are off-line compilers.

**Table 4.5:** Times for `iter`

Language	Time (s)	Ratio
Reference	2.516	1.000
C	0.630	0.250
Scheme	0.631	0.251
Haskell-a	73.023	29.023
Haskell-b	3.196	1.270
Haskell-c	11.221	4.460
Standard ML	1.676	0.666
Opal	12.759	5.071

**Table 4.6:** Times for `citer`

Language	Time (s)	Ratio
Reference	3.353	1.000
C	0.002	–
Scheme	0.003	–
Haskell-a	0.109	–
Haskell-b	0.006	–
Haskell-c	0.014	–
Standard ML	0.003	–
Opal	0.028	–

For the `iter` benchmark, we can see that our code generator produces reasonable code for such simple tasks as tail-call optimization and simple arithmetic. The C implementation runs much faster (as was to be expected), but our implementation competes well with the Haskell and Standard ML implementations, and is faster than the Opal version. The Scheme version is nearly as fast as the C version, but see Section 4.4.2 for remarks on the Scheme implementation used.

In the comparison for the `citer` benchmark, the numbers are very small, because of the limited accuracy of the `time` command. These numbers are included for completeness, not to draw any definitive conclusions from. Therefore, we have not included the ratios to our dynamic compiler for this benchmark program. This test shows how code generation times can affect the overall performance significantly when many closures are created.

The results for `odd-even` are similar to that of `iter`, since nearly all implementations perform tail-call optimizations for mutually recursive functions, as our system does. The only exception is Opal, which compiles the unannotated program into code which aborts with a stack overflow because the tail calls are not optimized into jumps. In order to make the measurement possible, the program

**Table 4.7:** Times for `odd-even`

Language	Time	Ratio
Reference	2.544	1.000
C	0.560	0.220
Scheme	0.556	0.219
Haskell-a	55.327	21.748
Haskell-b	1.875	0.737
Haskell-c	6.718	2.641
Standard ML	1.043	0.410
Opal	10.844	4.263

**Table 4.8:** Times for `fib`

Language	Time	Ratio
Reference	6.842	1.000
C	3.302	0.483
Scheme	3.316	0.485
Haskell-a	116.221	16.986
Haskell-b	8.994	1.315
Haskell-c	18.933	2.767
Standard ML	7.767	1.135
Opal	23.732	3.469

needs an annotation that instructs the compiler to inline (“unfold”) the function `even`. After inlining the function once, the compiler recognizes the self-recursion of function `odd` and generates iterative code. Note that it was necessary to remove the mutual recursion for the C version so that this benchmark could be run. Otherwise, the program aborts with a stack overflow, like the Opal version.

The most interesting fact about the `fib` benchmark is that our dynamically compiled system is faster than all languages except Scheme and C. The performance of Haskell and Standard ML is probably due to heap overflow checks and/or arithmetic overflow checks.

The results of the `nested` benchmark (see Table 4.9) for C, Scheme and Standard ML are quite similar to those of the `iter` benchmark (since both perform the same number of iterations). The result for Haskell is not very meaningful for comparison purposes, since the compiler notices that the evaluation of the inner loop is not necessary and can be optimized away. This is not true for a language like Standard ML, where arithmetic operations may throw exceptions and can therefore not get removed as dead code. For Opal, no measurements were made since the Opal language does not allow locally defined functions to be recursive.

**Table 4.9:** Times for nested

Language	Time	Ratio
Reference	2.744	1.000
C	0.629	0.229
Scheme	0.632	0.230
Haskell-a	0.002	–
Haskell-b	0.001	–
Haskell-c	0.001	–
Standard ML	1.669	0.608
Opal	–	–

#### 4.4.2 Language and Implementation Differences

Of course, the performance results presented in this section are to be interpreted carefully. The languages perform different work for similar tasks, because the language semantics differ. Details about the differences in the languages and how they affect the benchmarks are given below.

Some of the language implementations need special directives or program annotations in order to generate efficient code. In this case, we have tested both the unannotated and annotated versions.

**C.** The C versions performed very good for nearly all the benchmarks. This is of course due to the fact that the benchmarks exhibit extremely simple control flow and only basic arithmetic operations, both for which the C compiler optimizes very well. It should be kept in mind, though, that C lacks many of the features supported by all the other languages.

During the work on our prototype, version 4.0 of the GCC compiler suite was released. This version performs much better for many of the benchmarks than the old version we tested (3.3.5). The `fib` benchmark, for example, took 7.42 seconds with the older version.

Note that several of the implementations tested below work by compiling their input languages to C and then using the C compiler to generate machine code.<sup>1</sup> All these implementations have used the same compiler as the C programs, so all benefitted from the new compiler version.

**Scheme.** Scheme is dynamically typed, which requires often unnecessary type checks. Note that the Scheme implementation used does not support arbitrary precision integers and thus runs with machine integers. The implementation also does not perform overflow checks on integer arithmetic.

<sup>1</sup>Opal, Scheme and parts of the Standard ML programs are compiled using GCC.

## 4 Evaluation

Since the Scheme compiler is a whole-program compiler it can optimize away many (if not all, but we have not investigated this further) type checks, closure creations, heap allocations etc. This gives it an advantage over the other implementations, which support separate compilation (except the Standard ML compiler).

The Scheme compiler used compiles to C code, which in turn is compiled by the C compiler installed. We needed to add compiler options to pass to the C compiler, otherwise, the run times would have been much higher (up to a factor 15 for the benchmark `iter`).

**Haskell.** The Haskell programs need type annotations so that they work with machine-size integers instead of arbitrary precision integers. Without type annotations, the run times go up from a few seconds to about a minute for the simple `citer` and `odd-even` benchmarks. Also note that Haskell is the only language in the comparison which is lazy, whereas the other languages are strict. Extra care has to be taken when comparing the results because of this semantic difference. For the programs tested here, the compiler's strictness analyzer is able to optimize away all laziness. This would of course not be true for other, more complicated benchmark programs.

The Haskell compiler compiles to machine code on the platform we tested it on, which allows for more efficient calling conventions and various operations such as heap overflow checks than when compiling via C.

**Standard ML.** Since arithmetic operations in Standard ML are required to check for overflow, benchmarks using a lot of arithmetic operations perform slower than they could without such checks. The MLton compiler seems to optimize these overflow checks very well, as the resulting programs perform comparable to the C versions which do not do any overflow checking.

The Standard ML compiler used is a whole-program compiler (like the Scheme compiler), which gives it much better analysis and optimization possibilities. It also compiles to machine code instead of C, which allows for better machine-specific optimizations such as efficient overflow checking on arithmetic.

**Opal.** For Opal, the extremely poor performance for some of the benchmarks is due to the fact that simple arithmetic operations are not inlined and that arithmetic checks for overflows are done. For the manually overflow-checking Haskell programs we got similar results, which indicates that this is indeed the problem.

Like the Scheme version, Opal compiles to C which is in turn compiled to machine code by GCC, but unlike the Scheme implementation, Opal is designed for separate compilation, which only lets it perform less aggressive optimization.

### 4.4.3 Comment

The results presented in this section are of course not representative for the complete languages and language implementations, since only very small parts of the languages are tested. The results should therefore only be interpreted for orientation. Other language comparisons (for example, [15]) should be consulted for more complete and precise measurements.

## 4.5 Evaluation Conclusion

The benchmark programs used for testing the performance of the run-time code generation technique are not realistic applications. Nevertheless, we can draw several conclusions from the numbers presented in this chapter:

1. Dynamic code generation for simple arithmetic and basic control structures (conditionals, function calls) is efficiently possible, as the low compilation times show.
2. The code produced by a relatively simple compiler with nearly no optimizations is competitive with more sophisticated compilers.
3. The addition of optimizations, particularly those which exploit the treatment of free variables, is expected to improve run-time performance further. As long as these optimizations do not require too expensive analysis, the compilation times should remain low and therefore practical for on-line use.

## 5 Conclusion and Future Work

In this chapter, the work presented in this report is related to previous work in the areas of run-time code generation, multi-stage programming and partial evaluation. After that, planned future work extending the implementation presented here is discussed and we conclude.

### 5.1 Related Work

The work discussed in this section is related to ours and interconnected in several ways. On the one hand, some of the techniques are complementary to each other and are useful on their own, on the other, they depend on each other. For example, an efficient implementation of multi-stage programming languages relies on fast run-time code generation.

**Run-time Code Generation.** Similar to our approach, there exist dynamic code generation systems which can be used to programmatically generate machine code that can later be executed. Most of them are specific to individual implementations, such as various virtual machines for object-oriented languages like Self [5], Smalltalk [6] or Java [1, 4, 34]. Other systems concentrate on the task of code generation and are not bound to a specific language or implementation [8]. In other work, efficient run-time specialization has been realized with one of the following techniques. In the template-based approach, program code is pre-compiled into so-called templates. Templates consist of machine code for a function (or basic block, if the system works with smaller entities than functions) which contains holes. When instantiated, templates are copied and holes are patched with the constants for that particular instantiation. This approach is very fast, since it requires only a few instructions per generated machine instruction, but precludes most interesting dynamic optimizations, because of the fixed structure of templates. Sometimes, the code-templates are not generated by a compiler, but written by hand, as in the Synthesis operating system kernel [27, 21]. The second possibility is not to compile into templates, but instead into code that, when run, directly generates the target machine code [19, 20, 18]. Generating code-emitting code is also quite fast and has the advantage that optimizations can easily be incorporated. The work described in this report does not use any of the methods described, but the generation of code-generating code is an alternative we consider for future improvements.

**Partial Evaluation.** The run-time compilation of functions when their free variables are available can be considered as a form of partial evaluation. Partial evaluation transforms a program and part of its inputs into a different program (called a *residual* program), that has been specialized to the given (*early*) inputs and expects the rest of the inputs (the *late* inputs) in order to compute the final answer. This scheme can be generalized to more than two phases by partially evaluating a residual program to yield a version specialized to even more input values.

The combination of partial evaluation with suitable compilation techniques is also a promising approach [32].

**Multi-stage Programming.** In multi-stage programming languages [35], the programmer is allowed to specify the time at which expressions are created and evaluated. Expressions may contain “quoted” subexpressions which denote delayed computations. It is possible to “splice” values (already evaluated expressions) into such delayed expressions and to explicitly demand evaluation. This can be considered a means for manual specialization: expressions which are not constant over the whole time a program runs, but constant for a longer period of time can be evaluated once and spliced into some code construct that gets evaluated many times. The time savings in this case can be considerable.

## 5.2 Future Work

It is planned to implement several dynamic (that is, run-time) analysis and transformations which exploit the possibilities of run-time code generation for functional programs. A future version of the run-time compiler will be accompanied by a static pre-compiler, which annotates the abstract syntax trees with information that is important but expensive to compute. Analysis similar to binding time analysis could be able to identify relationships between parameters and the code influenced by their values. This will result in a system similar to Tempo, a specializer for the C programming language, but tailored to the needs of a declarative language.

On the theoretical side, the development of a formal foundation of dynamic analysis and transformation is planned. This theory of dynamic compilation is expected to capture the currently used patterns of dynamic and adaptive compilation, which are almost completely informal, so that they can be analyzed and developed formally.

It should be noted that the implementation method described in this report is targeted at call-by-value functional languages. Call-by-name or call-by-need languages create much more closures during run-time, because nearly every evaluation is packaged into a closure in order to delay computation. We have not investigated this fact further, but expect that the compilation overhead would be too high in this scenario to make run-time code generation practical. Further investigation is needed to confirm this.

### 5.3 Conclusion

We have presented an implementation of a functional programming language that uses run-time code generation to realize closures. The implementation was described in some detail, with emphasis on the constrained environment in which a run-time compiler works and how these limitations were considered in the implementation. Several optimizations possible within this framework have been described and the performance on example programs has been measured. The performance results suggest that the presented implementation technique is not only practical, but promises to improve when more analysis and transformations will be included. When compared to other state-of-the-art compilers, our dynamic compiler performs well, considering that it is still a prototype with probably some potential for tuning.

# A Example Program Source Code

## A.1 Benchmark Programs

These are the programs used with the run-time code generation closure implementation technique described in Chapter 3. The implementation has no concrete syntax at the moment, so the programs are presented in a pretty-printed output of the abstract syntax used internally.

### Iter

```
1   letrec
2     iter = (\ x -> if (x = 0)
3             then 23
4             else (iter (x - 1))
5             fi)
6   in
7     (iter 1000000000)
8   end
```

### Citer

```
1   letrec
2     citer = (\ x -> (\ n -> if (x = 0)
3                            then n
4                            else ((citer (x - 1)) (n + 1))
5                            fi))
6   in
7     ((citer 1000000) 0)
8   end
```

### Odd-even

```
1   letrec
2     odd = (\ x -> if (x = 0)
3             then 0
4             else (even (x - 1))
5             fi)
6     even = (\ x -> if (x = 0)
7             then 1
8             else (odd (x - 1))
9             fi)
10    start = (\ y -> (even y))
11  in
```

## A Example Program Source Code

```
12      (start 1000000000)
13      end
```

### Fib

```
1      letrec
2          fib = (\ x -> if (x < 2)
3                      then 1
4                      else ((fib (x - 1)) + (fib (x - 2)))
5                      fi)
6      in
7          (fib 42)
8      end
```

### Nested

```
1      letrec
2          outer = (\ n -> if (n = 0)
3                      then 1
4                      else letrec
5                          inner = (\ m -> if (m = 0)
6                                          then n
7                                          else (inner (m - 1))
8                                          fi)
9                      in
10                     let
11                         x = (inner 1000000)
12                     in
13                         (outer (n - 1))
14                     end
15                 end
16             fi)
17      in
18          (outer 1000)
19      end
```

## A.2 Opal Programs

Note that the Opal programs need to output the results, because the compiler would otherwise optimize away everything.

### Iter

```

1  SIGNATURE Iter
2  IMPORT  Com[void]      ONLY com
3          Void          ONLY void
4  FUN main : com[void]  -- top level command

1  IMPLEMENTATION Iter
2  IMPORT  Nat            ONLY nat ! 0 1 23 - + = %
3          Com            COMPLETELY
4          BasicIO        COMPLETELY
5
6  DEF main ==
7    writeLine(iter("1000000000!"))
8
9  FUN iter: nat -> nat
10 DEF iter(x) ==
11   IF x = 0 THEN 23
12   ELSE iter(x - 1)
13   FI

```

### Citer

```

1  SIGNATURE Citer
2  IMPORT  Com[void]      ONLY com
3          Void          ONLY void
4  FUN main : com[void]  -- top level command

1  IMPLEMENTATION Citer
2  IMPORT  Nat            ONLY nat ! 0 1 - + = %
3          Com            COMPLETELY
4          BasicIO        COMPLETELY
5
6  DEF main ==
7    writeLine(iter("1000000!", 0))
8
9  FUN iter: nat ** nat -> nat
10 DEF iter(x, n) ==
11   IF x = 0 THEN n
12   ELSE iter(x - 1, n + 1)
13   FI

```

### Odd-even

```

1  SIGNATURE OddEven

```

## A Example Program Source Code

```
2  IMPORT Com[void]      ONLY com
3      Void              ONLY void
4  FUN main : com[void]  -- top level command

1  IMPLEMENTATION OddEven
2  IMPORT Nat            ONLY nat ! 0 1 - = %
3      Com              COMPLETELY
4      BasicIO         COMPLETELY
5
6  DEF main ==
7      writeLine(start("1000000000!"))
8
9  FUN start: nat -> nat
10 DEF start(x) == even(x)
11
12 /$ unfold [even] $/
13
14 FUN even: nat -> nat
15 DEF even(x) ==
16     IF x = 0 THEN 1
17     ELSE odd(x - 1)
18     FI
19
20 FUN odd: nat -> nat
21 DEF odd(x) ==
22     IF x = 0 THEN 0
23     ELSE even(x - 1)
24     FI
```

### Fib

```
1  SIGNATURE Fib
2  IMPORT Com[void]      ONLY com
3      Void              ONLY void
4      Nat              ONLY nat
5  FUN main : com[void]  -- top level command

1  IMPLEMENTATION Fib
2
3  IMPORT Denotation    ONLY ++
4      Nat              ONLY nat ! 1 2 - + < %
5      NatConv         ONLY ‘
6      Com              COMPLETELY
7      BasicIO         COMPLETELY
8
9  DEF main ==
10     writeLine("done" ++ ‘(fib("42!"))’)
11
12
13  FUN fib : nat -> nat
```

```

14 DEF fib(n) ==
15     IF n < 2 THEN 1
16     ELSE fib(n - 1) + fib (n - 2)
17     FI

```

### Nested

No Opal version, since Opal does not permit local recursive functions.

## A.3 Standard ML Programs

### Iter

```

1 fun iter x = if x = 0 then 23 else iter (x - 1)
2 val _ = iter 1000000000

```

### Citer

```

1 fun iter x n = if x = 0 then n else iter (x - 1) (n + 1)
2 val _ = iter 1000000 0

```

### Odd-even

```

1 fun odd x = if x = 0 then 0 else even (x - 1)
2 and even x = if x = 0 then 1 else odd (x - 1)
3 and start x = even x
4 val _ = (start 1000000000)

```

### Fib

```

1 fun fib x = if x < 2 then 1 else fib (x - 1) + fib (x - 2)
2 val _ = (fib 42)
3

```

### Nested

```

1 fun outer n = if n = 0 then 1 else
2   let fun inner m = if m = 0 then n else inner (m - 1)
3       val x = inner 1000000
4   in
5     outer (n - 1)
6   end
7 val _ = (outer 1000)
8

```

## A.4 Haskell Programs

The Haskell programs all exist in three versions: without type annotations, with type annotations (restricting induction variables to type `Int`, which is much more efficient than the standard unbounded integer type `Integer`), and with type annotations and bounds-checking arithmetic operations. Since the versions are very similar to each other, we only give three versions for the first benchmark, `Iter`.

The Haskell programs also have to output their results in order to prevent over-optimization.

### `Iter`

#### Version without annotations

```
1  module Main where
2
3  main = putStrLn ("Result: " ++ show (iter 1000000000))
4
5  iter 0 = 23
6  iter x = iter (x - 1)
```

#### Version with type annotations

```
1  module Main where
2
3  main = putStrLn ("Result: " ++ show (iter 1000000000))
4
5  iter :: Int -> Int
6  iter 0 = 23
7  iter x = iter (x - 1)
```

#### Version with type annotations and bounds-checking arithmetic

```
1  module Main where
2
3  main = putStrLn ("Result: " ++ show (iter 1000000000))
4
5  iter :: Int -> Int
6  iter 0 = 23
7  iter x = iter (x `minus` 1)
8
9  minus :: Int -> Int -> Int
10 minus x y = if x < (-2147483648) + y then error "overflow" else x - y
```

### `Citer`

```
1  module Main where
2
3  main = putStrLn ("Result: " ++ show (iter 1000000 0))
4
5  iter 0 n = n
6  iter x n = iter (x - 1) (n + 1)
```

### Odd-even

```
1  module Main where
2
3  import Prelude hiding (even, odd)
4
5  main = putStrLn ("Result: " ++ show (start 1000000000))
6
7  start x = even x
8
9  odd 0 = 0
10 odd x = even (x - 1)
11
12 even 0 = 1
13 even x = odd (x - 1)
```

### Fib

```
1  module Main where
2
3  main = putStrLn ("Result: " ++ show (fib 42))
4
5  fib 0 = 1
6  fib 1 = 1
7  fib x = fib (x - 1) + fib (x - 2)
8
```

### Nested

```
1  module Main where
2
3  main = putStrLn ("Result: " ++ show (outer 1000))
4
5  outer 0 = 1
6  outer n = let inner 0 = n
7             inner m = inner (m - 1)
8             x = inner 1000000
9             in outer (n - 1)
10
```

## A.5 Scheme Programs

The zero at the end of the programs is required for the used Scheme implementation, because the value of the last expression in a file is used as the exit code of the program.

### Iter

```
1 (letrec ((iter (lambda (x)
2               (if (= x 0)
3                   23
4                   (iter (- x 1))))))
5   (display (iter 1000000000))
6   (newline))
7 0
```

### Citer

```
1 (letrec ((iter (lambda (x n)
2               (if (= x 0)
3                   n
4                   (iter (- x 1) (+ n 1))))))
5   (display (iter 1000000 0))
6   (newline))
7 0
```

### Odd-even

```
1 (letrec ((odd (lambda (x)
2                (if (= x 0)
3                    0
4                    (even (- x 1))))))
5   (even (lambda (x)
6          (if (= x 0)
7              1
8              (odd (- x 1))))))
9   (start (lambda (y) (even y))))
10  (display (start 1000000000))
11  (newline))
12 0
```

### Fib

```
1 (letrec ((fib (lambda (x)
2                (if (< x 2)
3                    1
4                    (+ (fib (- x 1)) (fib (- x 2)))))))
5   (display (fib 42))
6   (newline))
7 0
```

**Nested**

```

1  (letrec ((outer (lambda (n)
2                    (if (= n 0)
3                        1
4                        (letrec ((inner (lambda (m)
5                                    (if (= m 0)
6                                        n
7                                        (inner (- m 1))))))
8                                (let ((x (inner 1000000)))
9                                    (outer (- n 1))))))))))
10 (display (outer 1000))
11 (newline))
12 0

```

**A.6 C Programs****Iter**

```

1  #include <stdio.h>
2
3  int iter (int x) {
4      while (x > 0) {
5          x = x - 1;
6      }
7      return 23;
8  }
9
10 int
11 main (int argc, char * argv[]) {
12     printf ("Result: %d\n", iter (1000000000));
13     return 0;
14 }

```

**Citer**

```

1  #include <stdio.h>
2
3  int iter (int x, int n) {
4      while (x > 0) {
5          x = x - 1;
6          n = n + 1;
7      }
8      return n;
9  }
10
11 int
12 main (int argc, char * argv[]) {
13     printf ("Result: %d\n", iter (1000000, 0));
14     return 0;

```

## A Example Program Source Code

```
15 }
```

### Odd-even

```
1  #include <stdio.h>
2
3  int start (int x) {
4      even:
5      if (x == 0)
6          return 1;
7      else {
8          x = x - 1;
9          goto odd;
10     }
11     odd:
12     if (x == 0)
13         return 0;
14     else {
15         x = x - 1;
16         goto even;
17     }
18 }
19
20 int
21 main (int argc, char * argv[]) {
22     printf ("Result: %d\n", start (1000000000));
23     return 0;
24 }
```

### Fib

```
1  #include <stdio.h>
2
3  int
4  fib (int n)
5  {
6      if (n < 2)
7          return 1;
8      else
9          return fib (n - 1) + fib (n - 2);
10 }
11
12 int
13 main (int argc, char * argv[])
14 {
15     printf ("%d\n", fib (42));
16     return 0;
17 }
```

## Nested

```
1  #include <stdio.h>
2
3  int
4  outer (int n)
5  {
6      while (n > 0)
7          {
8              int m = 1000000;
9              while (m > 0)
10                 {
11                     m--;
12                 }
13             n--;
14         }
15     return 1;
16 }
17
18 int
19 main (int argc, char * argv[])
20 {
21     printf ("%d\n", outer (1000));
22     return 0;
23 }
```

## Bibliography

- [1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 280–290. ACM Press, 1998.
- [2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [3] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.
- [4] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.
- [5] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 146–160. ACM Press, 1989.
- [6] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM Press, 1984.
- [7] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-driven code generation. Technical Report 302, Indiana University Computer Science Department, February 1990.
- [8] Dawson R. Engler. vcode: A retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.
- [9] Marc Feeley and Guy Lapalme. Closure generation based on viewing lambda as epsilon plus compile. *Journal of Computer Languages*, 17(4), 1992.
- [10] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.
- [11] The GCC Developers. GNU Compiler Collection Homepage. Available on the World Wide Web: <http://www.gnu.org/gcc>, 2005. Last visited: 2005-08-04.
- [12] The GHC Developers. Glasgow Haskell Compiler Homepage. Available on the World Wide Web: <http://www.haskell.org/ghc>, 2005. Last visited: 2005-08-04.

- [13] The GHC Team. The Glorious Glasgow Haskell Compilation System User’s Guide, Version 6.4. World Wide Web, 2005. Available from: [http://www.haskell.org/ghc/docs/latest/html/users\\_guide/users-guide.html](http://www.haskell.org/ghc/docs/latest/html/users_guide/users-guide.html), last visited: 2005-04-11.
- [14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java™ Language Specification*. Addison-Wesley, 2nd edition, June 2000.
- [15] Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. van Groningen, Kevin Hammond, Bogumil Hausman, Melody Y. Ivory, Richard E. Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D. Lins, Sandra Loosemore, Niklas Røjemo, Manuel Serrano, Jean-Pierre Talpin, John Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and Peter Wentworth. Benchmarking implementations of functional languages with “pseudoknot”, a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.
- [16] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of the International Conference of Functional programming languages and computer architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
- [17] Richard Kelsey, William Clinger, Jonathan Rees, et al. Revised<sup>5</sup> report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(6):26–76, September 1998.
- [18] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 137–148. ACM Press, 1996.
- [19] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, June 1994.
- [20] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSS)*, 1996.
- [21] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, 1992.
- [22] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997. Revised edition.
- [23] The MLton Developers. MLton Standard ML Compiler Homepage. Available on the World Wide Web: <http://mlton.org>, 2005. Last visited: 2005-08-04.
- [24] The Opal Group. Opal Project Homepage. Available on the World Wide Web: <http://uebb.cs.tu-berlin.de/~opal>, 2004. Last visited: 2005-08-04.
- [25] Peter Pepper. *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer-Verlag, 2nd edition, 2003.
- [26] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, April 2003. Also available from: <http://www.haskell.org/definition/>, last visited: 2003-06-23.
- [27] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

## Bibliography

- [28] David Salomon. *Assemblers and Loaders*. Ellis Horwood Series in Computers and Their Applications. Ellis Horwood Ltd, 1993.
- [29] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, January 2000.
- [30] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, December 1999.
- [31] Jeffrey Mark Siskind. Jeffrey Mark Siskind’s Software. Available on the World Wide Web: <http://www.ece.purdue.edu/~qobi/software.html>, 2005. Last visited: 2005-08-04.
- [32] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–225, 1997.
- [33] Richard M. Stallman and the GCC Developer Community. *Using GCC: The GNU Compiler Collection Reference Manual*. GNU Press, 2003.
- [34] Sun Microsystems. The Java HotSpot Virtual Machine, v1.4.1, d2. Available on the World Wide Web: [http://java.sun.com/products/hotspot/docs/whitepaper/Java\\_HotSpot\\_WP\\_Final\\_4\\_30\\_01.html](http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html), September 2002.
- [35] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999.