The Categorical Abstract Machine: Basics and Enhancements

Ralf Hinze
Universität Bonn
Institut für Informatik III
Römerstraße 164
53177 Bonn 1
Germany

e-mail: `ralf.informatik.uni-bonn.de`

August 6, 1999

Abstract

The categorical abstract machine (CAM) is a well-known environment-based architecture for implementing strict functional languages. Existing literature about the CAM presumes more or less a category-theoretical background. Although we appreciate the firm grounds on which the CAM is built, we try to motivate the components of the CAM from an implementor's point of view. This undertaking is facilitated by the conceptional simplicity of the CAM and its proximity to conventional stack architectures.

We describe the translation of an augmented λ-calculus into CAM instructions. The basic compilation scheme has its didactic virtues but for a real implementation many improvements are necessary (and applicable). A first improvement concerns the treatment of subexpressions which do not access the environment. Further improvements are achieved via simple local transformations of the generated CAM code—this is one of the novel features presented in this report. They include β-reduction at compile-time, improving calls to local functions, and last call optimization.

1 Introduction

Since we want to introduce a particular implementation technique for functional programming languages, we should first say

1. what we mean by "functional" programming languages and
2. why such languages require a special implementation technique.

The first question is easy to answer. To deserve the attribute "functional", a programming language should treat functions as *first-class citizens*, i.e., functions should not be subject to any restrictions. In a functional language there are expressions to denote functions and functions can be passed as parameters or obtained as results. Without entering a discussion about the usefulness of this feature we would like to name one of its advantages. First-class functional values allow the unrestricted application of the *abstraction principle*. A sorting function can be made more widely usable by abstracting from a particular ordering relation (functions as parameters). We may abstract from frequently occurring recursion schemes such as applying a function to every member of a list (map) or inserting a binary operation between two adjacent list elements (fold) using higher order functions[1]. Special instances of these schemes are obtained by partially parameterizing the corresponding functions (functions as results). For a broader discussion of this topic we refer to standard textbooks about functional programming [2, 11, 13, 21, 24].

[1]The function $f : \sigma \to \tau$ is called higher order if σ or τ contains a functional type.

In this report we are only concerned with so-called *pure* functional languages, which adhere to the principle of referential transparency. This principle is a very fundamental property of mathematical reasoning and can be summarized as follows [27].

> The only thing that matters about an expression is its *value*. Moreover, in the same context the expression has the same value wherever it occurs.

Referential transparency precludes any side-effects (assignment, imperative I/O, exceptions), which in turn offers the possibility of rearranging the standard left to right order of evaluation.

Let us turn to the second question. Many imperative languages such as Pascal, which are implemented using conventional stack-based techniques, also allow functions to be passed as parameters. The crucial difference to functional languages lies in the possibility of *returning* functions as results of other functions. The following example illustrates the point.

$$add \quad = \quad \lambda x \rightarrow \lambda y \rightarrow +\ x\ y$$

If the function *add* is called with only one parameter, say *add* 3, we obtain a function which adds 3 to its argument. On a conventional stack machine the arguments of a function are pushed onto the stack prior to the call and are popped upon return. Thus, if the resulting nameless function is applied, the value of the free variable x no longer rests on the stack. Figuratively speaking, the free variable x is torn out of its context. This situation can never occur in languages where functions or procedures may only be passed as parameters.

The techniques which have been proposed to circumvent this problem can roughly be divided into two categories.

1. Graph-based implementation techniques (SK graph-reduction [30], G-machine [1, 16], TIM [10], ABC-machine [26]).

2. Environment-based implementation techniques (SECD machine [19], FAM [3], CAM [4, 5, 28]).

Despite the apparent differences, the two approaches have many things in common as a closer inspection reveals (cf. Section 5).

The idea underlying graph-based techniques is simple but radical: Since (free) variables pose problems, get rid of them. So-called bracket abstraction algorithms can be used to turn a (functional) expression into a variable free form. The origin of these algorithms must be traced back a long way in time [8]. They were rediscovered by Turner [30]. Bracket abstraction algorithms use a fixed set of combinators[2] to accomplish their task. Alternatively, one can transform a source program into a set of (super-) combinators using a technique called λ-lifting [23]. A combinator may be interpreted as a first-order rewrite rule. Thus the evaluation of an expression amounts to rewriting the corresponding combinatorial expression. In order to facilitate the sharing of expressions, graphs are used to represent combinator expressions, hence the name of the technique. Graph-based techniques are best suited for the implementation of *non-strict* functional languages like Miranda[3] [29] or Haskell [14]. Lazy evaluation is easy to achieve via normal order reduction of the graph; the sharing of subgraphs guarantees that expressions are evaluated at most once.

The idea underlying environment-based techniques is equally simple and radical: Since popping of values from the stack causes problems, never pop values. The resulting book-keeping mechanism is of course no longer called stack but environment instead, hence the name of the technique. A functional value is represented using a pair consisting of a pointer to the code and a pointer to the environment which records the values of the free variables. This bears a strong resemblance to stack-based implementations of imperative languages where procedure parameters are represented by a pointer to the code and a pointer into the stack (link to the activation record of the static predecessor). The first abstract machine which realized these ideas was the SECD machine by Landin [19]. The CAM resembles the SECD machine in many respects but it is conceptionally simpler and easier to

[2]A function not containing any free variables is called a combinator. Consequently a local function which refers to non-local variables is not a combinator.

[3]Miranda is a trademark of Research Software Ltd.

understand. It should be noted that, although the CAM and the SECD machine are environment-based, both of them also use stacks albeit for other purposes (remember that conventional machines also employ the stack for different purposes like the evaluation of arithmetic expressions). The fact that the environment contains the *values* of the free variables indicates that this technique is best-suited for the implementation of *strict* languages like SML [12]. Nevertheless, lazy evaluation can be built on top of eager evaluation.

The origins of the CAM can be traced back to the work of Lambek about the interpretation of λ-expressions in cartesian closed categories [18]. The ideas upon which the design of the CAM is based are due to Curien, who introduced categorical combinatory logic[4] [7]. When Cousineau and Curien realized that categorical combinators could be interpreted as instructions of a von Neumann machine, these ideas lead to the publication of the seminal paper about the CAM [4]. Due to its origin the CAM is tailored for the extreme, i.e., programs which make intensive use of higher-order functions. To be feasible for the normal case, i.e., many first order functions, improvements are necessary. Various optimizations were described by Suárez [28]. It is mainly his work this report is based upon.

The interesting point about the CAM lies in its conceptional proximity to classical stack architectures deviating only in the management of variable access. It is this proximity to classical stack architectures that makes many optimization techniques developed for the implementation of imperative languages (tail recursion optimization and its generalization last call optimization) practicable for the CAM as well. One of the objectives of this report is to point out this potential for optimization. In addition, we try to motivate the components of the CAM starting from first principles. Consequently a large part of the material presented is tutorial in nature. We largely ignore the category-theoretical background and introduce the ingredients from an implementor's point of view.

The report is organized as follows. Section 2 gives an account of the source language, which is essentially an augmented λ-calculus. Section 3 shows that once some fundamental design decisions have been made, the structure of the CAM develops quite naturally. The core of the CAM is presented in Section 4. A first improvement concerns the compilation of subexpressions which do not access the environment (Section 5). Further improvements instrumented through simple local transformations of the CAM code are presented in Section 6. Amongst others they include β-reduction at compile-time, improving calls to local functions, and last call optimization.

2 The Source Language

The source language which serves as a starting point for the compilation process is fairly conventional. It is essentially an augmented λ-calculus simple enough to be compiled directly into CAM code and expressive enough to translate a pure functional language into it. Readers familiar with the topic may safely skip this section.

The syntactic domains of the language are shown in Table 1. The domain **var** contains variables.

category	comment	meta variables
var	variables	x
$\mathbf{sys}_{(n)}$	predefined functions of arity $0 \leq n \leq 2$	$s_{(n)}$
con	constructors	c
exp	expressions	e
pat	patterns	p

Table 1: Syntactic domains for the source language

We do not bother how variables are represented, we simply assume that there are enough of them.

[4]Categorical combinatory logic can be viewed as "classical" combinatory logic augmented with products. Categorical combinators have been proposed as an alternative to SK combinators by Lins [20] revealing once again the close interconnection between graph-based and environment-based approaches.

Every realistic programming language offers predefined constants and functions, which constitute the domains $\mathbf{sys}_{(n)}$. The arity of a predefined function is indicated using bracketed subscripts, i.e., $s_{(2)}$ is a 2-argument function. In the remainder we require that the domain $\mathbf{sys}_{(0)}$ contains at least (the representations of) the natural numbers, characters, and the truth values *true* and *false*. The elements of the domain **con** are used to represent constructed values.

The domain **exp** contains the various syntactic constructs of the source language and will be further specified below. Finally, the domain **pat** comprises patterns which may be used in formal parameter positions—we permit ourselves this luxury because patterns of this kind can be easily translated into CAM code. We use x, $s_{(n)}$, c, e, and p as syntactic metavariables ranging over variables, predefined functions, constructors, expressions, and patterns.

The abstract syntax of the source language is given in Table 2. The source language contains the

exp	::=	**var**	variable
	|	$\mathbf{sys}_{(n)}\ \mathbf{exp}_1\ \cdots\ \mathbf{exp}_n$	application of a predefined function
	|	$()$	empty tuple
	|	$(\mathbf{exp}_1, \mathbf{exp}_2)$	pair
	|	**con exp**	constructed value
	|	$\mathbf{exp}_1\ \mathbf{exp}_2$	application
	|	$\lambda\mathbf{pat} \to \mathbf{exp}$	abstraction
	|	`if` $\mathbf{exp}_1$ `then` $\mathbf{exp}_2$ `else` $\mathbf{exp}_3$	conditional
	|	`case` **exp** `of` $\mathbf{con}_1\ \mathbf{pat}_1 \to \mathbf{exp}_1 \mid \cdots \mid \mathbf{con}_n\ \mathbf{pat}_n \to \mathbf{exp}_n$	case analysis
	|	`let` $\mathbf{pat}_1 = \mathbf{exp}_1$ `in` **exp**	local definition
	|	`letrec` $\mathbf{pat}_1 = \mathbf{exp}_1$ `in` **exp**	recursive local definition
pat	::=	**var**	variable
	|	$()$	empty pattern
	|	$(\mathbf{pat}_1, \mathbf{pat}_2)$	pair
	|	**var** `as` **pat**	layered pattern

Table 2: Abstract syntax of the source language

λ-calculus as a sublanguage, i.e., we are able to denote functional values. The expression

$$\lambda f \to \lambda x \to f\ (f\ x)$$

denotes a function which applies its first argument twice to its second. Function application is written without special syntax simply by sequencing the function and its argument. The expression $\lambda p \to e$ called abstraction corresponds to the function f with $f(p) = e$ but there is no need to invent a name for it. The formal parameter of an abstraction may be a very simple pattern composed of variables and pairs with the restriction that a variable may occur only once in a pattern, i.e., (a, b) is a legal pattern, (a, a) is not. We restrict ourselves to linear patterns of this kind (*irrefutable* patterns) because they cannot fail to match a (well-typed) argument obviating the need for a complex pattern matching process. The expression

$$\lambda(f, x) \to f\ (f\ x)$$

denotes a function which applies the first component of its argument twice to the second component of its argument. Note: There is a fundamental difference between

$$\lambda(a, b) \to +\ a\ b \quad \text{and} \quad \lambda a \to \lambda b \to +\ a\ b.$$

The first expression maps a pair to a natural number whereas the second maps a natural number to a function over the natural numbers.[5] Pairs may of course also appear in the body of an abstraction.

$$\lambda(a, b) \to (b, a)$$

[5] As a matter of fact the CAM is based on the correspondence between these two function definitions. We will see in the sequel that the CAM code generated for the bodies of the abstractions is identical.

is a function which swaps the components of a pair. The possibility of using pairs in patterns obviates the need for projection functions. With the help of layered patterns we can simultaneously access a pair and its components. In the expression

$$\lambda z \texttt{ as } (x, y) \rightarrow +\ (f\ z)\ x$$

the variable z serves as a shortcut for the pair (x, y).

The expression $\lambda p \rightarrow e$ is a variable binding construct very much like quantifiers in predicate logic. The variables in p are said to be bound by the λ-abstraction. A variable which is not bound is called free. In the expression $(\lambda(x, y) \rightarrow *\ x\ y)\ (x, z)$ the variable y occurs bound, z occurs free and x occurs both free and bound. We postpone a rigorous definition of these terms until Section 5.1.

Predefined functions must always be applied to the correct number of arguments. Partial parameterization, however, can be achieved using abstractions. So we are forced to write

$$\lambda x \rightarrow +\ 1\ x \quad \text{instead of} \quad +\ 1.$$

Local definitions are introduced by `let`- or `letrec`-expressions. As the name indicates, `letrec`-expressions must be used for recursive definitions. The scope of the defined variables extends over the definiens as well as over the definiendum. In analogy to patterns in formal parameter positions a local definition may define a variable only once, i.e., `let` $(x, y) = (1, 2)$ `in` x is legal, `let` $(x, x) = (1, 2)$ `in` x is not. We shall see in Section 6.5 that it is advantageous to use `letrec` for the binding of non-recursive abstractions as well. Hence a functional program or rather script can be simply represented by a `letrec`-expression. Note: Due to the evaluation strategy recursively defined non-functional values usually fail to terminate.

Most modern functional languages offer the possibility to define new datatypes, often called algebraic datatypes. A datatype definition describes essentially how elements of the datatype can be constructed. The SML definition of integer sequences

```
datatype sequence = nil | cons of int*sequence
```

tells us that the constructor `nil` is a `sequence` and `cons` applied to a pair consisting of an integer and a `sequence` is again a `sequence`. Elements of a datatype can be translated almost literally into constructed values of the source language with the slight exception that nullary constructors such as `nil` have to take the empty tuple as a dummy argument.

Functions over algebraic datatypes are usually defined via pattern matching. The test, whether a sequence contains a certain number, could be specified in SML as follows.

```
fun contains nil c = false
  | contains (cons(d,s)) c = if c=d then true
                                    else contains s c
```

If the patterns are exclusive and not nested we can translate them directly into a `case`-expression.

$$\begin{array}{lcl} \texttt{letrec}\ \mathit{contains} & = & \lambda t \rightarrow \lambda c \rightarrow \texttt{case}\ t\ \texttt{of} \\ & & \qquad\qquad \begin{array}{rcl} \mathit{nil}\ () & \rightarrow & \mathit{false} \\ \mathit{cons}\ (d, s) & \rightarrow & \texttt{if} = c\ d\ \texttt{then}\ \mathit{true}\ \texttt{else}\ \mathit{contains}\ s\ c \end{array} \\ \texttt{in}\ \ldots & & \end{array}$$

Each alternative of the `case`-expression describes how to handle the corresponding constructed value. Note: The `if`-construct could be defined in terms of the `case`-expression but for reasons of efficiency we include the former as a primitive.

To improve the readability of expressions we will make use of the abbreviations listed in Table 3.

3 Towards the CAM

Imperative languages can be implemented using a stack for storage management. We have seen in the introduction that this technique is not applicable to functional languages because a non-local variable may escape from its scope. Since a non-local variable is by definition a variable which occurs free

derived form	equivalent form
expressions	
$(e_1, e_2, \ldots, e_{n-1}, e_n)$	$((\ldots(e_1, e_2)\ldots, e_{n-1}), e_n)$
$e\ e_1\ e_2\ \cdots\ e_n$	$(\cdots((e\ e_1)\ e_2)\cdots\ e_n)$
$\lambda p_1\ \cdots\ p_n \to e$	$\lambda p_1 \to (\cdots(\lambda p_n \to e)\cdots)$
$\texttt{let}\ p_1 = e_1\ ;\ \ldots\ ;\ p_n = e_n\ \texttt{in}\ e$	$\texttt{let}\ (p_1, \ldots, p_n) = (e_1, \ldots, e_n)\ \texttt{in}\ e$
$\texttt{letrec}\ p_1 = e_1\ ;\ \ldots\ ;\ p_n = e_n\ \texttt{in}\ e$	$\texttt{letrec}\ (p_1, \ldots, p_n) = (e_1, \ldots, e_n)\ \texttt{in}\ e$
patterns	
$(p_1, p_2, \ldots, p_{n-1}, p_n)$	$((\ldots(p_1, p_2)\ldots, p_{n-1}), p_n)$

Table 3: Derived forms of expressions

in an abstraction, we must essentially decide how to deal with *free variables*. Note: Every variable eventually becomes free as λ-abstractions are traversed.

The classical approach also employed in the CAM is to maintain a data structure at run-time, the so-called *environment*, which records the bindings of the free variables. In contrast to a stack an environment is a monotonic data structure: It never shrinks. Nonetheless we have considerable freedom in representing the environment. It could be represented by a vector or by a list. Each choice has its pros and cons: Environments-as-lists are easy to extend whereas environments-as-vectors support constant time access to variables. We decide for the former: Environments are represented by lists or rather by nested pairs (the FAM uses vectors). For historical reasons pairs are nested in the left component, i.e., $(((), 1), 4)$ is an example for an environment. At compile-time we maintain a formal image of the environment which tells us how to access variables at run-time. To distinguish run-time from compile-time environments we use angle brackets for the latter, e.g., $\langle\langle\langle\rangle, y\rangle, x\rangle$.

An environment looks very much like an ordinary value, so why not treat it as such. Applying this idea we use the same *register* to store the environment and the computed value of an expression. The CAM code generated for the expression e in the compile-time environment ρ (notation: $\mathcal{C}[\![e]\!]\ \rho$) always satisfies the following invariant:

> If the register contains the run-time environment, then after the execution of the code it holds the value of the expression.

The CAM code may be interpreted as a mapping from environments to values. Because the environment is overwritten by a value, we are faced with a problem if the environment is required twice during a computation. We need a mechanism for saving and restoring environments. A suitable data structure for this purpose is a stack. The CAM code generated for an expression satisfies the further constraint:

> The stack may be used during the execution of the CAM code but afterwards it is in the same state as before.[6]

These invariants should be kept in mind when the compilation schemes are studied.

By now we have introduced all the components of the CAM. Let us summarize: The environment holds the values of the free variables and is organized as a list. The CAM comprises

- a register (for the environment and the computed value),
- a stack (for saving and restoring environments), and
- a code area.

The triple consisting of the register, the stack, and the pointer into the code area constitutes a CAM configuration. In the sequel we introduce the most important CAM instructions and show how to compile the expression $(\lambda x \to +\ x\ y)\ 4$ into CAM code (do not worry about the free variable y).

[6] This is what the warden of a youth hostel continues to tell.

3.1 Compiling a Variable

For a variable code which accesses the run-time environment is generated. The instruction Acc n fetches the n^{th} component of the environment. The CAM instructions are defined via their effects on a configuration. Every row of the following table specifies a transition (CAM instructions are separated by ";").

register	stack	code	register	stack	code
(v_1, v_2)	S	$\text{Acc } 0 \,;\, C$	v_2	S	C
(v_1, v_2)	S	$\text{Acc } (n+1) \,;\, C$	v_1	S	$\text{Acc } n \,;\, C$

The instruction Acc 0 expects the register to contain a pair and replaces it by its second component leaving the stack unchanged. The "program counter" is moved to the next instruction.

In the environment $\rho = \langle\langle\langle\rangle, y\rangle, x\rangle$ the variables x and y get translated to:

$$\begin{aligned} \mathcal{C}[\![x]\!]\ \rho &= \text{Acc } 0 \\ \mathcal{C}[\![y]\!]\ \rho &= \text{Acc } 1 \end{aligned}$$

The argument of Acc coincides with the de Bruijn number of a variable, i.e., the number of λ-abstractions between a variable and its binding place.

3.2 Compiling a Predefined Function

Nullary predefined functions or constants may safely ignore the environment.

register	stack	code	register	stack	code
v	S	$\text{Quote } s_{(0)} \,;\, C$	$s_{(0)}$	S	C

The Quote instruction corresponds to a load immediate operation.

$$\mathcal{C}[\![4]\!]\ \rho = \text{Quote } 4$$

Binary operators expect their first argument on top of the stack and their second in the register. This is reminiscent of conventional stack architectures especially if we consider the register as a cache for the topmost element of the stack. As the environment may be required for both arguments, the stack must be used for intermediate storage. For saving and restoring the environment the instructions Push and Swap are provided (":" prefixes an element to a stack).

register	stack	code	register	stack	code
v_1	$v_2 : S$	$+ \,;\, C$	$v_2 + v_1$	S	C
v	S	$\text{Push} \,;\, C$	v	$v : S$	C
v_1	$v_2 : S$	$\text{Swap} \,;\, C$	v_2	$v_1 : S$	C

The instruction Push copies the contents of the register onto the stack; Swap interchanges the contents of the register and the topmost element of the stack.

$$\begin{aligned} \mathcal{C}[\![+\ x\ y]\!]\ \rho &= \text{Push} \,;\, \mathcal{C}[\![x]\!]\ \rho \,;\, \text{Swap} \,;\, \mathcal{C}[\![y]\!]\ \rho \,;\, + \\ &= \text{Push} \,;\, \text{Acc } 0 \,;\, \text{Swap} \,;\, \text{Acc } 1 \,;\, + \end{aligned}$$

The initial Push instruction copies the environment onto the stack providing it for the computation of the second summand. The code for the first summand destroys the environment mapping it to a value; Swap restores the environment and simultaneously moves the first argument's value onto the stack. After the execution of the second argument's code, the configuration is such as the binary operation "+" expects it to be.

3.3 Compiling an Abstraction

Though we know that an abstraction denotes a function, it is not quite clear what the computed value of an abstraction should be. The answer is very simple: An abstraction evaluates to a frozen computation waiting to be applied to an argument later on.[7] Because an abstraction may contain free variables it is represented by a pair consisting of

- the current environment and
- a pointer to the CAM code of the body.

This pair (notation: $[s:\ell]$) is traditionally called *closure*. The CAM instruction CUR builds a closure; RETURN is used to return from a subroutine call.

register	stack	code	register	stack	code
v	S	$\textsc{Cur}\ \ell\,; C$	$[v:\ell]$	S	C
v	$C:S$	$\textsc{Return}\,; C'$	v	S	C

Closure building is a very cheap operation. This is the main reason why the CAM is well suited for the execution of highly functional programs where closure building occurs very frequently. Let $\rho' = \langle\langle\rangle, y\rangle$, then

$$\begin{array}{rcl}
\mathcal{C}[\![\lambda x \to +\ x\ y]\!]\ \rho' & = & \textsc{Cur}\ \ell \\
& & [\ \ell : \mathcal{C}[\![+\ x\ y]\!]\ \langle\rho', x\rangle\,; \textsc{Return}\ ] \\
& = & \textsc{Cur}\ \ell \\
& & [\ \ell : \textsc{Push}\,; \textsc{Acc}\ 0\,; \textsc{Swap}\,; \textsc{Acc}\ 1\,; +\,; \textsc{Return}\ ]
\end{array}$$

The square brackets indicate that the generated code is *not* consecutive. The bracketed code has rather the status of a subroutine, which is addressed via the label put at the beginning. The body of the abstraction must be compiled in an extended environment because the bound variable x appears free in the body.

The more abstractions are processed the deeper the nesting of the environment becomes. The access time grows linear to the depth of the environment. But since in practice the nesting of definitions is rather small, little run-time penalty is caused.

3.4 Compiling an Application

The application is treated like a binary operator. The instruction APP melts a frozen computation and performs an indirect jump to the body of the abstraction. The code sequence of the body is executed in the environment which is stored in the closure extended with the argument's value.

register	stack	code	register	stack	code
$[v_1:\ell]$	$v_2:S$	$\textsc{App}\,; C$	(v_1, v_2)	$C:S$	C_ℓ

The notation C_ℓ denotes the subroutine labeled with ℓ.

$$\begin{array}{rcl}
\mathcal{C}[\![(\lambda x \to +\ x\ y)\ 4]\!]\ \rho' & = & \textsc{Push}\,; \mathcal{C}[\![4]\!]\ \rho'\,; \textsc{Swap}\,; \mathcal{C}[\![\lambda x \to +\ x\ y]\!]\ \rho'\,; \textsc{App} \\
& = & \textsc{Push}\,; \textsc{Quote}\ 4\,; \textsc{Swap}\,; \textsc{Cur}\ \ell\,; \textsc{App} \\
& & [\ \ell : \textsc{Push}\,; \textsc{Acc}\ 0\,; \textsc{Swap}\,; \textsc{Acc}\ 1\,; +\,; \textsc{Return}\ ]
\end{array}$$

Note: The function and the argument are processed in reversed order. This compilation scheme deviates from the original description of the CAM but it facilitates code optimizations (cf. Section 6.4).

[7]That is to say, an abstraction is not evaluated at all. This fact is employed in some SML implementations of lazy lists. Functions with a dummy argument $\lambda() \to e$ serve as means to freeze a computation. A frozen computation is melted by applying it to the dummy value ().

3.5 Execution of the Code

Because our running example contains a free variable, we must supply an non-empty initial environment, e.g., $v = ((), 1)$. The consecutively numbered code and the execution of this code is displayed in Table 4. The empty stack is denoted by ε, the operator ":" prefixes an element to a stack. Fortunately the expected value of the expression coincides with the computed one.

CAM configuration			**CAM code**
register	stack	code	
v	ε	ℓ_1	ℓ_1 : PUSH
v	$v : \varepsilon$	ℓ_2	ℓ_2 : QUOTE 4
4	$v : \varepsilon$	ℓ_3	ℓ_3 : SWAP
v	$4 : \varepsilon$	ℓ_4	ℓ_4 : CUR ℓ_7
$[v : \ell_7]$	$4 : \varepsilon$	ℓ_5	ℓ_5 : APP
$(v, 4)$	$\ell_6 : \varepsilon$	ℓ_7	ℓ_6 : STOP
$(v, 4)$	$(v, 4) : \ell_6 : \varepsilon$	ℓ_8	ℓ_7 : PUSH
4	$(v, 4) : \ell_6 : \varepsilon$	ℓ_9	ℓ_8 : ACC 0
$(v, 4)$	$4 : \ell_6 : \varepsilon$	ℓ_{10}	ℓ_9 : SWAP
1	$4 : \ell_6 : \varepsilon$	ℓ_{11}	ℓ_{10} : ACC 1
5	$\ell_6 : \varepsilon$	ℓ_{12}	ℓ_{11} : +
5	ε	ℓ_6	ℓ_{12} : RETURN

Table 4: A sample execution

4 The Core of the Machine

4.1 More Details of the CAM

So far we have been rather vague about the values the register and the stack hold. A precise characterization of these values is displayed in Tables 5 and 6. The domain **lab** contains labels which are

domain	comment	meta variables
lab	labels	ℓ
val	values	v
env α	environments	ρ
ins	CAM instructions	I
cam	CAM code	C

Table 5: Syntactic domains for CAM values, formal environments, and CAM code

used to address CAM code sequences. (Intermediate) results of computations make up the domain **val**. Some of its constituents (constants, pairs, and closures) have already been used in the previous example. Tagged values are the semantic counterpart of algebraic datatypes. The first component of a tagged value called tagfield indicates which constructor was used; the second component contains the argument of the constructor. The expression (of type `sequence`)

$$\mathit{cons}\ (3, \mathit{cons}\ (2, \mathit{cons}\ (1, \mathit{nil}\ ())))$$

is represented by:

$$(\mathit{cons} : (3, (\mathit{cons} : (2, (\mathit{cons} : (1, (\mathit{nil} : ())))))))$$

val	::=	$\mathbf{sys}_{(0)}$	constant
	|	$()$	empty tuple
	|	$(\mathbf{val}, \mathbf{val})$	pair
	|	$(\mathbf{con} : \mathbf{val})$	tagged value
	|	$[\mathbf{val} : \mathbf{lab}]$	closure
	|	$[\mathbf{lab}]$	closure of a combinator
env α	::=	$\langle\rangle$	empty environment
	|	$\langle \mathbf{env}\ \alpha, \mathbf{pat}\rangle$	constructed environment
	|	$\langle \mathbf{env}\ \alpha, \mathbf{pat} \mapsto \alpha\rangle$	annotated environment
cam	::=	**ins**	single instruction
	|	**cam** ; **cam**	sequence
	|	**lab** : **cam**	labeled sequence

Table 6: Abstract syntax of CAM values, formal environments, and CAM code

Functions not containing free variables are represented simply by a pointer into the code, i.e., $[\ell]$ is a closure with an empty environment. Note: In principle there are only two kinds of values, namely scalars and pairs. The domain **env** α comprises formal or compile-time environments parameterized with the domain α of annotations. The domain **ins** contains CAM instructions (cf. Table 7); the domain **cam** consists of sequences of CAM instructions.

As shown in Table 7, the CAM instructions can be roughly divided into four groups. This classification is not always as straightforward as the table suggests but it may help in memorizing the different instructions. To access values stored in the environment, access instructions are used. A quick glance shows that in principle two of them would suffice, namely FST and SND, because ACC and REST can be defined in terms of them.

$$\begin{aligned} \textsc{Acc}\ n &:= \textsc{Fst}^n ; \textsc{Snd} \\ \textsc{Rest}\ n &:= \textsc{Fst}^n \end{aligned}$$

The stack operations PUSH and SWAP realize the saving mechanism. The register operations comprise commands which change the contents of the register possibly using values on the stack. The control operations affect the flow of control. The SKIP instruction—sometimes called NOP—is only included because it simplifies the presentation of the compilation schemes.

4.2 Compiling an Expression

We present the translation of a source expression into a sequence of CAM instructions using a set of compilation schemes. The different schemes, $\mathcal{E}$, $\mathcal{P}$, $\mathcal{C}$, $\mathcal{T}$, and $\mathcal{R}$, are described in Tables 8 and 9. We consider each of them in turn.

4.2.1 Compiling pure λ-expressions

Because patterns may occur in formal parameter positions, environments are LISP-like binary trees rather than lists. The body of the multiple abstraction

$$\lambda(a, b) \to \lambda(c, d) \to e$$

is compiled in the formal environment:

$$\rho \;=\; \langle\langle\langle\rangle, (a, b)\rangle, (c, d)\rangle$$

The pictorial representation of the environment displayed in figure 1 shows its tree-like structure.

The code generation for a variable boils down to a non-deterministic search in the formal environment. The sequence of access instructions—ACC n followed by a sequence of FST and SND

register	stack	code	register	stack	code
		access instructions			
(v_1, v_2)	S	$\textsc{Fst}\,; C$	v_1	S	C
(v_1, v_2)	S	$\textsc{Snd}\,; C$	v_2	S	C
(v_1, v_2)	S	$\textsc{Acc}\ 0\,; C$	v_2	S	C
(v_1, v_2)	S	$\textsc{Acc}\ (n+1)\,; C$	v_1	S	$\textsc{Acc}\ n\,; C$
v	S	$\textsc{Rest}\ 0\,; C$	v	S	C
(v_1, v_2)	S	$\textsc{Rest}\ (n+1)\,; C$	v_1	S	$\textsc{Rest}\ n\,; C$
		stack operations			
v	S	$\textsc{Push}\,; C$	v	$v : S$	C
v_1	$v_2 : S$	$\textsc{Swap}\,; C$	v_2	$v_1 : S$	C
		register operations			
v	S	$\textsc{Quote}\ s_{(0)}\,; C$	$s_{(0)}$	S	C
v	S	$\textsc{Clear}\,; C$	$()$	S	C
v	S	$\textsc{Prim}\ s_{(1)}\,; C$	$s_{(1)}(v)$	S	C
v_1	$v_2 : S$	$\textsc{Prim}\ s_{(2)}\,; C$	$s_{(2)}(v_2, v_1)$	S	C
v_1	$v_2 : S$	$\textsc{Cons}\,; C$	(v_2, v_1)	S	C
v	S	$\textsc{Cur}\ \ell\,; C$	$[v : \ell]$	S	C
v	S	$\textsc{Pack}\ c\,; C$	$(c : v)$	S	C
		control instructions			
v	S	$\textsc{Skip}\,; C$	v	S	C
v	S	$\textsc{Stop}\,; C$		all systems stop	
$[v_1 : \ell]$	$v_2 : S$	$\textsc{App}\,; C$	(v_1, v_2)	$C : S$	C_ℓ
v	$C' : S$	$\textsc{Return}\,; C$	v	S	C'
v	S	$\textsc{Call}\ \ell\,; C$	v	$C : S$	C_ℓ
true	$v : S$	$\textsc{Gotofalse}\ \ell\,; C$	v	S	C
false	$v : S$	$\textsc{Gotofalse}\ \ell\,; C$	v	S	C_ℓ
$(c_i : v_1)$	$v_2 : S$	$\textsc{Switch}\ [c_1 : \ell_1, \ldots, c_n : \ell_n]\,; C$	(v_2, v_1)	S	C_{ℓ_i}
v	S	$\textsc{Goto}\ \ell\,; C$	v	S	C_ℓ

Table 7: The instructions of the CAM

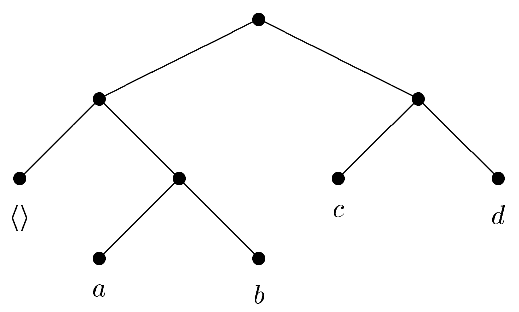

Figure 1: Graphical representation of the environment $\langle\langle\langle\rangle, (a, b)\rangle, (c, d)\rangle$

$\mathcal{E}[\![x]\!]\ \rho\ n$ generates code which accesses the variable x in the environment ρ at nesting level n. Before the execution of the code the register contains an instance of the environment, afterwards it holds the value of x.

$$
\begin{array}{rcl}
\mathcal{E} & : & \textbf{var} \to \textbf{env lab} \to \textbf{nat} \to \textbf{cam} \\
\mathcal{E}[\![x]\!]\ \langle\rangle\ n & = & \mathit{fail} \\
\mathcal{E}[\![x]\!]\ \langle\rho,p\rangle\ n & = & (\textsc{Acc}\ n\,;\mathcal{P}[\![x]\!]\ p)\ ?\ \mathcal{E}[\![x]\!]\ \rho\ (n+1) \\
\mathcal{E}[\![x]\!]\ \langle\rho,p\mapsto\ell\rangle\ n & = & (\textsc{Rest}\ n\,;\textsc{Call}\ \ell\,;\mathcal{P}[\![x]\!]\ p)\ ?\ \mathcal{E}[\![x]\!]\ \rho\ n
\end{array}
$$

$\mathcal{P}[\![x]\!]\ p$ generates code which accesses the variable x in the pattern p. Before the execution of the code the register contains an instance of the pattern, afterwards it holds the value of x.

$$
\begin{array}{rcll}
\mathcal{P} & : & \textbf{var} \to \textbf{pat} \to \textbf{cam} & \\
\mathcal{P}[\![x]\!]\ () & = & \mathit{fail} & \\
\mathcal{P}[\![x]\!]\ y & = & \textsc{Skip} & \text{if } x = y \\
 & = & \mathit{fail} & \text{otherwise} \\
\mathcal{P}[\![x]\!]\ (p_1,p_2) & = & (\textsc{Fst}\,;\mathcal{P}[\![x]\!]\ p_1)\ ?\ (\textsc{Snd}\,;\mathcal{P}[\![x]\!]\ p_2) & \\
\mathcal{P}[\![x]\!]\ (y\ \texttt{as}\ p) & = & \textsc{Skip} & \text{if } x = y \\
 & = & \mathcal{P}[\![x]\!]\ p & \text{otherwise}
\end{array}
$$

Table 8: The $\mathcal{E}$ and $\mathcal{P}$ compilation schemes

instructions—mirrors the access path to the variable. The non-deterministic search is realized via the (meta-) operations "*fail*" and "?". If the search is successful, a sequence of CAM instructions is returned, otherwise the special value "*fail*". The binary operation "?" returns the value of its first argument if it is not equal to "*fail*", otherwise the value of its second argument. Thus a sequential left to right search can be implemented with "?".

The compilation scheme $\mathcal{E}[\![x]\!]\ \rho\ n$ compiles the access of x in the environment ρ. The parameter n records the current depth of the environment. The scheme $\mathcal{P}[\![x]\!]\ p$ compiles the access within the pattern p. For the above environment we get:

$$
\begin{array}{rcl}
\mathcal{C}[\![c]\!]\ \rho & = & \mathcal{E}[\![c]\!]\ \rho^{(0)} \\
 & = & \textsc{Acc}\ 0\,;\mathcal{P}[\![c]\!]\ (c,d) \\
 & = & \textsc{Acc}\ 0\,;\textsc{Fst} \\
\mathcal{C}[\![b]\!]\ \rho & = & \mathcal{E}[\![b]\!]\ \rho\ 0 \\
 & = & \mathcal{E}[\![b]\!]\ \langle\langle\rangle,(a,b)\rangle\ 1 \\
 & = & \textsc{Acc}\ 1\,;\mathcal{P}[\![b]\!]\ (a,b) \\
 & = & \textsc{Acc}\ 1\,;\textsc{Snd}
\end{array}
$$

We postpone the discussion of the third equation in the definition of $\mathcal{E}$ until Section 4.2.4.

Since we can view function application and construction of pairs as special binary operations, their compilation is very similar to the treatment of predefined binary functions. We use the binary scheme $\mathcal{T}[\![e_1,e_2]\!]\ \rho$ to generate code which pushes the value of e_1 onto the stack and stores the value of e_2 into the register. Remember that function application is processed in reversed order. Thus the arguments of a curried function are evaluated from right to left.

$$
\begin{array}{rcl}
\mathcal{C}[\![f\ e_1\ e_2\ e_3]\!]\ \rho & = & \textsc{Push}\,;\mathcal{C}[\![e_3]\!]\ \rho\,;\textsc{Swap}\,;\textsc{Push}\,;\mathcal{C}[\![e_2]\!]\ \rho\,;\textsc{Swap}\,; \\
 & & \textsc{Push}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{Swap}\,;\mathcal{C}[\![f]\!]\ \rho\,;\textsc{App}\,;\textsc{App}\,;\textsc{App}
\end{array}
$$

Rearranging the order of evaluation poses no problems since we are only concerned with languages which adhere to the principle of referential transparency.

The compilation scheme $\mathcal{R}$ generates code for a subroutine, i.e., the "normal" code sequence followed by a RETURN instruction. In Section 6.6 we will define a more sophisticated variant of this

$\mathcal{M}[\![e]\!]$ is the CAM code for the closed expression e.

$$\begin{array}{rcl} \mathcal{M} & : & \mathbf{exp} \to \mathbf{cam} \\ \mathcal{M}[\![e]\!] & = & \mathcal{C}[\![e]\!]\ \langle\rangle\ ;\ \textsc{Stop} \end{array}$$

$\mathcal{C}[\![e]\!]\ \rho$ compiles code which evaluates the expression e in the environment ρ. Before the execution of the code the register contains an instance of the environment, afterwards it holds the value of e.

$$\begin{array}{rcl} \mathcal{C} & : & \mathbf{exp} \to \mathbf{env} \to \mathbf{cam} \\ \mathcal{C}[\![x]\!]\ \rho & = & \mathcal{E}[\![x]\!]\ \rho\ 0 \\ \mathcal{C}[\![s_{(0)}]\!]\ \rho & = & \textsc{Quote}\ s_{(0)} \\ \mathcal{C}[\![s_{(1)}\ e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho\ ;\ \textsc{Prim}\ s_{(1)} \\ \mathcal{C}[\![s_{(2)}\ e_1\ e_2]\!]\ \rho & = & \mathcal{T}[\![e_1, e_2]\!]\ \rho\ ;\ \textsc{Prim}\ s_{(2)} \\ \mathcal{C}[\![()]\!]\ \rho & = & \textsc{Clear} \\ \mathcal{C}[\![(e_1, e_2)]\!]\ \rho & = & \mathcal{T}[\![e_1, e_2]\!]\ \rho\ ;\ \textsc{Cons} \\ \mathcal{C}[\![c\ e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho\ ;\ \textsc{Pack}\ c \\ \mathcal{C}[\![e_1\ e_2]\!]\ \rho & = & \mathcal{T}[\![e_2, e_1]\!]\ \rho\ ;\ \textsc{App} \\ \mathcal{C}[\![\lambda p \to e]\!]\ \rho & = & \textsc{Cur}\ \ell \\ & & [\ \ell : \mathcal{R}[\![e]\!]\ \langle \rho, p\rangle\ ] \\ \mathcal{C}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!]\ \rho & = & \textsc{Push}\ ;\ \mathcal{C}[\![e_1]\!]\ \rho\ ;\ \textsc{Gotofalse}\ \ell_1\ ;\ \mathcal{C}[\![e_2]\!]\ \rho\ ;\ \textsc{Goto}\ \ell_2\ ; \\ & & \ell_1 : \mathcal{C}[\![e_3]\!]\ \rho\ ;\ \ell_2 : \textsc{Skip} \\ \multicolumn{3}{l}{\mathcal{C}[\![\texttt{case}\ e\ \texttt{of}\ c_1\ p_1 \to e_1 \mid \cdots \mid c_n\ p_n \to e_n]\!]\ \rho} \\ & = & \textsc{Push}\ ;\ \mathcal{C}[\![e]\!]\ \rho\ ;\ \textsc{Switch}\ [c_1 : \ell_1, \ldots, c_n : \ell_n]\ ; \\ & & \ell_1 : \mathcal{C}[\![e_1]\!]\ \langle \rho, p_1\rangle\ ;\ \textsc{Goto}\ \ell \\ & & \vdots \\ & & \ell_n : \mathcal{C}[\![e_n]\!]\ \langle \rho, p_n\rangle\ ; \\ & & \ell : \textsc{Skip} \\ \mathcal{C}[\![\texttt{let}\ p_1 = e_1\ \texttt{in}\ e]\!]\ \rho & = & \textsc{Push}\ ;\ \mathcal{C}[\![e_1]\!]\ \rho\ ;\ \textsc{Cons}\ ;\ \mathcal{C}[\![e]\!]\ \langle \rho, p_1\rangle \\ \mathcal{C}[\![\texttt{letrec}\ p_1 = e_1\ \texttt{in}\ e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \langle \rho, p_1 \mapsto \ell\rangle \\ & & [\ \ell : \mathcal{R}[\![e_1]\!]\ \langle \rho, p_1 \mapsto \ell\rangle\ ] \end{array}$$

$\mathcal{T}[\![e_1, e_2]\!]\ \rho$ compiles code which pushes the value of e_1 onto the stack and stores the value of e_2 into the register.

$$\begin{array}{rcl} \mathcal{T} & : & \mathbf{exp} \times \mathbf{exp} \to \mathbf{env} \to \mathbf{cam} \\ \mathcal{T}[\![e_1, e_2]\!]\ \rho & = & \textsc{Push}\ ;\ \mathcal{C}[\![e_1]\!]\ \rho\ ;\ \textsc{Swap}\ ;\ \mathcal{C}[\![e_2]\!]\ \rho \end{array}$$

$\mathcal{R}[\![e]\!]\ \rho$ is—at least for the moment—the code produced by $\mathcal{C}[\![e]\!]\ \rho$ followed by a Return instruction.

$$\begin{array}{rcl} \mathcal{R} & : & \mathbf{exp} \to \mathbf{env} \to \mathbf{cam} \\ \mathcal{R}[\![e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho\ ;\ \textsc{Return} \end{array}$$

Table 9: The $\mathcal{M}$, $\mathcal{C}$, $\mathcal{T}$, and $\mathcal{R}$ compilation schemes

scheme, which has the property that each *execution* of the code ends in a RETURN instruction. The $\mathcal{R}$ scheme is employed to compile the body of an abstraction. It should be clear that for each invocation of a compilation rule fresh labels must be used. The CAM code for some simple source expressions is shown below (the CAM code of an expression is arranged column-wise from left to right).

$\lambda x \to +\ 1\ x$		$\lambda f\ x \to f\ (f\ x)$			$\lambda(f,x) \to f\ (f\ x)$		
	CUR ℓ		CUR ℓ_1	ACC 1		CUR ℓ	FST
	STOP		STOP	APP		STOP	APP
ℓ :	PUSH	ℓ_1 :	CUR ℓ_2	SWAP	ℓ :	PUSH	SWAP
	QUOTE 1		RETURN	ACC 1		PUSH	ACC 0
	SWAP	ℓ_2 :	PUSH	APP		ACC 0	FST
	ACC 0		PUSH	RETURN		SND	APP
	PRIM +		ACC 0			SWAP	RETURN
	RETURN		SWAP			ACC 0	

The example in the first column demonstrates how to translate partially parameterized, predefined functions. Multiple abstractions result in sequences of CUR instructions (second column); the use of pairs in abstractions complicates the access of variables (third column). It should be noted, however, that the access *time* is the same regardless to the use of multiple abstractions or pairs.

4.2.2 Compiling an alternative

Conditional and unconditional jumps are used to implement the `if-then-else`-construct. The compilation scheme for the alternative bears a strong resemblance to code schemes for the `if`-statement in imperative languages.

$\lambda n \to$ `if` $\geq n\ 0$ `then` n `else` $-\ n$						
	CUR ℓ_1	ACC 0		GOTOFALSE ℓ_2		PRIM $-$
	STOP	SWAP		ACC 0	ℓ_3 :	RETURN
ℓ_1 :	PUSH	QUOTE 0		GOTO ℓ_3		
	PUSH	PRIM $\geq$	ℓ_2 :	ACC 0		

Note: Lazy variants of the logical conjunction and disjunction can also be implemented via conditional jumps (short-circuited evaluation).

$e_1 \wedge e_2 \quad := \quad$ `if` e_1 `then` e_2 `else` *false*

$e_1 \vee e_2 \quad := \quad$ `if` e_1 `then` *true* `else` e_2

4.2.3 Compiling algebraic datatypes

We have already seen how elements of an algebraic datatype are represented at run-time. The tagfield is set with the instruction PACK and is discriminated via the SWITCH instruction, which behaves like a multi-way jump. The compilation scheme of the `case`-expression generalizes the rule for the `if-then-else`-construct: There are in general more alternatives and each alternative includes a variable binding mechanism (for the constructor's argument).

$\lambda s \to$ `case` s `of` $nil\ () \to nil\ ()\ \mid\ cons\ (c,t) \to t$							
	CUR ℓ_1		ACC 0		PACK *nil*		SND
	STOP		SWITCH $[nil : \ell_2, cons : \ell_3]$		GOTO ℓ_4	ℓ_4 :	RETURN
ℓ_1 :	PUSH	ℓ_2 :	CLEAR	ℓ_3 :	ACC 0		

4.2.4 Compiling a local definition

A non-recursive local definition can be understood as a combination of functional abstraction and application, i.e., the expression

`let` $p_1 = e_1$ `in` e_2

is equivalent to the β-redex

$$(\lambda p_1 \rightarrow e_2)\ e_1$$

The equivalence suggests a first compilation rule for the `let`-construct.

$$\begin{array}{rcl} \mathcal{C}[\![\texttt{let}\ p_1 = e_1\ \texttt{in}\ e_2]\!]\ \rho & = & \textsc{Push}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{Swap}\,;\textsc{Cur}\ \ell\,;\textsc{App} \\ & & [\ \ell : \mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle\,;\textsc{Return}\ ] \end{array}$$

The generated code sequence can be easily improved if we take a closer look at the operational behavior of the code. When the body of the abstraction is executed, the register contains an instance of the formal environment $\langle\rho, p_1\rangle$. The indirect call of the subroutine is obviously unnecessary, so we content ourselves with the construction of the environment.

$$\mathcal{C}[\![\texttt{let}\ p_1 = e_1\ \texttt{in}\ e_2]\!]\ \rho \quad = \quad \textsc{Push}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{Cons}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle$$

This is already an example for a simple code transformation, a topic we will pursue further in Section 6. It should be noted that local definitions are evaluated whether they are needed or not.

let $a = 5$ in $*\ a\ a$			
PUSH QUOTE 5	CONS PUSH	ACC 0 SWAP	ACC 0 PRIM $*$

The most difficult thing to explain is the compilation of recursive local definitions. A recursive definition can be viewed as a finite description of an infinite expression. The generated CAM code has the same flavor: It is in principle infinite but is represented finitely using labels and references. The defining expression of the `letrec`-construct is compiled as a subroutine marked with label ℓ. The association between the label ℓ and the defined pattern p is recorded in the environment (notation: $\langle\rho, p \mapsto \ell\rangle$). The use of the extended environment for the translation of both expressions reflects the scoping rules.

At run-time the generated subroutine is addressed via an ordinary subroutine call. Prior to the call, the proper environment must be restored. Because the environment only grows, we must simply forget the entries which have been added since the code was entered. This is instrumented by the REST n instruction, which is equal to a sequence of n FST instructions. An example for the compilation of a recursive definition is given below.

letrec $even = \lambda n \rightarrow$ if $= n\ 0$ then $true$ else $\neg\ even\ (dec\ n)$ in $even$ 56			
PUSH QUOTE 56 SWAP REST 0 CALL ℓ_1 APP STOP	ℓ_1 : CUR ℓ_2 RETURN ℓ_2 : PUSH PUSH ACC 0 SWAP QUOTE 0	PRIM $=$ GOTOFALSE ℓ_3 QUOTE $true$ GOTO ℓ_4 ℓ_3 : PUSH ACC 0 PRIM dec	SWAP REST 1 CALL ℓ_1 APP PRIM $\neg$ ℓ_4 : RETURN

The first call to the defining expression is executed in the initial environment (REST 0 is equal to SKIP). For the recursive call, which is situated in the body of the abstraction, the environment must be reduced by one entry.

In contrast to the `let`-construct, which is evaluated in a call by value fashion, the `letrec`-construct realizes a call by name regime. That is to say, the defining expression is evaluated as many times as the defined variables are accessed. The `letrec`-construct should be used whenever the defining expression is an abstraction, because an abstraction is never evaluated—there is no loss in time—and the code is addressed directly not using the environment—there is a gain in space efficiency.

The code generated for multiple recursive definitions like `letrec` $p_1 = e_1\,;\ \ldots\,; p_n = e_n$ `in` e is rather clumsy. One can do better by directly addressing the functions instead of building pairs and subsequently accessing the components. From now on we will use the generalized scheme shown in Table 10.

$\mathcal{C}[\![\texttt{letrec}\ p_1 = e_1\ ;\ \ldots\ ; p_n = e_n\ \texttt{in}\ e]\!]\ \rho$
$= \quad \mathcal{C}[\![e]\!]\ \rho'$
$\quad\quad \begin{bmatrix} \ell_1 : \mathcal{R}[\![e_1]\!]\ \rho' \\ \vdots \\ \ell_n : \mathcal{R}[\![e_n]\!]\ \rho' \end{bmatrix}$
where
$\rho' = \langle \ldots \langle \rho, p_1 \mapsto \ell_1 \rangle \ldots, p_n \mapsto \ell_n \rangle$

Table 10: The $\mathcal{C}$ scheme revisited for multiple recursive definitions

5 r-Closed Expressions

The compilation schemes which we have introduced in the last section are conceptionally very simple but they are of course too simple-minded to be used in a real implementation. In the following we will introduce several techniques which aim at improving the quality of the generated code (classically called *code optimizations*).

A first improvement concerns the compilation of subexpressions which do not access the environment. Consider the following example.

$$\mathcal{C}[\![+\ x\ 3]\!]\ \rho \quad = \quad \text{PUSH}\,; \mathcal{C}[\![x]\!]\ \rho\,; \text{SWAP}\,; \text{QUOTE}\ 3\,; \text{PRIM}\ +$$

The PUSH instruction copies the initial environment onto the stack to make it available for the evaluation of the second summand. But once the environment has been restored, it is overwritten by the QUOTE instruction. The saving of the environment is obviously unnecessary and we can improve the code as follows.

$$\mathcal{C}[\![+\ x\ 3]\!]\ \rho \quad = \quad \mathcal{C}[\![x]\!]\ \rho\,; \text{MOVE}\,; \text{QUOTE}\ 3\,; \text{PRIM}\ +$$

The MOVE instruction moves the contents of the register onto the stack. The example shows that the improved translation essentially saves PUSH instructions. The elimination of some PUSHes may not seem worth the effort but it should be kept in mind that the generated CAM code is usually further expanded into real machine code. If the target machine belongs to the von Neumann class, then the PUSH instruction corresponds to an allocation of a register or a memory cell, an operation which should occur as seldom as possible. The innocent looking PUSH instruction has another interesting implication. As PUSH is the only operation which duplicates a value, it necessitates the implementation of some form of garbage collection.

For those readers familiar with graph-based implementation techniques (for non-strict functional languages), a comparison between SK combinator code and CAM code might be helpful. With the help of the combinators S, K, and I it is possible to simulate a β-reduction, i.e., the substitution of an argument for a bound variable. The combinator S moves or rather copies the argument to the leaves of the expression[8] in the body, I accepts the argument, and K rejects it. The procedure is reminiscent of the CAM's saving mechanism. The instructions PUSH and SWAP move the environment to the leaves, ACC uses the environment, and QUOTE rejects it. The only difference lies in the number of arguments which are moved: The SK machine takes only one argument at a time, the CAM takes all arguments simultaneously.

A first improvement of the combinator code introduces the combinators B and C; they are used to move the argument only to the positions where it is needed. The improvement we are concerned with in this section aims at the same purpose. The effect on the code size is of course not equally impressive.

[8] We appeal to the abstract syntax tree of an expression rather than to its linear representation.

5.1 r-Closedness

A closed expression, i.e., an expression not containing free variables, can be computed without any reference to the environment. Since `letrec`-bound variables are not addressed via the environment, we need the more technical notion of r-closedness.

> An expression is called r-closed if the environment is not necessary[9] for its computation.

The following expression exemplifies the difference between closed and r-closed terms.

$$\begin{array}{ll}\texttt{letrec} & \mathit{twice} = \lambda f\ x \to f\ (f\ x)\,; \\ & \mathit{inc} = \lambda n \to +\ 1\ n \\ \multicolumn{2}{l}{\texttt{in}\ \mathit{twice}\ \mathit{twice}\ \mathit{inc}\ 0}\end{array}$$

The expression *twice twice inc* 0 and each of its subexpressions are (in the given context) r-closed but not closed because the `letrec`-bound variables *twice* and *inc* are compiled to simple subroutine calls for the execution of which the initial environment is unnecessary. It should be noted that the notion of r-closedness is intentionally defined very operationally adapted to the improvements we have in mind.

Furthermore, observe that the improved compilation of r-closed expressions may benefit from optimizations which aim at using registers or stack positions instead of the environment, i.e., the fewer variables are held in the environment the more the technique becomes applicable.

Let us turn to the question of determining whether an expression satisfies the property of r-closedness or not. In analogy to the notion of closed expressions, we call an expression r-closed if it does not contain r-free variables—this second definition is consistent with the one given earlier.

> A variable x occurs r-free in the expression e if x is held in in the environment and if it is necessary for the computation of e.

Note that a variable need not be part of an expression to occur r-free in the expression. In the subexpression $*\ y\ y$ of

$$\lambda x \to \texttt{letrec}\ y = +\ 1\ x\ \texttt{in} * y\ y$$

the variable x occurs r-free (imagine that the occurrences of y are in-line expanded to $+\ 1\ x$). It should be clear by now that the notion of r-closedness is context dependent, e.g., in the context of

$$\lambda x \to \texttt{let}\ y = +\ 1\ x\ \texttt{in} * y\ y$$

only the variable y occurs r-free in $*\ y\ y$. We obtain the set of r-free variables from the set of free variables by

- subtracting the `letrec`-bound variables and by
- adding for each subtracted `letrec`-bound variable the r-free variables of the defining right hand side.

This procedure precisely reflects the operational behavior of the CAM code. A formal definition of r-closedness can be found in Table 11 ($\mathbb{P}S$ denotes the powerset of S). The function r-free$[\![e]\!]\ \eta$ computes the r-free variables of the expression e relative to the environment η. The environment associates `letrec`-bound patterns with the r-free variables of their corresponding right hand sides. By means of the environment we can distinguish between `letrec`-bound variables (second equation) and λ-, `let`-, or `case`-bound variables (first equation).

The last equation of r-free needs a bit of explanation. Due to the scoping rules the extended environment must already be employed in its definition. Consequently the new environment η' is

[9] The word "necessary" should not be taken too literally. The body of the abstraction

$$\lambda n \to \texttt{if}\ \mathit{true}\ \texttt{then}\ 0\ \texttt{else}\ n$$

is *not* r-closed although the `else` branch is never entered.

vars$[\![p]\!]$ denotes the set of variables occurring in p.

$$
\begin{array}{rcl}
\text{vars} & : & \mathbf{pat} \rightarrow \mathbb{P}\ \mathbf{var} \\
\text{vars}[\![x]\!] & = & \{x\} \\
\text{vars}[\![()]\!] & = & \emptyset \\
\text{vars}[\![(p_1, p_2)]\!] & = & \text{vars}[\![p_1]\!] \cup \text{vars}[\![p_2]\!] \\
\text{vars}[\![x \texttt{ as } p]\!] & = & \{x\} \cup \text{vars}[\![p]\!]
\end{array}
$$

r-free$[\![e]\!]\ \eta$ computes the set of r-free variables occurring in e relative to the environment η.

$$
\begin{array}{rcll}
\text{r-free} & : & \mathbf{exp} \rightarrow \mathbf{env}\ (\mathbb{P}\ \mathbf{var}) \rightarrow \mathbb{P}\ \mathbf{var} & \\
\text{r-free}[\![x]\!]\ \langle \eta, p \rangle & = & \{x\} & \text{if } x \in \text{vars}[\![p]\!] \\
 & = & \text{r-free}[\![x]\!]\ \eta & \text{otherwise} \\
\text{r-free}[\![x]\!]\ \langle \eta, p \mapsto V \rangle & = & V & \text{if } x \in \text{vars}[\![p]\!] \\
 & = & \text{r-free}[\![x]\!]\ \eta & \text{otherwise} \\
\text{r-free}[\![s_{(n)}\ e_1\ \cdots\ e_n]\!]\ \eta & = & \text{r-free}[\![e_1]\!]\ \eta \cup \cdots \cup \text{r-free}[\![e_n]\!]\ \eta & \\
\text{r-free}[\![()]\!]\ \eta & = & \emptyset & \\
\text{r-free}[\![(e_1, e_2)]\!]\ \eta & = & \text{r-free}[\![e_1]\!]\ \eta \cup \text{r-free}[\![e_2]\!]\ \eta & \\
\text{r-free}[\![c\ e]\!]\ \eta & = & \text{r-free}[\![e]\!]\ \eta & \\
\text{r-free}[\![e_1\ e_2]\!]\ \eta & = & \text{r-free}[\![e_1]\!]\ \eta \cup \text{r-free}[\![e_2]\!]\ \eta & \\
\text{r-free}[\![\lambda p \rightarrow e]\!]\ \eta & = & \text{r-free}[\![e]\!]\ \langle \eta, p \rangle \setminus \text{vars}[\![p]\!] & \\
\multicolumn{4}{l}{\text{r-free}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\ \eta} \\
 & = & \text{r-free}[\![e_1]\!]\ \eta \cup \text{r-free}[\![e_2]\!]\ \eta \cup \text{r-free}[\![e_3]\!]\ \eta & \\
\multicolumn{4}{l}{\text{r-free}[\![\texttt{case } e \texttt{ of } c_1\ p_1 \rightarrow e_1 \mid \cdots \mid c_n\ p_n \rightarrow e_n]\!]\ \eta} \\
 & = & \text{r-free}[\![e]\!]\ \eta \cup \text{r-free}[\![e_1]\!]\ \langle \eta, p_1 \rangle \setminus \text{vars}[\![p_1]\!] \cup \cdots & \\
 & & \cup\, \text{r-free}[\![e_n]\!]\ \langle \eta, p_n \rangle \setminus \text{vars}[\![p_n]\!] & \\
\text{r-free}[\![\texttt{let } p_1 = e_1 \texttt{ in } e]\!]\ \eta & = & (\text{r-free}[\![e]\!]\ \langle \eta, p_1 \rangle \setminus \text{vars}[\![p_1]\!]) \cup \text{r-free}[\![e_1]\!]\ \eta & \\
\multicolumn{4}{l}{\text{r-free}[\![\texttt{letrec } p_1 = e_1\ ;\ \ldots\ ;\ p_n = e_n \texttt{ in } e]\!]\ \eta} \\
 & = & \text{r-free}[\![e]\!]\ \eta' & \\
 & & \text{where} & \\
 & & \eta' = \langle \ldots \langle \eta, p_1 \mapsto \text{r-free}[\![e_1]\!]\ \eta' \rangle \ldots, p_n \mapsto \text{r-free}[\![e_n]\!]\ \eta' \rangle &
\end{array}
$$

r-closed$[\![e]\!]\ \eta$ defines the property of r-closedness relative to the environment η.

$$
\begin{array}{rcl}
\text{r-closed} & : & \mathbf{exp} \rightarrow \mathbf{env}\ (\mathbb{P}\ \mathbf{var}) \rightarrow \mathbf{bool} \\
\text{r-closed}[\![e]\!]\ \eta & = & \text{r-free}[\![e]\!]\ \eta = \emptyset
\end{array}
$$

Table 11: The computation of r-free variables

defined with the help of a recursive equation and denotes the least fixed point of the following chain of environments.

$$\begin{array}{rcl}\eta_0 & = & \langle \ldots \langle \eta, p_1 \mapsto \emptyset \rangle \ldots, p_n \mapsto \emptyset \rangle \\ & \vdots & \\ \eta_{n+1} & = & \langle \ldots \langle \eta, p_1 \mapsto \text{r-free}[\![e_1]\!]\ \eta_n \rangle \ldots, p_n \mapsto \text{r-free}[\![e_n]\!]\ \eta_n \rangle \end{array}$$

Since r-free is monotone with respect to the environment and the domain under consideration is finite (an expression contains only a finite number of variables), a least fixed point always exists and is finite itself (the number of iterations is equal to the longest acyclic path in the static calling graph of the `letrec`-expression). For the expression e with

$$\begin{array}{lll} e = \lambda a\ b \rightarrow & \texttt{letrec} & f = \ldots a \ldots g \ldots ; \\ & & g = \ldots h \ldots ; \\ & & h = \ldots b \ldots g \ldots \\ & \texttt{in}\ \ldots & \end{array}$$

we obtain the following chain (r-free$[\![e]\!]$ $\langle\rangle$):

$$\begin{array}{rcl}\eta_0 & = & \langle\langle\langle\langle\rangle, f \mapsto \emptyset\rangle, g \mapsto \emptyset\rangle, h \mapsto \emptyset\rangle \\ \eta_1 & = & \langle\langle\langle\langle\rangle, f \mapsto \{a\}\rangle, g \mapsto \emptyset\rangle, h \mapsto \{b\}\rangle \\ \eta_2 & = & \langle\langle\langle\langle\rangle, f \mapsto \{a\}\rangle, g \mapsto \{b\}\rangle, h \mapsto \{b\}\rangle \\ \eta_3 & = & \langle\langle\langle\langle\rangle, f \mapsto \{a, b\}\rangle, g \mapsto \{b\}\rangle, h \mapsto \{b\}\rangle \\ \eta_4 & = & \eta_3 \end{array}$$

After the third iteration the fixed point is reached.

The expensive determination of the fixed point can be simplified if the `letrec`-definition satisfies the following property: The equations are mutually recursive, i.e., each defining expression depends on every remaining definition (the static calling graph consists of a single maximal strong component). The source code transformation which establishes this property is called *dependency analysis*. If the source language has a polymorphic type system (à la Hindley and Milner [22]), the analysis must be carried out anyway prior to the type inference process. The above example is transformed to the nested expression:

$$\begin{array}{lll} e = \lambda a\ b \rightarrow & \texttt{letrec} & g = \ldots h \ldots ; \\ & & h = \ldots b \ldots g \ldots\ \texttt{in} \\ & \texttt{letrec} & f = \ldots a \ldots g \ldots \\ & \texttt{in}\ \ldots & \end{array}$$

For expressions of this kind the last equation of r-free can be simplified as indicated in Table 12. The

$$\boxed{\begin{array}{l} \text{r-free}[\![\texttt{letrec}\ p_1 = e_1\ ;\ \ldots\ ; p_n = e_n\ \texttt{in}\ e]\!]\ \eta \\ \quad = \quad \text{r-free}[\![e]\!]\ \eta_1 \\ \qquad\quad \text{where} \\ \qquad\quad \eta_0 = \langle \ldots \langle \eta, p_1 = \emptyset \rangle \ldots, p_n = \emptyset \rangle \\ \qquad\quad V = \text{r-free}[\![e_1]\!]\ \eta_0 \cup \cdots \cup \text{r-free}[\![e_n]\!]\ \eta_0 \\ \qquad\quad \eta_1 = \langle \ldots \langle \eta, p_1 = V \rangle \ldots, p_n = V \rangle \end{array}}$$

Table 12: The definition of r-free revisited for truly recursive definitions

modified definition reflects the fact that the defining expressions are truly recursive.

5.2 Compiling an Expression

We have seen that the property of r-closedness depends on the context. In the remainder we mark the parts of an expression that are r-closed with a star, i.e., e^* identifies the subexpression e as r-closed (relative to the given context).

The new instructions which are required for the compilation of r-closed expressions are displayed in table 13. Table 14 shows the modified compilation schemes. Again we will consider each of them

register	stack	code	register	stack	code
		stack operations			
v	S	$\textsc{Move}\,; C$	$()$	$v : S$	C
v_1	$v_2 : S$	$\textsc{Pop}\,; C$	v_2	S	C
		register operations			
v_1	$v_2 : S$	$\textsc{Snoc}\,; C$	(v_1, v_2)	S	C
v	S	$\textsc{Comb}\ \ell\,; C$	$[\ell]$	S	C
		control instructions			
$[\ell]$	$v : S$	$\textsc{App}\,; C$	v	$C : S$	C_ℓ
true	S	$\textsc{GotoIfalse}\ \ell\,; C$	*true*	S	C
false	S	$\textsc{GotoIfalse}\ \ell\,; C$	*false*	S	C_ℓ
$(c_i : v)$	S	$\textsc{SwitchI}\ [c_1 : \ell_1, \ldots, c_n : \ell_n]\,; C$	v	S	C_{ℓ_i}

Table 13: Some more instructions

in turn.

5.2.1 Compiling pure λ-expressions

An r-closed expression does not make use of the environment at run-time. Consequently we do not need the environment at compile-time with the exception of `letrec`-caused-entries $(p \mapsto \ell)$. The annotation ρ^* indicates that only entries of the form $p \mapsto \ell$ are valid in ρ. The formal environment $\langle \rho^*, p\rangle$, which is called simple environment, represents the run-time environment consisting only of an instance of p. The CAM code generated for the expression e in the restricted environment ρ^* (notation: $\mathcal{C}[\![e]\!]\ \rho^*$) satisfies the following invariant.

> The register may contain an arbitrary value. After the execution of the code it holds the value of the expression.

The compilation scheme $\mathcal{C}[\![e]\!]\ \langle \rho^*, p\rangle$ has a special reading as well.

> If the register contains an instance of the pattern, then after the execution of the code it holds the value of the expression.

An r-closed variable must be `letrec`-bound. Furthermore, the defining expression of the variable does not contain any external references. The simplified compilation scheme $\mathcal{E}^*$ takes these facts into account: Only entries of the form $p \mapsto \ell$ are examined and the subroutine is called directly without prior restoration of the environment. The original $\mathcal{E}$ scheme must be extended for the case of simple environments. The access path for a variable in p is shortened by a $\textsc{Snd}$ instruction ($\textsc{Rest}\ n$ is used instead of $\textsc{Acc}\ n$).

The $\mathcal{T}$ scheme distinguishes between two situations: The second argument is r-closed (or both) or only the first argument is r-closed. In the former case the first expression is compiled in the initial environment, the value is moved from the register onto the stack and the second expression is compiled in an undefined environment. In the latter case the order of the evaluation is reversed and the reversal is compensated by a $\textsc{Swap}$ instruction.

The $\textsc{Move}$ instruction resembles a $\textsc{Push}$ instruction with the difference that $\textsc{Push}$ duplicates a value whereas $\textsc{Move}$ actually moves a value. After a $\textsc{Push}$ instruction the register is “alive”, after a

$$
\begin{array}{rcl}
\mathcal{E}^*[\![x]\!]\ \langle\rangle & = & \mathit{fail} \\
\mathcal{E}^*[\![x]\!]\ \langle\rho,p\rangle & = & \mathcal{E}^*[\![x]\!]\ \rho \\
\mathcal{E}^*[\![x]\!]\ \langle\rho,p\mapsto\ell\rangle & = & (\textsc{Call}\ \ell\,;\mathcal{P}[\![x]\!]\ p)\ ?\ \mathcal{E}^*[\![x]\!]\ \rho \\
\hline
\mathcal{E}[\![x]\!]\ \langle\rho^*,p\rangle\ n & = & \textsc{Rest}\ n\,;\mathcal{P}[\![x]\!]\ p \\
\hline
\mathcal{C}[\![x^*]\!]\ \rho & = & \mathcal{E}^*[\![x]\!]\ \rho \\
\mathcal{C}[\![e_1^*\ e_2]\!]\ \rho & = & \mathcal{C}[\![e_2]\!]\ \rho\,;\textsc{Move}\,;\mathcal{C}[\![e_1]\!]\ \rho^*\,;\textsc{App} \\
\mathcal{C}[\![e_1\ e_2^*]\!]\ \rho & = & \textsc{Move}\,;\mathcal{C}[\![e_2]\!]\ \rho^*\,;\textsc{Swap}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{App} \\
\mathcal{C}[\![(\lambda p\to e)^*]\!]\ \rho & = & \textsc{Comb}\ \ell \\
 & & [\ \ \ell:\mathcal{R}[\![e]\!]\ \langle\rho^*,p\rangle\ \ ] \\
\multicolumn{3}{l}{\mathcal{C}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2^*\ \texttt{else}\ e_3^*]\!]\ \rho} \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{Gotoifalse}\ \ell_1\,;\mathcal{C}[\![e_2]\!]\ \rho^*\,;\textsc{Goto}\ \ell_2\,; \\
 & & \ell_1:\mathcal{C}[\![e_3]\!]\ \rho^*\,;\ell_2:\textsc{Skip} \\
\multicolumn{3}{l}{\mathcal{C}[\![\texttt{case}\ e\ \texttt{of}\ (c_1\ p_1\to e_1)^*\mid\cdots\mid(c_n\ p_n\to e_n)^*]\!]\ \rho} \\
 & = & \mathcal{C}[\![e]\!]\ \rho\,;\textsc{Switchi}\ [c_1:\ell_1,\ldots,c_n:\ell_n]\,; \\
 & & \ell_1:\mathcal{C}[\![e_1]\!]\ \langle\rho^*,p_1\rangle\,;\textsc{Goto}\ \ell \\
 & & \vdots \\
 & & \ell_n:\mathcal{C}[\![e_n]\!]\ \langle\rho^*,p_n\rangle\,; \\
 & & \ell:\textsc{Skip} \\
\mathcal{C}[\![\texttt{let}\ p_1=e_1\ \texttt{in}\ e]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho\,;\mathcal{C}[\![e]\!]\ \langle\rho^*,p_1\rangle \qquad \text{if } \lambda p_1\to e \text{ is r-closed} \\
\mathcal{C}[\![\texttt{let}\ p_1=e_1^*\ \texttt{in}\ e]\!]\ \rho & = & \textsc{Move}\,;\mathcal{C}[\![e_1]\!]\ \rho^*\,;\textsc{Cons}\,;\mathcal{C}[\![e]\!]\ \langle\rho,p_1\rangle \\
\hline
\mathcal{T}[\![e_1,e_2^*]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{Move}\,;\mathcal{C}[\![e_2]\!]\ \rho^* \\
\mathcal{T}[\![e_1^*,e_2]\!]\ \rho & = & \mathcal{C}[\![e_2]\!]\ \rho\,;\textsc{Move}\,;\mathcal{C}[\![e_1]\!]\ \rho^*\,;\textsc{Swap}
\end{array}
$$

Table 14: The $\mathcal{E}^*$, $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{T}$ compilation schemes for r-closed expressions

Move instruction it is "dead". The distinction is important if the CAM code is further expanded to machine code or for compile-time garbage collection.

Function application is treated differently from pairs and predefined functions because we do not want function and argument to be reordered (in view of further improvements in Section 6.4).

An r-closed abstraction, classically called combinator, evaluates to a very simple closure (notation: $[\ell]$), which contains only a reference to the CAM code of the body. This closure is created by the Comb instruction. The App instruction must distinguish between a normal and a simple closure. In the latter case the App instruction boils down to an indirect jump. This distinction facilitates further code improvements as described in Section 6.4 but complicates the expansion of CAM code into machine code. The examples of Section 4.2.1 get compiled to the following code (we omit Skip and Rest 0).

$\lambda x \to +\ 1\ x$		$\lambda f\ x \to f\ (f\ x)$			$\lambda(f,x) \to f\ (f\ x)$		
	Comb ℓ		Comb ℓ_1	Swap		Comb ℓ	App
	Stop		Stop	Rest 1		Stop	Swap
ℓ :	Move	ℓ_1 :	Cur ℓ_2	App	ℓ :	Push	Fst
	Quote 1		Return	Swap		Push	App
	Swap	ℓ_2 :	Push	Rest 1		Snd	Return
	Prim +		Push	App		Swap	
	Return		Acc 0	Return		Fst	

Note that if we bear in mind that Acc 0 is equal to Snd and Rest 1 is equal to Fst, the code generated for the function bodies in the second and third example is identical.

5.2.2 Compiling an alternative

If neither of the branches needs the environment, the initial PUSH instruction can be omitted. The GOTOIFALSE (branch immediate) instruction acts like a GOTOFALSE without restoring the environment. The Heaviside function exemplifies the new compilation scheme.

$\lambda n \rightarrow$ `if` $\leq n$ 0 `then` 0 `else` 1			
COMB ℓ_1 STOP ℓ_1 : MOVE	QUOTE 0 PRIM $\leq$ GOTOIFALSE ℓ_2	QUOTE 0 GOTO ℓ_3 ℓ_2 : QUOTE 1	ℓ_3 : RETURN

5.2.3 Compiling algebraic datatypes

In analogy to the `if`-construct we can save a PUSH instruction if neither of the branches contains other references than to the arguments of the corresponding constructor. The example of 4.2.2 benefits from this improvement.

$\lambda s \rightarrow$ `case` s `of` nil () $\rightarrow$ nil () $\mid$ $cons$ $(c, t) \rightarrow t$		
COMB ℓ_1 STOP ℓ_1 : SWITCHI $[nil : \ell_2, cons : \ell_3]$	ℓ_2 : QUOTE () PACK nil GOTO ℓ_4	ℓ_3 : SND ℓ_4 : RETURN

5.2.4 Compiling a local definition

The CAM code for the expression `let` $p_1 = e_1$ `in` e can be drastically simplified if e contains only references to variables in p_1 by just sequencing the code fragments of e_1 and e. The first example of Section 4.2.4 is a candidate for this improvement.

`let` $a = 5$ `in` $*\ a\ a$	
QUOTE 5 PUSH	SWAP PRIM $*$

As a final example, we give the improved code for the *even* function of Section 4.2.4.

`letrec` $even = \lambda n \rightarrow$ `if` $= n$ 0 `then` $true$ `else` $\neg$ $even$ $(dec\ n)$ `in` $even$ 56			
QUOTE 56 MOVE CALL ℓ_1 APP STOP	ℓ_1 : COMB ℓ_2 RETURN ℓ_2 : PUSH MOVE QUOTE 0	PRIM = GOTOFALSE ℓ_3 QUOTE $true$ GOTO ℓ_4 ℓ_3 : PRIM dec	MOVE CALL ℓ_1 APP PRIM $\neg$ ℓ_4 : RETURN

6 Peephole Optimization

The purpose of this section is to show how the generated CAM code can be further improved. The technique we employ is a very simple one called *peephole optimization* and works as follows: We examine a short sequence of the code and try to replace it by an equivalent sequence which is either shorter or faster. This process is repeated until no more improvements are applicable. Typically one improvement spawns opportunities for additional improvements.

The improvements with the exception of last call optimization are already described elsewhere [28] but they are tightly coupled with the compilation process in the cited paper. We hope that a separation of the phases (code generation and optimization) helps towards a better understanding of the topic. Note that most of the optimizations are only applicable because we reversed the order in which functions and arguments are compiled.

6.1 Replacement Systems

Before we look at the different sources of improvements, we would like to give a short account of the theoretical background. Peephole optimizations can be viewed as an instance of what we call a *replacement system*.

Replacement systems are similar to string rewriting systems [9] and Markov algorithms [25]. Formally, a replacement system is a pair (Σ, R), where Σ is an alphabet and $P \subseteq \Sigma^* \times \Sigma^*$ is a set of ordered pairs of words over Σ. The elements of R are called *optimization rules* and are denoted by $\alpha \to \beta$. We assume that the optimization rules are given in a linear order.

$$\alpha_1 \to \beta_1, \ldots, \alpha_n \to \beta_n$$

Note that since we want to view a single CAM instruction as a letter of the alphabet, we do not demand the alphabet to be finite (of course it must be decidable). Consequently the set of rules need not be finite as well (albeit given a word α we must be able to determine effectively whether R contains a rule of the form $\alpha \to \beta$).

A replacement system translates a word u into a word v. At each step of the replacement process the leftmost subword which matches a left hand side is replaced by the corresponding right hand side. The rules are tried in the order given. The process terminates if none of the rules is applicable. Formally, the relation $u \Rightarrow v$ (u yields directly v) holds iff

1. there is a rule $\alpha_i \to \beta_i \in R$ and there are words u_1 and u_2 such that $u = u_1\alpha_i u_2$ and $v = u_1\beta_i u_2$ and
2. there is no rule $\alpha_j \to \beta_j \in R$ such that there are words v_1 and v_2 with $u = v_1\alpha_j v_2$ and $|v_1| < |u_1| \ \vee \ (|v_1| = |u_1| \ \wedge \ j < i)$.

The second condition implies that the leftmost occurrence of the left hand side in u must be replaced. As an immediate consequence of the definition, there is at most one word v with the property $u \Rightarrow^* v \wedge \neg\exists w \ \ v \Rightarrow w$. If there is no such word, the replacement system loops.

In contrast to string rewriting systems replacement systems are deterministic. The former must satisfy non-trivial properties like confluence to guarantee that the final outcome is determined. Markov algorithms first impose an order on the rules and then on the position of the left hand sides in the string, whereas the situation is reversed with replacement systems.

A naïve implementation of a replacement system would repeatedly scan a given word from left to right looking for instances of left hand sides. Of course there is a more efficient way. After a replacement has taken place, we may in some cases safely ignore the left context in search of the next replacement. To be more precise, let

$$u_1\alpha u_2 \quad \Rightarrow \quad u_1\beta u_2$$

be the last step. The position where the next search may safely start is given by the current position (first letter of β) plus an offset k. The offset k depends only on the last rule which was applied and can be determined in advance (the definition actually underestimates the possible offset).

$$\begin{aligned} k \quad = \quad \min(\quad & \{ -|w_1| \mid \exists \gamma \to \delta \in R \quad \exists w_1, w_2, w_4 \in \Sigma^* \\ & \qquad\qquad \text{with } w_1\beta w_2 = \gamma w_4 \text{ and } |w_1| < |\gamma| \,\} \\ & \cup \{ |w_3| \mid \exists \gamma \to \delta \in R \quad \exists w_2, w_3, w_4 \in \Sigma^* \\ & \qquad\qquad \text{with } \beta w_2 = w_3\gamma w_4 \text{ and } |w_3| < |\beta| \,\} \\ & \cup \{|\beta|\}) \end{aligned}$$

The offset is negative if there is a left hand side γ with which β overlaps from the right (there must be an overlap, otherwise α was not the leftmost occurrence of a left hand side in the preceding step). In the worst case, k is equal to the length of the longest left hand side minus 1. If there is no such overlap but an overlap from the left, the offset is positive but smaller than $|\beta|$. If the right hand side β has nothing in common with the left hand sides, we may safely start the search at the first letter of u_2.

Since the optimization rules we will introduce in the subsequent sections consider at most two instructions at a time, the offset k has a lower bound of -1.

6.2 Converse Operations

Algebraic properties of predefined operations can be used to improve the code. Let us take a look at the $\mathcal{T}$ scheme. If the first argument is r-closed but not the second, the arguments are compiled in reversed order. The reversal is compensated by a trailing SWAP instruction. This instruction is clearly unnecessary if the subsequent operation is commutative.

$$\begin{aligned}\mathcal{C}[\![+\ 3\ x]\!]\ \rho &= \mathcal{C}[\![x]\!]\ \rho\,;\textsc{Move}\,;\textsc{Quote}\ 3\,;\underline{\textsc{Swap}\,;\textsc{Prim}\ +}\\ &= \mathcal{C}[\![x]\!]\ \rho\,;\textsc{Move}\,;\textsc{Quote}\ 3\,;\overline{\textsc{Prim}\ +}\end{aligned}$$

Many operations possess a simple converse operation (f^c is called the converse operations of f iff $f(x_1, x_2) = f^c(x_2, x_1)$), which can be used instead of the sequence SWAP ; PRIM $s_{(2)}$. The converse operations of some primitives are displayed below with $sub(m, n) = n - m$ and $div(m, n) = n/m$.

$s_{(2)}$	$+$	$-$	$*$	$/$	$<$	$\leq$	$=$	$\geq$	$>$	$\wedge$	$\vee$
$s^c_{(2)}$	$+$	sub	$*$	div	$>$	$\geq$	$=$	$\leq$	$<$	$\wedge$	$\vee$

Consequently we get the following optimization rules.

code	improved code	offset
SWAP ; CONS	SNOC	-1
SWAP ; SNOC	CONS	-1
SWAP ; PRIM $s_{(2)}$	PRIM $s^c_{(2)}$	-1

Attention must be paid not to modify parts marked with a label. It is *not* safe to replace SWAP ; ℓ : CONS by ℓ : SNOC.

6.3 Access Instructions

The optimization rules presented in this section are mainly of cosmetic nature and could be built directly into the compilation schemes albeit with a loss of clarity. The improvements only affect the size of the CAM code but have little or no impact on the run-time behavior.

The SKIP instruction and the REST 0 instructions (camouflaged SKIP) are clearly superfluous. We have already omitted them in the preceding sections. The REST 1 and ACC 0 instructions have simpler variants FST and SND which can be used instead.

code	improved code	offset
SKIP		-1
REST 0		-1
REST 1	FST	-1
ACC 0	SND	-1

Tuples are represented by nested pairs. The structure of the nesting was chosen carefully so that the instructions ACC and REST can be used to access components of a tuple.

$$\begin{aligned}\mathcal{C}[\![\lambda(x_1, x_2, x_3, x_4, x_5) \rightarrow x_1]\!]\ \rho &= \textsc{Comb}\ \ell\\ &\quad [\ \ell : \textsc{Fst}\,;\textsc{Fst}\,;\textsc{Fst}\,;\textsc{Fst}\,;\textsc{Return}\ ]\\ \mathcal{C}[\![\lambda(x_1, x_2, x_3, x_4, x_5) \rightarrow x_3]\!]\ \rho &= \textsc{Comb}\ \ell\\ &\quad [\ \ell : \textsc{Fst}\,;\textsc{Fst}\,;\textsc{Snd}\,;\textsc{Return}\ ]\end{aligned}$$

The sequence of access instructions can be condensed into a REST 4 respectively an ACC 2 instruction. The following rules will do the job.

code	improved code	offset
FST ; FST	REST 2	0
FST ; SND	ACC 1	1
REST n ; FST	REST $(n + 1)$ if $n \geq 2$	0
REST n ; SND	ACC n if $n \geq 2$	1

Note that these rules suffice—a rule like REST m ; REST $n \to$ REST $(m+n)$ is not necessary—as the compiler generates only access sequences of the following form.

$$(\text{ACC } n \mid \text{REST } n)\,;\,(\text{FST} \mid \text{SND})^*$$

6.4 Abstraction and Application

The biggest potential for optimizations lies in the combination of functional abstraction and application. In Section 4.2.4 we have already used the equivalence of

$$\texttt{let}\ p_1 = e_1\ \texttt{in}\ e_2$$

with the β-redex

$$(\lambda p_1 \to e_2)\ e_1$$

for the derivation of a compilation rule for the `let`-construct. Nevertheless, the generated CAM code for the two expressions differs significantly in size and speed.

$$\begin{array}{rcl}
\mathcal{C}[\![(\lambda p_1 \to e_2)\ e_1]\!]\ \rho & = & \text{PUSH}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\text{SWAP}\,;\text{CUR}\ \ell\,;\text{APP} \\
 & & [\ \ell : \mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle\,;\text{RETURN}\ ] \\
\mathcal{C}[\![\texttt{let}\ p_1 = e_1\ \texttt{in}\ e_2]\!]\ \rho & = & \text{PUSH}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\text{CONS}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle
\end{array}$$

In what follows we will show how to derive the second sequence from the first one. It is instructive to follow the execution of CUR ℓ; APP (left column). The value of e_1 is located on the stack (v_2); the register contains the current environment (v_1).

register	stack	code	register	stack	code
v_1	$v_2 : S$	CUR ℓ; APP ; C	v_1	$v_2 : S$	SNOC ; CALL ℓ ; C
$[v_1 : \ell]$	$v_2 : S$	APP ; C	(v_1, v_2)	S	CALL ℓ ; C
(v_1, v_2)	$C : S$	C_ℓ	(v_1, v_2)	$C : S$	C_ℓ

The code fragment effectively cons'es the environment with the value and calls the subroutine labeled with ℓ. The intermediate building of the closure $[v_1 : \ell]$ can be spared if one uses the SNOC and the CALL instruction instead (right column). The APP instruction is a very complex operation and should be avoided whenever possible. Thus we obtain the following code for $(\lambda p_1 \to e_2)\ e_1$.

$$\begin{array}{rl}
= & \text{PUSH}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\underline{\text{SWAP}\,;\text{SNOC}}\,;\text{CALL}\ \ell \\
 & [\ \ell : \mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle\,;\text{RETURN}\ ] \\
= & \text{PUSH}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\text{CONS}\,;\text{CALL}\ \ell \\
 & [\ \ell : \mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle\,;\text{RETURN}\ ]
\end{array}$$

Since the CALL instruction is the only one which refers to the subroutine, we could replace CALL by the code of the subroutine (exclusive RETURN) and dispose the subroutine itself. We resist the temptation to do so because the inline expansion would require some sort of bookkeeping mechanism (how many instructions refer to a label) and because it does not go well together with a further optimization we have in mind (cf. Section 6.6).

If the abstraction is r-closed, the combination of abstraction and application gets compiled to the following code.

$$\begin{array}{rcl}
\mathcal{C}[\![(\lambda p_1 \to e_2)^*\ e_1]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho\,;\text{MOVE}\,;\text{COMB}\ \ell\,;\text{APP} \\
 & & [\ \ell : \mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle\,;\text{RETURN}\ ]
\end{array}$$

The sequence COMB ℓ; APP may be improved in a similar way as CUR ℓ; APP. At this point the POP instruction comes into play.

register	stack	code	register	stack	code
v_1	$v_2 : S$	COMB ℓ; APP ; C	v_1	$v_2 : S$	POP ; CALL ℓ ; C
$[\ell]$	$v_2 : S$	APP ; C	v_2	S	CALL ℓ ; C
v_2	$C : S$	C_ℓ	v_2	$C : S$	C_ℓ

The $\textsc{Pop}$ instruction is inverse to $\textsc{Push}$ and $\textsc{Move}$, that is to say, $\textsc{Pop}$ compensates the effect of $\textsc{Push}$ and $\textsc{Move}$. Thus we obtain:

$$
\begin{array}{rl}
= & \mathcal{C}[\![e_1]\!]\ \rho\,;\ \underline{\textsc{Move}\,;\ \textsc{Pop}}\,;\ \textsc{Call}\ \ell \\
 & [\ \ell : \mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle\,;\ \textsc{Return}\ ] \\
= & \mathcal{C}[\![e_1]\!]\ \rho\,;\ \textsc{Call}\ \ell \\
 & [\ \ell : \mathcal{C}[\![e_2]\!]\ \langle\rho, p_1\rangle\,;\ \textsc{Return}\ ]
\end{array}
$$

If we expanded the subroutine call, we would obtain exactly the code sequence which is generated by the optimized $\mathcal{C}$ scheme for the `let`-construct.

Iterated applications and abstractions require iterated application of the optimization rules.

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathcal{C}[\![(\lambda a\ b \to -\ b\ a)\ 7\ 8]\!]\ \rho} \\
= & \textsc{Quote}\ 8\,;\ \textsc{Move}\,;\ \textsc{Quote}\ 7\,;\ \textsc{Move}\,;\ \underline{\textsc{Comb}\ \ell_1\,;\ \textsc{App}}\,;\ \textsc{App} \\
 & [\ \ell_1 : \textsc{Cur}\ \ell_2\,;\ \textsc{Return}\ ] \\
 & [\ \ell_2 : \textsc{Push}\,;\ \textsc{Acc}\ 0\,;\ \textsc{Swap}\,;\ \textsc{Rest}\ 1\,;\ \textsc{Prim}\ -\,;\ \textsc{Return}\ ] \\
= & \textsc{Quote}\ 8\,;\ \textsc{Move}\,;\ \textsc{Quote}\ 7\,;\ \underline{\textsc{Move}\,;\ \textsc{Pop}}\,;\ \textsc{Call}\ \ell_1\,;\ \textsc{App} \\
= & \textsc{Quote}\ 8\,;\ \textsc{Move}\,;\ \textsc{Quote}\ 7\,;\ \overline{\textsc{Call}\ \ell_1}\,;\ \textsc{App}
\end{array}
$$

The body of the subroutine labeled with ℓ_1 consists only of a single instruction. In this special case we replace $\textsc{Call}$ by the respective instruction.

$$
\begin{array}{rl}
= & \textsc{Quote}\ 8\,;\ \textsc{Move}\,;\ \textsc{Quote}\ 7\,;\ \underline{\textsc{Cur}\ \ell_2\,;\ \textsc{App}} \\
= & \textsc{Quote}\ 8\,;\ \textsc{Move}\,;\ \textsc{Quote}\ 7\,;\ \overline{\textsc{Snoc}\,;\ \textsc{Call}\ \ell_2}
\end{array}
$$

The code corresponds with a minor exception (the order of the arguments 7 and 8 is reversed) exactly to the code generated for the uncurried variant.

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathcal{C}[\![(\lambda(a,b) \to -\ b\ a)\ (7,8)]\!]\ \rho} \\
= & \textsc{Quote}\ 7\,;\ \textsc{Move}\,;\ \textsc{Quote}\ 8\,;\ \textsc{Cons}\,;\ \textsc{Call}\ \ell_2 \\
 & [\ \ell_2 : \textsc{Push}\,;\ \textsc{Snd}\,;\ \textsc{Swap}\,;\ \textsc{Fst}\,;\ \textsc{Prim}\ -\,;\ \textsc{Return}\ ]
\end{array}
$$

The example shows that curried functions can be used without loss of efficiency. Again, if we expanded the subroutine call, we would obtain the code sequence the following expressions get compiled to.

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathcal{C}[\![\texttt{let}\ a = 7\,;\ b = 8\ \texttt{in}\ -\ b\ a]\!]\ \rho} \\
= & \mathcal{C}[\![\texttt{let}\ a = 7\ \texttt{in}\ \texttt{let}\ b = 8\ \texttt{in}\ -\ b\ a]\!]\ \rho \\
= & \textsc{Quote}\ 7\,;\ \textsc{Move}\,;\ \textsc{Quote}\ 8\,;\ \textsc{Cons}\,;\ \textsc{Push}\,;\ \textsc{Snd}\,;\ \textsc{Swap}\,;\ \textsc{Fst}\,;\ \textsc{Prim}\ -
\end{array}
$$

The optimization rules introduced in this section are summarized below.

code	improved code	offset
$\textsc{Move}\,;\ \textsc{Pop}$		-1
$\textsc{Cur}\ \ell\,;\ \textsc{App}$	$\textsc{Snoc}\,;\ \textsc{Call}\ \ell$	-1
$\textsc{Comb}\ \ell\,;\ \textsc{App}$	$\textsc{Pop}\,;\ \textsc{Call}\ \ell$	-1
$\textsc{Call}\ \ell\,[\ \ell : I\,;\ \textsc{Return}\ ]$	$I\,[\ \ell : I\,;\ \textsc{Return}\ ]$	-1

Note that the expanded $\textsc{Call}$ instruction must not be identical with I (this restriction is necessary to guarantee the termination of the replacement process).

6.5 Local Function Definitions

A `letrec`-bound function applied to arguments is a combination of functional abstraction and application in disguise. Hence all of the optimizations introduced in the last section are also applicable in this context. Let us assume that the local function satisfies the property of r-closedness.

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathcal{C}[\![\texttt{letrec}\ f^* = \lambda p \to e_1\ \texttt{in}\ f\ e_2]\!]\ \rho} \\
= & \mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\ \textsc{Move}\,;\ \underline{\textsc{Call}\ \ell_1}\,;\ \textsc{App} \\
 & [\ \ell_1 : \textsc{Comb}\ \ell_2\,;\ \textsc{Return}\ ] \\
 & [\ \ell_2 : \mathcal{C}[\![e_1]\!]\ \langle\langle\rho, f \mapsto \ell_1\rangle^*, p\rangle\,;\ \textsc{Return}\ ] \\
= & \mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\ \textsc{Move}\,;\ \underline{\textsc{Comb}\ \ell_2\,;\ \textsc{App}} \\
= & \mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\ \underline{\textsc{Move}\,;\ \textsc{Pop}}\,;\ \textsc{Call}\ \ell_2 \\
= & \mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\ \overline{\textsc{Call}\ \ell_2}
\end{array}
$$

The calling sequence resembles the one on a conventional stack architecture: The single parameter is loaded into the register, which serves as a cache for the topmost stack element, and the subroutine is called. If f is not r-closed, we obtain the following code sequence.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}[\![\texttt{letrec}\ f = \lambda p \rightarrow e_1\ \texttt{in}\ f\ e_2]\!]\ \rho} \\
= & \textsc{Push}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\textsc{Swap}\,;\underline{\textsc{Rest}\ 0}\,;\textsc{Call}\ \ell_1\,;\textsc{App} \\
 & [\ \ell_1 : \textsc{Cur}\ \ell_2\,;\textsc{Return}\ ] \\
 & [\ \ell_2 : \mathcal{C}[\![e_1]\!]\ \langle\langle\rho, f \mapsto \ell_1\rangle, p\rangle\,;\textsc{Return}\ ] \\
= & \textsc{Push}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\textsc{Swap}\,;\underline{\textsc{Call}\ \ell_1}\,;\textsc{App} \\
= & \textsc{Push}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\textsc{Swap}\,;\underline{\textsc{Cur}\ \ell_2\,;\textsc{App}} \\
= & \textsc{Push}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\underline{\textsc{Swap}\,;\textsc{Snoc}}\,;\textsc{Call}\ \ell_2 \\
= & \textsc{Push}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, f \mapsto \ell_1\rangle\,;\textsc{Cons}\,;\textsc{Call}\ \ell_2
\end{array}
$$

The parameter is paired with the current environment, then the subroutine is called. If the call to f is situated in an abstraction, the environment must be restored by an REST n instruction with $n > 0$, which prevents the application of the SWAP ; SNOC rule resulting in a slightly longer code sequence.

Thus, after expanding the first CALL instruction we carry out the same simplifications as in Section 6.4 with the minor difference that REST n possibly prevents some optimizations.[10]

We have said in Section 2 that non-recursive functions should also be bound by `letrec` rather than `let`. We are now in a position to justify the advice by looking at the code sequence generated for a `let`-bound function.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}[\![\texttt{let}\ f = \lambda p \rightarrow e_1\ \texttt{in}\ f\ e_2]\!]\ \rho} \\
= & \textsc{Push}\,;\textsc{Cur}\ \ell\,;\textsc{Cons}\,;\textsc{Push}\,;\mathcal{C}[\![e_2]\!]\ \langle\rho, f\rangle\,;\textsc{Swap}\,;\textsc{Acc}\ 0\,;\textsc{App} \\
 & [\ \ell_1 : \mathcal{C}[\![e_1]\!]\ \langle\rho, p\rangle\,;\textsc{Return}\ ]
\end{array}
$$

The code does not offer any opportunity for improvements and is consequently inferior to the code generated for the `letrec`-construct.

Functions with multiple arguments can be improved by applying the optimization rules repeatedly.

`letrec` $f = \lambda x\ y\ z \rightarrow +\ (*\ x\ y)\ z$ `in` $f\ 3\ 4\ 5$				
QUOTE 5		SNOC	PUSH	SWAP
MOVE		SNOC	REST 2	SND
QUOTE 4		CALL ℓ	SWAP	PRIM +
MOVE		STOP	ACC 1	RETURN
QUOTE 3	ℓ :	PUSH	PRIM $*$	

Note that we have omitted the subroutines which are not addressed anywhere (dead code elimination). The code is nearly identical to the code generated for `letrec` $f = \lambda(x, y, z) \rightarrow +\ (*\ x\ y)\ z$ `in` $f\ (3, 4, 5)$ the only difference being the order in which the arguments are processed. Thus the use of curried functions, which are superior to their uncurried counterparts because of their greater flexibility, does not result in a loss of efficiency. The code of the *even* function (cf. Section 5.2.4) also benefits from

[10] With the following exception our technique subsumes the optimizations described in [28]. If the call to f lies within an abstraction (the nesting level n is indicated by the bracketed superscript $f^{(n)}$) and the argument is r-closed, we get the following code sequence,

$$\mathcal{C}[\![f^{(n)}\ e^*]\!]\ \rho \quad = \quad \textsc{Move}\,;\mathcal{C}[\![e]\!]\ \rho^*\,;\textsc{Swap}\,;\textsc{Rest}\ n\,;\textsc{Snoc}\,;\textsc{Call}\ \ell$$

whereas Suárez obtains:

$$\mathcal{C}[\![f^{(n)}\ e^*]\!]\ \rho \quad = \quad \textsc{Rest}\ n\,;\textsc{Move}\,;\mathcal{C}[\![e]\!]\ \rho^*\,;\textsc{Cons}\,;\textsc{Call}\ \ell$$

A SWAP instruction is saved by reversing the argument and the restoration of the environment. This improvement is difficult to achieve in our framework because we must insist on the special order in which the argument and the function call are compiled. Otherwise, the CUR ℓ; APP rule would no longer be applicable.

the improvements described in this section.

$\texttt{letrec}\ even = \lambda n \to \ \texttt{if} = n\ 0\ \texttt{then}\ true$ $\texttt{else}\ \neg\ even\ (dec\ n)$ $\texttt{in}\ even\ 56$			
QUOTE 56 CALL ℓ_1 STOP	ℓ_1 : PUSH MOVE QUOTE 0 PRIM =	GOTOFALSE ℓ_2 QUOTE *true* GOTO ℓ_3 ℓ_2 : PRIM *dec*	CALL ℓ_1 PRIM $\neg$ ℓ_3 : RETURN

6.6 Last Call Optimization

If the last instruction executed in the body of a function is a call to the same function, the call is termed *tail recursive*. It is well-known that tail recursive calls can be replaced by jumps. After this transformation, a tail recursive function, i.e., a function where every recursive call is tail recursive, runs with constant stack space. On conventional stack architectures this behavior is achieved by deallocating the stack used by the function *prior* to the recursive call. This technique is commonly called *tail recursion optimization*. If the technique is generalized to arbitrary calls in a tail position, it is termed *last call optimization*.

Last call optimization is particularly easy to achieve in our setting because we must deal only with single argument functions—the environment being the only argument. The single argument is placed into the register, no additional stack space is allocated. The following example illustrates the point (we do not bother that f never terminates).

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{letrec}\ f^* = \lambda p \to f\ e_1\ \texttt{in}\ e_2]\!]\ \rho \\
&\quad = \ \mathcal{C}[\![e_2]\!]\ \langle \rho, f \mapsto \ell_1 \rangle \\
&\qquad\quad [\ \ell_1 : \textsc{Comb}\ \ell_2\,;\ \textsc{Return}\ ] \\
&\qquad\quad [\ \ell_2 : \mathcal{C}[\![e_1]\!]\ \langle \langle \rho, f \mapsto \ell_1 \rangle^*, p \rangle\,;\ \textsc{Call}\ \ell_2\,;\ \textsc{Return}\ ]
\end{aligned}
$$

Upon the entry of the subroutine labeled with ℓ_2 the register contains an instance of p. Prior to the recursive call the instance is replaced by the value of e_1 (the stack is unchanged). Thus the CAM automatically supports the stack recovery mechanism of last call optimization. If the code sequence CALL ℓ_2 ; RETURN is replaced by a single GOTO ℓ_2 instruction, we obtain a (nonterminating) loop.

$$
\begin{aligned}
&\quad = \ \ldots \\
&\qquad\quad [\ \ell_2 : \mathcal{C}[\![e_1]\!]\ \langle \langle \rho, f \mapsto \ell_1 \rangle^*, p \rangle\,;\ \textsc{Goto}\ \ell_2\ ]
\end{aligned}
$$

It is easy to see that the above transformation is correct. We may assume that the body of a subroutine (C) only affects the register (mapping v_1 to v_2) leaving the stack unchanged.

register	stack	code	register	stack	code
v_1	$C_1 : S$	CALL ℓ; RETURN [$\ell : C$; RETURN]	v_1	$C_1 : S$	GOTO ℓ [$\ell : C$; RETURN]
v_1	RETURN : $C_1 : S$	C; RETURN	v_1	$C_1 : S$	C; RETURN
v_2	RETURN : $C_1 : S$	RETURN	v_2	$C_1 : S$	RETURN
v_2	$C_1 : S$	RETURN	v_2	S	C_1
v_2	S	C_1			

Thus we achieve last call optimization simply by applying the following optimization rule.

code	improved code	offset
CALL ℓ; RETURN	GOTO ℓ	1

In order to improve the applicability of this rule, we have to change the $\mathcal{R}$ scheme. The RETURN instruction is moved into the branches of alternatives and `case`-expressions (Table 15).

$$\begin{array}{lll}
\mathcal{R}[\![\texttt{if } e_1 \texttt{ then } e_2^* \texttt{ else } e_3^*]\!]\ \rho & & \\
& = & \mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{GotoIfalse}\ \ell_1\,;\mathcal{R}[\![e_2]\!]\ \rho^*\,;\ell_1 : \mathcal{R}[\![e_3]\!]\ \rho^* \\
\mathcal{R}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\ \rho & & \\
& = & \textsc{Push}\,;\mathcal{C}[\![e_1]\!]\ \rho\,;\textsc{Gotofalse}\ \ell_1\,;\mathcal{R}[\![e_2]\!]\ \rho\,;\ell_1 : \mathcal{R}[\![e_3]\!]\ \rho \\
\mathcal{R}[\![\texttt{case } e \texttt{ of } (c_1\ p_1 \to e_1)^* \mid \cdots \mid (c_n\ p_n \to e_n)^*]\!]\ \rho & & \\
& = & \mathcal{C}[\![e]\!]\ \rho\,;\textsc{Switchi}\ [c_1 : \ell_1, \ldots, c_n : \ell_n]\,; \\
& & \ell_1 : \mathcal{R}[\![e_1]\!]\ \langle \rho^*, p_1 \rangle\,; \\
& & \vdots \\
& & \ell_n : \mathcal{R}[\![e_n]\!]\ \langle \rho^*, p_n \rangle\,; \\
\mathcal{R}[\![\texttt{case } e \texttt{ of } c_1\ p_1 \to e_1 \mid \cdots \mid c_n\ p_n \to e_n]\!]\ \rho & & \\
& = & \textsc{Push}\,;\mathcal{C}[\![e]\!]\ \rho\,;\textsc{Switch}\ [c_1 : \ell_1, \ldots, c_n : \ell_n]\,; \\
& & \ell_1 : \mathcal{R}[\![e_1]\!]\ \langle \rho, p_1 \rangle\,; \\
& & \vdots \\
& & \ell_n : \mathcal{R}[\![e_n]\!]\ \langle \rho, p_n \rangle\,; \\
\mathcal{R}[\![e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho\,;\textsc{Return}
\end{array}$$

Table 15: The $\mathcal{R}$ compilation scheme for last call optimization

The mutual recursive definitions of *even* and *odd* serve as an example for the effects of last call optimization.

$$\begin{array}{lrcll}
\texttt{letrec} & even & = & \lambda n \to & \texttt{if} = n\ 0\ \texttt{then}\ true \\
& & & & \texttt{else}\ odd\ (dec\ n)\,; \\
& odd & = & \lambda n \to & \texttt{if} = n\ 0\ \texttt{then}\ false \\
& & & & \texttt{else}\ even\ (dec\ n) \\
\texttt{in}\ even\ 56 & & & &
\end{array}$$

	QUOTE 56		PRIM =	ℓ_3 :	PUSH		RETURN
	CALL ℓ_3		GOTOFALSE ℓ_2		MOVE	ℓ_4 :	PRIM *dec*
	STOP		QUOTE *true*		QUOTE 0		GOTO ℓ_1
ℓ_1 :	PUSH		RETURN		PRIM =		
	MOVE	ℓ_2 :	PRIM *dec*		GOTOFALSE ℓ_4		
	QUOTE 0		GOTO ℓ_3		QUOTE *false*		

6.7 Miscellaneous

The optimization rules should be chosen very carefully in order to improve the most frequently generated code sequences. Many algebraic properties of CAM instructions do not serve well as optimization rules because the code schemes never generate the respective sequences.

code	improved code	offset
PUSH ; POP		−1

The above rule is never applicable. In some cases it is questionable whether the effect is worth the effort because the code sequences in question are seldom generated.

code	improved code	offset
PUSH ; SWAP	PUSH	0

The following expression shows one of the rare chances to apply the rule above.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}[\![*\ n\ (+\ n\ 1)]\!]\ \langle \rho^*, n \rangle} \\
= & \underline{\textsc{Push}\,;\textsc{Swap}}\,;\textsc{Move}\,;\textsc{Quote}\ 1\,;\textsc{Prim}\ +\,;\textsc{Prim}\ * \\
= & \overline{\textsc{Push}}\,;\textsc{Move}\,;\textsc{Quote}\ 1\,;\textsc{Prim}\ +\,;\textsc{Prim}\ *
\end{array}$$

code	improved code		offset
access instructions			
SKIP			-1
REST 0			-1
REST 1	FST		-1
ACC 0	SND		-1
FST ; FST	REST 2		0
FST ; SND	ACC 1		1
REST n ; FST	REST $(n+1)$	if $n \geq 2$	0
REST n ; SND	ACC n	if $n \geq 2$	1
stack operations			
PUSH ; SWAP	PUSH		0
MOVE ; POP			-1
register operations			
SWAP ; CONS	SNOC		-1
SWAP ; SNOC	CONS		-1
SWAP ; PRIM $s_{(2)}$	PRIM $s^c_{(2)}$		-1
control instructions			
CUR ℓ ; APP	SNOC ; CALL ℓ		-1
COMB ℓ ; APP	POP ; CALL ℓ		-1
CALL ℓ [$\ell : I$; RETURN]	I [$\ell : I$; RETURN]		-1
CALL ℓ ; RETURN	GOTO ℓ		1

Table 16: Optimization rules

The optimization rules are summarized in Table 16.

In the remainder we name some of the advantages and disadvantages of peephole optimizations in contrast to source code transformations like partial evaluation.

It is obvious that the compilation of a λ-expression to a sequence of CAM instructions is a structure loosing mapping. Consequently it is much harder—although not impossible—to mimic source code transformations like $fst(e_1, e_2) \Rightarrow e_1$ on the level of CAM instructions.

Since the granularity of CAM instructions is much finer, peephole optimizations sometimes do not correspond to a transformation on the source code level. Furthermore, a single optimization rule may be applicable to code sequences stemming from different compilation schemes.

The optimization rules introduced in Section 6.2 serve as an example. The SWAP instruction does not only appear in the optimized $\mathcal{T}$ scheme but also in the ordinary $\mathcal{T}$ scheme. If the second argument of $\mathcal{T}$ compiles to the empty sequence, the code can be further improved.

$$\begin{array}{rcl} \mathcal{C}[\![\texttt{let}\ a = 5\ \texttt{in} * a\ a]\!]\ \rho & = & \textsc{Quote}\ 5\,;\textsc{Push}\,;\underline{\textsc{Swap}\,;\textsc{Prim}\ *} \\ & = & \textsc{Quote}\ 5\,;\textsc{Push}\,;\textsc{Prim}\ * \end{array}$$

In general, source code transformations such as common subexpression elimination, reduction in strength and code motion are complementary to target code transformations rather than competing.

7 Perspectives

We have seen in the introduction that free variables occurring in an abstraction prevent us from using a conventional stack architecture. The solution was to dispose the stack as a bookkeeping-mechanism for variables and to use environments instead. This reaction is quite extreme, it should be clear that the stack could be used nonetheless in many cases. A variable which has a free occurrence in an abstraction is held in the environment (access time linear to the nesting level), otherwise it is put onto the stack (constant access time). This distinction is already made in Cardelli's FAM, where variables are classified as global or local. Suárez similarly distinguishes between persistent and ephemeral variables. Note that this improvement does not only reduce the size of the environment (speeding up the access to persistent variables) but as a side-effect also increases the number of r-closed expressions (cf. Section 5) resulting in a more compact code.

The notion of persistency is relative. If the body of an abstraction is entered, a persistent variable may turn to an ephemeral one. In case the variable is accessed three or more times it may be worthwhile to copy it onto the stack. This also opens new perspectives in optimizing (local) function calls. Let us assume that all the variables occurring in the body of a (local) function are ephemeral. Thus they can safely be copied onto the stack. Fully parameterized (recursive) calls to this function can be compiled more efficiently by directly pushing the arguments onto the stack and entering the code of the body after the initial copy sequence. This scheme corresponds to the usual calling mechanism in stack-based implementations. Function calls implemented in this fashion can be optimized even more by a general stack recovery mechanism called *stack trimming*. Stack trimming can be viewed as a generalization of last call optimization and works as follows. The arguments of a function are pushed onto the stack using a special ordering. The parameter which occurs in the rightmost position and consequently lives for the longest time is pushed first and so forth. *Prior* to (recursive) calls in the body stack space is freed by removing those variables from the stack which are not accessed any longer *after* the call. This technique applies to every function call in the body last call optimization being only the special case of the last call.

8 Related Work

Existing literature on the CAM [4, 5, 6] with the notable exception of Suárez [28] deals with the translation of functional languages into CAM code on a very high and abstract level showing only the principle suitability of the CAM as a target machine.

In [28] an optimizing compiler is presented for the CAML language, a variant of SML. The improvements include detection of r-closed expressions, β-reduction at compile-time, improving calls to local functions, and local variables classification. Except for the latter we perform the same improvements albeit in a different setting.

The improvements described in [28] are tightly coupled with the compilation process whereas we separate the phases of generating code and improving it. Insofar our work can be interpreted as a paraphrases of [28]. Besides better readability and verifibility our approach allows the easy integration of classical optimization techniques such as last call optimization.

9 Acknowledgements

Thanks are due to Holger Berse, Wolfram Burgard, Ulrike Griefahn, Jürgen Kalinski, Stefan Kurtz, Stefan Lüttringhaus-Kappel, Jörg Prante, and Peter Thiemann for their detailed comments and helpful suggestions on an earlier draft of this report.

References

[1] Lennart Augustsson. A compiler for lazy ML. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas*, pages 218–227. ACM, 1984.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice Hall International, 1988.

[3] L. Cardelli. The functional abstract machine. Technical Report 107, Bell Laboratories, 1983.

[4] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In Jouannaud [17], pages 50–64. LNCS 201.

[5] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[6] Guy Cousineau. The Categorical Abstract Machine. In Huet [15], pages 25–45. ISBN 0-201-17234-8.

[7] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Pitman, Series in Theoretical Computer Science, 1986.

[8] H.B. Curry and R. Feys. *Combinatory Logic, Volume 1.* North-Holland, Amsterdam New York Oxford, 1958.

[9] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 6, pages 243–320. Elsevier Science Publishers B.V. (North Holland), 1990.

[10] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture: Portland, OR*, pages 34–46. Springer-Verlag, 1987. LNCS 274.

[11] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley Publishing Company, 1988.

[12] Robert Harper and Robin Milner. The Definition of Standard ML, Version 2. Technical report, University of Edinburgh, 1988.

[13] Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda.* Teubner, Stuttgart, 1992.

[14] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. Research Report CSC/89/R5, Department of Computer Science, University of Glasgow, Glasgow, 1989.

[15] Gerard Huet, editor. *Logical Foundations of Functional Programming*. Addison-Wesley Publishing Company, 1990. ISBN 0-201-17234-8.

[16] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, pages 58–69. SIGPLAN, 1984. SIGPLAN Notices Vol. 19, No. 6, June 1984.

[17] Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture Nancy, France, September 1985*. Springer-Verlag, 1985. LNCS 201.

[18] J. Lambek. From lambda-calculus to cartesian closed categories. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 376–402. Academic-Press, 1980.

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

[20] R.D. Lins. On the efficiency of categorical combinators as a rewriting system. *Software - Practice and Experience*, 17(8):547–559, 1987.

[21] Bruce MacLennan. *Functional programming: Practice and Theory.* Addison-Wesley Publishing Company, 1990.

[22] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[23] S.L. Peyton Jones and D.R. Lester. *Implementing functional languages: a tutorial.* International Series in Computer Science. Prentice Hall International, 1992.

[24] Chris Reade. *Elements of Functional Programming.* Addison-Wesley Publishing Company, 1989.

[25] Arto Salomaa. *Computation and Automata.* Cambridge University Press, 1985.

[26] Sjaak Smetsers, Eric Nöcker, John van Groningen, and Rinus Plasmeijer. Generating efficient code for lazy functional languages. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture: Cambridge, MA*, pages 593–617. Springer-Verlag, 1991. LNCS 523.

[27] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[28] Ascánder Suárez. Compiling ML into CAM. In Huet [15], pages 47–73. ISBN 0-201-17234-8.

[29] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In Jouannaud [17], pages 1–26. LNCS 201.

[30] D.A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.

Contents

List of Tables