The Lazy Virtual Machine specification

Technical report: UU-CS-2004-052

Daan Leijen
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
daan@cs.uu.nl

August 22, 2005

1 Introduction

This chapter describes the Lazy Virtual Machine (LVM). Just like the JVM [12], it defines a portable instruction set and file format. However, it is specifically designed to execute languages with non-strict (or lazy) semantics. Is there need for such system? After all, there are many compilers and interpreters for lazy languages, for example, GHC developed at the Glasgow university, the HUGS interpreter by Mark Jones, NHC from York university (UK), the HBC compiler developed at Chalmers, and Clean from Nijmegen. One may think that with this diversity of systems there is no need anymore for other compilers, as most implementation issues have been resolved.

However, the current compilers and interpreters have become large systems that are hard to adapt – it has become difficult to experiment with new type systems, language constructions, compiler transformations, profiling, or debugging tools. In particular, the work on embedded languages as described in the previous two chapters gave rise to experimentation with various extensions to the Haskell language. Eventually, this lead to the development of the LVM:

- A small portable system that can be easily adapted to support different (experimental) languages and type systems.
- A simple and robust instruction set that is an easy target for compiler front-ends.
- Efficient interpretation or JIT compilation is possible.
- A toolkit that translates an *untyped*, rich intermediate language (λ_{core}) to LVM instructions. Note that we use untyped expressions in order to experiment with extensions that are hard to type at this level, examples include *type indexed records* [25] and dependent types [1].

The LVM is currently implemented on top of the OCaml runtime system [10, 11]. The system runs on many platforms, including Windows, various Unix's, MacOSX and 64-bit platforms like the DEC alpha. The LVM is used as a backend for the experimental HX system [26] and the Helium compiler – this compiler implements a very large subset of Haskell and is currently used to teach first year students Haskell at Utrecht University.

The design and implementation of the LVM is as simple and modular as possible. However, simplicity does not imply that it is a toy system; the implementation is full fledged including support for exotic features as (asynchronous) exceptions, concurrency, a foreign function interface, generational garbage collection, and execution traces.

This chapter will focus on the translation of the intermediate language to LVM instructions, and on the operational semantics of the instructions themselves. Many items of this chapter have been described before, and the main contribution of this chapter is the *design* of a 'real world' instruction set, operational semantics, and translation scheme as a whole. More specifically:

- We define a naive and straightforward translation scheme from the low-level λ_{lvm} language to LVM instructions. Instead of defining many *optimized* translation schemes [14, 6, 23], we define a small set of rewrite rules on instructions that achieve the same effect. The correctness of the rewrite rules is relatively easy to prove with the operational semantics. In contrast, an optimized translation scheme is much harder to prove correct, as one has to show a correspondence between the operational semantics of the λ_{lvm} language and the generated instructions. Furthermore, the rewrite rules are most of the time even more effective than optimized translation rules, as the rewrite rules sometimes find optimization opportunities between instructions that are unrelated at the language level.

- The low-level λ_{lvm} language has an operational reading and directly reflects the capabilities of the abstract machine. As such, we are able to reason about denotationally equivalent expressions that have a different operational behaviour.

- We define simple operational semantics for the instructions. A state is determined by just three items: the current code, the heap, and the stack. Besides being simpler than many other instruction sets [15, 14, 6, 23], the instructions also map directly onto C instructions, reducing the number of bugs in an implementation and improving our understanding of the relationship between abstract machine and concrete implementation.

- As the instructions can be so closely related to an actual implementation, we can reason about implementation techniques that are normally only described informally, or explained via pictures. Examples are exception handling, returning constructors in registers, and the reason why `seq` frames are a necessary addition to the STG machine [15].

2 An overview

The compilation of a high level functional language to LVM instructions goes via a number of intermediate languages: λ_{core}, λ_{lvm}, and extended LVM instructions. The process is sketched in figure 1. The actual compilation steps involved are:

- Translate the source language to λ_{core}; an enriched lambda calculus that corresponds closely to the intermediate core language of the GHC compiler [15, 21].

- *Normalise*: translate the λ_{core} language to the λ_{lvm} language, a more restricted form of λ_{core} that maps conveniently to LVM instructions.

- *Compile*: translate λ_{lvm} to LVM instructions that contain *pseudo* instructions. These pseudo instructions are used in the next phases to calculate correct stack and code offsets, but should be removed completely when generating the final instruction stream.

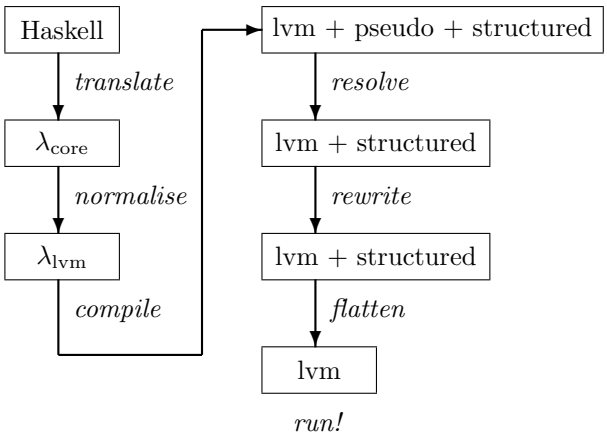

Figure 1: Compilation scheme

- *Resolve*: use and remove pseudo instructions that resolve stack offsets of local variables and arguments.
- *Rewrite*: optimize the instruction stream using rewrite rules.
- *Flatten*: remove all structure from the instructions and generate a flat stream of instructions that can efficiently be interpreted. This phase uses (and removes) the last pseudo instructions by calculating code offsets for jumps.
- *Execute*: execute the instruction streams according to the state transition rules.

The next sections describe all these steps in detail, but to give a feel of what each of these steps do, we describe each step in the context of a small example. We start with the following Haskell program:

$$
\begin{aligned}
&\mathit{main} \;=\; \mathit{const}\ \mathit{inc}\ (\lambda\, x \to x)\ 42 \\
&\qquad \textbf{where} \\
&\qquad\quad \mathit{inc}\ x \;=\; x \;+\; 1 \\
&\qquad\quad \mathit{const}\ x\ y \;=\; x
\end{aligned}
$$

The front-end language, in this case Haskell, is translated into an enriched lambda calculus, called λ_{core}. In this example, only the Haskell specific `where` binding is translated into a `let` binding but in general more transformation can be necessary; pattern matching compilation for example. Here is the translated program in λ_{core}:

$$
\begin{aligned}
&\mathit{main} \;= \textbf{let}\ \mathit{inc}\ x \;=\; x \;+\; 1 \\
&\qquad\qquad\quad\ \ \mathit{const}\ x\ y \;=\; x \\
&\qquad\qquad \textbf{in}\ \ \mathit{const}\ \mathit{inc}\ (\lambda\, x \to x)\ 42
\end{aligned}
$$

The *Normalise* step translates λ_{core} to a restricted form, called λ_{lvm}. The λ_{lvm} language is specifically designed to reflect the capabilities of the abstract machine and maps easily onto LVM instructions. In our example, all functions are lifted to top-level and binary application is translated into vector application:

$$\begin{array}{lcl} \mathit{id}\ x & = & x \\ \mathit{inc}\ x & = & x\ +\ 1 \\ \mathit{const}\ x\ y & = & x \\ \mathit{main} & = & \mathit{const}\ \mathit{inc}\ \mathit{id}\ 42 \end{array}$$

Finally, the program is in a form where it can be compiled to LVM instructions, here are the instructions for the *id* function:

$$\mathit{id} \ \mapsto\ \mathbf{instr}(\mathsf{ArgChk}(1); \mathsf{Atom}(\mathsf{Param}(x); \mathsf{PushVar}(x); \mathsf{NewAp}(1));\ \mathsf{Enter})$$

However, the compiled program does not just consist of LVM instructions, but also contains *pseudo* instructions like Param and Atom. These instructions are used during the next phases to calculate stack and code offsets. The stack offsets are determined in the *resolve* step:

$$\mathit{id} \ \mapsto\ \mathbf{instr}(\mathsf{ArgChk}(1); \mathsf{PushVar}(0); \mathsf{NewAp}(1); \mathsf{Slide}(1,1); \mathsf{Enter})$$

As we can see, the pseudo instructions Param and Atom have dissappeared, and variable references are replaced by a stack offset. After the stack offsets have been resolved, the instruction stream is rewritten according to simple rewrite rules. This step is strictly used to optimize the instruction stream. It is interesting to see that just few rewrite rules suffice to completely replace complicated translation schemes [22, 6, 23]. The optimized instruction stream is:

$$\mathit{id} \ \mapsto\ \mathbf{instr}(\mathsf{ArgChk}(1); \mathsf{Enter})$$

The final step flattens the instruction stream by calculating code offsets and it removes the remaining pseudo instructions. The complete code for our example becomes[1]:

$$\begin{array}{lcl} \mathit{id} & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(1); \mathsf{Enter}) \\ \mathit{inc} & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(1); \mathsf{PushInt}(1); \mathsf{PushVar}(1); \mathsf{AddInt}; \mathsf{Return}) \\ \mathit{const} & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(2); \mathsf{PushVar}(0); \mathsf{Slide}(1,2); \mathsf{Enter}) \\ \mathit{main} & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(0); \mathsf{PushInt}(42); \mathsf{PushCode}(\mathit{id}); \\ & & \quad \mathsf{PushCode}(\mathit{inc}); \mathsf{EnterCode}(\mathit{const})) \end{array}$$

The program can now be executed by the abstract LVM machine. The state of the machine is determined by the current code, the stack, and the heap. The initial state of the machine is always the Enter instruction with the value *main* on the stack. In our example, the initial heap *hp* just consists of the compiled functions. Note that we write $hp[p \mapsto x]$ if the heap *hp* contains a pointer p that points to value x. Here is a complete execution trace of our example:

[1] Actually, some functions are compiled into more efficient code, but for clarity we use the unoptimized version.

	Code	Stack	Heap
$\Longrightarrow$	Enter	$[main]$	$hp[main \mapsto \mathbf{instr}(...)]$
$\Longrightarrow$	ArgChk(0); PushInt(42); ...	$[main]$	hp
$\Longrightarrow$	PushInt(42); PushCode(id); ...	[]	hp
$\Longrightarrow$	PushCode(id); PushCode(inc); ...	$[42]$	hp
$\Longrightarrow$	PushCode(inc); EnterCode$(const)$	$[id, 42]$	hp
$\Longrightarrow$	EnterCode$(const)$	$[inc, id, 42]$	$hp[const \mapsto \mathbf{instr}(...)]$
$\Longrightarrow$	PushVar(0); Slide$(1, 2)$; ...	$[inc, id, 42]$	hp
$\Longrightarrow$	Slide$(1, 2)$; Enter	$[inc, inc, id, 42]$	hp
$\Longrightarrow$	Enter	$[inc, 42]$	$hp[inc \mapsto \mathbf{instr}(...)]$
$\Longrightarrow$	ArgChk(1); PushInt(1); ...	$[inc, 42]$	hp
$\Longrightarrow$	PushInt(1); AddInt; ...	$[42]$	hp
$\Longrightarrow$	AddInt; Return	$[1, 42]$	hp
$\Longrightarrow$	Return	$[43]$	hp
$\Longrightarrow$	terminate with an integer	$[43]$	hp

The next section describes the abstract machine and instructions in detail. Section 4 describes the λ_{lvm} and λ_{core} languages. Section 5 describes the translation from λ_{lvm} to LVM instructions, together with the rewrite rules. The chapter closes with an assessment of a real implementation of the LVM and is followed by conclusions and two appendices that describe the format of binary LVM files.

3 The abstract machine

In this section we look at the operational semantics of the LVM instructions. The semantics of the LVM is given by state transitions [24]. The state of the LVM is determined by a triple: the current instructions *is*, the stack *st*, and the heap *hp*.

The instruction sequence *is* consists of instructions and arguments of instructions. The empty sequence is written as [] and an initial instruction with arguments x and y, is written as $\mathsf{Instr}(x, y) : is$. Arguments and instructions have the same size and the previous expression is equivalent to $\mathsf{Instr} : x : y : is$.

The stack *st* is a sequence of values. The empty stack is written as [] and a non-empty stack with an initial value x as $x : st$. The n^{th} value on the stack is written as $st[n]$ where $st[0]$ is the top of the stack. Besides values, the stack also contains a chain of stack *markers*, each taking two stack slots. A marker is associated with the value next on the stack. A marker together with its value is called a *frame*. There exist three kinds of markers, update markers (**upd**), continuation markers (**cont**), and catch markers (**catch**).

The heap *hp* is a map from pointers p to heap values. We write $hp[p \mapsto x]$ if the heap *hp* contains a pointer p that points to value x. The extension of the heap with a fresh pointer p to value x is written as $hp \circ [p \mapsto x]$. The update of an existing pointer p with value x is written as $hp \bullet [p \mapsto x]$.

Heap values are tagged and have varying sizes. Although we give a short description, the exact meaning of each heap value becomes clear during the description of the instruction set. There exist six kinds of heap values:

$\mathbf{instr}(is)$	A sequence of instructions *is*.
$\mathbf{ap}(x_1, ..., x_n)$	An updateable application block.
$\mathbf{nap}(x_1, ..., x_n)$	A non-updateable application block.
$\mathbf{con}_t(x_1, ..., x_n)$	A constructor with tag t and arguments x_1 to x_n.
$\mathbf{inv}_n$	An invalid block of size n.
$\mathbf{raise}(x)$	An exception block, raises exception x when entered.

The *initial heap* contains all global values. All instruction arguments that refer to a global value are fixed by the runtime loader to contain the proper heap pointer. The special value *inv* will point to an invalid block of size 0: $inv \mapsto \mathbf{inv}_0$

The initial state of the abstract machine consists of: the Enter instruction, a stack that just contains (a pointer to) the function *main*, and the initial heap.

Since the state of the abstract machine is so simple, it maps directly onto current hardware. Instruction streams can be modelled with a simple instruction pointer as instructions are never modified. The stack can be modelled with an array and a stack pointer. Only the

heap requires extensive runtime support, but that seems unavoidable in any garbage collected language.

3.1 Basic instructions

We first introduce a minimal set of instructions that support a minimal subset of λ_{lvm}. New instructions are introduced as the need arises by taking new language features into account. We first treat a minimal useful subset of the λ_{lvm} language, just consisting of top-level values with (partial) function applications.

All `let`-bound local variables and function parameters reside on the stack. Three instructions manipulate the stack: PushVar pushes a copy of a value that resides on the stack (i.e. a local variable or parameter), PushCode pushes a pointer to a top-level value, and Slide slides out unused values.

	Code	Stack	Heap
	PushCode(f) : is	st	hp
$\Longrightarrow$	is	$f : st$	hp
	PushVar(ofs) : is	st	hp
$\Longrightarrow$	is	$st[ofs] : st$	hp
	Slide(n, m) : is	$x_1 : ... : x_n : ... : x_{n+m} : st$	hp
$\Longrightarrow$	is	$x_1 : ... : x_n : st$	hp

The parameters of a function are pushed on the stack in a right-to-left order. This is dual to most imperative languages that use left-to-right order, like Java and ML. The most notable exception is the C language that uses a right-to-left calling convention in order to support functions with a variable number of arguments. However, for any higher-order language that allows partial applications, it is necessary to use this calling convention. The following example illustrates why partial applications force a right-to-left order.

$$
\begin{array}{ll}
id\ x & = x \\
const\ x\ y & = x \\
apply\ f\ x & = f\ x \\
main & = apply\ (const\ id)\ const
\end{array}
$$

With a right-to-left order, everything works well – inside *apply*, the argument x is pushed (which is *const*) and f is entered. This is actually the expression *const id* that pushes *id* and enters *const* with a proper stack: $id : const : []$, where parameter x is *id* and parameter y is *const*. If a left-to-right order is used, the partial application *const id* somehow has to insert its argument between arguments already residing on the stack. This can not be done without whole-program analysis and might even be impossible to do in general.

Partial applications combined with polymorphism also lead to the famous *argument check*. In a higher-order, polymorphic language it is not always possible to determine at a call

site if a function is partially applied or not. In the previous example, this is the case for the parameter f in the *apply* function. For this reason, each function checks the number of arguments itself with the ArgChk instruction, which is always the first instruction of a top-level value. If there are enough arguments on the stack, execution continues. If there are not enough arguments on the stack, we immediately return with a functional value as a result.

	Code	Stack	Heap
$n \leq m$	$\mathsf{ArgChk}(n) : is$	$f : x_1 : ... : x_m : st$	hp
$\Longrightarrow$	is	$x_1 : ... : x_m : st$	hp
(1) $n > m$	$\mathsf{ArgChk}(n) : is$	$f : x_1 : ... : x_m : []$	hp
$\Longrightarrow$	$[]$	$f : x_1 : ... : x_m : []$	hp

(1) termination with a functional value.

Due to polymorphism, it is not always possible to determine at a call site which particular function is called. Therefore, the Enter instruction is able to enter any kind of value that resides on top of the stack. Right now, we only have instruction values but we will later add more values that can be entered.

	Code	Stack	Heap
	$\mathsf{Enter} : is$	$f : st$	$hp[f \mapsto \mathbf{instr}(is_f)]$
$\Longrightarrow$	is_f	$f : st$	hp

Note that we *enter* a function instead of *calling* it. Every function application in λ_{lvm} is a *tail* call and, since we don't need to return, there is no need to push a return address either. It is necessary however to remove any local variables and parameters of the calling function that are still on the stack with the Slide instruction. Besides keeping the stack from growing, it is essential for our definition of the ArgChk instruction – if the local variables or parameters are not squeezed out, they are misinterpreted by the argument check as if they are extra parameters! This subtle requirement was first observed by Mountjoy in the context of the STG machine [13].

Here are some examples of functions that can be compiled with the current instruction set:

$$
\begin{array}{lcl}
id\ x & = & x \\
swap\ x\ f & = & f\ x \\
main & = & swap\ id\ id
\end{array}
$$

The final value of this program is the functional value id. With the compilation scheme from section 5 we get the following initial heap:

$$
\begin{array}{lcl}
id & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(1); \mathsf{Enter}) \\
swap & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(2); \mathsf{PushVar}(0); \mathsf{PushVar}(2); \mathsf{Slide}(2,2); \mathsf{Enter}) \\
main & \mapsto & \mathbf{instr}(\mathsf{ArgChk}(0); \mathsf{PushCode}(id); \mathsf{PushCode}(id); \\
 & & \quad \mathsf{PushCode}(swap); \mathsf{Enter})
\end{array}
$$

In the above program, it is clear that the *swap* function is called with sufficient arguments. This special case can be optimized with the EnterCode instruction. If a known function is called with sufficient arguments, the argument check of the called function can be skipped. This is called the 'direct entry point' convention in the STG machine [15]. The EnterCode instruction performs this optimization and enters a known function with sufficient arguments.

	Code	Stack	Heap
	$\mathsf{EnterCode}(f) : is$	st	$hp[f \mapsto \mathbf{instr}(\mathsf{ArgChk}(n) : is_f)]$
$\Longrightarrow$	is_f	st	hp

This instruction is essentially what a C compiler would use to implement tail calls: a jump! In contrast, the Enter instruction performs an indirect jump based on the kind of value that is entered – object oriented people would probably call this a 'virtual method tail-call'.

3.2 Local definitions

In this section we extend the instruction set to deal with local `let` and `letrec` bindings. A `let` binding is non-strict and delays evaluation of its right-hand side. The NewNap instruction allocates a (non-updateable) application node in the heap that contains the function to be called and its arguments.

	Code	Stack	Heap
	$\mathsf{NewNap}(n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{nap}(x_1, ..., x_n)]$

Now that we have introduced a new heap value, we need to extend the Enter instruction to deal with this new value. When the Enter instruction sees a (non-updateable) application node, the values are moved to the stack, and the top of the stack is entered again.

	Code	Stack	Heap
	$\mathsf{Enter} : is$	$p : st$	$hp[p \mapsto \mathbf{nap}(x_1, ..., x_n)]$
$\Longrightarrow$	$\mathsf{Enter} : is$	$x_1 : ... : x_n : st$	hp

3.3 Sharing

Although the NewNap instruction delays the evaluation of an expression, it is not lazy since it doesn't *share* the result. Take for example the following program:

$$main \;=\; \texttt{let}\; x = nfib\; 10\; \texttt{in}\; x + x$$

The expression *nfib* 10 is calculated twice if the `let` binding uses the NewNap instruction. To share the computation, we use graph reduction instead of simple tree reduction. The

NewAp instruction allocates an *updateable* application node in the heap. When this node is evaluated it is *updated* with its evaluated value, thus sharing the result of the computation. The Enter instruction puts a special update marker on the stack as a reminder that the node has to be updated with its evaluated value. The ArgChk instruction looks for these update frames – if there are insufficient arguments, the updateable application node is overwritten with a non-updateable one. At the moment, the only weak-head-normal-form values are functional values, but in the following sections we will see how updateable application nodes can also be overwritten by integers for example.

	Code	Stack	Heap
	$\mathsf{NewAp}(n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
	$\mathsf{Enter} : is$	$p : st$	$hp[p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	$\mathsf{Enter} : is$	$x_1 : ... : x_n : \mathbf{upd} : p : st$	hp
$n > m$	$\mathsf{ArgChk}(n) : is$	$f : x_1 : ... : x_m : \mathbf{upd} : p : st$	hp
$\Longrightarrow$	$\mathsf{ArgChk}(n) : is$	$f : x_1 : ... : x_m : st$	$hp \bullet [p \mapsto \mathbf{nap}(f, x_1, ..., x_m)]$

The argument check instruction suddenly looks expensive: previously, the number of arguments on the stack was equal to the depth of the stack, but now it seems the argument check has to search the stack for an update marker to determine the number of arguments! Fortunately, we can use some conventional compiler technology to overcome this inefficiency.

An implementation uses a frame pointer *fp* that points to the top frame on the stack. Now we also see why a marker takes up two stack slots: one slot is the real marker while the second is a link back to the previous stack frame. When a frame is pushed, the current frame pointer is saved in the marker and the frame pointer is updated to point to the new top frame. When a frame is popped, the frame pointer is updated with the back-link. The argument check can now substract the frame pointer from the stack pointer to obtain the number of arguments on the stack.

Not only local values should be shared but top-level values that take no arguments should be shared too. These values are called *constant applicative forms* or caf's. The initial heap contains an **ap** node for each caf. In the previous example, *main* takes no arguments and its initial heap nodes are:

$$
\begin{aligned}
main &\mapsto \mathbf{ap}(main') \\
main' &\mapsto \mathbf{instr}(\mathsf{ArgChk}(0); ...)
\end{aligned}
$$

Note that we can't generate an EnterCode instruction for function calls with zero arguments: their values point to either an **ap** node, or to any other value with which they are updated! We can certainly not jump to their instructions directly.

3.4 Recursive values

Recursive values are constructed in two steps: dummy values are allocated first, and later initialized. This allows the values to refer to each other. The AllocAp instruction allocates an application node without initializing its fields. Later the Pack(N)Ap instruction initializes the fields.

	Code	Stack	Heap
	$\mathsf{AllocAp}(n) : is$	st	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{inv}_n]$
$p = st[ofs - n]$	$\mathsf{PackAp}(ofs, n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$p = st[ofs - n]$	$\mathsf{PackNap}(ofs, n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{nap}(x_1, ..., x_n)]$

3.5 Algebraic data types

The LVM supports open ended algebraic data types. Constructor blocks are allocated just like application blocks. The AllocCon and PackCon are used to construct recursive constructor definitions.

	Code	Stack	Heap
	$\mathsf{NewCon}(t, n) : is$	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
	$\mathsf{AllocCon}(t, n) : is$	st	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{con}_t(...)]$
$p = st[ofs - n]$	$\mathsf{PackCon}(ofs, n) : is$	$x_1 : ... : x_n : st$	$hp[p \mapsto \mathbf{con}_t(...)]$
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$

When the Enter instruction sees a constructor value, it behaves like the Return instruction:

	Code	Stack	Heap
	$\mathsf{Enter} : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	$\mathsf{Return} : is$	$p : st$	hp
	$\mathsf{Return} : is$	$p : \mathbf{upd} : u : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	$\mathsf{Return} : is$	$p : st$	$hp \bullet [u \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
(1)	$\mathsf{Return} : is$	$p : []$	hp
$\Longrightarrow$	$[]$	$p : []$	hp

(1) Termination with a constructor value.

The Return instruction is used when the final value is known to be a constructor. Just like

the ArgChk instruction, the Return instruction looks for frames on the stack where an update frame causes the value to be updated with the constructor value. When the stack is empty, execution stops with the constructor value as the result.

Note that we have two instructions that look at the stack configuration, ArgChk and Return, and one instruction that looks at the type of an heap value, Enter.

3.6 Strict evaluation

Before describing how values of algebraic data types are matched, we first look at their evaluation. The `let!` binding strictly evaluates its right-hand side before evaluating the body. A *continuation* marker is pushed on the stack before the evaluation of the right-hand side. When the right-hand side is evaluated, execution resumes at the instructions in the continuation frame.

	Code	Stack	Heap
	$\mathsf{PushCont}(n) : is$	st	hp
$\Longrightarrow$	is	$\mathbf{cont} : \text{drop } n \; is : st$	hp
	$\mathsf{Return} : is$	$p : \mathbf{cont} : is' : st$	hp
$\Longrightarrow$	is'	$p : st$	hp
$n > m$	$\mathsf{ArgChk}(n) : is$	$f : x_1 : ... : x_m : \mathbf{cont} : is' : st$	hp
$\Longrightarrow$	is'	$p : st$	$hp \circ [p \mapsto \mathbf{nap}(f, x_1, ..., x_m)]$

Continuation frames resemble conventional calling conventions closely – a C compiler pushes a return address before calling a function. The STG machine [15] also uses plain return adresses instead of continuation frames. This seems impossible at first sight – The argument check builds a partial application block if there are insufficient arguments, which is checked by looking at the top frame. If only a plain return address is pushed instead of a frame, the number of arguments can't be determined as the return address is misinterpreted as just another argument! However, the STG machine only evaluates expressions that are scrutinized by a `case` expression. These expressions can never have a functional type, and the STG machine therefore never reaches this machine configuration – quite a subtly dependency on the host language. Indeed, the STG machine has special **seq** frames to support the polymorphic *seq* function of Haskell that *can* take functional values as its argument. The **seq** frames are treated like our continuation frames. As we always use explicit continuation frames, the *seq* function can be expressed directly in the λ_{lvm} language: *seq* x y = `let!` z = x `in` y

3.7 Matching

Once a value is evaluated to weak head normal form, it can be matched. The MatchCon instruction matches on algebraic datatypes.

	Code	Stack	Heap
$\exists i.\ t = t_i$	$\mathsf{MatchCon}(n, o, t_1, o_1, ..., t_n, o_n) : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_m)]$
$\Longrightarrow$	drop o_i is	$x_1 : ... : x_m : st$	hp
$\forall i.\ t \neq t_i$	$\mathsf{MatchCon}(n, o, t_1, o_1, ..., t_n, o_n) : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_m)]$
$\Longrightarrow$	drop o is	$p : st$	hp

The MatchCon instruction pops the argument p when a constructor matches. This opens up the possibility of an important optimization. Many constructors are allocated in the heap and immediately deconstructed with a match. The ReturnCon instruction tries to avoid many of these allocations. ReturnCon behaves denotationally exactly like a NewCon followed by a Return:

$$\mathsf{ReturnCon}(t, n) \ \Rightarrow\ \mathsf{NewCon}(t, n); \mathsf{Return}$$

However, there exist a more efficient implementation that sometimes avoids an expensive heap allocation. This is called the 'return in registers' convention in the STG machine [15].

	Code	Stack	Heap
(1)	$\mathsf{ReturnCon}(t, n) : is$	$x_1 : ... : x_n : \mathbf{cont} : is' : st$	hp
$\Longrightarrow$	drop o_i is''	$x_1 : ... : x_n : st$	hp
	$\mathsf{ReturnCon}(t, n) : is$	st	hp
$\Longrightarrow$	$\mathsf{NewCon}(t, n) : \mathsf{Return} : is$	$x_1 : ... : x_n : st$	hp

(1) $is' = \mathsf{MatchCon}(n, o, t_1, o_1, ..., t_n, o_n) : is'' \wedge \exists\, i.\ t = t_i$

In the special but common case that a constructor returns immediately into a MatchCon instruction, the ReturnCon instruction avoids the allocation of the constructor in the heap. In all other cases, it behaves like a NewCon/Return pair. This happens for example when there is an update frame before the continuation or when the constructor is not immediately matched after being evaluated. The 'return in register' convention is no longer used in GHC as it lead to too much complexity in the generated code. For the LVM this doesn't seem the case and the LVM interpreter uses this optimization on every constructor that is returned.

3.8 Synchronous exceptions

Any robust programming language needs to handle exceptional situations. The LVM instruction set supports exception handling at a fundamental level for two reasons. The first reason is efficiency – since exceptional situations are exceptional, normal execution shouldn't be penalized. Furthermore, LVM instructions, like division, can raise exceptions themselves and thus, the LVM needs a standard mechanism for raising exceptions.

The Catch instruction installs an exception handler. The instruction pushes a **catch** frame on the stack. When an exception is raised, execution is continued at the exception handler. When no exception is raised, the **catch** frame is simply ignored by other instructions that look for stack frames, i.e. ArgChk and Return.

	Code	Stack	Heap
	Catch : is	$h : st$	hp
$\Longrightarrow$	is	**catch** : $h : st$	hp
$n > m$	ArgChk(n) : is	$x_1 : ... : x_m$: **catch** : $h : st$	hp
$\Longrightarrow$	ArgChk(n) : is	$x_1 : ... : x_m : st$	hp
	Return : is	x : **catch** : $h : st$	hp
$\Longrightarrow$	Return : is	$x : st$	hp

Note that a **catch** frame should immediately follow another frame or the end of the stack. If this is not the case, the Return instruction could end up in an undefined configuration. In practice, an implementation can actually deal quite easily with **catch** frames that don't follow another frame directly. When the Return instruction pops the **catch** frame, it also pops any values up to the next frame on the stack.

An exception is raised explicitly with the Raise instruction. It unwinds the stack until it finds a **catch** frame. Execution is continued at the exception handler with the exception as its argument.

	Code	Stack	Heap
	Raise : is	x : **catch** : $h : st$	hp
$\Longrightarrow$	Enter : is	$h : x : st$	hp
(1)	Raise : is	x : []	hp
$\Longrightarrow$	[]	x : []	hp
	Raise : is	x : **upd** : $p : st$	hp
$\Longrightarrow$	Raise : is	$x : st$	$hp \bullet [p \mapsto \textbf{raise}(x)]$
	Raise : is	x : **cont** : $is' : st$	hp
$\Longrightarrow$	Raise : is	$x : st$	hp
	Raise : is	$x : y : st$	hp
$\Longrightarrow$	Raise : is	$x : st$	hp

(1) Termination with an exceptional value.

Again, we assume that there is another frame immediately following the Catch frame. Otherwise, the Raise instruction has to pop any values following the Catch frame to prevent that they are treated as extra arguments by the Enter instruction. Note that having explicit continuation frames helps out in practice to find more information about the exception at runtime. In our LVM implementation we have special functions that inspect the stack when an exception occurs – the update and continuation markers give the execution trace that lead to the exception. This proved especially useful in the Helium compiler that is used mainly for educational purposes where good error messages are highly important.

When the Raise instruction encounters an update frame it updates the value with a **raise** block – indeed, if a value raises an exception once, it will always raise that exception when evaluated, and should be updated with that exception. When a **raise** block is entered, it raises the exception again.

	Code	Stack	Heap
	Enter : is	$p : st$	$hp[p \mapsto \mathbf{raise}(x)]$
$\Longrightarrow$	Raise : is	$x : st$	hp

An example of an exceptional situation is a stack overflow. In the LVM the ArgChk instruction conservatively checks for thousand available stack slots. If there are fewer, a stack overflow exception is raised.

	Code	Stack	Heap
$free(st) < 1000$	ArgChk(n) : is	st	hp
$\Longrightarrow$	Raise : is	$p : st$	$hp \circ [p \mapsto stackoverflow]$

The *stackoverflow* heap block is just a constructor of the predefined *Exception* data type and contains for example the source and line number of the function where the stack overflowed. Together with the execution trace, this normally pins down an unbounded recursion.

3.9 Blackholing

Certain forms of infinite loops can be detected at runtime. In particular, if we enter an updateable application node, we should not re-enter that node again during the update. To prevent this kind of infinite loop, we can overwrite an application node with a **raise** node when we enter it. When the value is finally updated, the **raise** node is overwritten again with the computed value. Whenever the value is re-entered during the update, a *black hole* exception is raised automatically. Here is the refined Enter rule for application nodes.

	Code	Stack	Heap
	Enter : is	$p : st$	$hp[p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	Enter : is	$x_1 : ... : x_n : \mathbf{upd} : p : st$	$hp \bullet [p \mapsto \mathbf{raise}(blackhole)]$

However, it is fairly expensive to overwrite all application nodes whenever they are entered. A more efficient technique is to delay the overwrite, called lazy blackholing. With this technique, execution is sometimes stopped to do lazy blackholing, where every value in an update frame on the stack is overwritten with a blackhole. Since this kind of infinite loop always grows the stack, a good moment to do this is when the stack needs to be extended, but it can also be done during garbage collection or when a thread yields. We can describe this technique formally by a generic rule that allows us to execute a Blackhole instruction at any time. This instruction saves the current stack pointer and then walks the stack, updating any update frames with a blackhole.

	Code	Stack	Heap
	is	st	hp
$\Longrightarrow$	$\mathsf{Blackhole}(st) : is$	st	hp
	$\mathsf{Blackhole}(st) : is$	[]	hp
$\Longrightarrow$	is	st	hp
	$\mathsf{Blackhole}(st') : is$	$\mathbf{upd} : p : st$	hp
$\Longrightarrow$	$\mathsf{Blackhole}(st') : is$	st	$hp \bullet [p \mapsto \mathbf{raise}(blackhole)]$
	$\mathsf{Blackhole}(st') : is$	$\_ : st$	hp
$\Longrightarrow$	$\mathsf{Blackhole}(st') : is$	st	hp

3.10 Garbage collection

Another generic rule that can be applied at any time is the garbage collection rule. This rule models a garbage collector as part of the abstract machine.

	Code	Stack	Heap
(1)	is	st	hp
$\Longrightarrow$	is	st	hp'

(1) Where hp' is constrained to the *reachable* pointers:
$hp' = [\, p \mapsto x \mid p \mapsto x \in hp \ \wedge \ p \in reachable(st, hp) \,]$

The *reachable(st,hp)* predicate returns all pointers that can be reached from the stack st in the heap hp. Note that the reachable set includes all pointers can potentially be used later, and it is a superset of the *live* pointers that encompasses the set the pointers that are actually used later on. As such it is a conservative estimate of liveness.

However, we should always try to keep the reachable set as small as possible to avoid space leaks. For one thing, we can see that it is actually a good strategy to perform lazy blackholing just before a garbage collection as it makes the reachable set potentially smaller. This was first observed by Jones [8]. Other techniques are stack stubbing where we overwrite values in the stack with dummy values if we can statically determine that the values are never referenced again. This happens often in alternatives of a `match` statement:

	Code	Stack	Heap
	$\mathsf{Stub}(n) : is$	$x_0 : ... : x_{n-1} : st$	hp
$\Longrightarrow$	is	$x_0 : ... : inv : st$	hp

The garbage collection rule is also essential to prove the correctness of the rewrite rules presented in the next section. As an illustrative example, we show that the important rule for avoiding allocation of application nodes is correct:

$$\mathsf{NewAp}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \Rightarrow \mathsf{Slide}(n, m); \mathsf{Enter}$$

We can prove that this transformation is correct by showing that it leads to the same machine configuration at runtime. Here, we need the garbage collection rule to discard the intermediate application node.

	Code	Stack	Heap
	$\mathsf{NewAp}(n); \mathsf{Slide}(1, m); \mathsf{Enter}$	$x_1 : ... : x_n : ... : x_{n+m} : st$	hp
$\Longrightarrow$	$\mathsf{Slide}(1, m); \mathsf{Enter}$	$p : ... : x_{n+m} : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	Enter	$p : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
$\Longrightarrow$	Enter	$x_1 : ... : x_n : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
	{garbage collection}		
$\Longrightarrow$	Enter	$x_1 : ... : x_n : st$	hp
$\Longleftarrow$	$\mathsf{Slide}(n, m); \mathsf{Enter}$	$x_1 : ... : x_n : ... : x_{n+m} : st$	hp

4 The LVM language

Now that we have described the instruction set in detail, it is time to look more closely at how we can map a functional language onto these instruction. We define a low level language called λ_{lvm}, that closely relates to the LVM instructions. The abstract syntax for λ_{lvm} is given in figure 3.10. Allthough the form of expressions is restrictive, any enriched lambda calculus expression can be translated into a λ_{lvm} expression.

Just like the STG language [15], we attach an operational reading to λ_{lvm}: `let` and `letrec` bind expressions to variables, `let!` evaluates expressions to weak head normal form and `match` distinguishes evaluated values.

The λ_{lvm} language does not contain lambda-expressions or local function definitions – we assume that all functions have been lambda-lifted to toplevel [7]. This means that λ_{lvm} functions contain no free variables and a program consists just of a set of combinator definitions.

The `let!` expression is a strict version of `let`. It evaluates its right hand side to weak-head-normal-form before evaluating the body of the expression. The usual `case` expression of lazy languages is easily translated into a `let!` and `match` pair:

`case` *e* `of` *alts*
$\Rightarrow$
`let!` *x* `=` *e* `in match` *x* `with` *alts*

The `let!` binding is also used to translate strict languages to λ_{lvm}. The `letrec` binding of O'Caml and ML can only be used with recursive *functions*, which are lifted to toplevel, and present no problem.

4.0.1 Primitive expressions

Primitive expressions are functions that are not expressed in λ_{lvm} itself. They may consist of instructions, like integer addition, statically linked functionality like the `sin` function, or user imported foreign functions. All these variants are accomodated with the *prim* expression. This has proved very convenient in the implementation of the compiler, as it can treat these expressions uniformly up to code generation time.

Separate primitive declarations describe the different kinds. The syntax is exactly the same as foreign import declarations – here are some examples:

```
foreign instruction  "AddInt" addInt :: Int -> Int -> Int
foreign import       "sin"    sin    :: Double -> Double
```

Program	*program*	→	$\{$*top* $\{$*var*$\}^*$ `=` *expr*`;`$\}^*$
Expression	*expr*	→	`let!` *var* `=` *expr* `in` *expr*
		|	`match` *var* `with` `{` $\{$*pat* `->` *expr*`;`$\}^+$ `}`
		|	$prim^n$ $\{$*atom*$\}^n$
		|	*let* `in` *expr*
		|	*atom*
Let	*let*	→	`letrec` `{` $\{$*var* `=` *atom*`;`$\}^+$ `}`
		|	`let` *var* `=` *atom*
Atomic	*atom*	→	*let* `in` *atom*
		|	*id* $\{$*atom*$\}^*$
		|	con_t^n $\{$*atom*$\}^n$
		|	*literal*
Pattern	*pat*	→	*var*
		|	con_t^n $\{$*var*$\}^n$
		|	*literal*
Literal	*literal*	→	*int* | *float* | *bytes*
Identifier	*id*	→	*var* | *top*
Variable	*var*	→	local identifier (x)
Global	*top*	→	top level identifier (f)
Constructor	con_t^n	→	constructor with tag t and arity n
Primitive	$prim^n$	→	instruction or foreign function of arity n
Integer	*int*	→	integer (i)
Float	*float*	→	floating point number
Bytes	*bytes*	→	a sequence of bytes (packed string)
Notation	$\{p\}^*$	→	zero or more p
	$\{p\}^+$	→	one or more p
	$\{p\}^n$	→	exactly n occurrences of p

Figure 2: Abstract syntax of the λ_{lvm} language

Note that instructions are also described as foreign functions – they just have an extremely efficient calling convention and encoding!

4.0.2 Atomic expressions

The distinction between atomic and normal expressions is more than a syntactic convenience. During execution, the instructions that are generated for an atomic expression will always succeed and terminate[2]. In contrast, a `let!`, `match` or *prim* expression can raise an exception or go into an infinite loop. This is the reason why `let` expressions can only contain atomic expressions on their right-hand side. In contrast with the STG language, `let` expressions can contain nested `let` expressions on their right-hand side (as they are also atomic).

The STG paper [15] recommends special compilation techniques to avoid the creation of nested `let` bindings. With the LVM language, this can be avoided as nested `let` bindings can be compiled directly. Consider the following Haskell expression:

$$f\ =\ \texttt{let}\ x\ =\ [1,2]\ \texttt{in}\ e$$

This is translated to:

$$f\ =\ \texttt{let}\ x\ =\ (\texttt{let}\ y\ =\ \mathit{Cons}\ 2\ \mathit{Nil}\ \texttt{in}\ \mathit{Cons}\ 1\ y)\ \texttt{in}\ e$$

The STG machine can not deal with nested `let` bindings and will implicitly lift it to top-level, as in:

$$f_y\ =\ \mathit{Cons}\ 2\ \mathit{Nil}$$
$$f\ =\ \texttt{let}\ x\ =\ \mathit{Cons}\ 1\ f_y\ \texttt{in}\ e$$

However, it is hard to garbage collect a top level binding without arguments and it is not recommended to lift bindings to top level in general [20]. The translation recommended in the STG paper therefore, is to float the `let` binding one level up:

$$f\ =\ \texttt{let}\ y\ =\ \mathit{Cons}\ 2\ \mathit{Nil}\ \texttt{in}\ \texttt{let}\ x\ =\ \mathit{Cons}\ 1\ y\ \texttt{in}\ e$$

Both programs are denotationally equivalent but operationally different. Under the compilation scheme presented in the next section, the first program slides out the y value from the stack and therefore uses slightly less stack space with slightly more work. In contrast to the STG machine, both programs will construct the *Cons* 2 *Nil* node, even when x is never demanded. Since everything in the LVM language is explicit and has a formal operational reading, we are able to explicitly express all three variants and reason about their operational behaviour.

[2] Modulo fatal situations like heap exhaustion.

4.0.3 Strictness and speculative evaluation

In general, we can not float other constructs like `let!` or `match` since they might fail or perform an unbounded amount of computation. It is possible when a strictness analyser determines that the value is demanded later, but in that case it is easier to transform the `let` binding into a `let!` binding, which *can* have full expressions at its right-hand side.

If the strictness analyser can not prove that a value is demanded but if we are reasonably sure that the expression uses a bounded amount of computation, we could *speculatively* evaluate the expression. The value is computed eagerly but if it fails or uses too much resources, in terms of time or space, it is suspended. Currently, this is still an area of research [16] but we plan to add the atomic `let$` construct for speculative bindings. Again, the operational semantics described later in this chapter allows us to reason very specifically about the operational behaviour of eager evaluation.

4.1 Translating $\lambda_{\mathbf{core}}$ to $\lambda_{\mathbf{lvm}}$

It is convenient to use an intermediate language that is less restrictive than λ_{lvm} in a compiler. We define λ_{core} as an enriched lambda calculus with lambda expressions, free variables, no distinction between atomic expresions and normal expression, binary application, and unsaturated constructors and primitives.

The λ_{core} language can be mapped to the λ_{lvm} by applying the following transformations:

- Replace binary application with vector application.

  $(... ((id\ e_1)\ e_2) ...)\ e_n \quad \Rightarrow \quad id\ e_1 ...\ e_n$

- Saturate all applications to constructors and primitives.

  $con_t^n\ e_1 ...\ e_m \qquad \mid (m < n) \Rightarrow \quad \backslash x_{(m+1)} ...\ x_n\ .\ con_t^n\ e_1 ...\ e_m\ x_{(m+1)} ...\ x_n$

- Introduce a `let` expression for all anonymous lambda expressions.

  $\backslash x_1 ...\ x_n\ \texttt{.}\ e \quad \Rightarrow \quad \texttt{let}\ x\ x_1 ...\ x_n\ =\ e\ \texttt{in}\ x$

- Introduce a `let` expression for all non-atomic arguments.

  $e\ (\texttt{match}\ x\ \texttt{with}\ alts) \quad \Rightarrow \quad \texttt{let}\ y\ =\ (\texttt{match}\ x\ \texttt{with}\ alts)\ \texttt{in}\ e\ y$

- Introduce a `let` expression for all applications to terms that are not variables or constructors:

  $e\ x_1 \ldots x_n \quad \Rightarrow \quad \texttt{let}\ x\ =\ e\ \texttt{in}\ x\ x_1 \ldots x_n$

- Pass all free variables in non-atomic expressions as explicit arguments. This corresponds essentially to lambda-lifting [7, 18, 4], and leads to an environment-less machine.

  $f\ x\ =\ \texttt{let}\ y\ =\ (\texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z)\ \texttt{in}\ y$
  $\Rightarrow$
  $f\ x\ =\ \texttt{let}\ y\ x\ =\ (\texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z)\ \texttt{in}\ y\ x$

- Lift all local functions and non-atomic right-hand sides of `let` bindings to top-level.

  $f\ x\ =\ \texttt{let}\ y\ x\ =\ (\texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z)\ \texttt{in}\ y\ x$
  $\Rightarrow$
  $f_y\ x\ =\ \texttt{let!}\ z\ =\ 1/x\ \texttt{in}\ z$
  $f\ x\ =\ f_y\ x$

5 Compilation scheme

The compilation scheme translates λ_{lvm} into LVM instructions. In order to make the translation as clear as possible, the compilation scheme uses a few pseudo instructions to delay offset computations. This allows us to move the complexity of computing stack offsets of local variables to a separate *resolve* phase. The pseudo instructions are:

- $\mathsf{Param}(x)$ declares a local variable x that resides on the stack as an argument. This instruction allows the *resolve* phase to calculate the correct stack offset for x.
- $\mathsf{Var}(x)$ declares a local variable x that is bound to the current top of the stack.
- $\mathsf{Eval}(is)$. After executing instructions is, execution is continued at the next instruction. It is translated during code generation into $(\mathsf{PushCont}(ofs); is)$. Eval is introduced to delay the computation of the code offset ofs which is only known at code generation time.
- $\mathsf{Atom}(is)$. This instruction is used for translating expressions that result in a single value on the stack. During *resolve* it is translated into the instructions $(is; \mathsf{Slide}(1, m))$ where m intermediate values are slided out of the stack. Atom is used to delay the computation of the correct value for m which is only known during the *resolve* phase.
- $\mathsf{Init}(is)$. This instruction is used for translating the initialization of `letrec` bindings. The instructions is don't compute any value on the stack. During *resolve* it is translated into the instructions $(is; \mathsf{Slide}(0, m))$ where m intermediate values are slided out of the stack.

5.1 Program

A LVM program is translated with the $\mathcal{P}$ scheme.

$$\begin{aligned}&\mathcal{P}[\![\, f_1\ args_1 = e_1\,;\ \ldots;\ f_n\ args_n = e_n\,;\,]\!] \quad \Rightarrow\\ &\quad \textbf{let}\ index(f_i) = i\\ &\quad \textbf{let}\ arity(f_i) = |args_i|\\ &\quad \textbf{let}\ code(f_i) = \mathcal{T}[\![\, args_i = e_i\,]\!]\end{aligned}$$

The $\mathcal{P}$ scheme translates a program into three functions, *code* gives the code for a function, *arity* returns the number of parameters and *index* returns the index used in binary LVM files.

Each top level value is translated with the $\mathcal{T}$ scheme. The $\mathcal{T}$ scheme emits the pseudo instruction Param for each argument in order to resolve the stack offsets of each argument

during the *resolve* phase. As signified by the Atom instruction, a single value is computed on top of the stack that is subsequently entered by the Enter instruction.

$$\mathcal{T}[\![\, x_1 \ldots x_n = e \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{ArgChk}(n);\ \mathsf{Atom}(\mathsf{Param}(x_n);\ \ldots;\ \mathsf{Param}(x_1);\ \mathcal{E}[\![\, e \,]\!]);\ \mathsf{Enter}$$

Each top level value first checks the number of arguments with an argument check instruction. Quite often however, the compiler *can* determine if there are sufficient arguments when the function is called. The rewrite rules that are given later in this chapter will emit an EnterCode instruction if a function call is saturated. This instruction enters a function just beyond the ArgChk instruction since we know that the check will succeed. For this reason, every supercombinator *always* has to start with the ArgChk instruction or otherwise the EnterCode instruction will enter the function at the wrong location!

5.2 Expressions

Expressions are translated with the $\mathcal{E}$ scheme.

$$\mathcal{E}[\![\, let\ \texttt{in}\ e \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{L}[\![\, let \,]\!]; \mathcal{E}[\![\, e \,]\!]$$
$$\mathcal{E}[\![\, \texttt{let!}\ x = e\ \texttt{in}\ e' \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{Eval}(\mathsf{Atom}(\mathcal{E}[\![\, e \,]\!]);\ \mathsf{Enter});\ \mathsf{Var}(x);\ \mathcal{E}[\![\, e' \,]\!]$$
$$\mathcal{E}[\![\, \texttt{match}\ x\ \texttt{with}\ \{\ alts\ \} \,]\!]$$
$$\quad \mathsf{PushVar}(x);\ \mathcal{M}[\![\, alts \,]\!]$$
$$\mathcal{E}[\![\, prim^n\ a_1 \ldots a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{Call}(prim, n)$$
$$\mathcal{E}[\![\, a \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a \,]\!]$$

Let bindings are translated with the $\mathcal{L}$ scheme. A strict binding first evaluates its right hand side, leaving the result on the stack and continues with the evaluation of the body. A `match` statement pushes the value to be matched and uses the $\mathcal{M}$ scheme to translate the alternatives. A primitive call is handled by the Call instruction. Atomic expressions are translated with the $\mathcal{A}$ scheme.

5.3 Atomic expressions

The $\mathcal{A}$ scheme wraps the instructions in an Atom pseudo instruction to slide out any intermediate local variables arising from nested `let` expressions.

$$\mathcal{A}[\![\, a \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{Atom}(\mathcal{A}'[\![\, a \,]\!])$$

The $\mathcal{A}'$ scheme translates atomic expressions without entering them, effectively delaying their computation.

$$\mathcal{A}'[\![\, let\ \mathtt{in}\ a \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{L}[\![\, let \,]\!]; \mathcal{A}'[\![\, a \,]\!];$$
$$\mathcal{A}'[\![\, x\ a_1 \ldots a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{PushVar}(x);\ \mathsf{NewAp}(n+1);$$
$$\mathcal{A}'[\![\, f\ a_1 \ldots a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{PushCode}(f);\ \mathsf{NewAp}(n+1);$$
$$\mathcal{A}'[\![\, con_t^n\ a_1 \ldots a_n \,]\!] \quad \Rightarrow$$
$$\quad \mathcal{A}[\![\, a_n \,]\!];\ \ldots\ ;\ \mathcal{A}[\![\, a_1 \,]\!];\ \mathsf{NewCon}(t, n);$$
$$\mathcal{A}'[\![\, i \,]\!] \quad \Rightarrow$$
$$\quad \mathsf{PushInt}(i);$$

Note that this simple translation scheme is quite inefficient – it allocates an application node for every function call. Take for example the following expression:

$$swap\ f\ x\ y\ =\ f\ y\ x$$

Using the simple translation scheme, *swap* is translated into:

```
ArgChk(3); Atom(
  Param(y); Param(x); Param(f);
  Atom(PushVar(x); NewAp(1));
  Atom(PushVar(y); NewAp(1));
  Atom(PushVar(f); NewAp(1));
  NewAp(3))
Enter
```

After *resolve*, this instruction stream becomes:

```
ArgChk(3);
PushVar(1); NewAp(1); Slide(1, 0);
PushVar(3); NewAp(1); Slide(1, 0);
PushVar(2); NewAp(1); Slide(1, 0);
NewAp(3); Slide(1, 3);
Enter
```

Instead of just pushing the arguments on the stack and entering the function f, the code first builds an application node with application nodes for each variable, which is subsequently

entered, unpacked and, only than, the function f is entered! Fortunately, we can use some simple rewrite rules on the instruction stream to remove these inefficiencies. Using the rewrite rules from section 5.8, the instruction stream becomes much more efficient:

$$\mathsf{ArgChk}(3); \mathsf{PushVar}(1); \mathsf{PushVar}(3); \mathsf{PushVar}(2); \mathsf{Slide}(3,3); \mathsf{Enter}$$

We made the compilation scheme as simple and straightforward as possible and let the compiler do its optimizations on the LVM language and the instruction streams. It is quite easy to prove that transformations on the LVM language and instruction stream are correct. For example, the above transformation is simply a matter of applying the operational semantics described in section 3. In contrast, proving that the compilation scheme is correct is much harder – we have to show a correspondence between the operational semantics of the LVM language and the translated instructions. By making the compilation scheme naive, we hope that it becomes at least 'obviously' correct.

5.4 Let expressions

$$\begin{array}{l}
\mathcal{L}[\![\, \texttt{let}\ x = a \,]\!] \quad \Rightarrow \\
\qquad \mathcal{A}[\![\, a \,]\!]; \mathsf{Var}(x) \\
\mathcal{L}[\![\, \texttt{letrec}\ \{\ x_1 = a_1;\ ...;\ x_n = a_n;\ \}\,]\!] \quad \Rightarrow \\
\qquad \mathsf{Atom}(\mathcal{U}[\![\, a_1 \,]\!]);\ \mathsf{Var}(x_1);\ ...;\ \mathsf{Atom}(\mathcal{U}[\![\, a_n \,]\!]);\ \mathsf{Var}(x_n); \\
\qquad \mathsf{Init}(\mathcal{I}[\![\, x_1 = a_1 \,]\!]);\ ...;\ \mathsf{Init}(\mathcal{I}[\![\, x_n = a_n \,]\!])
\end{array}$$

The rule for `letrec` first allocates uninitialized values for its bindings using the $\mathcal{U}$ scheme and binds the stack slots to its local variables using the Var pseudo instruction. Later, the values are initialized using the $\mathcal{I}$ scheme. The rule for `let` is not concerned with recursive bindings and immediately allocates a value.

The $\mathcal{U}$ scheme allocates an uninitialized application- or constructor node that later initialized. This allows the different bindings in a `letrec` expression to refer to each other.

$$\begin{array}{l}
\mathcal{U}[\![\, \mathit{let}\ \texttt{in}\ a \,]\!] \quad \Rightarrow \\
\qquad \mathcal{U}[\![\, a \,]\!] \\
\mathcal{U}[\![\, \mathit{id}\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow \\
\qquad \mathsf{AllocAp}(n+1); \\
\mathcal{U}[\![\, \mathit{con}_t^n\ a_1\ ...\ a_n \,]\!] \quad \Rightarrow \\
\qquad \mathsf{AllocCon}(t,n);
\end{array}$$

Later, the $\mathcal{I}$ scheme is used to initialize each node with the proper values.

$$\mathcal{I}[\![\, x = \mathit{let}\ \texttt{in}\ a \,]\!] \quad \Rightarrow$$

$$
\begin{array}{l}
\quad \mathcal{L}[\![\ let\ ]\!]; \mathcal{I}[\![\ x = a\ ]\!] \\
\mathcal{I}[\![\ x = x'\ a_1\ ...\ a_n\ ]\!] \quad \Rightarrow \\
\quad \mathcal{A}[\![\ a_n\ ]\!];\ \ldots;\ \mathcal{A}[\![\ a_1\ ]\!];\ \mathsf{PushVar}(x');\ \mathsf{PackAp}(x, n+1); \\
\mathcal{I}[\![\ x = f\ a_1\ ...\ a_n\ ]\!] \quad \Rightarrow \\
\quad \mathcal{A}[\![\ a_n\ ]\!];\ \ldots;\ \mathcal{A}[\![\ a_1\ ]\!];\ \mathsf{PushCode}(f);\ \mathsf{PackAp}(x, n+1); \\
\mathcal{I}[\![\ x = con_t^n\ a_1\ ...\ a_n\ ]\!] \quad \Rightarrow \\
\quad \mathcal{A}[\![\ a_n\ ]\!];\ \ldots;\ \mathcal{A}[\![\ a_1\ ]\!];\ \mathsf{PackCon}(x, n);
\end{array}
$$

5.5 Matching

A `match` is translated with the $\mathcal{M}$ scheme.

$$
\begin{array}{l}
\mathcal{M}[\![\ pat_1\ \texttt{->}\ e_1\ \texttt{;}\ ...\texttt{;}\ pat_n\ \texttt{->}\ e_n\ ]\!] \quad \mid \exists\, i.\ pat_i \text{ is a constructor pattern} \ \Rightarrow \\
\quad \mathsf{MatchCon}(\mathcal{P}[\![\ pat_1\ \texttt{->}\ e_1\ ]\!],\ ...,\ \mathcal{P}[\![\ pat_n\ \texttt{->}\ e_n\ ]\!]) \\
\mathcal{M}[\![\ pat_1\ \texttt{->}\ e_1\ \texttt{;}\ ...\texttt{;}\ pat_n\ \texttt{->}\ e_n\ ]\!] \quad \mid \exists\, i.\ pat_i \text{ is an integer pattern} \ \Rightarrow \\
\quad \mathsf{MatchInt}(\mathcal{P}[\![\ pat_1\ \texttt{->}\ e_1\ ]\!],\ ...,\ \mathcal{P}[\![\ pat_n\ \texttt{->}\ e_n\ ]\!])
\end{array}
$$

The patterns result in a list of tuples, the first element containing the value to be matched and the second the instructions to be executed. The code generation phase will arrange the code correctly.

Each pattern is compiled with the $\mathcal{P}$ scheme.

$$
\begin{array}{l}
\mathcal{P}[\![\ con_t^n\ x_1\ ...\ x_n\ \texttt{->}\ e\ ]\!] \quad \Rightarrow \\
\quad \langle t,\ \mathsf{Atom}(\mathsf{Param}(x_n);\ ...;\ \mathsf{Param}(x_1);\ \mathcal{E}[\![\ e\ ]\!])\rangle \\
\mathcal{P}[\![\ i\ \texttt{->}\ e\ ]\!] \quad \Rightarrow \\
\quad \langle i,\ \mathsf{Atom}(\mathcal{E}[\![\ e\ ]\!])\rangle \\
\mathcal{P}[\![\ x\ \texttt{->}\ e\ ]\!] \quad \Rightarrow \\
\quad \langle x,\ \mathsf{Atom}(\mathsf{Param}(x);\ \mathcal{E}[\![\ e\ ]\!])\rangle
\end{array}
$$

Note that the Param instruction is used to bind the values of a matched constructor. As we saw in section 3.7, it is quite important that a `match` automatically unpacks the constructor as it allows us to return the constructor on the stack sometimes without allocation (the *return in registers* convention).

5.6 Optimized schemes

Although we tried to make the compilation scheme as straightforward as possible, some transformations are hard to apply during a different phase. For example, the following rule discards a stack push of a value that has just been evaluated to be matched. However, it can only do so if the bound variable is not used in the alternatives. This is a good example of where we need both high level information (is x used in the alternatives?) and low-level information (we can skip a PushVar instruction).

$$\mathcal{E}[\![\,\texttt{let!}\ x = e\ \texttt{in}\ \texttt{match}\ x\ \texttt{with}\ alts\,]\!] \quad \mid x \notin fv(alts) \Rightarrow$$
$$\mathsf{Eval}(\mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Enter});\ \mathcal{M}[\![\,alts\,]\!]$$

Another important optimization removes superfluous continuation frames. This is especially important for efficient arithmetic. For example:

$$\begin{aligned} discriminant\ a\ b\ c = {} & \texttt{let!}\ ac = a * c\ \texttt{in} \\ & \texttt{let!}\ ac4 = 4 * ac\ \texttt{in} \\ & \texttt{let!}\ b2 = b * b\ \texttt{in}\ b2 + ac4 \end{aligned}$$

If we suppose that a, b and c are already in weak head normal form and that $*$ and $+$ expand to the primitive MulInt and AddInt instructions, we would get the following instruction sequence (after some rewriting):

$$\begin{aligned} & \mathsf{ArgChk}(3) \\ & \mathsf{Eval}(\mathsf{PushVar}(c); \mathsf{PushVar}(a); \mathsf{MulInt}; \mathsf{Slide}(1,0); \mathsf{Enter}) \\ & \mathsf{Var}(ac); \\ & \mathsf{Eval}(\mathsf{PushVar}(ac); \mathsf{PushInt}(4); \mathsf{MulInt}; \mathsf{Slide}(1,0); \mathsf{Enter}) \\ & \ldots \end{aligned}$$

However, the result of MulInt is already in weak head normal form and entering the value will only return immediately to the continuation frame pushed by Eval. A much better instruction sequence is possible:

$$\begin{aligned} & \mathsf{ArgChk}(3) \\ & \mathsf{PushVar}(c); \mathsf{PushVar}(a); \mathsf{MulInt}; \\ & \mathsf{Var}(ac); \\ & \mathsf{PushVar}(ac); \mathsf{PushInt}(4); \mathsf{MulInt}; \\ & \ldots \end{aligned}$$

In general, when an expression is evaluated that is already in weak head normal form, we don't need to evaluate it again.

$$\mathcal{E}[\![\,\texttt{let!}\ x = e\ \texttt{in}\ e'\,]\!] \quad \mid whnf(e) \Rightarrow$$
$$\mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Var}(x); \mathcal{E}[\![\,e'\,]\!]$$

The *whnf* predicate determines whether the expression e puts a value in weak head normal form on the stack. We assume that every primitive operation *prim* has an associated type t which is annotated with a (`!`) when the result is always in weak head normal form. The function *whnf* can be conservatively defined as:

$$
\begin{array}{lll}
whnf\ (let\ \texttt{in}\ e) & = & whnf(e) \\
whnf\ (\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ e') & = & whnf(e') \\
whnf\ (\texttt{match}\ x\ \texttt{with}\ alts) & = & whnfAlts\ alts \\
whnf\ (x\ a_1\ ...\ a_n) & = & False \\
whnf\ (con_t^n\ a_1\ ...\ a_n) & = & True \\
whnf\ (i) & = & True \\
whnf\ (prim^n\ a_1\ ...\ a_n) & \mid\ prim\ ::\ t_1\ \rightarrow\ ...\ \rightarrow\ t_n\ \rightarrow\ t\texttt{!} & = True \\
 & \mid\ otherwise & = False
\end{array}
$$

$$
\begin{array}{lll}
whnfAlts\ (\texttt{\{}\ alt_1\ \texttt{;}\ ...\ \texttt{;}\ alt_n\ \texttt{\}}) & = & whnfAlt(alt_1)\ \&\ ...\ \&\ whnfAlt(alt_n) \\
whnfAlt\ (pat\ \texttt{->}\ e) & = & whnf(e)
\end{array}
$$

5.7 Resolve stack offsets

The *resolve* phase resolves all offsets of local variables and removes the Param, Var, Init and Atom pseudo instructions. Guided by these pseudo instructions, the algorithm simulates the stack and calculates the correct offsets for each variable.

5.7.1 The resolve monad

We use a monadic formulation of the algorithm. The monad type is defined as:

$$\textbf{newtype}\ M\ a\ =\ M\ (\langle Env,\ Depth\rangle\ \rightarrow\ \langle a,\ Depth\rangle)$$

The monad uses an enviroment, *Env* that maintains the mapping from local variables to their stack location. The monad also has a state *Depth* that contains the current depth of the (simulated) stack.

The monadic functions are defined as usual [5]:

$$
\begin{array}{l}
return\ x\ = \\
\quad M\ (\backslash\langle env,\ depth\rangle \rightarrow \langle x,\ depth\rangle) \\
(M\ m)\ \texttt{>>=}\ f\ = \\
\quad M\ (\backslash\langle env,\ depth\rangle \rightarrow \\
\quad\ \textbf{let}\ \langle x,\ depth'\rangle\ =\ m\ \langle env,\ depth\rangle \\
\quad\quad\ (M\ fm)\ =\ f\ x \\
\quad\ \textbf{in}\ fm\ \langle env,\ depth'\rangle
\end{array}
$$

The *push* and *pop* non-proper morphisms simulate stack movements.

$$
\begin{aligned}
&\mathit{pop}\ n\ = \\
&\qquad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow \langle (),\ \mathit{depth} - n\rangle) \\
&\mathit{push}\ n\ = \\
&\qquad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow \langle (),\ \mathit{depth} + n\rangle)
\end{aligned}
$$

The *depth* function returns the current stack depth.

$$
\begin{aligned}
&\mathit{depth}\ = \\
&\qquad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow \langle \mathit{depth},\ \mathit{depth}\rangle\ )
\end{aligned}
$$

Variables are bound using *bind* and the function *offset* returns their current offset relative to the top of the stack.

$$
\begin{aligned}
&\mathit{offset}\ x\ = \\
&\qquad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow \langle \mathit{depth}\ -\ \mathit{env}[x],\ \mathit{depth}\rangle) \\
&\mathit{bind}\ x\ (M\ m)\ = \\
&\qquad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow m\ \langle \mathit{env}\ \oplus\ \{\ x\ \mapsto\ \mathit{depth}\},\ \mathit{depth}\rangle)
\end{aligned}
$$

Addressing variables relative to the top of the stack removes the need for a separate *base pointer*, which is still used in some C compilers to aid debuggers.

5.7.2 The algorithm

An instruction stream is resolved by the *resolves* function.

$$
\begin{aligned}
&\mathit{resolves}\ (\mathsf{Param}(x) : \mathit{instrs})\ = \\
&\qquad \mathbf{do}\{\ \mathit{push}\ 1;\ \mathit{bind}\ x\ (\mathit{resolves}\ \mathit{instrs})\ \} \\
&\mathit{resolves}\ (\mathsf{Var}(x) : \mathit{instrs})\ = \\
&\qquad \mathit{bind}\ x\ (\mathbf{do}\{\ \mathit{resolves}\ \mathit{instrs}\ \}) \\
&\mathit{resolves}\ (\mathit{instr} : \mathit{instrs})\ = \\
&\qquad \mathbf{do}\{\ \mathit{is} \leftarrow \mathit{resolve}\ \mathit{instr} \\
&\qquad\qquad \mathit{iss} \leftarrow \mathit{resolves}\ \mathit{instrs} \\
&\qquad\qquad \mathit{return}\ (\mathit{is}\ +\!\!+\ \mathit{iss})\ \}
\end{aligned}
$$

Individual instructions are resolved by the *resolve* function. Note that we allow ourselves some freedom by reusing the PushVar instruction such that it can contain either an argument name or resolved stack offset.

$$
\begin{aligned}
&\mathit{resolve}\ \mathsf{PushVar}(x)\ = \\
&\qquad \mathbf{do}\{\ \mathit{ofs} \leftarrow \mathit{offset}\ x; \\
&\qquad\qquad \mathit{push}\ 1; \\
&\qquad\qquad \mathit{return}\ [\mathsf{PushVar}(\mathit{ofs})]\ \} \\
&\mathit{resolve}\ \mathsf{PackAp}(x, n)\ = \\
&\qquad \mathbf{do}\{\ \mathit{ofs} \leftarrow \mathit{offset}\ x; \\
&\qquad\qquad \mathit{pop}\ n;
\end{aligned}
$$

```
            return [PackAp(ofs, n)] }
resolve PackCon(x, n) =
      do{ ofs ← offset x;
            pop n;
            return [PackCon(ofs, n)] }
resolve Eval(is) =
      do{ push 3;
            is′ ← resolves is;
            pop 3;
            return [Eval(is′)] }
resolve Atom(is) =
      do{ resolveSlide 1 is }
resolve Init(is) =
      do{ resolveSlide 0 is }
resolve (MatchCon(alts)) =
      do{ pop 1;
            alts′ ← sequence (map resolveAlt alts);
            return [MatchCon(alts′)] }
resolve instr =
      do{ effect instr; return [instr] }
```

The *resolveSlide n is* function slides out any dead values on the stack, only preserving the top n stack values.

```
resolveSlide n is =
      do{ d0 ← depth;
            is′ ← resolves is;
            d1 ← depth;
            let m = d1 − d0 − n
            pop m;
            return (is′ ++ [Slide(n, m)]) }
```

Alternatives are resolved with *resolveAlt*. Note that every alternative should return with the same stack depth.

```
resolveAlt ⟨t, is⟩ =
      do{ is′ ← resolves is; return ⟨t, is⟩ }
```

Most instructions are not transformed but they do have an effect on the stack. The *effect* function simulates this effect in the resolve monad.

$$
\begin{aligned}
&\mathit{effect}\ \mathsf{PushCode}(f) &&= \mathit{push}\ 1\\
&\mathit{effect}\ \mathsf{AllocAp}(n) &&= \mathit{push}\ 1\\
&\mathit{effect}\ \mathsf{AllocCon}(t,n) &&= \mathit{push}\ 1\\
&\mathit{effect}\ \mathsf{NewAp}(n) &&= \mathbf{do}\{\ \mathit{pop}\ n;\ \mathit{push}\ 1\ \}\\
&\mathit{effect}\ \mathsf{NewCon}(t,n) &&= \mathbf{do}\{\ \mathit{pop}\ n;\ \mathit{push}\ 1\ \}\\
&\mathit{effect}\ \mathsf{AddInt} &&= \mathbf{do}\{\ \mathit{pop}\ 2;\ \mathit{push}\ 1\ \}\\
&... &&\\
&\mathit{effect}\ \mathit{instr} &&= \mathit{return}\ ()
\end{aligned}
$$

Using pseudo instructions together with this simple algorithm, we have cleanly separated stack offset resolving from the translation scheme from the LVM language to instructions.

5.8 Rewrite rules

The rewrite rules transform a sequence of instructions into a more efficient sequence of instructions with the same semantic effect. As described in section 5.3, the rewrite rules describe important optimizations since the compilation scheme is quite naïve.

There are two essential rewrite rules that push instructions following a match into the branches of the match. This is needed since the branches are not able to jump to those instructions. The transformation is safe, since every alternative leaves the stack at the same depth.

$$
\begin{aligned}
&\mathsf{MatchCon}(\mathit{alt}_1, ..., \mathit{alt}_n); \mathit{instrs} \quad \Rightarrow\\
&\quad \mathsf{MatchCon}(\mathit{alt}_1; \mathit{instrs}\ , ...,\ \mathit{alt}_n; \mathit{instrs})\\
&\mathsf{MatchInt}(\mathit{alt}_1, ..., \mathit{alt}_n); \mathit{instrs} \quad \Rightarrow\\
&\quad \mathsf{MatchInt}(\mathit{alt}_1; \mathit{instrs}\ , ...,\ \mathit{alt}_n; \mathit{instrs})
\end{aligned}
$$

This transformation duplicates code, but fortunately, the *instrs* are always a Slide followed by an Enter, as we can see from the translation schemes. Moreover, subsequent transformations are more effective when these instructions are directly in scope.

The first optimizing rules transform partial and saturated applications. The first rule emits NewNap instructions for a known partial application – this instruction will not push an expensive update frame. The second rule uses EnterCode for saturated applications to a known top level function. This instruction behaves just like Enter except that an implementation can safely skip the expensive argument check for the entered function (a *direct entry point*).

$$
\begin{aligned}
&\mathsf{PushCode}(f); \mathsf{NewAp}(n) \quad \mid \mathit{arity}(f) > (n-1) \Rightarrow\\
&\quad \mathsf{PushCode}(f); \mathsf{NewNap}(n)
\end{aligned}
$$

$$\mathsf{PushCode}(f); \mathsf{Slide}(n, m); \mathsf{Enter} \quad \mid \mathit{arity}(f) = (n-1) \,\&\, \mathit{arity}(f) \neq 0 \Rightarrow \\ \mathsf{Slide}(n-1, m); \mathsf{EnterCode}(f)$$

If an application node is entered immediately after building it, we can safely enter the application directly without building the application node at all!

$$\mathsf{NewAp}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \Rightarrow \\ \mathsf{Slide}(n, m); \mathsf{Enter}$$
$$\mathsf{NewNap}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \Rightarrow \\ \mathsf{Slide}(n, m); \mathsf{Enter}$$

These two simple rewrite rules remove the need for the usual translation schemes: one for expressions that will be entered and one for `let`-bound expressions [14].

The following rule moves the Slide instruction up in order to prevent space leaks while calling external functions.

$$\mathsf{Call}(\mathit{prim}, n); \mathsf{Slide}(1, m); \mathsf{Enter} \Rightarrow \\ \mathsf{Slide}(n, m); \mathsf{Call}(\mathit{prim}, n); \mathsf{Enter}$$

Expressions of the form (`let!` $x = e$ `in` x) lead to code that pushes variable x and subsequently discard the original binding. We can instead discard the push and leave the original binding in place.

$$\mathsf{PushVar}(0); \mathsf{Slide}(1, m) \quad \mid m \geq 1 \Rightarrow \\ \mathsf{Slide}(1, m-1)$$

The previous rule naturally generalizes to a sequence of n pushes:

$$\mathsf{PushVar}_1(n-1); ...; \mathsf{PushVar}_n(n-1); \mathsf{Slide}(n, m) \quad \mid m \geq n \Rightarrow \\ \mathsf{Slide}(n, m-n)$$

If a value is entered that is already in weak head normal form, we can directly use the Return instruction. We assume that all primitive functions have a type that ends with a (`!`) when they return a strict result. This is the case for many primitive operations and instructions.

$$\mathsf{Call}(\mathit{prim}, n); \mathsf{Enter} \quad \mid \mathit{prim} :: t_1 \rightarrow ... \rightarrow t_n \rightarrow t\,! \Rightarrow \\ \mathsf{Call}(\mathit{prim}, n); \mathsf{Return}$$

Commonly, a constructor or literal is returned. The LVM has the special ReturnCon and ReturnInt instructions that can potentially execute without extra heap allocation resulting from building a new constructor. Instead of building a new constructor that is immediately entered, the constructor is kept on the stack (see section 3.7). This is the 'return in registers' convention as described in the STG machine paper [15].

$$\mathsf{NewCon}(t, n); \mathsf{Slide}(1, m); \mathsf{Enter} \Rightarrow \\ \mathsf{Slide}(n, m); \mathsf{ReturnCon}(t, n)$$

$$\mathsf{PushInt}(i); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow$$
$$\quad \mathsf{Slide}(0, m); \mathsf{ReturnInt}(i)$$

An LVM interpreter can cheaply test a variable to see if it is already in a weak head normal form. The EvalVar instruction can use this in order to avoid creating a continuation frame that is immediately popped.

$$\mathsf{Eval}(\mathsf{PushVar}(ofs); \mathsf{Slide}(1, 0); \mathsf{Enter}) \quad \Rightarrow$$
$$\quad \mathsf{EvalVar}(ofs - 3)$$

Quite often, we can merge slides, arising from instructions pushed into an alternative.

$$\mathsf{Slide}(n0, m0); \mathsf{Slide}(n1, m1) \qquad \mid n1 \leq n0 \Rightarrow$$
$$\quad \mathsf{Slide}(n1, m0 + m1 - (n0 - n1))$$

The last rules deal with instructions that have no effect and primitive instructions. By treating instructions like AddInt as a primitive call, the compiler can be simplified since it doesn't need special code to deal with built-in operations. In a sense, these instructions are just like external calls except that they have a very efficient calling convention and encoding.

$$\mathsf{Call}(prim, n) \qquad \mid prim = \texttt{instr}\ instr :: t_1 \to ... \to t_n \to t \Rightarrow$$
$$\quad instr$$
$$\mathsf{NewAp}(n) \qquad \mid n \leq 1 \Rightarrow$$
$$\quad -$$
$$\mathsf{Slide}(n, 0) \quad \Rightarrow$$
$$\quad -$$

Together, the above rules have proven to be quite effective in optimizing the instructions generated by the naive translation scheme. Careful study of the generated code shows that hardly any improvements on this level are attainable. The simple translation scheme in combination with these rewrite rules also make the compiler much simpler. Furthermore, the rewrite rules even seem to perform *better* than optimized translation schemes as the rewrite rules sometimes find optimization opportunities between instructions that are unrelated at the language level.

5.9 Code generation

The code generation phase resolves the code offsets relative to program counter.

$$codegens\ is =$$
$$\quad concat\ (map\ codegen\ is)$$
$$codegen\ \mathsf{Eval}(is) =$$
$$\quad \textbf{let}\ is' = codegens\ is$$
$$\quad \textbf{in}\ [\mathsf{PushCont}(size\ is')] +\!\!+ is'$$
$$codegen\ \mathsf{PushCode}(f) =$$

```
        [PushCode(index(f))]
codegen EnterCode(f) =
        [EnterCode(index(f))]
codegen MatchCon(alts) =
        let iss  = map (codegen . snd) alts
            tags = map fst alts
            ofss = scanl (+) 0 (map size iss)
        in [MatchCon(length alts, 0, zip tags ofss)] ++ concat iss
codegen instr =
        [instr]
```

For simplicity, the rule for MatchCon(*alts*) assumes that there are no (default) variable patterns inside *alts*. The actual rule used in the compiler is inconvenient to formulate but still straightforward. After the code resolve phase, we are done and the compiler can write a binary LVM file that can be loaded by the interpreter.

5.10 More optimization: superfluous stack movements

An important optimization is to reduce the number of superfluous stack movements. Due to the close relation of λ_{lvm} with the LVM instruction set, it is possible to perform this optimization at the language level instead of at the instruction level.

As an example of unnecessary stack pushes we look at a definition of the S combinator.

```
combS f g x = let z = g x in f x z
```

After translating, resolving, and rewriting this program, it is compiled into:

```
ArgChk(3);
PushVar(2 (x)); PushVar(2 (g)); NewAp(2);
PushVar(0 (z)); PushVar(4 (x)); PushVar(3 (f));
Slide(3, 4); Enter
```

However, the variable z is pushed on the stack immediately after building it and later discarded with the Slide instruction. Better code can be obtained by inlining the definition of z.

```
combS f g x = f x (g x)
```

This program uses the application node immediately and discards the superfluous PushVar instruction.

```
ArgChk(3);
PushVar(2 (x)); PushVar(2 (g)); NewAp(2);
PushVar(3 (x)); PushVar(2 (f));
```

Slide$(3, 3)$; Enter

Et voilá, we can optimize stack movements (and remove dead variables) by using a standard inliner. The inliner for the LVM language can be much simpler than a full fledged inliner [19] since we will not instantiate across lambda expressions but only perform local substitutions. This property makes it also easier to analyse whether work or code is ever duplicated.

It is always beneficial to inline *trivial* expressions since they duplicate neither work, nor code. Trivial expressions consist of:

- literals (*literal*),
- variables (x),
- constructors with no arguments (con_t^0).

For other expressions, we need to determine how often the binder *occurs*. The occurrence analysis can be as simple as counting the number of syntactic occurrences. If code duplication is not perceived as a problem, we can refine the analysis by taking the maximum of the occurrences inside alternatives instead of the sum. If a binder occurs only once, we can safely inline it (since lambda expressions are not part of the LVM language). When a binder has no occurrences, the binding can be removed entirely.

5.10.1 Inlining strict bindings

We look again at the example program *discriminant* from section 5.6:

$$\begin{aligned} \mathit{discriminant}\ a\ b\ c = {} & \texttt{let!}\ ac = a * c\ \texttt{in} \\ & \texttt{let!}\ ac4 = 4 * ac\ \texttt{in} \\ & \texttt{let!}\ b2 = b * b\ \texttt{in}\ b2 + ac4 \end{aligned}$$

The optimized instruction sequence was:

ArgChk(3)
PushVar(c); PushVar(a); MulInt;
Var(ac);
PushVar(ac); PushInt(4); MulInt;
...

This example can be optimized a little bit more since it still pushes variable ac allthough it already resides on the stack. An optimal instruction sequence would be:

ArgChk(3)
PushVar(c); PushVar(a); MulInt;
Var(ac);

$\mathsf{PushInt}(4); \mathsf{MulInt};$
...

Unfortunately, our simple inliner will not inline the binding for *ac* since `let!` bindings can not be inlined in general. However, we can define some side conditions under which the inlining of `let!` bindings is possible.

First, we extend the grammar and allow `let!` expressions as atomic expressions – of course, this is in general unsafe and should only be used 'internally'. Together with the grammar extension, the translation scheme for atomic expressions is also extended:

$$\begin{array}{l}\mathcal{A}'[\![\,\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\,]\!] \qquad \mid \mathit{whnf}(e)\ \Rightarrow \\ \quad \mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Var}(x); \mathcal{A}'[\![\,e'\,]\!]; \\ \mathcal{A}'[\![\,\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\,]\!] \quad \Rightarrow \\ \quad \mathsf{Eval}(\mathsf{Atom}(\mathcal{E}[\![\,e\,]\!]); \mathsf{Enter}); \mathsf{Var}(x); \mathcal{A}'[\![\,e'\,]\!];\end{array}$$

When an evaluated expression is both *pure* and *total*, we can transform `let!` bindings into `let` bindings. The standard inliner can now inline `let!` expressions via those `let` bindings.

$$\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \qquad \mid \mathit{pure}(e)\ \&\ \mathit{total}(e)\ \Rightarrow \quad \texttt{let}\ x\ \texttt{=}\ (\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ x)\ \texttt{in}\ e'$$

The $\mathit{pure}(e)$ predicate ensures that the expression has no side effect and the $\mathit{total}(e)$ predicate ensures that the expression can not fail or loop. These conditions can probably only be approximated in practice but they can be determined exactly for many common primitive expressions like comparison and bitwise operations. However it fails for operations that can raise exceptions – like addition, multiplication and division. Note that the `let!` binding inside the `let` is still needed in order to emit an Eval instruction during compilation.

The above approach works for expressions that are both pure and total but many times we don't know enough about the expression to ensure those predicates. Other strict expressions can be inlined only if the following conditions hold:

1. the inliner never duplicates code (to avoid duplication of an impure expression).
2. the binding is used once.
3. the binding is used before any other primitive function, `let!`, or `match` construct.

The first two conditions are intrisic properties of the inliner. The last condition, is formalized with the *firstuse* predicate.

$$\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \qquad \mid \mathit{once}\ x\ e'\ \&\ \mathit{firstuse}\ x\ e'\ \Rightarrow \quad \texttt{let}\ x\ \texttt{=}\ (\texttt{let!}\ x\ \texttt{=}\ e\ \texttt{in}\ x)\ \texttt{in}\ e'$$

Note that the *firstuse* predicate is extremely dependent on the exact translation scheme that is used – after a strict binding is inlined as a `let`, we must be sure that no other strict binding gets inlined beyond the previous one.

The *firstuse* predicate is defined in terms of the *first* function that has as its second argument a possible continuation that starts out as *False*. As soon as a primitive operation, `let!`, or `match` is encountered, the continuation is set to *False* again to avoid inlining a binding beyond that construct.

$$
\begin{array}{ll}
\mathit{firstuse}\ x\ e & = \mathit{first}\ x\ \mathit{False}\ e \\
 & \\
\mathit{first}\ x\ c\ x & = \mathit{True} \\
\mathit{first}\ x\ c\ (y\ a_1\ ...\ a_n) & = \mathit{firsts}\ x\ c\ [y, a_1, ..., a_n] \\
\mathit{first}\ x\ c\ (con_t^n\ a_1\ ...\ a_n) & = \mathit{firsts}\ x\ c\ [a_1, ..., a_n] \\
\mathit{first}\ x\ c\ (prim^n\ a_1\ ...\ a_n) & = \mathit{firsts}\ x\ \mathit{False}\ [a_1, ..., a_n] \\
\mathit{first}\ x\ c\ (\texttt{match}\ y\ \texttt{with}\ \mathit{alts}) & = \mathit{False} \\
\mathit{first}\ x\ c\ (\texttt{let!}\ y\ \texttt{=}\ e\ \texttt{in}\ e') & = \mathit{first}\ x\ \mathit{False}\ e \\
\mathit{first}\ x\ c\ (\texttt{let}\ y\ \texttt{=}\ e\ \texttt{in}\ e') & = \mathit{firsts}\ x\ c\ [e, e'] \\
\mathit{first}\ x\ c\ (\texttt{letrec}\ \texttt{\{}\ y_1\ \texttt{=}\ e_1\ \texttt{;}\ ...\ \texttt{;}\ y_n\ \texttt{=}\ e_n\ \texttt{\}}\ \texttt{in}\ e') & = \mathit{firsts}\ x\ c\ [e_1, ..., e_n, e'] \\
\mathit{first}\ x\ c\ \mathit{other} & = c \\
 & \\
\mathit{firsts}\ x\ c\ es & = \mathit{foldl}\ (\mathit{first}\ x)\ c\ es
\end{array}
$$

In combination with the rewrite rules, the described optimizations remove all superfluous stack movements – there is no need for complex reorderings of bindings. Note that we could achieve this by distinguishing between normal expressions and atomic expressions, and allowing atomic expressions as arguments. This constrasts with the STG language for example, that only allows variables as arguments. In our case, λ_{lvm} directly reflects the capabilities of the abstract machine.

6 Assessment

We have implemented a LVM interpreter on top of the O'Caml runtime system [11], which is well known for its portability and the efficient bytecode interpreter. By taking advantage of this excellent system, we were able to build an LVM interpreter in a relatively short time frame.

There is also a core compiler that translates λ_{core} programs into LVM files using the compilation rules described in section 5. The compiler is still very naive and doesn't perform any 'essential' optimizations like simplification, inlining or strictness analysis. Even though we tried to keep the LVM instruction set and compilation scheme as simple as possible, the total line count of the core compiler is still about 7000 lines of Haskell which is a bit disappointing. On the other hand, the core compiler has a very modular structure and it is easy to use as the backend for a real compiler or as a platform to experiment with new transformation algorithms. It is currently used as a backend to the Helium compiler and the experimental HX system [26].

To assess the performance of the interpreted LVM instruction set, we ran some preliminary benchmarks. Since each benchmark is rather small the results should be interpreted with care. However, we believe that the benchmarks will at least give an indication whether the performance of the an LVM interpreter is acceptable in practice. The following three programs were tested.

nfib 27 Calculates the 27th *nfib* number.

```
nfib :: Int -> Int
nfib 0  = 1
nfib 1  = 1
nfib n  = 1 + nfib (n-1) + nfib (n-2)
```

queens 9 Finds the number of ways to put 9 queens on a 9×9 checkboard where no queen threatens another.

```
queens n     = length (qqueens n n)

qqueens k 0  = [[]]
qqueens k n  = [ (x:xs) | xs <- qqueens k (n-1)
                        , x  <- [1..k], safe x 1 xs ]

safe x d []     = True
safe x d (y:ys) = x /= y && x+d /= y
                 && x-d /= y && safe x (d+1) ys
```

sieve 1000 Calculates the 1000th prime number using the sieve of Erasthones.

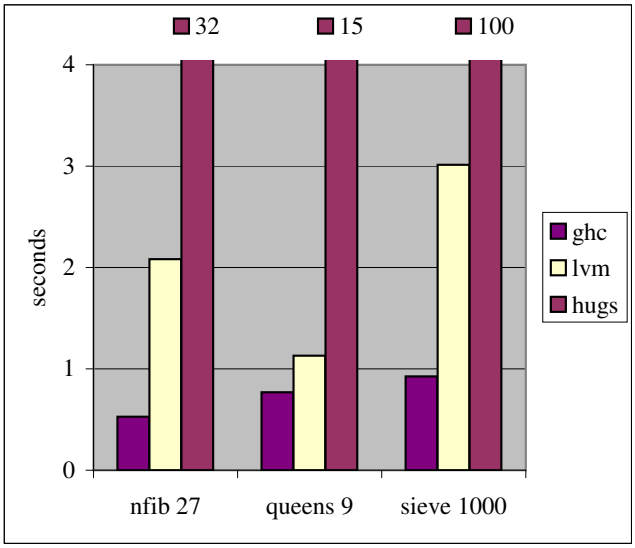

Figure 3: Benchmarks

```
sieve n  = last (take n (ssieve [3,5..]))
         where
          ssieve (x:xs)  = x:ssieve (filter (noDiv x) xs)
          noDiv x y      = (mod x y /= 0)
```

Each program was translated with the Hugs interpreter (May 1999), the GHC compiler (5.02) and the LVM core compiler. GHC was run without the `-O` flag but it still does simplification and inlining. Since the core compiler can not parse full Haskell, each program was manually desugared into enriched lambda expressions before compilation. All programs were run on a 266Mhz PentiumII PC with 128Mb RAM.

Figure 6 shows the running times of each program. Note that the running times of the programs run with Hugs are outside the scale of the y-axis. Perhaps not surprisingly, the LVM performs about 15 to 30 times better on these programs than Hugs. What is more surprising is that the interpreted, non-inlined, unsimplified LVM programs run just 3 times as slow as GHC compiled programs. The `queens` benchmark is even just 25% faster when compiled with GHC. Of course, the programs are too small to be used as realistic benchmarks but the results still give us confidence that the interpreter approach can be successful in practice.

We also measured how the LVM performs if the core compiler would have a simple strictness analyser and inliner. We naively hand-optimized the programs for the LVM, trying to emulate a simple strictness analyser and inliner. Here is for example the optimized source for *nfib*:

```
nfib :: Int -> Int
```

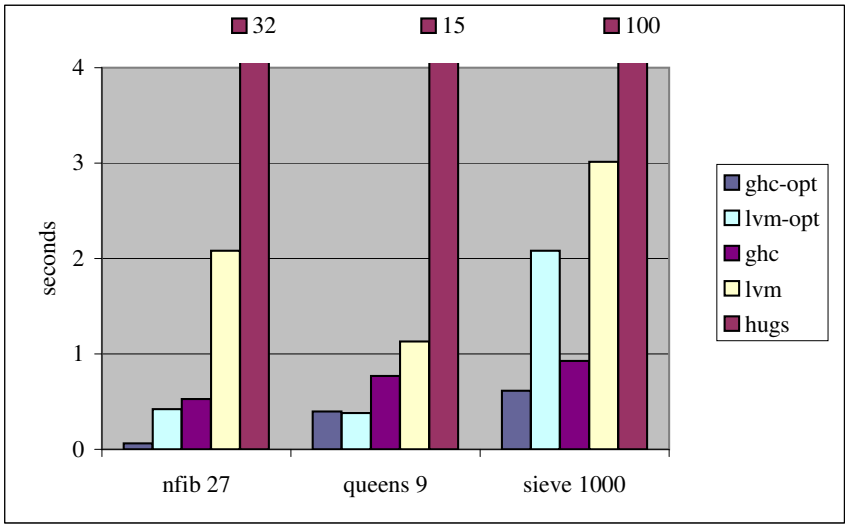

Figure 4: Benchmarks

```
nfib n = match n with
           0 -> 1
           1 -> 1
           n -> let! n2  = primSubInt n 2 in
                let! nf2 = nfib n2 in
                let! n1  = primSubInt n 1 in
                let! nf1 = nfib n1 in
                let! m   = primAddInt nf1 nf2 in
                primAddInt 1 m
```

Figure 6 shows the benchmarks with the optimized compilers. The *ghc-opt* programs are compiled with GHC with the `-O` flag while the *lvm-opt* programs are the hand-optimized sources compiled for the LVM. Optimized GHC is *much* faster on the *nfib* and *sieve* benchmarks but, surprisingly, the *queens* benchmark runs faster with the optimized LVM. We don't know for sure why the *queens* program performs so well, it might be a cache effect or it might be linked to the 'return in registers' convention that can avoid heap allocation – maybe the LVM avoids an expensive allocation in a critical part of the algorithm.

7 Appendix: The module format

An LVM module consists of 8-bit bytes. Multi byte values are stored in the big-endian format where the most significant byte comes first. There are two multi byte values:

int_{32}	A 32 bit signed integer.
$float_{64}$	A 64 bit IEEE floating point value.

Besides these *raw* values, there are also *encoded* values that are also stored as int_{32} values:

int	An encoded signed integer value. An encoded integer n is represented by the int_{32} value $\overline{n} = 2n + 1$.
rec	An encoded signed record index. An encoded record index r is represented by the int_{32} value $\overline{r} = 2r$.

Record indices and numbers are easily distinguished now – record indices are even integers while numbers are stored as odd integers.

The encoded values *int* and *rec* can also be typed:

enum t	An enumeration value of type t stored as an *int*. For example, *enum flags*
rec t	A record index *rec* that points to a record of type t. For example, *rec code* is a record index that points to a *code* record.

7.1 Records

The LVM format consists of *records*. These records are always aligned on 32 bits and should be padded with zero bytes if they don't align properly. A *length* is always the number of bytes, while a *count* is always the number of logical units.

Every LVM module consists of a *header* record, a number of program records and a *footer* record.

struct *lvm-file*
{ **struct** *header* header**;**
record [records count] program-records**;**
struct *footer* footer**;**
}

The header contains the number of program records, *records count*. Records are indexed

with *rec* values that are 1-based indices in the *program-records* array. An index of zero is used when no information is available.

7.2 Header and Footer

The header contains the *records count* and the total length of those records.

struct *header*
{ int_{32} header kind = $_{x}$1F4C564D (= ▼LVM);
int header length;
int total length;
int runtime major version;
int runtime minor version;
int records count;
int records length;
rec module module information;
... ...;
}

The runtime version numbers correspond to the LVM runtime version for which this file was build. The module major version is incremented on each non-compatible interface change, whereas the minor version is incremented for each new build.

struct *footer*
{ int_{32} footer kind = $_{x}$1E4C564D (= ▲LVM);
int length = 4;
int total length;
}

The footer marks the end of the LVM file and enables stand-alone executables. The LVM runtime has a special option that concatenates the runtime with all the needed LVM module files. When this program is invoked, the runtime loads its own image and looks if it ends with a footer, if so, it traces all catenated modules and executes them – et voilá, a portable method for stand-alone executables.

7.3 A Record

A record starts with the *kind* and the *length* of the record (always a multiple of 4). The length doesn't include the kind and length field.

A *standard* record starts with an *enum standard-kind* while a *custom* record starts with a *rec kind*. The LVM ignores custom records but they can be used by a compiler to encode

extra information – for example, algebraic data declarations.

struct *record*
{ *enum standard-kind* or *rec kind* record-kind**;**
int record length**;**
... ...**;**
}

Standard record kinds include:

enum *standard-kind*
{ name = 0**;**
kind = 1**;**
bytes = 2**;**
code = 3**;**
value = 4**;**
constructor = 5**;**
import = 6**;**
module = 7**;**
extern = 8**;**
externtype = 9**;**
}

The following records are all described without their standard header, i.e. the kind and length. To distinguish them from a **struct**, we use the special **record** declaration.

7.4 Byte records

A *byte* record contains a number of raw bytes. There exist four kinds of byte records: *name*, *kind*, *bytes* and *externtype* records.

A *name* record contains a serie of bytes that are used for a static (link-time) names or identifiers.

record *name*
{ *int* name length**;**
byte[name length] name**;**
byte[...] padding**;**
}

A *bytes* record also contains a serie of bytes. These are used for dynamic (run-time) enitities, like big integers or strings.

record *bytes*

{ *int* bytes length**;**
byte[bytes length] bytes**;**
byte[...] padding**;**
}

The kind of a custom record is described by a *kind* record. A *kind* record contains a serie of bytes that hold the static name of a custom kind.

record *kind*
{ *int* kind length**;**
byte[kind length] kind name**;**
byte[...] padding**;**
}

The type of an *extern* declaration is an *externtype* record. An *externtype* record is just a static string describing the type of an external function.

record *externtype*
{ *int* type length**;**
byte[type length] type**;**
byte[...] padding**;**
}

7.5 Instruction records

An *instruction* record constains LVM instructions. There is only one instance of an instruction record, namely *code*.

record *code*
{ int_{32}[record length/4] instructions**;**
}

7.6 Structured records

A *structured* record consists of *rec* and *int* values. All records that are not instruction- or byte records belong to this group. Structured records are either *standard* records or *custom* records.

Structured records have predefined fields but can also contain *custom* values encoded as *rec* or *int* values. Custom values are ignored by the LVM but can be used by a compiler to encode more information, like type signatures or inline declarations. Potential custom values are notated with three dots – "...".

A structured *declaration* record starts with a *name* and access *flags*. The *flags* determine the external visibility of a record.

enum *flags*
{ private = 0**;**
public = 1**;**
}

7.7 Standard records

record *value*
{ *rec name* name**;**
enum flags flags**;**
int arity**;**
rec value enclosing value**;**
rec code code**;**
... ...**;**
}

record *constructor*
{ *rec name* name**;**
enum flags flags**;**
int arity**;**
int tag**;**
... ...**;**
}

7.7.1 Import records

record *import*
{ *rec name* name**;**
enum flags flags**;**
rec module imported module**;**
rec name imported name**;**
enum standard-kind or *rec kind* imported record kind**;**
... ...**;**
}

A *module* declaration contains the version numbers of the module that it was linked to at compile time.

record *module*

```
{  rec name    name;
   int         major version;
   int         minor version;
   ...         ...;
}
```

7.7.2 Extern declarations

A *extern* declaration contains the signature of an external function.

```
record extern
{  rec name              name;
   enum flags            flags;
   int                   arity;
   rec externtype        external type;
   rec name              external library name;
   rec name or int       external name or ordinal;
   enum name-mode        name-mode;
   enum link-mode        link-mode;
   enum call-mode        calling convention;
   ...                   ...;
}
```

There are three *link-mode*s. *static* linkage is used for static libraries, *dynamic* for dynamic link libraries and *runtime* for functions that are referenced by address. The first argument of a *runtime* function is always the address of this function.

```
enum link-mode
{  static      = 0;
   dynamic     = 1;
   runtime     = 2;
}
```

The *call-mode* is either the C calling convention (*ccall*) or the *stdcall* (or *pascal*) calling convention (used on windows platforms).

```
enum call-mode
{  ccall               = 0;
   stdcall (pascal)    = 1;
}
```

The *name-mode* gives the mode of a name. Mode *decorate* decorates the name according to the calling convention. The *ccall* convention for example prefixes a name with an underscore. If mode *ordinal* is specified, the external name should contain an ordinal instead of a *rec*

name. The ordinal is the index of a function in a (dynamic) library, used for example in the windows system libraries. The *normal* mode leaves the name as it is.

enum *name-mode*
{ normal = 0**;**
decorate = 1**;**
ordinal = 2**;**
}

The type of an *extern* declaration is a *externtype* record, that just consists of a string of bytes. The type is interpreted as an ASCII string where each character describes the type of each argument. The first character describes the type of the result.

character	c-type	lvm-type
`a`	`value`	any LVM value
`c`	`char`	*int*
`i`	`int`	*int*
`I`	`long`	*int*
`f`	`float`	*float*
`d`	`double`	*float*
`D`	`long double`	*float*
`F`	`double` (or `long double`)	*float*
`u`	`unsigned int`	*int*
`U`	`unsigned long`	*int*
`p`	`void*`	*ptr*
`z`	`char*`	*string*
`Z`	`wchar_t*`	*string*
`v`	void	()
`1`	8 bit value	*int*
`2`	16 bit value	*int*
`4`	32 bit value	*int32*
`8`	64 bit value	*int64*
`n`	`long` (or `int`)	*native-int*

7.8 Custom records

A custom record always starts with a *rec kind* instead of a standard *int* kind. A custom record is either a *declaration* record, starting with a name and flags, or an *anonymous* record that starts with a zero index for the name.

record *custom*
{ *rec name* name**;**
enum flags flags**;**
... ...**;**

}

record *custom*
{ *rec name* name = 0**;**
... ...**;**
}

8 Appendix: The instruction set

All instructions and their arguments are 32 bits. Besides uniformity and simplicity, it has the advantage of executing much faster on current hardware architectures. The disadvantage is slightly larger code than bytecode oriented formats like the Java VM for example. We plan to define a compressed LVM format for distributing modules over slow networks or execution on mobile devices.

The basic types of values are:

int_{32} – a 32 bit signed integer.
$float_{64}$ – a 64 bit IEEE floating point value.
rec_{32} – a 32 bit signed record index (1-based).

The int_{32}, and rec_{32} types have only 30 bits of garanteed significance. The rec_{32} type is an index into the standard records of the module format (see section 7).

n,m – int_{32} values.
ofs,i – int_{32} values.
d – $float_{64}$ value.
f – rec_{32}, value record index.
c – rec_{32}, constructor record index[3].
b – rec_{32}, bytes record index.

8.1 Stack

(0)	ArgChk(n)	The argument satisfaction check – are there n arguments on the stack?
(1)	PushCode(f)	Push a function or CAF at record f.
(2)	PushCont(ofs)	Push a continuation frame to the code at *ofs* relative to the current location.
(3)	PushVar(n)	Push a local variable at stack location n.
(4)	PushInt(i)	Push a 32 bit signed integer i.
(5)	PushFloat(d)	Push a 64 bit IEEE floating point value d.
(6)	PushBytes(b)	Push the bytes at record b.
(7)	Slide(n, m)	Slide the top n values over the next m values.
(8)	Stub(n)	Overwrite the local variable at $st[n]$ with an *inv* value.

[3] hackers extension: a negative or zero index i is interpreted as an anonymous constructor with tag $|i|$.

8.2 Functions

(9)	AllocAp(n)	Allocate an uninitialized application node with n fields.
(10)	PackAp(n, m)	Create an updateable application node at stack location n with m values.
(11)	PackNap(n, m)	Create a non-updateable application node at stack location n with m values.
(12)	NewAp(n)	Allocate and initialize an updateable application node with n values from the stack.
(13)	NewNap(n)	Allocate and initialize a non-updateable application node with n values from the stack.

8.3 Control

(14)	Enter	Enter the value at the top of the stack.
(15)	Return	Return the whnf value at the top of the stack.
(16)	Catch	Install the exception handler at the top of the stack.
(17)	Raise	Raise the exception at the top of the stack.
(18)	Call(c, n)	Call an external function c with n arguments.

8.4 Alternatives

(19)	AllocCon(c, n)	Allocate a constructor c with n uninitialized fields.
(20)	PackCon(n, m)	Initialize a constructor node at stack location n with m values.
(21)	NewCon(c, n)	Allocate and initialize a constructor c with n values from the stack.
(22)	UnpackCon(n)	Move n fields from the constructor at the top of the stack to the stack.
(23)	TestCon(c, *ofs*)	Test the tag of the constructor at the top of the stack with the tag of the constructor c. If it is not equal, jump to the code at *ofs* relative to the current location (ie. the start of the next instruction).

8.5 Integers

(24)	TestInt(i, *ofs*)	Test the integer at the top of the stack with i. If it is not equal, jump to the code at *ofs* relative to the current location.
(25)	AddInt	Add two integers at the top of the stack; pop the integers and push the result.
(26)	SubInt	Subtract.
(27)	MulInt	Multiply.

(28) DivInt — Euclidean division.
(29) ModInt — Euclidean modulus.
(30) QuotInt — Truncated Quotient.
(31) RemInt — Truncated Remainder.
(32) AndInt — Bitwise and.
(33) XorInt — Bitwise xor.
(34) OrInt — Bitwise or.
(35) ShrInt — Bitwise arithmetic shift right (pad with highest bit).
(36) ShlInt — Bitwise shift left.
(37) ShrNat — Bitwise unsigned shift right (pad with zeros).
(38) NegInt — Negate.

8.6 Comparison

(39) EqInt — Equal.
(40) NeInt — Not equal.
(41) LtInt — Lower.
(42) GtInt — Greater.
(43) LeInt — Lower or equal.
(44) GeInt — Greater or equal.

8.7 General sums and products

(45) Alloc — Allocate a new heap block with the size at $st[1]$ and the tag at $st[0]$.
(46) New(n) — Allocate and initialize a new heap block with n values with the tag at $st[0]$.
(47) GetField — Push field $st[1]$ of the heap block at $st[0]$ on the top of the stack.
(48) SetField — Set field $st[1]$ of the heap block at $st[0]$ to the value at $st[2]$.
(49) GetTag — Push the tag of the heap block on the top of the stack.
(50) GetSize — Push the size of the heap block on the top of the stack.
(51) Pack(n) — Initialize the heap block on top of the stack with n values at the following stack locations and pop them all.
(52) Unpack(n) — Move n fields from the heap block at $st[0]$ to the stack.

8.8 Optimized stack

(53) PushVar0 ⇒ PushVar(0)
(54) PushVar1 ⇒ PushVar(1)
(55) PushVar2 ⇒ PushVar(2)
(56) PushVar3 ⇒ PushVar(3)

(57) PushVar4 $\Rightarrow$ PushVar(4)
(58) PushVars2($n1, n2$)
$\Rightarrow$ PushVar($n1$) : PushVar($n2$)
(59) PushVars3($n1, n2, n3$)
$\Rightarrow$ PushVars2($n1, n2$) : PushVar($n3$)
(60) PushVars4($n1, n2, n3, n4$)
$\Rightarrow$ PushVars3($n1, n2, n3$) : PushVar($n4$)

8.9 Optimized functions

(61) NewAp1 $\Rightarrow$ NewAp(1)
(62) NewAp2 $\Rightarrow$ NewAp(2)
(63) NewAp3 $\Rightarrow$ NewAp(3)
(64) NewAp4 $\Rightarrow$ NewAp(4)
(65) NewNap1 $\Rightarrow$ NewNap(1)
(66) NewNap2 $\Rightarrow$ NewNap(2)
(67) NewNap3 $\Rightarrow$ NewNap(3)
(68) NewNap4 $\Rightarrow$ NewNap(4)

8.10 Optimized constructors

(69) NewCon0(c) $\Rightarrow$ NewCon($c, 0$)
(70) NewCon1(c) $\Rightarrow$ NewCon($c, 1$)
(71) NewCon2(c) $\Rightarrow$ NewCon($c, 2$)
(72) NewCon3(c) $\Rightarrow$ NewCon($c, 3$)

8.11 Optimized control

(73) EnterCode(c) $\Rightarrow$ PushCode(c) : Enter. Enter the function declared at constant c. The stack is required to hold at least all the arguments of the function.
(74) EvalVar(n) $\Rightarrow$ PushCont(6) : PushVar($n + 3$) : Slide($1, 0$) : Enter. Push a continuation frame and enter the variable at offset n on the stack.
(75) ReturnCon(c, n) $\Rightarrow$ NewCon(c, n) : Slide($1, ?$) : Enter. Return a constructor c with n fields on the stack.
(76) ReturnInt(i) $\Rightarrow$ PushInt(i) : Slide($1, ?$) : Enter. Return the integer i.
(77) ReturnCon0(c) $\Rightarrow$ ReturnCon($c, 0$).

8.12 Optimized alternatives

(78) MatchCon(n, ofs, c_1, ofs_1, ..., c_n, ofs_n)
Pop the constructor on top of the stack and match it with c_1 to c_n, jumping to ofs_i when matching. Jump to ofs when no match is found.

(79) SwitchCon(n, ofs, ofs_1, ..., ofs_n)
Pop the constructor on top of the stack and switch on its tag. Jump to ofs when the tag is greater than n.

(80) MatchInt(n, ofs, i_1, ofs_1, ..., i_n, ofs_n)
Pop the int on top of the stack and match it with i_1 to i_n, jumping to the corresponding ofs_i when matching. Jump to ofs when no match is found.

(81) MatchFloat(n, ofs, d_1, ofs_1, ..., d_n, ofs_n)

(82) Match(n, ofs, tag_1, $arity_1$, ofs_1, ..., tag_n, $arity_n$, ofs_n)

8.13 Floating point

(83) ReturnFloat(d) $\Rightarrow$ PushFloat(d) : Slide(1, ?) : Enter. Return the float d.
(84) AddFloat Add.
(85) SubFloat Subtract.
(86) MulFloat Multiply.
(87) DivFloat Divide.
(88) NegFloat Negate.

8.14 Floating point comparison

(89) EqFloat Equal.
(90) NeFloat Not equal.
(91) LtFloat Lower.
(92) GtFloat Greater.
(93) LeFloat Lower or equal.
(94) GeFloat Greater or equal.

8.15 General sum and products

	Code	Stack	Heap
	$\mathsf{Alloc} : is$	$t : n : st$	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{con}_t(inv_1, ..., inv_n)]$
	$\mathsf{New}(n) : is$	$t : x_1 : ... : x_n : st$	hp
$\Longrightarrow$	is	$p : st$	$hp \circ [p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
	$\mathsf{Pack}(n) : is$	$p : x_1 : ... : x_n : st$	$hp[p \mapsto \mathbf{con}_t(y_1, ..., y_n)]$
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
	$\mathsf{UnPack}(n) : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	is	$x_1 : ... : x_n : st$	hp
$0 \leq i < n$	$\mathsf{GetField} : is$	$p : i : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	is	$x_{i+1} : st$	hp
$0 \leq i < n$	$\mathsf{SetField} : is$	$p : i : x : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_{i+1}, ..., x_n)]$
$\Longrightarrow$	is	st	$hp \bullet [p \mapsto \mathbf{con}_t(x_1, ..., x, ..., x_n)]$
	$\mathsf{GetTag} : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	is	$t : st$	hp
	$\mathsf{GetSize} : is$	$p : st$	$hp[p \mapsto \mathbf{con}_t(x_1, ..., x_n)]$
$\Longrightarrow$	is	$n : st$	hp

9 Appendix: Primitive operations

This section gives an overview of primitive data types and operations in the LVM runtime.

9.1 Exceptions

The `Exception` data types are in principle open-ended but the following exceptions are pre-defined by the system.

```
type BString  = Bytes

data Exception
  = HeapOverFlow                        -- heap overflow
  | StackOverflow   Int                 -- stack overflow
  | Signal          SignalException     -- interrupt occurred
  | Runtime         RuntimeException    -- runtime system exception
  | Arithmetic      ArithmeticException -- arithmetic exception
  | System          SystemException     -- operating system exceptions
  | InvalidArgument BString             -- invalid argument passed
  | Assert          BString             -- assertion failed
  | NotFound                            -- no object is found
  | UserError       BString             -- general failure (raised by "error")

data RuntimeException
  = PatternFailure    BString           -- pattern match failure
  | NonTermination    BString           -- non terminating program
  | OutOfBounds       BString           -- field access out of bounds
  | Exit              Int               -- exiting program
  | InvalidOpcode     Int               -- invalid opcode
  | LoadError         BString BString   -- runtime loader exception
  | RuntimeError      BString           -- general failure

data SystemException
  = EndOfFile                           -- end of input reached
  | BlockedOnIO                         -- blocked I/O channel
  | SystemError       Int BString       -- general system error

data ArithmeticException
  = FloatInvalidOperation       -- invalid float operation
  | FloatDivideByZero           -- float division by zero
  | FloatOverflow               -- float has overflowed
  | FloatUnderflow              -- float has underflowed
  | FloatInexact                -- float result is inexact
  | FloatDenormal               -- denormalized float value
  | DivideByZero                -- integer division by zero
```

```
  | Overflow                    -- integer overflow
  | Underflow                   -- integer underflow
  | InvalidOperation            -- general arithmetic error
  | UnEmulated                  -- cannot emulate float instruction
  | NegativeSquareRoot          -- square root of negative number
  | FloatStackOverflow          -- float hardware stack has overflowed
  | FloatStackUnderflow         -- float hardware stack has underflowed

data SignalException
  = SignalNone                  -- runtime: no signal
  | SignalGarbageCollect        -- runtime: GC needed
  | SignalYield                 -- runtime: thread should yield
  | SignalLost                  -- runtime: lost signal
  | SignalKeyboard              -- interactive interrupt (ctrl-c)
  | SignalKeyboardStop          -- interactive stop (ctrl-break)
  | SignalFloatException        -- floating point exception
  | SignalSegmentationViolation -- invalid memory reference
  | SignalIllegalInstruction    -- illegal hardware instruction
  | SignalAbort                 -- abnormal termination
  | SignalTerminate             -- termination
  | SignalKill                  -- termination (can not be ignored)
  | SignalKeyboardTerminate     -- interactive termination
  | SignalAlarm                 -- timeout
  | SignalVirtualAlarm          -- timeout in virtual time
  | SignalBackgroundRead        -- terminal read from background process
  | SignalBackgroundWrite       -- terminal write from background process
  | SignalContinue              -- continue process
  | SignalLostConnection        -- connection lost
  | SignalBrokenPipe            -- open ended pipe
  | SignalProcessStatusChanged  -- child process terminated
  | SignalStop                  -- stop process
  | SignalProfiler              -- profiling interrupt
  | SignalUser1                 -- application defined signal 1
  | SignalUser2                 -- application defined signal 2
```

9.2 Bytes

Converting byte arrays to character lists.

```
value prim_string_of_chars( long count, value chars );
value prim_chars_of_string( value string );
long  prim_string_length( value string );
```

9.3 IO

Basic input and output.

```
long prim_flag_mask( long flags );
long prim_open( const char* fname, long sysflags );
void prim_close( long handle );

value prim_open_descriptor( long handle, bool output );
void  prim_close_channel( value channel )
void  prim_set_binary_mode( value channel, bool binary );
bool  prim_flush_partial( value channel );
void  prim_flush( value channel );
void  prim_output_char( value outchannel, char c );
void  prim_output( value outchannel, const char* buffer
                , long start, long count );
long  prim_input_char( value inchannel );
```

The `prim_flag_mask` function converts a portable flag into a system flag mask used by `prim_open`. The portable flags are:

```
enum open_flags {
  Open_readonly = 0,
  Open_writeonly,
  Open_append,
  Open_create,
  Open_truncate,
  Open_exclusive,
  Open_binary,
  Open_text,
  Open_nonblocking
};
```

9.4 Floating point

Primitive floating point operations.

```
data RoundMode  = RoundNear
                | RoundUp
                | RoundDown
                | RoundZero

void  fp_set_round_mode( value mode );
value fp_get_round_mode( void );


data ArithmeticException
  = FloatInvalidOperation        -- invalid float operation
  | FloatDivideByZero            -- float division by zero
  | FloatOverflow                -- float has overflowed
  | FloatUnderflow               -- float has underflowed
  | FloatInexact                 -- float result is inexact
  | ...

long fp_sticky_mask( value exn );
long fp_get_sticky( void );
long fp_set_sticky( long sticky );

long fp_trap_mask( value exn );
long fp_get_traps( void );
long fp_set_traps( long traps );

void fp_reset( void );


bool  signal_is_trapped( value exception );
bool  signal_trap( value exception );
void  signal_untrap( value exception );
```

References

[1] L. Augustsson. Cayenne – a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[2] R. T. Boute. The Euclidean definition of the functions div and mod. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 14, pages 127–144, New York, NY, USA, Apr. 1992. ACM press.

[3] W. J. Cody et al. A proposed radix- and word-length-independent standard for floating-point arithmetic. In *IEEE Micro*, volume 4, pages 86–100, Aug. 1984.

[4] J. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, University of Oxford, 1984.

[5] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[6] T. Johnsson. Efficient compilation of lazy evaluation. In *proceedings of the ACM Conference on Compiler Construction*, pages 58–69, Montreal, Canada, June 1984.

[7] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *proceedings of Functional Programming Languages and Computer Architecture, LNCS*, volume 201, pages 190–203, Nancy, France, Sept. 1985.

[8] M. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, May 1994.

[9] D. E. Knuth. *The Art of Computer Programming, Vol 1, Fundamental Algorithms*. Addison-Wesley, 1972.

[10] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[11] X. Leroy. Le système caml special light: modules et compilation efficace en caml. Technical Report 2721, INRIA, France, Nov. 1995.

[12] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, Apr. 1999.

[13] J. Mountjoy. The spineless tagless g-machine, naturally. In *Proceedings of the third ACM SIGPLAN International Conference on Functional programming*, Baltimore, Maryland, 1998.

[14] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, International series in computer science, 1986.

[15] S. Peyton Jones. Implementing non-strict languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, Apr. 1992.

[16] S. Peyton Jones and R. Ennals. Optimistic evaluation: a fast evaluation strategy for non-strict programs. submitted to ICFP'03, Mar. 2003.

[17] S. Peyton Jones and J. Hughes (eds.). *Report on the language Haskell'98*, Feb. 1998.

[18] S. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in Haskell. *Software – Practice and Experience*, 21(5):479–506, 1991.

[19] S. Peyton Jones and S. Marlow. Secrets of the glasgow haskell compiler inliner. In *Workshop on Implementing Declarative Languages*, 1999. revised version submitted to JFP, Feb 2001.

[20] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *International Conference on Function Programming (ICFP)*, Philadelphia, May 1996.

[21] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, Sept. 1998.

[22] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL*, volume 20, pages 71–84, 1993.

[23] M. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[24] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science department, Aarhus University, Sept. 1981.

[25] M. Shields. *Static Types for Dynamic Documents*. PhD thesis, Oregon Graduate Institute, Feb. 2001.

[26] M. Shields and S. Peyton Jones. First-class modules for Haskell. In *Ninth International Conference on Foundation of Object-Oriented Languages (FOOL 9)*, Portland, Oregon, Dec. 2001.

[27] G. L. Steele Jr. *Common LISP: The Language, 2nd edition*. Digital Press, Woburn, MA, USA, 1990.

[28] S. Tucker Taft and R. A. Duff (eds.). *Ada95 Reference Manual: Language and Standard Libraries*. International Standard ISO/IEC 8652:1995(E), 1997.

[29] N. Wirth. The programming language Oberon. *Software Practice and Experience*, 19(9), 1988. The Oberon language report.