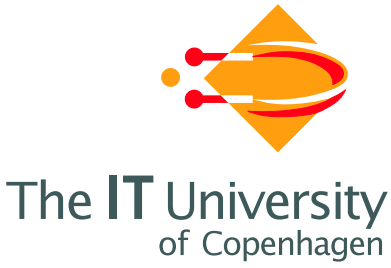

A Region-Based Abstract Machine for the ML Kit

Technical Report

Martin Elsman
Niels Hallenberg

IT University Technical Report Series TR-2002-18

ISSN 1600–6100 August 2002

ISSN 1600–6100

ISBN 87–7949–025–5

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web www.it-c.dk

A Region-Based Abstract Machine for the ML Kit

Martin Elsman* (mael@dina.kvl.dk)
Royal Veterinary and Agricultural University of Denmark

Niels Hallenberg (nh@it-c.dk)
IT University of Copenhagen

August 20, 2002

Abstract

This document describes the implementation of a bytecode backend for the ML Kit, a Standard ML compiler based on region inference. The purpose of the bytecode backend is to enable the ML Kit to produce portable and compact code that can be executed in environments where control of memory usage is a concern.

The document includes (1) a presentation of the target of the new backend, namely a region-based abstract machine called the Kit Abstract Machine (KAM), (2) a description of the intermediate language RegExp used in the ML Kit, (3) a presentation of yet an intermediate language LiftExp, in which all functions are lifted to top level, (4) a translation from the language RegExp to the language LiftExp, and finally, (5) a translation from the intermediate language LiftExp into KAM instructions.

The focus of this document is not on providing a formal definition of the KAM or of the translation into KAM instructions. Instead, this report is meant to document—sometimes informally—the implementation and the design of the many features of the KAM.

Contents

*Part time at the IT University of Copenhagen

1 Introduction

The ML Kit is a Standard ML compiler based on region inference [TBE+01]. Region inference inserts, at compile time, allocation and deallocation directives into the program [TB00, TB98, BT01]. When the program is executed, memory is allocated and deallocated according to the memory directives, without the need for traversing memory. In this basic setting, the programmer is provided with good control of memory.

Region based programming is promising in the area of embedded systems with real time requirements to the software running and where memory is a limited resource. Writing software for embedded systems in a type safe functional programming language with a rich module system helps shortening development time and helps providing reliable embedded systems based on reusable components. Today, software for embedded systems are often written in programming languages such as C and C++, which, for the first part, have type systems that are not rich enough to prevent the program from executing certain erroneous operations, and for the second part, have no support for verifying safety of the constructed system with respect to dynamic memory allocation and deallocation.

The region based programming model is also promissing in situations where programs run shortly and are executed often; in such situations region inference can provide very good cache behaviour and thus high efficiency. This latter situation arises often in server-centric and net-centric applications, such as Web servers, where clients request the execution of programs on a remote server.

To get practical experience with the latter of the above scenarios, we have initiated the *SMLserver project*, which aims at building an efficient multi-threaded Web server platform for the Standard ML programming language [EH02].

In this document, we describe the implementation of an abstract machine, called the Kit Abstract Machine (KAM), and a new backend for the ML Kit, which compiles the ML Kit intermediate language RegExp into bytecode for the KAM.[1] The KAM is a stack based machine where memory usage is controlled using a small set of region primitives. The KAM is an essential part of the SMLserver project.

2 Region Based Memory Management

In this section we introduce the region based memory management scheme, which is the basis for the operations provided by the abstract machine that we present in Section 3.

In the region memory model, the store consists of a stack of regions. Region inference turns all value producing expressions e in the program into e `at` ρ, where ρ is a region variable, which denotes a region in the store at runtime. Moreover, when e is an expression in the source program, region inference may turn e into the target expression `letregion` ρ `in` e' `end`, where e' is the target of analyzing sub-expressions in e and ρ is a fresh region variable. At runtime, first an empty region is pushed on the stack and bound to ρ. Then, the sub-expression e' is executed, perhaps using ρ for allocation. Finally, upon reaching `end`, the region is deallocated from the stack. Safety of region inference guarantees that a region is not freed until after the last use of a value located in that region [TT94, TT97].

Functions in the target language can be declared to take regions as arguments and may thus, depending on the actual regions that are passed to the function, produce values in different regions for each call.

[1] The ML Kit also has a backend that generates efficient native machine code for the X86 architecture.

After region inference has annotated a program fragment, a series of region analyses [BTV96] are applied, as to figure out the exact representation of regions. An analysis called *multiplicity inference* determines which regions are stored into at most once. Such regions are called *finite regions* and the term *infinite regions* is used for regions into which many values can be stored simultaneously. Another analysis, called *physical size inference*, determines, for each finite region, the maximal physical size in words for a value stored in that region. Multiplicity inference and physical size inference enables many regions to be allocated directly on the runtime stack. Only regions with infinite multiplicity need be allocated in the region heap.

Another kind of regions are *word-regions*, which have the property that they hold values that can be represented in only one word (e.g., integers and booleans). Such regions can be implemented in registers and spilled on the stack in case of lack of registers.

Yet an important analysis is the *storage mode analysis*, which attempts at finding out when a region is certain to contain no live data and thus can be reset before a new value is stored in the region. Storage mode analysis and multiplicity inference, are polymorphic analyses.

3 The Kit Abstract Machine

The *Kit Abstract Machine* (KAM) operates on a stack, a region heap, and a series of registers. Programs that may be executed by the KAM are constructed from a set of *instructions*, which manipulate the stack, the region heap, and the registers.

Instructions that operate on the stack include instructions for pushing and popping and for accessing and updating arbitrary elements on the stack; the stack is used, not only for activation records, but also for finite regions and region descriptions for infinite regions. A *region description* on the stack points to a region in the region heap. A region in the region heap may grow dynamically as values are stored in the region and the region may even be reset to the empty region using a dedicated primitive.

These dynamic properties of regions are implemented by representing a region as a linked list of constant-sized region pages, which are thunks of memory allocated from the operating system. When a region is deallocated or reset, region pages in the region are stored in a *region free list*, also from which region pages are obtained when more memory is requested for allocation. The KAM, as described in this section, however, does not model regions at this level of detail; instead, it is assumed that regions in the region heap extend to hold an arbitrary number of values. For more information on the implementation of the region heap, see [EH95, Hal99, Tof98, HET02].

Besides from the stack and the region heap, the KAM operates on four KAM *registers*, a *program counter* (*pc*), which points at the instructions being executed, an *accumulator* (*acc*), to hold the last computed value, a *stack pointer* (*sp*), which points at the top of the stack, and an *environment* (*env*), which holds the closure for the function currently being executed. Program flow in the KAM works by either incrementing the program pointer to point at the next instruction or have the program pointer point at an instruction following a labeled instruction.

The implementation of the KAM is motivated by the Moscow ML and O'Caml runtime systems [Ler90].

3.1 Grammar for KAM

The grammar for the KAM is based on a set of basic semantic objects, which are shown in Figure 1. The set Offset denotes offsets on the stack, which are negative, and offsets in allocated memory,

$$
\begin{array}{rcl}
lab & \in & \text{Label} \\
d, m, n & \in & \text{Int} \\
o & \in & \text{Offset} = \text{Int}
\end{array}
$$

Figure 1: Basic semantic objects for the KAM

which are positive. The set Label is a set of labels for identifying functions and local addresses in KAM code.

The grammar for the KAM language is shown in Figure 2. A KAM *program* (P) is a sequence of *functions*, each of which is a labeled *block* (B) of *instructions* (I).

In the following, we assume stack pointers to be word aligned, that is, the two least significant bits of a stack pointer are zero. The two bits have a special meaning in pointers to finite regions and region descriptions on the stack. The first bit is used to hold the multiplicity (finite or infinite) and the second to hold the storage mode (`atbot` or `attop`). Setting the infinite bit is done with the instruction `StackAddrInfBit`. Setting and clearing the storage mode bit is done with the instructions `SetAtbotBit` and `ClearAtbotBit`.

The KAM has instructions for allocating and deallocating regions and all instructions that potentially allocate memory take regions as parameters (on the stack). Many of the instructions also take parameters explicitly.

Lists are represented unboxed. The `nil` value is distinguishable from any boxed value (i.e., pointers). The instruction `IfCons`(*true_lbl*) tests whether the accumulator holds a value of the form `::`v, for any v, and if so jumps to *true_lbl*. In the implementation, the `Cons` and `Decons` instructions are implemented as `Nop` instructions; the instructions are included in the presentation to increase clarity.

The dynamic semantics of each of the KAM instructions is presented in the following section.

3.2 Dynamic Semantics for KAM

In the following sections, we present the dynamic semantics for the KAM by showing, for each instruction, the state of the KAM before and after the instruction is executed. The state of the KAM consists of the stack, the region heap, and the four registers. Program execution starts at the first instruction in the function labelled "main".

I	::=	`Nop`	no operation
	|	`Immed`(d) | `Nil`	basic constants
	|	`Cons` | `Decons`	list construction and deconstruction
	|	`Select`(o)	record selection
	|	`Push` | `Pop` | `Pop`(n) | `PushLbl`(lab)	stack operations
	|	`SelectStack`(o) | `StackAddr`(o)	
	|	`Label`(lab) | `Jmp`(lab)	intra-function control flow
	|	`IfEq`(d, lab) | `IfCons`(lab)	
	|	`ApplyFunCall`(lab, n) | `ApplyFunJmp`(lab, n_1, n_2)	calls to known function
	|	`ApplyFnCall`(n) | `ApplyFnJmp`(n_1, n_2)	calls to unknown function
	|	`Return`(n_1, n_2)	function return
	|	`SelectEnv`(o) | `EnvToAcc`	environment manipulation
	|	`LetregionFin`(n)	finite region allocation
	|	`LetregionInf` | `EndregionInf`	region allocation and deallocation
	|	`StackAddrInfBit`(o)	region descriptor
	|	`ClearAtbotBit` | `SetAtbotBit`	storage mode manipulation
	|	`Alloc`(n) | `AllocIfInf`(n) | `AllocSatInf`(n)	uninitialized allocation
	|	`AllocSatIfInf`(n) | `AllocAtbot`(n)	
	|	`BlockAlloc`(n) | `BlockAllocIfInf`(n)	record allocation
	|	`BlockAllocSatInf`(n) | `Block`(n)	
	|	`BlockAllocSatIfInf`(n) | `BlockAllocAtbot`(n)	
B	::=	I `;` B	instruction blocks
	|	ε	empty block
P	::=	`fun` lab `is` B `;` P	known function
	|	`fn` lab `is` B `;` P	unknown function
	|	ε	empty program

Figure 2: The grammar for the KAM

3.3 Basic Constants, Constructors, and Record Selection

Ref.	Instruction	*acc*	*env*	*pc*	*sp*	Stack
1	`Nop`	v	ϵ	pc	sp	s
		v	ϵ	$pc+1$	sp	s
2	`Immed`(d)	v	ϵ	pc	sp	s
		d	ϵ	$pc+1$	sp	s
3	`Nil`	v	ϵ	pc	sp	s
		`nil`	ϵ	$pc+1$	sp	s
4	`Cons`	v	ϵ	pc	sp	s
		$::v$	ϵ	$pc+1$	sp	s
5	`Decons`	$::v$	ϵ	pc	sp	s
		v	ϵ	$pc+1$	sp	s
6	`Select`(o)	p	ϵ	pc	sp	s
		v_o	ϵ	$pc+1$	sp	s

1. The instruction `Nop` does not change the state, except by increasing the program counter.
2. The accumulator is loaded with the integer d.
3. The accumulator is loaded with the `nil` constant.
4. The accumulator is tagged with the constructor tag.
5. The accumulator containing the constructor tag is detagged.
6. The value v_o from the record $(v_1, \ldots, v_o, \ldots, v_n)$ pointed at by p is copied into the accumulator.

3.4 Stack Operations

The stack grows upwards from low addresses to hign addresses. The stack contains function frames, return labels, closures, and arguments. Figure 3 shows an example stack where a function f is called from another function g with the body of f currently being evaluated. To access values on the stack, negative offsets from the stack pointer *sp*are used.

Ref.	Instruction	*acc*	*env*	*pc*	*sp*	Stack
	`Push`	v	ϵ	pc	sp	s
		v	ϵ	$pc+1$	$sp+1$	v, s
	`Pop`	_	ϵ	pc	sp	v, s
		v	ϵ	$pc+1$	$sp-1$	s
	`Pop`(n)	v	ϵ	pc	sp	$v_1, \ldots, v_n, s$
		v	ϵ	$pc+1$	$sp-n$	s
	`PushLbl`(lab)	v	ϵ	pc	sp	s
		v	ϵ	$pc+1$	$sp+1$	lab, s
1	`SelectStack`(o)	_	ϵ	pc	sp	$v_1, \ldots, v_{(-o)}, s$
		$v_{(-o)}$	ϵ	$pc+1$	sp	$v_1, \ldots, v_{(-o)}, s$
2	`StackAddr`(o)	_	ϵ	pc	sp	s
		$sp+o$	ϵ	$pc+1$	sp	s

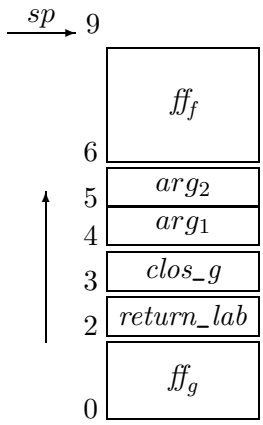

Figure 3: The Figure shows an example stack where the function frame for g starts at address 0, a return label is stored at address 2, etc. The function f has been called from inside the body of function g. The function frame for f starts at address 6.

1. The value $v_{(-o)}$ on the stack is copied into the accumulator. The offset o is a negative offset from sp, see Figure 3. To access a value stored at address 7 in Figure 3, the instruction `SelectStack`(-2) is used.

2. The accumulator is set to the stack address $sp + o$. This instruction is used to move a value that denotes a finite region into the accumulator. Recall that stack offsets are negative.

3.5 Inter-Function Control Flow

The `Label`(lab) instruction does not change the state—it marks the succeeding instruction. A jump to a label lab is performed by simply changing the code pointer pc to the address denoted by the label.

Ref.	Instruction	*acc*	*env*	*pc*	*sp*	Stack
	`Jmp`(lab)	v	ϵ	pc	sp	s
		v	ϵ	lab	sp	s
	`IfEq`(d, lab)	d	ϵ	pc	sp	s
		d	ϵ	lab	sp	s
	`IfEq`(d, lab)	d'	ϵ	pc	sp	s
		d'	ϵ	$pc+1$	sp	s
	`IfCons`(lab)	$::v$	ϵ	pc	sp	s
		$::v$	ϵ	lab	sp	s
	`IfCons`(lab)	v	ϵ	pc	sp	s
		v	ϵ	$pc+1$	sp	s

3.6 Function Calls and Returns

The following table shows the semantics of the various KAM instructions for function applications and returns.

Ref.	Instruction	*acc*	*env*	*pc*	*sp*	Stack
1	`ApplyFnCall`(n)	v_n	ϵ_{caller}	pc	sp	$v_{n-1}, \ldots, v_1, \epsilon_{\text{callee}}, s$
		v_n	ϵ_{callee}	lab	$sp+1$	$v_n, \ldots, v_1, \epsilon_{\text{caller}}, s$
2	`ApplyFnJmp`(m, n)	v_m	ϵ_{caller}	pc	sp	$v_{m-1}, \ldots, v_1, \epsilon_{\text{callee}}, v'_n, \ldots, v'_1, s$
		v_m	ϵ_{callee}	lab	$sp-n$	$v_m, \ldots, v_1, s$
3	`ApplyFunCall`(lab, n)	v_n	ϵ_{caller}	pc	sp	$v_{n-1}, \ldots, v_1, \epsilon_{\text{callee}}, s$
		v_n	ϵ_{callee}	lab	$sp+1$	$v_n, \ldots, v_1, \epsilon_{\text{caller}}, s$
4	`ApplyFunJmp`(lab, m, n)	v_m	ϵ_{caller}	pc	sp	$v_{m-1}, \ldots, v_1, \epsilon_{\text{callee}}, v'_n, \ldots, v'_1, s$
		v_m	ϵ_{callee}	lab	$sp-n$	$v_m, \ldots, v_1, s$
5	`Return`(m, n)	r_n	$\epsilon_{\text{current}}$	pc	sp	$r_{n-1}, \ldots, r_1, v_m, \ldots, v_1, \epsilon_{\text{caller}}, lab_{\text{ret}}, s$
		r_n	ϵ_{caller}	lab_{ret}	$sp-m-2$	$r_{n-1}, \ldots, r_1, s$

1. The label lab is extracted from the closure $\epsilon_{\text{callee}} = (lab, f_1, \ldots, f_l)$. The stack s takes the form lab_{ret}, s', where lab_{ret} is a return label and s' is the remainder of the stack.

2. The label lab is extracted from the closure $\epsilon_{\text{callee}} = (lab, f_1, \ldots, f_l)$. The arguments for the current function $v'_n, \ldots, v'_1$ are removed from the stack. The closure ϵ_{caller} is thrown away because it is a tail call—we return to the caller of the current function. The stack s takes the form $\epsilon_{\text{ret}}, lab_{\text{ret}}, s'$, where ϵ_{ret} is a stored return environment, lab_{ret} is a return label, and s' is the remainder of the stack.

3. Here s takes the form lab_{ret}, s' where lab_{ret} is a return label and s' is the remainder of the stack. The arguments $v_n, \ldots, v_1$ appear with ordinary arguments allocated on top of the stack.

4. Here s takes the form $\epsilon_{\text{ret}}, lab_{\text{ret}}, s'$, where ϵ_{ret} is a stored return environment, lab_{ret} is a return label, and s' is the remainder of the stack. The old arguments $v'_n, \ldots, v'_1$ are thrown away, which include both ordinary arguments and region arguments.

5. If the function from which we are returning is a region polymorphic function, some of the arguments $v_m, \ldots, v_1$ are region arguments.

3.7 Environment Manipulation

Ref.	Instruction	*acc*	*env*	*pc*	*sp*	Stack
1	`SelectEnv`(o)	acc	$v_1, \ldots, v_o, \ldots, v_n$	pc	sp	s
		v_o	$v_1, \ldots, v_o, \ldots, v_n$	$pc+1$	sp	s
2	`EnvToAcc`	acc	ϵ	pc	sp	s
		ϵ	ϵ	$pc+1$	sp	s

1. The value v_o in the environment (i.e., closure) is copied into the accumulator.

2. The environment pointer ϵ is copied into the accumulator.

3.8 Regions and Region Descriptors

Ref.	Instruction	*acc*	*env*	*pc*	*sp*	Stack
1	`LetregionFin`(n)	acc	ϵ	pc	sp	s
		sp	ϵ	$pc+1$	$sp+n$	$0_1, \ldots, 0_n, s$
2	`LetregionInf`	acc	ϵ	pc	sp	s
		sp	ϵ	$pc+1$	$sp + size_{rDesc}$	$R_1, \ldots, R_{size_{rDesc}}, s$
3	`EndregionInf`	acc	ϵ	pc	sp	$R_1, \ldots, R_{size_{rDesc}}, s$
		acc	ϵ	$pc+1$	$sp - size_{rDesc}$	s
4	`StackAddrInfBit`(o)	acc	ϵ	pc	sp	s
		$sp + o$+`inf_bit`	ϵ	$pc+1$	sp	s
5	`ClearAtbotBit`	ρ	ϵ	pc	sp	s
		ρ'	ϵ	$pc+1$	sp	s
6	`SetAtbotBit`	ρ	ϵ	pc	sp	s
		ρ'	ϵ	$pc+1$	sp	s

1. The finite region is allocated directly on the stack, that is, the stack pointer is offset by n. The accumulator *acc* is set to point at the first allocated word on the stack, which is the previous value of *sp*.
2. The infinite region is allocated in the region heap and a region descriptor of size $size_{rDesc}$ is pushed on the stack. The accumulator *acc* is set to point at the first allocated word on the stack, which is the previous value of *sp*.
3. The infinite region is deallocated from the region heap and the region descriptor is removed from the stack.
4. The accumulator is set to the stack address $sp + o$ with the infinite bit set. The stack address will contain the first word of a region descriptor
5. The region pointer ρ' is ρ with the atbot bit cleared.
6. The region pointer ρ' is ρ with the atbot bit set.

3.9 Allocating in Regions

Ref.	Instruction	*acc*	*env*	*pc*	*sp*	Stack
1	`Alloc`(n)	ρ	ϵ	pc	sp	s
		p	ϵ	$pc+1$	sp	s
2	`AllocIfInf`(n)	ρ	ϵ	pc	sp	s
		p	ϵ	$pc+1$	sp	s
3	`AllocSatInf`(n)	ρ	ϵ	pc	sp	s
		p	ϵ	$pc+1$	sp	s
4	`AllocSatIfInf`(n)	ρ	ϵ	pc	sp	s
		p	ϵ	$pc+1$	sp	s
5	`AllocAtbot`(n)	ρ	ϵ	pc	sp	s
		p	ϵ	$pc+1$	sp	s

1. The pointer p is a pointer to a block of n words allocated attop in the infinite region ρ in the region heap.

2. The pointer p is a pointer to a block of n words either allocated attop in ρ if ρ is infinite or allocated directly on the stack.

3. The pointer p is a pointer to a block of n words allocated attop or atbot in the infinite region ρ, depending on the storage mode bit of ρ.

4. The pointer p is a pointer to a block of n words allocated attop or atbot in the region ρ, depending on the storage mode bit of ρ. The region ρ may be finite or infinite.

5. The pointer p is a pointer to a block of n words allocated atbot in the infinite region ρ.

For each alloc-instruction there is a block-instruction that besides allocating a block in region ρ copies the n top words from the stack into the block.

Ref.	**Instruction**	***acc***	***env***	***pc***	***sp***	**Stack**
1	`BlockAlloc`(n)	ρ	ϵ	pc	sp	$v_n, \ldots, v_1, s$
		p	ϵ	$pc+1$	$sp-n$	s
1	`BlockAllocIfInf`(n)	ρ	ϵ	pc	sp	$v_n, \ldots, v_1, s$
		p	ϵ	$pc+1$	$sp-n$	s
1	`BlockAllocSatInf`(n)	ρ	ϵ	pc	sp	$v_n, \ldots, v_1, s$
		p	ϵ	$pc+1$	$sp-n$	s
1	`BlockAllocSatIfInf`(n)	ρ	ϵ	pc	sp	$v_n, \ldots, v_1, s$
		p	ϵ	$pc+1$	$sp-n$	s
1	`BlockAllocAtbot`(n)	ρ	ϵ	pc	sp	$v_n, \ldots, v_1, s$
		p	ϵ	$pc+1$	$sp-n$	s
2	`Block`(n)	p	ϵ	pc	sp	$v_n, \ldots, v_1, s$
		p	ϵ	$pc+1$	$sp-n$	s

1. The pointer p is a pointer to a record $(v_1, \ldots, v_n)$ allocated in region ρ. The record is allocated as specified for the alloc–instructions above.

2. The pointer p points at a block of size n in which the record $(v_1, \ldots, v_n)$ is created. The `Block` instruction is used for finite regions.

4 The Language RegExp

In this section we present the source language RegExp[2], which is the language passed on from region inference and the region representation analyses [BTV96] to the back end. We give only a short introduction to the language; please see [Hal99, Chapter 2] for a full presentation of RegExp.

In Section 5 we present the language LiftExp and in Section 6, we show how RegExp programs are translated into LiftExp programs, in which all functions are lifted (hoisted) to top level. In Section 7 we show how LiftExp programs are compiled into KAM programs.

The semantic objects for the RegExp language are shown in Figure 4. The grammar is shown in Figure 6. The grammar resembles a limited version of the core language of Standard ML and shares many of the same properties. For instance, evaluation is call-by-value and evaluation order is left to right.[3]

[2]The term RegExp is an abbreviation for Region Expression.

[3]The ML Kit implements all features of Standard ML; the RegExp language presented here is a subset of the RegExp language used in the ML Kit.

$$
\begin{array}{rcl}
x, f, y & \in & \text{LamVar} \\
d, n & \in & \text{Int} \\
\rho & \in & \text{RegVar} \\
fv & \in & \text{Var} = \text{LamVar} \ \cup \ \text{RegVar} \\
\mathit{free} & \in & \text{List(Var)}
\end{array}
$$

Figure 4: The semantic objects used in RegExp

$$\text{List}(A) = \cup\{[x_1, \ldots, x_n] | x_i \neq x_j, i \neq j, x_k \in A\}$$

Figure 5: Each element can appear only once in a list. The order in which elements appear in a list matters; so for instance, the two lists $[x_1, x_2]$ and $[x_2, x_1]$ are different.

The syntactic classes are *allocation directives* (a), *region binders* (b), *constants* (c), *boxed expressions* (be), *patterns* (pat), *binary operators* (bop), and *expressions* (e). We use RegExp to denote the set of all RegExp terms.

Value variables, ranged over by x, f and y are bound either by the `let`-expressions or by function-constructs (λ- and `letrec`-expressions). There are two types of constants; integers are ranged over by d and the constant `nil` is used for the construction of lists. Region variables, ranged over by ρ, are bound by `letregion`-expressions and `letrec`-expressions.

To simplify the presentation, we assume that the free variables of ordinary functions (λ) and for `letrec` bound functions have been computed beforehand. The free variables are represented by *free*. We use an *ordered list* for the free variables, see Figure 5.

For a full treatment of RegExp, including a dynamic semantics for the language, consult [Hal99, Chapter 2].

4.1 Example Program

We shall use the following Standard ML program as a running example.

```
fun foldl f b xs =
  case xs of
    [] => b
  | x::xs' => foldl f (f x b) xs'
```

The same program in RegExp is shown in Figure 7.

The example program is taken from [Hal99, Chapter 2 and 3], where one can also find a full discussion of the region annotations (e.g., multiplicities, storage modes and call kinds).

5 The Language LiftExp

It is possible to compile RegExp programs directly to bytecode. However, to avoid one complicated compilation phase, compilation is split into two phases. The first phase lifts all functions to top level,

a	::=	$sma\ \rho$	allocation point
sma	::=	`attop` | `atbot` | `sat`	storage mode annotation
b	::=	$\rho : m$	region binder
m	::=	$n \mid \infty$	multiplicity
c	::=	d | `nil`	constant
be	::=	$(e_1, \ldots, e_n) \mid \lambda^{free}\ \langle x_1, \ldots, x_n \rangle$ `=>` e	boxed expressions
pat	::=	c | `::` x	patterns
bop	::=	`+` | `-` | `<` | ...	primitive binary operations
ck	::=	`funjmp` | `funcall` | `fnjmp` | `fncall`	call kinds
e	::=	x	value variable
	|	$be\ a$	boxed expression
	|	c	constant
	|	`::` e	list construction
	|	$e\ bop\ e$	primitive binary operation
	|	`#`$n(e)$	record selection
	|	`letrec` $f^{free}\ \langle x_1, \ldots, x_n \rangle\ [b_1, \ldots, b_m]\ a$ `=` e `in` e `end`	function binding
	|	$e_{ck}\ e$	unknown function application
	|	$f_{ck}\ \langle e_1, \ldots, e_n \rangle\ [a_1, \ldots, a_m]$	known function application
	|	`letregion` b `in` e `end`	region binding
	|	`let val` $\langle x_1, \ldots, x_n \rangle$ `=` e_1 `in` e_2 `end`	value binding
	|	`case` e_1 `of` pat `=>` e_2 `|` pat `=>` e_3	case construct
	|	$\langle e_1, \ldots, e_n \rangle$	unboxed record

Figure 6: The grammar for RegExp

```
letrec foldl[ ] ⟨f⟩ [r7 : 4, r8 : 4] attop r1 =
  λ[f,r8,foldl] ⟨b⟩ attop r7 =>
    λ[b,f,foldl] ⟨xs⟩ attop r8 =>
      (case xs of
         nil => b
       | :: (v942) =>
           let
             val x = #0(v942)
             val xs' = #1(v942)
           in
             letregion r22:4 in
               (letregion r24:4 in
                  (foldl_funcall ⟨f⟩ [atbot r24, atbot r22])_fncall
                  ⟨(f_fncall ⟨x⟩)_fncall ⟨b⟩⟩
                end (*r24*))_fncall ⟨xs'⟩
             end (*r22*)
           end (*let*))
in
  foldl_funcall ⟨sum⟩ [atbot r4, atbot r5]
end (*letrec*)
```

Figure 7: Our running example program written in RegExp. We assume *sum*, $r1$, $r4$, and $r5$ to be previously defined.

thereby making closures more explicit in the program. Moreover, in LiftExp, region annotations appear on the constructs on which they are used by the code generator. In this way, two less complicated phases are obtained, each of which is easy to debug and maintain.

The language LiftExp, presented here, is very similar to the language ClosExp used in the other backends of the ML Kit [Hal99, Chapter 3]. Actually, the implementation of the bytecode compiler uses the ClosExp language, so LiftExp is used for presentation only.

5.1 Calling Conventions

A *calling convention*, which specifies how arguments are passed to a function, is specified by a triple:

$$ce \in \text{CallConv} = \text{LamVar} \times \text{List(LamVar)} \times \text{List(RegVar)}$$

We shall often write calling conventions on the form

$$\{\text{clos} = x,\ \text{args} = [x_1, \ldots, x_n],\ \text{regargs} = [\rho_1, \ldots, \rho_m]\}$$

where $x, x_1, \ldots, x_n$, $n \geq 0$, are value variables and $\rho_1, \ldots, \rho_m$, $m \geq 0$, are region variables. We use ce to range over calling conventions and CallConv to denote the set of all calling conventions. We use the notation $\{\text{clos} = x,\ \text{args} = [x_1, \ldots, x_n]\}$ as an abbreviation for calling conventions of the form $\{\text{clos} = x,\ \text{args} = [x_1, \ldots, x_n],\ \text{regargs} = []\}$. We use *compatible* calling conventions for all functions: Ordinary arguments and region arguments are passed on the stack. Free variables are always passed in a closure record, and the KAM uses a dedicated register (env) to hold the closure record. We note that a list may be empty (see Figure 5). We use CallConv to denote the set of possible calling conventions.

A calling convention fully describes how both ordinary and `letrec` functions are called. With all functions at top level and with the calling convention specified above, we get the following grammar for top-level declarations:

$$\begin{array}{rcl} \mathit{topdec} & ::= & \lambda^{\texttt{fun}}_{\mathit{lab}}\ \mathit{cc}\ \texttt{=>}\ e \\ & | & \lambda^{\texttt{fn}}_{\mathit{lab}}\ \mathit{cc}\ \texttt{=>}\ e \end{array}$$

The two constructs are identical except that we syntactically differentiate between `letrec` (abbreviated `fun`) and ordinary functions (abbreviated `fn`). The label *lab* is a unique identifier (i.e., name) for the function; $\mathit{lab} \in \text{Label}$. The set TopDec denotes the set of top-level declarations ranged over by *topdec*.

Application points must follow the calling convention for the called function. We use the following grammar at application points:

$$\begin{array}{rcl} e & ::= & e_{ck}\ \langle e_1, \ldots, e_n \rangle \\ & | & \mathit{lab}_{ck}\ \langle e_1, \ldots, e_n \rangle\ \langle e'_1, \ldots, e'_m \rangle\ \langle e_{\text{clos}} \rangle \end{array}$$

Here e_{ck} and e_{clos} evaluate to a closure and a shared closure, respectively. Ordinary arguments are written $\langle e_1, \ldots, e_n \rangle$ and region arguments are written $\langle e'_1, \ldots, e'_m \rangle$. Notice that *ck* ranges over call kinds.

We note that RegExp is in so-called *K-normal form* (see [BTV96]), thus, all value-creating expressions are either bound to a variable, ranged over by x and f, or the value is a constant. In LiftExp, we do not use the extra `let`-bindings introduced in RegExp. The target machine is a stack machine where we want as few variables as possible, that is, only the `let`-bindings introduced by the programmer. The RegExp language has been carefully designed so that unnecessary `let`-bindings can be avoided.

5.2 Functions

With all functions at top level, λ-expressions and `letrec` expressions are replaced with two new boxed expressions. The λ-expression is replaced with a boxed expression that explicitly builds a closure:

$$\mathit{be} \quad ::= \quad \lambda^{\mathit{lab}}\ [e_1, \ldots, e_n]$$

This boxed expression evaluates to a closure record $(\mathit{lab}, v_1, \ldots, v_n)$, where v_i is the value resulting from evaluating e_i, $i = 1..n$. The label *lab* refers to the function at top level. The `letrec` construct is replaced by the boxed expression

$$\mathit{be} \quad ::= \quad [e_1, \ldots, e_n]_{\text{sclos}}$$

which builds a shared closure record $(v_1, \ldots, v_n)$. The LiftExp language also has a `letrec` construct, which makes the scope for a function identifier explicit:

$$e \ ::= \ \texttt{letrec}\ f_{\mathit{lab}}\ \texttt{=}\ \mathit{be}\ a\ \texttt{in}\ e\ \texttt{end}$$

The label *lab* connects the top-level function $\lambda^{\texttt{fun}}_{\mathit{lab}}$ with f. We shall often omit the label in examples where f and the label are identical. The boxed expression *be* is always a shared closure, which, in this case, is bound to the variable f.

5.3 Storage Mode Annotations

In the LiftExp language, the set of storage modes is refined to make code generation easier. The storage modes `attop`, `atbot`, and `sat` are extended to eight new *storage mode annotations*, namely `attop_ff`, `attop_fi`, `attop_lf`, `attop_li`, `atbot_lf`, `atbot_li`, `sat_ff`, and `sat_ff`. The letter combinations added to the storage modes have the following meanings:

- `ff:` a formal region parameter with finite multiplicity
- `fi:` a formal region parameter with infinite multiplicity
- `lf:` a `letregion` bound region variable with finite multiplicity
- `li:` a `letregion` bound region variable with infinite multiplicity

The set SMA is the set of storage mode annotations ranged over by *sma*. Consult [Hal99, Chapter 3] for more details on the storage mode annotations.

5.4 Grammar for LiftExp

The grammar for LiftExp is shown in Figure 8. The semantic objects for the grammar are shown in Figure 9.

Our running example translated into the LiftExp language is shown in Figure 10 and Figure 11. The translation results in the creation of four top-level functions.

6 Translating RegExp Programs Into LiftExp Programs

The translation from RegExp programs into LiftExp programs, denoted by the translation function $\mathcal{L}$, is fairly straightforward. Before we present the translation, however, we introduce the notion of a translation environment and a few auxiliary translation functions.

6.1 Translation Environments

The translation from RegExp programs to LiftExp programs uses a value environment (*ve*), a function environment for `letrec` bound functions (*fe*), and a region environment for mapping region variables into representation information (*re*).

Value environments take the following form:

$$ve \in VE = \mathrm{Var} \rightarrow \mathrm{Var} \cup \{\texttt{\#}n(x) | n \in \mathbb{N} \wedge x \in \mathrm{Var}\}$$

When translating the body of a function f, free variables of f are by *ve* mapped into entries of the form `#`$n(env)$. Function environments take the form:

$$fe \in FE = \mathrm{Var} \rightarrow \mathrm{Label}$$

Function environments connects a function identifier with the associated top-level function, which is uniquely identified by a label.

A region environment (*re*) maps a region variable into a member of the set {`ff`,`fi`,`lf`,`li`}, depending on the representation of the region, namely, how the region is bound and its multiplicity; see Section 5.3:

$$re \in RE = \mathrm{RegVar} \rightarrow \{\texttt{ff}, \texttt{fi}, \texttt{lf}, \texttt{li}\}$$

The auxiliary translation function **convert_sma** converts a storage mode `attop`, `atbot`, or `sat` into a storage mode annotation, based on the representation information associated with a region variable in a region environment:

topdec	::=	$\lambda^{\texttt{fun}}_{lab}$ *cc* `=>` *e*	known function
	|	$\lambda^{\texttt{fn}}_{lab}$ *cc* `=>` *e*	unknown function
	|	$topdec_1$ `;` $topdec_2$	sequence
ck	::=	`funjmp` | `funcall` | `fnjmp` | `fncall`	call kind
sma	::=	`attop_li` | `attop_lf` | `attop_fi` | `attop_ff`	storage mode
	|	`atbot_li` | `atbot_lf` | `sat_ff` | `sat_ff`	annotation
a	::=	*sma e*	allocation
m	::=	$n \mid \infty$	multiplicity
b	::=	$\rho : m$	region binder
c	::=	*d* | `nil`	constant
pat	::=	*c* | `::` *x*	pattern
bop	::=	`+` | `-` | `<` | ...	binary primitive operation
be	::=	$(e_1, \ldots, e_n)$	boxed record
	|	$\lambda^{lab}\ [e_1, \ldots, e_n]$	closure for unknown function
	|	$[e_1, \ldots, e_n]_{\text{sclos}}$	closure for known functions
e	::=	*x*	value variable
	|	ρ	region variable
	|	*be a*	boxed expression
	|	*c*	constant
	|	`::` *e*	list construction
	|	e_1 *bop* e_2	binary operation
	|	`#`$n(e)$	record selection
	|	`letrec` f_{lab} `=` *be a* `in` *e* `end`	function binding
	|	$e_{ck}\ \langle e_1, \ldots, e_n \rangle$	unknown function call
	|	$lab_{ck}\ \langle e_1, \ldots, e_n \rangle\ \langle a_1, \ldots, a_m \rangle\ \langle e_{clos} \rangle$	known function call
	|	`letregion` *b* `in` *e* `end`	region binding
	|	`let val` $\langle x_1, \ldots, x_n \rangle$ `=` e_1 `in` e_2 `end`	value binding
	|	`case` e_1 `of` *pat* `=>` e_2 `|` *pat* `=>` e_3	case construct
	|	$\langle e_1, \ldots, e_n \rangle$	unboxed record

Figure 8: The grammar for LiftExp. The calling convention *cc* is defined in Section 5.1. Allocation points (a) are restricted to the forms *sma* ρ, *sma* x, and *sma* (`#`n(*env*)).

$topdec$	$\in$	TopDec	e	$\in$	LiftExp
x, f, y, env	$\in$	LamVar	lab	$\in$	Label
d	$\in$	Int	ρ	$\in$	RegVar
cc	$\in$	CallConv	xv	$\in$	Var = LamVar $\cup$ RegVar

Figure 9: The semantic objects used in LiftExp

$\lambda^{\texttt{fun}}_{foldl}$ {clos=env, args=[f], regargs=[$r7, r8$]} `=>` λ^{fn_b} [f, $r8$, env] `attop_ff` $r7$

$\lambda^{\texttt{fn}}_{fn\_b}$ {clos=env, args=[b]} `=>` λ^{fn_xs} [b, `#1`(env), `#3`(env)] `attop_ff #2`(env)

$\lambda^{\texttt{fn}}_{main}$ { } `=>`
  `letrec` $foldl_{foldl}$ `=` [ ]$_{sclos}$ `attop_li` $r1$
  `in`
    $foldl_{\texttt{funcall}}$ $\langle sum \rangle$ $\langle$`attop_lf` $r4$`,`$\texttt{atbot\_lf}$ $r5\rangle$ $\langle foldl\rangle$
  `end`

Figure 10: The first three top-level functions in our running example program translated into LiftExp. In the `letrec` binding we introduce both the label *foldl* and the variable *foldl* (holding the empty shared closure). In the application, the first term *foldl* is the label and the shared closure is accessed in the third bracket. Actually, the shared closure is empty so it should be omitted and will be in the implementation. The closure record *env* for *fn_b* is (*fn_b*, *f*, *r*8, *foldl*).

$\lambda^{\texttt{fn}}_{fn\_xs}$ {clos=env, args=[xs]} `=>`
  (`case` xs `of`
     `nil =>` `#1`(env)
   `|` `::` ($v942$) `=>`
       `let`
         `val` x `=` `#0`($v942$)
         `val` xs' `=` `#1`($v942$)
       `in`
         `letregion` $r22$:4 `in`
           (`letregion` $r24$:4 `in`
             ($foldl_{\texttt{funcall}}$ $\langle$`#2`(env)$\rangle$ [$\texttt{atbot\_lf}$ $r24$, $\texttt{atbot\_lf}$ $r22$] $\langle$`#3`(env)$\rangle$)$_{\texttt{fncall}}$
             $\langle$(`#2`(env)$_{\texttt{fncall}}$ $\langle x\rangle$)$_{\texttt{fncall}}$ $\langle$`#1`(env)$\rangle\rangle$
           `end (*`r24`*))`$_{\texttt{fncall}}$ $\langle xs'\rangle$
         `end (*`r22`*)`
       `end (*`let`*))`

Figure 11: The main function *fn_xs*. The label in the call to *foldl* is known at compile time and is therefore not free in the function, however, the shared closure *foldl* is free in the function. The closure record *env* is (*fn_xs*, *b*, *f*, *foldl*).

convert_sma: $RE \times \{\texttt{atbot},\texttt{attop},\texttt{sat}\} \times \text{RegVar} \rightarrow \text{SMA}$
convert_sma$(re,\texttt{sat},\rho) = \texttt{sat\_}re(\rho)$
convert_sma$(re,\texttt{atbot},\rho) = \texttt{atbot\_}re(\rho)$
convert_sma$(re,\texttt{attop},\rho) = \texttt{attop\_}re(\rho)$

When entries are added to a region environment, we shall make use of the auxiliary translation function **mult** to generate the appropriate region environment entries for a particular region variable:

mult: $\{\texttt{f},\texttt{l}\} \times \text{Mult} \rightarrow \{\texttt{ff},\texttt{fi},\texttt{lf},\texttt{li}\}$
mult$(\texttt{f},n) = \texttt{ff}$
mult$(\texttt{f},\infty) = \texttt{fi}$
mult$(\texttt{l},n) = \texttt{lf}$
mult$(\texttt{l},\infty) = \texttt{li}$

The function takes two arguments, a token (`f` or `l`), which specifies whether the region is `letregion` bound or a formal region parameter, and a multiplicity. A region variable that appears free in a function inherits the storage mode from the region environment in which the region variable is defined at the function declaration.

6.2 Top-Level Declarations

The translation function $\mathcal{L}^{\mathcal{T}}$ translates the RegExp language into the LiftExp language using an auxiliary translation function $\mathcal{L}$:

$$\mathcal{L} : \text{RegExp} \rightarrow VE \rightarrow FE \rightarrow RE \rightarrow \text{LiftExp}$$

To simplify the presentation, the translation $\mathcal{L}^{\mathcal{T}}$ makes use of two functions **add_new_fn** and **add_new_fun**, which are used to collect all functions in a sequence at top level as described by the grammar for top-level declarations (see Figure 8):

add_new_fn : Label $\times$ CallConv $\times$ LiftExp $\rightarrow \{\texttt{()}\}$
add_new_fun : Label $\times$ CallConv $\times$ LiftExp $\rightarrow \{\texttt{()}\}$

We write `()` for unit and $\{\}$ for an empty calling convention. As a side effect, the functions update a *topdec* list reference. The top-level translation function $\mathcal{L}^{\mathcal{T}}$ is defined as follows:

$$\begin{aligned}\mathcal{L}^{\mathcal{T}}(e) = (\ & \mathit{topdec} \texttt{ := } \mathit{nil};\\ & \textbf{add\_new\_fn}(\text{"main"}, \{\}, \mathcal{L}\ [\![e]\!]\ \{\}\ \{\}\ \{\});\\ & \texttt{!}\mathit{topdec}\ )\end{aligned}$$

Here "main" is the label used for the main function and e is the RegExp expression.

6.3 Variables and Constants

$\mathcal{L}\ [\![x]\!]\ ve\ fe\ re = ve(x)$

$\mathcal{L}\ [\![\rho]\!]\ ve\ fe\ re = ve(\rho)$

$\mathcal{L}\ [\![c]\!]\ ve\ fe\ re = c$

6.4 Boxed Expressions

$$
\begin{array}{l}
\mathcal{L}\ [\![(e_1, \ldots, e_n)\ sma\ \rho]\!]\ ve\ fe\ re = \\
\quad \textbf{let} \\
\quad\quad \textbf{val}\ e'_i = \mathcal{L}\ [\![e_i]\!]\ ve\ fe\ re \qquad i = 1..n \\
\quad\quad \textbf{val}\ v = \mathcal{L}\ [\![\rho]\!]\ ve\ fe\ re \\
\quad\quad \textbf{val}\ sma' = \textbf{convert\_sma}(sma, re, \rho) \\
\quad \textbf{in} \\
\quad\quad (e'_1, \ldots, e'_n)\ sma'\ v \\
\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{L}\ [\![\lambda^{[f_1,\ldots,f_m]}\ \langle x_1, \ldots, x_n\rangle\ \texttt{=>}\ e\ sma\ \rho]\!]\ ve\ fe\ re = \\
\quad \textbf{let} \\
\quad\quad \textbf{val}\ lab = \textbf{new\_label}(\text{"anon"}) \\
\quad\quad \textbf{val}\ cc = \{\text{clos} = env, \text{args} = [x_1, \ldots, x_n]\} \\
\quad\quad \textbf{val}\ ve' = \{f_i \mapsto \texttt{\#}i(env)\}_{i=1..m} \cup \{x_i \mapsto x_i\}_{i=1..n} \\
\quad\quad \textbf{val}\ v = \mathcal{L}\ [\![\rho]\!]\ ve\ fe\ re \\
\quad\quad \textbf{val}\ e' = \mathcal{L}\ [\![e]\!]\ ve'\ fe\ re \\
\quad\quad \textbf{val}\ \_ = \textbf{add\_new\_fn}(lab,\ cc,\ e') \\
\quad\quad \textbf{val}\ sma' = \textbf{convert\_sma}(sma, re, \rho) \\
\quad \textbf{in} \\
\quad\quad \lambda^{lab}\ [ve(f_1), \ldots, ve(f_m)]\ sma'\ v' \\
\quad \textbf{end}
\end{array}
$$

The auxiliary function **new_label** generates a new label distinct from other labels generated so far.

6.5 Binary Operators, List Construction, and Record Selection

$$
\begin{array}{l}
\mathcal{L}\ [\![e_1\ bop\ e_2]\!]\ ve\ fe\ re = \\
\quad \textbf{let} \\
\quad\quad \textbf{val}\ e'_1 = \mathcal{L}\ [\![e_1]\!]\ ve\ fe\ re \\
\quad\quad \textbf{val}\ e'_2 = \mathcal{L}\ [\![e_2]\!]\ ve\ fe\ re \\
\quad \textbf{in} \\
\quad\quad e'_1\ bop\ e'_2 \\
\quad \textbf{end}
\end{array}
$$

$$\mathcal{L}\ [\![\texttt{::}\, e]\!]\ ve\ fe\ re = \texttt{::}\ (\mathcal{L}\ [\![e]\!]\ ve\ fe\ re)$$

$$\mathcal{L}\ [\![\texttt{\#}n(e)]\!]\ ve\ fe\ re = \texttt{\#}n(\mathcal{L}\ [\![e]\!]\ ve\ fe\ re)$$

6.6 Recursive Function Bindings

$$
\begin{array}{l}
\mathcal{L}\ [\![\texttt{letrec}\ f^{[y_0,\ldots,y_m]}\ \langle x_1, \ldots, x_n\rangle\ [\rho_0 : m_0, \ldots, \rho_l : m_l]\ sma\ \rho\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}]\!]\ ve\ fe\ re = \\
\quad \textbf{let} \\
\quad\quad \textbf{val}\ lab = \textbf{new\_label}(\text{"}f\text{"}) \\
\quad\quad \textbf{val}\ sma' = \textbf{convert\_sma}(sma, re, \rho) \\
\quad\quad \textbf{val}\ re' = re + \{\rho_0 \mapsto \textbf{mult}(\texttt{f}, m_0), \ldots, \rho_l \mapsto \textbf{mult}(\texttt{f}, m_l)\} \\
\quad\quad \textbf{val}\ v' = \mathcal{L}\ [\![\rho]\!]\ ve\ fe\ re
\end{array}
$$

$$
\begin{array}{l}
\quad \textbf{val } cc = \{\text{clos} = env, \text{args} = [x_1, \ldots, x_n], \text{regargs} = [\rho_0, \ldots, \rho_l]\} \\
\quad \textbf{val } ve' = \{f \mapsto env\} \cup \{y_i \mapsto \#i(env)\}_{i=0..m} \cup \{x_i \mapsto x_i\}_{i=1..n} \cup \{\rho_i \mapsto \rho_i\}_{i=1..l} \\
\quad \textbf{val } fe' = fe + \{f \mapsto lab\} \\
\quad \textbf{val } e_1' = \mathcal{L}\ [\![e_1]\!]\ ve'\ fe'\ re' \\
\quad \textbf{val } \_ = \textbf{add\_new\_fun}(lab,\ cc,\ e_1') \\
\textbf{in} \\
\quad \texttt{letrec } f_{lab} \texttt{ = } [ve(y_0), \ldots, ve(y_m)]_{\text{sclos}}\ sma'\ v' \\
\quad \texttt{in} \\
\qquad \mathcal{L}\ [\![e_2]\!]\ (ve + \{f \mapsto f\})\ fe'\ re \\
\quad \texttt{end} \\
\textbf{end}
\end{array}
$$

The function f is in scope in the body e_1, so we map f into the closure variable env in ve'. Moreover, we extend the function environment with information about the label associated with the variable f, so that the label can be obtained at the functions call sites. We do not use fresh variables for the arguments $x_1, \ldots, x_n$ and $\rho_0, \ldots, \rho_l$.

6.7 Function Applications

$$
\begin{array}{l}
\mathcal{L}\ [\![e_{ck}\ \langle e_1, \ldots, e_n\rangle]\!]\ ve\ fe\ re = \\
\quad \textbf{let} \\
\qquad \textbf{val } e' = \mathcal{L}\ [\![e]\!]\ ve\ fe\ re \\
\qquad \textbf{val } e_i' = \mathcal{L}\ [\![e_i]\!]\ ve\ fe\ re \qquad i = 1..n \\
\quad \textbf{in} \\
\qquad e_{ck}'\ \langle e_1', \ldots, e_n'\rangle \\
\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{L}\ [\![f_{ck}\ \langle e_1, \ldots, e_n\rangle\ [sma_0\ \rho_0, \ldots, sma_l\ \rho_l]]\!]\ ve\ re\ fe = \\
\quad \textbf{let} \\
\qquad \textbf{val } e_{\rho_i} = \mathcal{L}\ [\![\rho_i]\!]\ ve\ fe\ re \qquad i = 0..l \\
\qquad \textbf{val } f\_lab = fe(f) \\
\qquad \textbf{val } e_i' = \mathcal{L}\ [\![e_i]\!]\ ve\ fe\ re \qquad i = 1..n \\
\qquad \textbf{val } sma_i' = \textbf{convert\_sma}(sma_i, re, \rho_i) \qquad i = 0..l \\
\quad \textbf{in} \\
\qquad f\_lab_{ck}\ \langle e_1', \ldots, e_n'\rangle\ \langle sma_0'\ e_{\rho_0}, \ldots, sma_l'\ e_{\rho_l}\rangle\ \langle ve(f)\rangle \\
\quad \textbf{end}
\end{array}
$$

Notice that the label for f is obtained by looking in fe.

6.8 Letregion

$$
\begin{array}{l}
\mathcal{L}\ [\![\texttt{letregion } \rho\texttt{:}m \texttt{ in } e \texttt{ end}]\!]\ ve\ fe\ re = \\
\quad \texttt{letregion } \rho\texttt{:}m \\
\quad \texttt{in} \\
\qquad \mathcal{L}\ [\![e]\!]\ (ve + \{\rho \mapsto \rho\})\ fe\ (re + \{\rho \mapsto \textbf{mult}(\texttt{1}, m)\}) \\
\quad \texttt{end}
\end{array}
$$

6.9 Declaring Value Variables

$$\mathcal{L}\ [\![\texttt{let val}\ \langle x_1,\ldots,x_n\rangle\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}]\!]\ ve\ fe\ re =$$
$$\texttt{let}$$
$$\quad \texttt{val}\ \langle x_1,\ldots,x_n\rangle\ \texttt{=}\ \mathcal{L}\ [\![e_1]\!]\ ve\ fe\ re$$
$$\texttt{in}$$
$$\quad \mathcal{L}\ [\![e_2]\!]\ (ve + \{x_i \mapsto x_i\}_{i=1..n})\ fe\ re$$
$$\texttt{end}$$

6.10 Unboxed Records

$$\mathcal{L}\ [\![\langle e_1,\ldots,e_n\rangle]\!]\ ve\ fe\ re =$$
$$\langle \mathcal{L}\ [\![e_1]\!]\ ve\ fe\ re,\ldots,\mathcal{L}\ [\![e_n]\!]\ ve\ fe\ re\rangle$$

6.11 Case

$$\mathcal{L}\ [\![\texttt{case}\ e\ \texttt{of}\ pat_1\ \texttt{=>}\ e_1\ \texttt{|}\ pat_2\ \texttt{=>}\ e_2]\!]\ ve\ fe\ re =$$
$$\textbf{let}$$
$$\quad \textbf{fun pat\_ve}\ (c,\ ve) = ve$$
$$\quad |\ \textbf{pat\_ve}\ (\texttt{::}\ x,\ ve) = ve + \{x \mapsto x\}$$
$$\textbf{in}$$
$$\quad \texttt{case}\ \mathcal{L}\ [\![e]\!]\ ve\ fe\ re\ \texttt{of}$$
$$\quad\quad pat_1\ \texttt{=>}\ \mathcal{L}\ [\![e_1]\!]\ (\textbf{pat\_ve}(pat_1,ve))\ fe\ re$$
$$\quad |\ pat_2\ \texttt{=>}\ \mathcal{L}\ [\![e_2]\!]\ (\textbf{pat\_ve}(pat_2,ve))\ fe\ re$$
$$\textbf{end}$$

7 Compiling LiftExp Programs Into KAM Programs

LiftExp programs are compiled into programs for the Kit Abstract Machine (KAM), which is presented in Section 3. Recall that the KAM is a stack based machine with four registers (*env*, *acc*, *sp*, *pc*), a stack, and a region heap.

The compilation function $\mathcal{C}^{\mathcal{T}}$ compiles LiftExp top-level expressions into KAM programs:

$$\mathcal{C}^{\mathcal{T}} : \text{TopDec} \ \rightarrow\ \text{KAMProg}$$

Here KAMProg denotes the set of KAM programs, ranged over by P. The compilation function $\mathcal{C}^{\mathcal{T}}$ uses the auxiliary compilation function $\mathcal{C}$:

$$\mathcal{C} : \text{LiftExp} \ \rightarrow\ \text{CompEnv} \ \rightarrow\ \text{Int} \ \rightarrow\ \text{KAMBlock}$$

Here KAMBlock, ranged over by B, denotes the set of possible sequences of KAM instructions. The compilation function takes as argument a compiler environment (see Section 7.1) and an integer, which denotes the stack pointer offset in the current functions activation record. The next few sections define auxiliary functions used by the compilation function $\mathcal{C}$.

7.1 Compiler Environments

A compiler environment *ce* maps lambda and region variables into how these variables are accessed at runtime:

$$ce : (\text{LamVar} \ \cup \text{RegVar}) \ \rightarrow\ \{\texttt{reg\_i}(o), \texttt{reg\_f}(o), \texttt{stack}(o), \texttt{env}(o), \texttt{env}\}$$

We use CompEnv to denote the set of all possible compiler environments. The possible values in the range of a compiler environment have the following meanings:

Environment value	Description
`reg_i`(o)	The region variable denotes an infinite region where the region descriptor is at offset o in the current activation frame.
`reg_f`(o)	The region variable denotes a finite region where the region itself is at offset o in the current activation frame.
`stack`(o)	The variable is a local variable and present in the current activation record with offset o.
`env`(o)	The variable is free in the function and its associated value is located in the environment register with offset o.
`env`	The variable is bound to the environment register.

Given a variable x, a compiler environment ce, and the top of the stack relative to the start of the current activation record sp, the auxiliary translation function **access** returns appropriate code for accessing the variable x:

fun access(x, ce, sp) =
  case $ce(x)$ **of**
    `stack`(o) ⇒ `SelectStack`($-sp + o$)
  | `env`(o) ⇒ `SelectEnv`(o)
  | `reg_i`(o) ⇒ `StackAddrInfBit`($-sp + o$)
  | `reg_f`(o) ⇒ `StackAddr`($-sp + o$)
  | `env` ⇒ `EnvToAcc`

7.2 Allocating in Regions

In this section we present an auxiliary compilation function **block_alloc** for allocating memory in regions and storing content from the stack into the allocated memory. The concept of storage mode annotations were introduced in Section 5.3. Storage mode annotations are categorized according to the table in Figure 12.

When generating code for boxed expressions, code for allocating blocks of memory in regions and for copying values into the blocks need be generated (see Section 7.5). The code returned by **alloc_block** assumes that the accumulator *acc* holds a pointer to a region and that the stack contains the values to be copied into the block. The function takes as argument a storage mode annotation *sma* and the number n of words to allocate.

fun alloc_block(sma, n) =
  case sma **of**
    `attop_li` ⇒ `BlockAlloc`(n)
  | `attop_lf` ⇒ `Block`(n)
  | `attop_ff` ⇒ `BlockAllocIfInf`(n)
  | `attop_fi` ⇒ `BlockAlloc`(n)
  | `sat_ff` ⇒ `BlockAllocSatInf`(n)
  | `sat_ff` ⇒ `BlockAllocSatIfInf`(n)
  | `atbot_li` ⇒ `BlockAllocAtbot`(n)
  | `atbot_lf` ⇒ `Block`(n)

sma	Description
`attop_li`	The region is `letregion` bound and infinite. Memory is allocated attop with the KAM instruction `Alloc`.
`attop_lf`	The region is `letregion` bound and finite so storage has already been set aside on the stack.
`attop_ff`	The region is an actual region argument to the current function or another function and free in the current function. The region can either be finite or infinite and the generated code must check the actual multiplicity before allocating.
`attop_fi`	The region is infinite and memory is allocated with the KAM instruction `Alloc`. A formal region parameter to a region polymorphic function may either be finite or infinite but in the case that the function stores more than one time in the region (denoted by the last `i` in the annotation) then the region is known to be infinite.
`sat_ff`	The region is infinite. The generated code must check the storage mode bit to see wheter or not the region should be reset before allocating in it.
`sat_ff`	The region is either finite or infinite. The generated code must check the multiplicity; if infinite then the storage mode must be tested and if this is atbot then the region is reset before allocation. The test on multiplicity and storage mode may be performed simultaneously.
`atbot_li`	The region is `letregion` bound and infinite. The region is reset before memory is allocated in the region.
`atbot_lf`	The region is `letregion` bound and finite. Memory has already been allocated on the stack and no resetting is necessary.

Figure 12: Categorization of storage mode annotations

sma	Change in storage mode
`attop_li`	No change is necessary because the region is `letregion` bound, thus the atbot bit is not set.
`attop_lf`	The region is finite and the atbot bit is therefore insignificant.
`attop_ff`	The region may or may not be infinite so we have to check the multiplicity and then clear the atbot bit if the multiplicity is infinite. Actually, we may blindly clear the atbot bit no matter the multiplicity.
`attop_fi`	The region is infinite and the atbot bit is cleared.
`sat_ff`	The atbot bit is already set appropriately.
`sat_ff`	The atbot bit is already set appropriately.
`atbot_li`	The atbot bit is set.
`atbot_lf`	The region is finite and the atbot bit is therefore insignificant.

Figure 13: Change in storage mode when an actual region is passed as argument to a region polymorphic function

After executing the generated code, the accumulator contains a pointer to the block, allocated either in the region heap or on the stack.

7.3 Setting Storage Modes

Storage modes are set when passing regions as arguments to `letrec` bound functions. Figure 13 specifies how storage bits are set according to storage mode annotations. Notice, that the atbot bit is never set on `letregion` bound regions when they are created so we do not have to explicitly clear the atbot bit for those regions.

The auxiliary compilation function **set_sm** assumes that the accumulator holds a pointer to a region and afterwards the accumulator holds the same pointer with the storage mode set.

fun set_sm(*sma*) =
  case *sma* **of**
    `attop_li` ⇒ `Nop`
  | `attop_lf` ⇒ `Nop`
  | `attop_ff` ⇒ `ClearAtbotBit`
  | `attop_fi` ⇒ `ClearAtbotBit`
  | `sat_ff` ⇒ `Nop`
  | `sat_ff` ⇒ `Nop`
  | `atbot_li` ⇒ `SetAtbotBit`
  | `atbot_lf` ⇒ `Nop`

In the following we show the function $\mathcal{C}$, which compiles LiftExp expressions into sequences of KAM instructions.

7.4 Variables and Constants

$\mathcal{C}\ [\![x]\!]\ ce\ sp =$ **access**(x, ce, sp)
$\mathcal{C}\ [\![\rho]\!]\ ce\ sp =$ **access**(ρ, ce, sp)
$\mathcal{C}\ [\![c]\!]\ ce\ sp =$ `Immed`(c)

7.5 Boxed Expressions

There are three kinds of boxed expressions: records, closures, and shared closures. Code generation for records and shared closures are identical.

$\mathcal{C}\ [\![(e_1, \ldots, e_n)\ sma\ e]\!]\ ce\ sp =$
  $\mathcal{C}\ [\![e_1]\!]\ ce\ sp$;
  `Push;`
  $\vdots$
  $\mathcal{C}\ [\![e_n]\!]\ ce\ (sp + n - 1)$;
  `Push;`
  $\mathcal{C}\ [\![e]\!]\ ce\ (sp + n)$;
  alloc_block(sma, n)

$\mathcal{C}\ [\![\lambda^{lab}\ [e_1, \ldots, e_n]\ sma\ e]\!]\ ce\ sp =$
  `PushLbl`(lab);
  $\mathcal{C}\ [\![e_1]\!]\ ce\ (sp + 1)$;
  `Push;`
  $\vdots$
  $\mathcal{C}\ [\![e_n]\!]\ ce\ (sp + n)$;
  `Push;`
  $\mathcal{C}\ [\![e]\!]\ ce\ (sp + n + 1)$;
  alloc_block$(sma, n + 1)$

$\mathcal{C}\ [\![[e_1, \ldots, e_n]_{\text{sclos}}\ sma\ e]\!]\ ce\ sp =$
  $\mathcal{C}\ [\![(e_1, \ldots, e_n)\ sma\ e]\!]\ ce\ sp$

7.6 Binary Operators and List Construction

$\mathcal{C}\ [\![e_1\ bop\ e_2]\!]\ ce\ sp =$
  $\mathcal{C}\ [\![e_1]\!]\ ce\ sp$;
  `Push;`
  $\mathcal{C}\ [\![e_2]\!]\ ce\ (sp + 1)$;
  `Bop`

$\mathcal{C}\ [\![\texttt{::}\ e]\!]\ ce\ sp =$
  $\mathcal{C}\ [\![e]\!]\ ce\ sp$;
  `Cons`

7.7 Record Selection

$\mathcal{C}\ [\![\texttt{\#}n(e)]\!]\ ce\ sp =$
  $\mathcal{C}\ [\![e]\!]\ ce\ sp$;
  `Select`(n)

7.8 Declaration of Recursive Functions

$\mathcal{C}\ [\![\texttt{letrec}\ f_{lab}\ \texttt{=}\ be\ a\ \texttt{in}\ e\ \texttt{end}]\!]\ ce\ sp =$

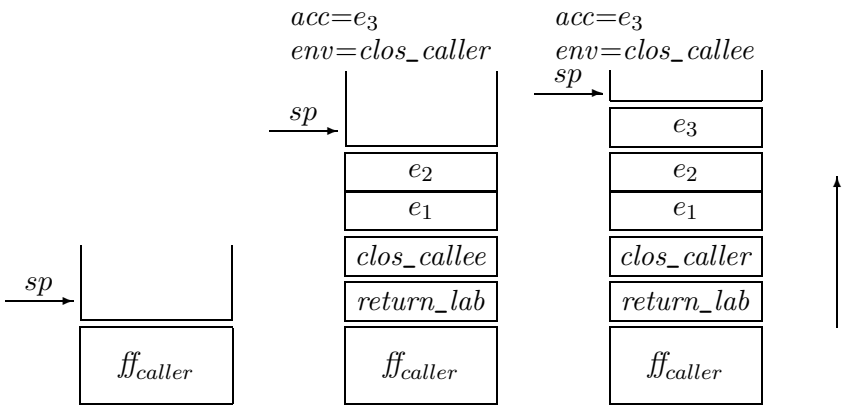

Figure 14: At left we have the stack before the application. At center we have the stack, *env* and *acc* before the `ApplyFnCall` instruction. At right the stack, *env* and *acc* at entry to the called function. The stack grows upwards. Three arguments are passed in the function call.

$\mathcal{C}\ [\![be\ a]\!]\ ce\ sp$;
`Push;`
$\mathcal{C}\ [\![e]\!]\ (ce + \{f \mapsto \texttt{stack}(sp)\})\ (sp+1)$;
`Pop`

7.9 Function Applications

The LiftExp language features four kinds of applications: ordinary calls and tail calls for unknown and known functions. Functions may pass any number of arguments.

Generally, the return address is stored on the stack together with the closure of the calling function just below the arguments. The KAM features four function-call instructions, namely `ApplyFnCall`, `ApplyFnJmp`, `ApplyFunCall`, and `ApplyFunJmp`, which, in turn, are explained below.

$\mathcal{C}\ [\![e_{\text{fncall}}\ \langle e_1, \ldots, e_n\rangle]\!]\ ce\ sp =$
let
  val *return_lbl* = **new_label**("return_lbl")
in
  `PushLbl(`*return_lbl*`);`
  $\mathcal{C}\ [\![e]\!]\ ce\ (sp+1)$;
  `Push;`
  $\mathcal{C}\ [\![\langle e_1, \ldots, e_n\rangle]\!]\ ce\ (sp+2)$;
  `ApplyFnCall(`n`);`
  `Label(`*return_lbl*`)`
end

Figure 14 shows the stack before the `ApplyFnCall` instruction and after the instruction (i.e., at entry to the called function). All arguments are passed on the stack. The closure for the calling function is also stored on the stack so that it can be restored at return from the function.

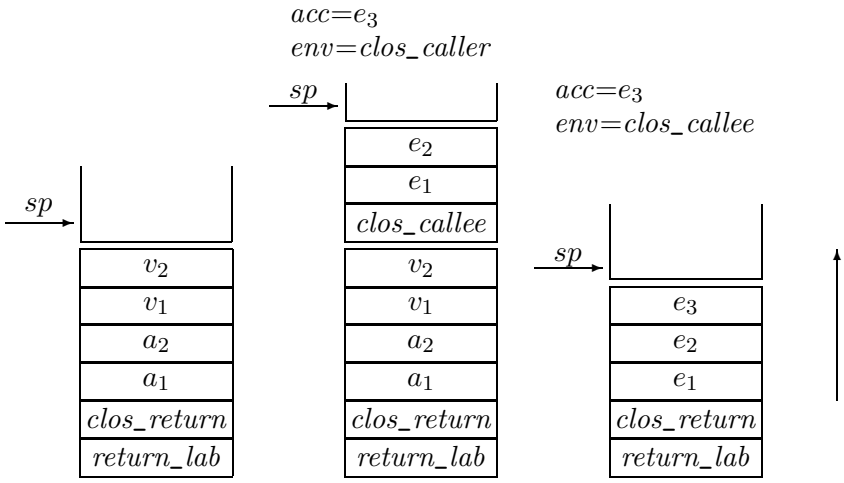

Figure 15: At left we have the stack (with $sp = 4$) before the application. At center we have the stack, *env* and *acc* before the `ApplyFnJmp` instruction. At right the stack, *env* and *acc* at entry to the called function. The stack grows upwards. Three arguments are passed in the function call. The values a_1 and a_2 are the original arguments passed to the function that we are leaving. The values v_1 and v_2 are let-bound variables that have not been popped from the stack yet; the `ApplyFnJmp` instruction must therefore also pop v_1 and v_2.

$\mathcal{C}\ [\![e_{\text{fnjmp}}\ \langle e_1, \ldots, e_n\rangle]\!]\ ce\ sp =$
  $\mathcal{C}\ [\![e]\!]\ ce\ sp$;
  `Push;`
  $\mathcal{C}\ [\![\langle e_1, \ldots, e_n\rangle]\!]\ ce\ (sp+1)$;
  `ApplyFnJmp`(n,sp)

Figure 15 shows the stack before the application, before the `ApplyFnJmp` instruction and after (i.e., at entry to the called function). There are two arguments (a_1 and a_2) passed to the current function in the example. Allthough the application is in tail position, it may happen that local variables are still pushed on the stack (v_1 and v_2), which must also be popped by the `ApplyFnJmp` instruction.

A known function call is similar to an unknown function call except that both ordinary and region arguments are passed on the stack. We set the storage mode on region pointers, as described in Section 7.3, before they are pushed on the stack. Figure 16 shows the stack, environment and accumulator at various stages in a call with four region arguments and three ordinary arguments.

$\mathcal{C}\ [\![lab_{\text{funcall}}\ \langle e_1, \ldots, e_n\rangle\ \langle sma_1 e'_1, \ldots, sma_m e'_m\rangle\ \langle e_{\text{clos}}\rangle]\!]\ ce\ sp =$
  let
    val *return_lbl* = **new_label**("return_lbl")
  in
    `PushLbl`$(return\_lbl)$;
    $\mathcal{C}\ [\![e_{\text{clos}}]\!]\ ce\ (sp+1)$;
    `Push;`
    $\mathcal{C}\ [\![e'_1]\!]\ ce\ (sp+2)$;
    set_sm(sma_1);

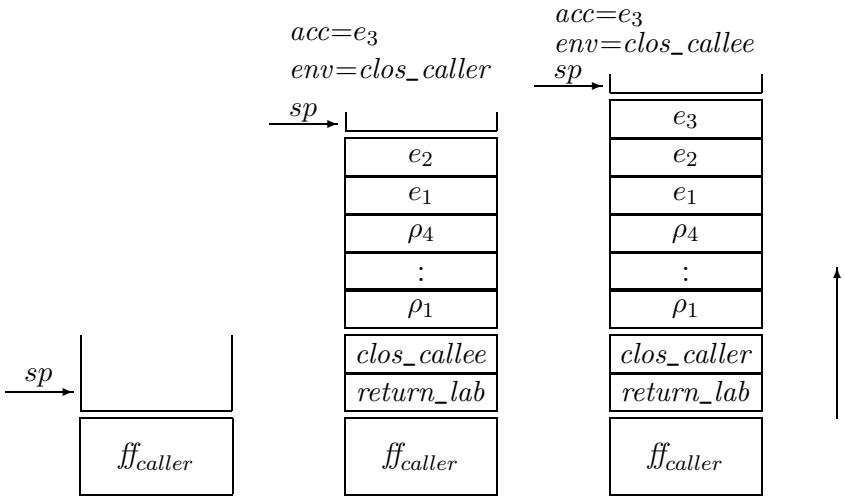

Figure 16: At left we have the stack before the application. At center we have the stack, *env* and *acc* before the `ApplyFunCall` instruction. At right the stack, *env* and *acc* at entry to the called function. The stack grows upwards. Three ordinary arguments and four region arguments are passed to the function.

```
    Push;
     ⋮
    C [[e'_m]] ce (sp + m + 1) ;
    set_sm(sma_m);
    Push;
    C [[⟨e_1, ..., e_n⟩]] ce (sp + m + 2) ;
    ApplyFunCall(lab,n + m);
    Label(return_lbl)
end
```

A tail call to a known function may pass region arguments as well as ordinary arguments. For certain kinds of tail calls it is possible to figure out that some region argument is already possitioned correctly on the stack, making it possible to avoid an update to the stack. The present implementation makes no attempt at figuring this out; experiments with the x86 backend showed that the extra complexity did not help much. For more information on this possible optimization, see [TBE+98, Chapter 14].

$$
\begin{aligned}
&\mathcal{C}\ [\![lab_{\text{funjmp}}\ \langle e_1,\ldots,e_n\rangle\ \langle sma_1\ e'_1,\ldots,sma_m\ e'_m\rangle\ \langle e_{\text{clos}}\rangle]\!]\ ce\ sp = \\
&\quad \mathcal{C}\ [\![e_{\text{clos}}]\!]\ ce\ sp\ ; \\
&\quad \texttt{Push;} \\
&\quad \mathcal{C}\ [\![e'_1]\!]\ ce\ (sp+1)\ ; \\
&\quad \textbf{set\_sm}(sma_1); \\
&\quad \texttt{Push;} \\
&\quad \vdots \\
&\quad \mathcal{C}\ [\![e'_m]\!]\ ce\ (sp+m)\ ;
\end{aligned}
$$

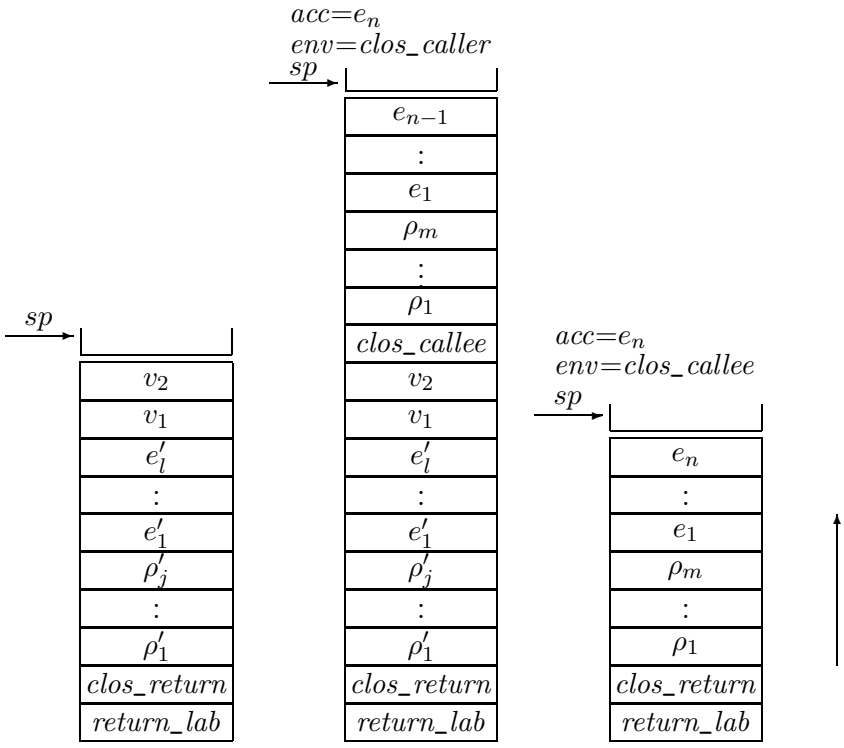

Figure 17: At left we have the stack before the application where $e'_1, \ldots, e'_l$ are ordinary arguments to the current function and $\rho'_1, \ldots, \rho'_j$ are region arguments to the current function. At center we have the stack, *env*, and *acc* before the `ApplyFunJmp` instruction, where $e_1, \ldots, e_n$ are the ordinary arguments in the call. The region arguments $\rho'_1, \ldots, \rho'_m$ to pass in the call are positioned below the ordinary arguments on the stack. At right the stack, *env*, and *acc* at entry to the called function. The values v_1 and v_2 are let-bound and are popped by the `ApplyFunJmp` instruction before jumping. The stack grows upwards.

set_sm(sma_m);
`Push;`
$\mathcal{C}\ [\![\langle e_1, \ldots, e_n \rangle]\!]\ ce\ (sp + m + 1)$;
`ApplyFunJmp`(lab,$n + m$,sp)

Figure 17 shows the stack, environment, and accumulator at various stages in the call.

7.10 Letregion

$\mathcal{C}\ [\![$`letregion` ρ : n `in` e `end`$]\!]\ ce\ sp =$
`LetregionFin`(n);
$\mathcal{C}\ [\![e]\!]\ (ce + \{\rho \mapsto$ `reg_f`$(sp)\})\ (sp + n)$;
`Pop`(n)

$\mathcal{C}\ [\![$`letregion` ρ : ∞ `in` e `end`$]\!]\ ce\ sp =$
`LetregionInf;`

$\mathcal{C}\ [\![e]\!]\ (ce{+}\{\rho \mapsto \texttt{reg\_i}(sp)\})\ (sp + size_{rDesc})$;
`EndregionInf`

7.11 Declaring Variables

$\mathcal{C}\ [\![\texttt{let val}\ \langle x_1, \ldots, x_n\rangle\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}]\!]\ ce\ sp =$
$\mathcal{C}\ [\![e_1]\!]\ ce\ sp$;
`Push;`
$\mathcal{C}\ [\![e_2]\!]\ (ce + \{x_1 \mapsto \texttt{stack}(sp), \ldots, x_n \mapsto \texttt{stack}(sp + n - 1)\})\ (sp + n)$;
`Pop`(n)

$\mathcal{C}\ [\![\langle e_1, \ldots, e_n\rangle]\!]\ ce\ sp =$
$\mathcal{C}\ [\![e_1]\!]\ ce\ sp$;
`Push;`
$\vdots$
$\mathcal{C}\ [\![e_{n-1}]\!]\ ce\ (sp + n - 2)$;
`Push;`
$\mathcal{C}\ [\![e_n]\!]\ ce\ (sp + n - 1)$

7.12 Case

$\mathcal{C}\ [\![\texttt{case}\ e\ \texttt{of}\ pat_1\ \texttt{=>}\ e_1\ \texttt{|}\ pat_2\ \texttt{=>}\ e_2]\!]\ ce\ sp =$
let
  fun comp_sel(pat,e) =
    case pat **of**
      $c \Rightarrow \mathcal{C}\ [\![e]\!]\ ce\ sp$
    | $::x \Rightarrow$ `Decons;`
      `Push;`
      $\mathcal{C}\ [\![e]\!]\ (ce + \{x \mapsto \texttt{stack}(sp)\})\ (sp + 1)$;
      `Pop`
  fun comp_match(pat,$true\_lbl$) =
    case pat **of**
      $c \Rightarrow$ `IfEq`(c,$true\_lbl$)
    | $::x \Rightarrow$ `IfCons`($true\_lbl$)
  val $true\_lbl$ = **new_label**("true_lbl")
  val $join\_lbl$ = **new_label**("join_lbl")
in
  $\mathcal{C}\ [\![e]\!]\ ce\ sp$;
  comp_match(pat_1,$true\_lbl$);
  comp_sel(pat_2,e_2);
  `Jmp`($join\_lbl$);
  `Label`($true\_lbl$)
  comp_sel(pat_1,e_1);
  `Label`($join\_lbl$)
end

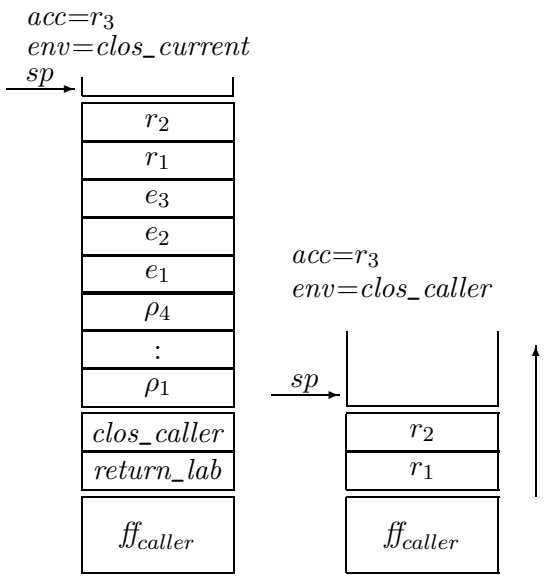

Figure 18: At left we have the stack, *env*, and *acc* before the `Return` instruction. Four region arguments, three ordinary arguments have been passed to the current function and three values are returned. At right we have the stack, *env*, and *acc* after the `Return` instruction. The stack grows upwards.

7.13 Top-Level Declarations

The `Return` instruction removes arguments passed to the current function and replaces return values on the stack. Figure 18 shows the stack before and after a `Return` instruction. Arguments on the stack are ordinary arguments, region arguments, or both.

$$
\begin{array}{l}
\mathcal{C}^{\mathcal{T}}\ [\![\lambda^{\texttt{fun}}_{lab}\ cc\ \texttt{=>}\ e]\!] = \\
\quad \texttt{fun}\ lab\ \texttt{is} \\
\qquad \mathcal{C}\ [\![e]\!]\ (\{\rho_1 \mapsto \texttt{stack}(0), \ldots, \rho_l \mapsto \texttt{stack}(l-1), \\
\qquad\qquad x_1 \mapsto \texttt{stack}(l), \ldots, x_n \mapsto \texttt{stack}(l+n-1), \\
\qquad\qquad env \mapsto \texttt{env}\})\ (l+n)\ \texttt{;} \\
\qquad \texttt{Return}(l+n,\textbf{no\_returns}(e))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}^{\mathcal{T}}\ [\![\lambda^{\texttt{fn}}_{lab}\ cc\ \texttt{=>}\ e]\!] = \\
\quad \texttt{fun}\ lab\ \texttt{is} \\
\qquad \mathcal{C}\ [\![e]\!]\ (\{x_1 \mapsto \texttt{stack}(0), \ldots, x_n \mapsto \texttt{stack}(n-1), env \mapsto \texttt{env}\})\ n\ \texttt{;} \\
\qquad \texttt{Return}(n,\textbf{no\_returns}(e))
\end{array}
$$

where *cc* is on the following form: {clos=*env*, args=$[x_1, \ldots, x_n]$, regargs=$[\rho_1, \ldots, \rho_l]$}). The function **no_returns** returns the number of values that e evaluates to.

7.14 Example Program

The result of translating the example program shown in Figures 10 and 11 into KAM code is shown in Figure 19. The figure does not show the main function. There are many posible peephole

optimizations that can be applied to the generated code, including eliminations of jumps to jumps and combination of stack- and push-instructions.

8 Extensions to the KAM

Besides the KAM instructions mentioned in the previous sections, an implementation of the KAM for compiling Standard ML must provide instructions for managing exceptions and various primitives.

Exceptions are implemented using a register to point to the current exception handler on the stack. Each exception handler on the stack is associated with a closure, which represents the actual handler, and exception handlers on the stack are linked, so that a previous exception handler can be restored upon raising an exception. The runtime system uses C's `longjmp` and `setjmp` features to make it possible to raise exceptions from primitives written in C. Consult [EH95] for more information about the implementation of the exception mechanism.

9 SMLserver

SMLserver is a Web server platform that allows dynamic Web pages written in Standard ML to be served efficiently via bytecode interpretation within the Web server. SMLserver builds on AOLserver, which is a Web server with a rich set of utility features including URL filtering, virtual hosting, script scheduling, and easy connectivity to a variety of database systems. SMLserver thus has easy access to all of these features.

There are at least two advantages of using Standard ML for Web applications:

1. Standard ML is a type safe language, which means that more bugs are discovered early in the development process.

2. The rich set of language features, including polymorphism, higher order functions, and the Standard ML modules language, provides means for increasing source-code reusability.

More information about SMLserver is available from the SMLserver tutorial [EH02] and from the SMLserver Web page `http://www.smlserver.org`.

10 Conclusion

In this document we have presented the Kit Abstract Machine (KAM), a region-based abstract machine, which serves as a target for the region-based Standard ML compiler, the ML Kit. The document also presents a translation from an intermediate region-explicit language RegExp into the language LiftExp, in which closures are made explicit and functions are hoisted to top level. Finally, the document presents how LiftExp programs are compiled into code for the KAM.

The KAM is available as a target for the ML Kit compiler and it is an essential part of the SMLserver project, which aims at building an efficient multi-threaded Web server platform for the Standard ML programming language [EH02].

References

[BT01] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, 258:299–392, 2001.

```
fun foldl is
   PushLbl(fn_b);
   SelectStack(-2); (*f*)
   Push;
   SelectStack(-3); (*r8*)
   Push;
   EnvToAcc; (*env*)
   Push;
   SelectStack(-5);
   BlockAllocIfInf(4);
   Return(3,1);
fn fn_b is
   PushLbl(fn_xs);
   SelectStack(-2); (*b*)
   Push;
   SelectEnv(1);
   Push;
   SelectEnv(3);
   Push;
   SelectEnv(2);
   BlockAllocIfInf(4);
   Return(1,1);
fn fn_xs is (*xs->s(0)*)
   SelectStack(-1); (*xs*)
   IfEq(nil,l_true);
   Decons;
   Push; (*v942*)
   SelectStack(-1); (*v942*)
   Select(0);
   Push; (*x*)
   SelectStack(-2); (*v942*)
   Select(1);
   Push; (*xs'*)
   LetregionFin(4); (*r22*)
   PushLbl(l_return1) ; (*[fncall_1*)
   LetregionFin(4); (*r24*)
   PushLbl(l_return2) ; (*[fncall_2*)
   LetregionFin(2); (*r25*)
   PushLbl(l_return3) ; (*[funcall*)
   SelectEnv(3);
   Push; (*clos_funcall*)
   StackAddr(-9); (*r24*)
   Nop; atbot_lf
   Push; atbot_lf(*r24*)
   StackAddr(-15); (*r22*)
   Nop; atbot_lf
   Push; atbot_lf(*r22*)
   SelectEnv(2); (*arg funcall*)
   ApplyFunCall(foldl,3);
Label(l_return3); (*funcall]*)
   Pop(2); (*r25*)
   Push; (*clos fncall_2*)
   PushLbl(l_return4) ; (*[fncall_4*)
   PushLbl(l_return5) ; (*[fncall_5*)
   SelectEnv(2);
   Push; (*clos fncall_5*)
   SelectStack(-10); (*x*)
   ApplyFnCall(1);
Label(l_return5); (*fncall_5]*)
   Push; (*clos fncall_4*)
   SelectEnv(1); (*arg fn_call4*)
   ApplyFnCall(1);
Label(l_return4); (*fncall_4]*)
   ApplyFnCall(1);
Label(l_return2); (*fncall_2]*)
   Pop(4); (*r24*)
   Push; (*clos fncall_1*)
   SelectStack(-5); (*xs'*)
   ApplyFnCall(1);
Label(l_return1); (*fncall_1]*)
   Pop(4); (*r22*)
   Pop; (*x*)
   Pop; (*xs'*)
   Pop; (*v942*)
   Jmp(l_join);
Label(l_true);
   SelectEnv(1);
Label(l_join);
   Return(1,1);
```

Figure 19: The functions *foldl*, *fn_b* and *fn_xs* compiled into KAM code. Comments are added to the program in the style of Standard ML.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 171–183, St. Petersburg, Florida, January 1996. ACM Press.

[EH95] Martin Elsman and Niels Hallenberg. An optimizing backend for the ML Kit using a stack of regions. Student Project, Department of Computer Science, University of Copenhagen (DIKU), July 5 1995.

[EH02] Martin Elsman and Niels Hallenberg. *SMLserver—A Functional Approach to Web Publishing*, February 2002.

[Hal99] Niels Hallenberg. Combining Garbage Collection and Region Inference in The ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen, June 1999.

[HET02] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. ACM Press, June 2002. Berlin, Germany.

[Ler90] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, February 1990.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998. (plus 24 pages of electronic appendix).

[TB00] Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, May 2000.

[TBE+98] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, Dept. of Computer Science, University of Copenhagen, 1998.

[TBE+01] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. *Programming with Regions in the ML Kit (for Version 4)*. The IT University of Copenhagen, 2001. (234 pages). Available via `http://www.it-c.dk/research/mlkit`.

[Tof98] Mads Tofte. A Brief Introduction to Regions. In *Proceedings on the 1998 ACM International Symposium on Memory Management (ISMM '98)*, pages 186–195, 1998.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.