LVM, the lazy virtual machine

DAAN LEIJEN
University of Utrecht
Dept. of Computer Science
PO.Box 80.089, 3508 TB Utrecht
The Netherlands
daan@cs.uu.nl,http://www.cs.uu.nl/~daan/lvm.html
Revision : 1.12

November 12, 2002

1 Introduction

The Lazy Virtual Machine (LVM), like the JVM, defines a portable instruction set and file format. However, it is specifically designed to execute languages with non-strict (or lazy) semantics. These languages are hard to map to stock hardware and require an extensive runtime system. By providing a standard runtime environment we hope to make it easier to experiment with new lazy languages, profilers or debuggers.

Prominent features of the LVM are:

- A portable file format and instruction set.
- High level and simple instructions – ie. easy to compile to from a lazy language.
- Efficient interpretation or JIT compilation is possible.
- Interaction with existing C and Java libraries.

The LVM is implemented on top of the OCaml runtime system and there is a toolkit that translates enriched lambda expressions to bytecode files. The system currently runs on many platforms, including Windows, various Unix's, MacOSX and 64-bit platforms like the DEC alpha.

The current LVM implementation is an interpreter and the LVM modules are generated with a very naive compiler that performs no inlining or simplification. However, preliminary bench marks (queens, sieve) show that it performs quite well in practice – it is orders of magnitude faster than Hugs and about three times as slow as unoptimised GHC code.

The instruction set of the LVM is closely based on the STG machine. The most notable differences are that the machine is environment-less but not tag-less. Both changes were made to make efficient interpretation possible. The design of the LVM instruction set was further influenced by the implementation of the once existing STG-Hugs interpreter written by Alastair Reid. Allthough it initiated the design of the LVM, it now differs fundamentally on many aspects, like partial applications, uniform values and continuation frames.

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 1: Abstract syntax of the LVM language

2 The LVM language

Since the LVM is supposed to be (functional) language neutral, we define a small LVM language that is used to explain the instruction set and to define a formal compilation scheme. The abstract syntax for the LVM language is given in figure 1. Allthough the form of expressions is restrictive, any enriched lambda calculus expression can be translated into a LVM expression. As such, the LVM language can be seen as the assembly language of the lambda calculus.

Just like the STG language (Peyton Jones, 1992), we attach an operational reading to the LVM language: `let` and `letrec` perform heap allocation, `let!` evaluates expressions and `match` distinguishes evaluated values. The LVM languages doesn't contain lambda-expressions or local function definitions – all functions have been lambda-lifted to toplevel (Johnsson, 1985). This means that no LVM function contains free variables and a program consists of recursive equations.

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*
⇒

`let!` x `=` e `in match` x `with` *alts*

Strict languages can easily be translated into the LVM language. Every language `let` binding becomes a LVM `let!` binding. The `letrec` binding of O'Caml and ML can only be used on recursive *functions* which are lifted to toplevel and present no problem.

The distinction between atomic and normal expressions is more than a syntactic convenience. An atomic expression can always be constructed without failure but a `let!`, `match` or *prim* expression can fail or loop. This is the reason why `let` expressions can only contain atomic expressions on their right-hand side.

2.1 Translating into the LVM language

The following transformations should be applied to translate enriched lambda calculus into the LVM language:

- 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.

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

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

  $\backslash x_1\ ...\ x_n\ .\ e \quad \Rightarrow \quad$ `let` $x\ x_1\ ...\ x_n\ =\ e$ `in` x

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

  e (`match` x `with` *alts*) $\quad \Rightarrow \quad$ `let` $y\ =$ (`match` x `with` *alts*) `in` $e\ y$

- Introduce a `let` expression for all applications that are not applied to a variable or constructor.

  $e\ x_1\ ...\ x_n \quad \Rightarrow \quad$ `let` $x\ =\ e$ `in` $x\ x_1\ ...\ x_n$

- Pass all free variables in non-atomic expressions as explicit arguments. This leads to an environment-less LVM machine. This transformation corresponds essentially to lambda-lifting.

  $f\ x\ =$ `let` $y\ =$ (`let!` $z\ =\ 1/x$ `in` z) `in` y
  $\Rightarrow$
  $f\ x\ =$ `let` $y\ x\ =$ (`let!` $z\ =\ 1/x$ `in` z) `in` $y\ x$

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

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

2.2 Let floating

An explicit let floating pass can often be avoided since the LVM language allows `let` expressions as atomic expressions Consider the following Haskell expression:

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

A straightforward translation gives:

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

Another way to translate the expression is to float the `let` binding one level up:

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

Both programs are almost equivalent under the compilation scheme presented in the next section. The only difference is that 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. If the same behaviour as the STG is desired, the binding should be lifted to the top level:

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

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 (Peyton Jones *et al.*, 1996).

2.3 Strictness and speculative evaluation

In general, we can not float up 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 but we plan to add the atomic `let$` construct for speculative bindings.

2.4 Compilation scheme

The compilation scheme translates the LVM language 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 seperate *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 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 dead 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 dead values are slided out of the stack.

2.4.1 Program

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

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

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 that is subsequently entered by the Enter instruction.

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

Each top level value first checks the number of arguments with an argument check instruction. This is necessary in a higher-order language since it is not always possible to determine at a call site if a function is partially applied or not. For example:

$$apply\ f\ x\ =\ f\ x$$

Is f partially applied or not? This can not be determined without a whole-program analysis. For this reason, each function checks the number of arguments itself with the ArgChk instruction, building a partial application node if there are too few arguments. Many times however, the compiler *can* determine if there are enough arguments. The rewrite rules that are given later in this document 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 this instruction or otherwise the EnterCode instruction will enter the function at a random location!

2.4.2 Expressions

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

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

2.4.3 Atomic expressions

The $\mathcal{A}$ scheme wraps the instructions in an Atom pseudo instruction to slide out any dead 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.

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

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:

$$\begin{array}{l}\mathsf{ArgChk}(3); \mathsf{Atom}(\\ \quad \mathsf{Param}(y); \mathsf{Param}(x); \mathsf{Param}(f); \\ \quad \mathsf{Atom}(\mathsf{PushVar}(x); \mathsf{NewAp}(1)); \\ \quad \mathsf{Atom}(\mathsf{PushVar}(y); \mathsf{NewAp}(1)); \\ \quad \mathsf{Atom}(\mathsf{PushVar}(f); \mathsf{NewAp}(1)); \\ \quad \mathsf{NewAp}(3)) \\ \mathsf{Enter}\end{array}$$

After *resolve*, this instruction stream becomes:

$$\begin{array}{l}\mathsf{ArgChk}(3); \\ \mathsf{PushVar}(1); \mathsf{NewAp}(1); \mathsf{Slide}(1,0); \\ \mathsf{PushVar}(3); \mathsf{NewAp}(1); \mathsf{Slide}(1,0); \\ \mathsf{PushVar}(2); \mathsf{NewAp}(1); \mathsf{Slide}(1,0); \\ \mathsf{NewAp}(3); \mathsf{Slide}(1,3); \\ \mathsf{Enter}\end{array}$$

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 2.6, 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 dumb, we hope that it becomes at least 'obviously' correct.

2.4.4 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 it's 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.

$\mathcal{U}[\![\, let \texttt{ in } a \,]\!] \quad \Rightarrow$
$\quad \mathcal{U}[\![\, a \,]\!]$
$\mathcal{U}[\![\, id\ a_1 \ldots a_n \,]\!] \quad \Rightarrow$
$\quad \mathsf{AllocAp}(n+1);$
$\mathcal{U}[\![\, con_t^n\ a_1 \ldots a_n \,]\!] \quad \Rightarrow$
$\quad \mathsf{AllocCon}(t, n);$

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

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

2.4.5 Matching

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

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

Each pattern is compiled with the $\mathcal{P}$ scheme. Note that the Param instruction is used to bind the values of a matched constructor.

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

2.4.6 Optimized schemes

Allthough we tried to put as little smartness as possible into the compilation scheme, 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. 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).

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

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

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

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):

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

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

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

In general, when an expression is evaluated that already returns a strict result, we don't need to enter that result again.

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

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 where the result type is annotated with a (!) when the result is always in weak head normal form. The function *whnf* can be conservatively defined as:

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

$$
\begin{array}{lcl}
\mathit{whnfAlts}\ (\texttt{\{}\ \mathit{alt}_1\ \texttt{;}\ \ldots\texttt{;}\ \mathit{alt}_n\ \texttt{\}}) & = & \mathit{whnfAlt}(\mathit{alt}_1)\ \&\ \ldots\ \&\ \mathit{whnfAlt}(\mathit{alt}_n) \\
\mathit{whnfAlt}\ (\mathit{pat}\ \texttt{->}\ e) & = & \mathit{whnf}(e)
\end{array}
$$

2.5 Resolve rules

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.

2.5.1 The resolve monad

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

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

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 (Hutton and Meijer, 1996):

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

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

$$
\begin{array}{l}
\mathit{pop}\ n\ = \\
\quad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow \langle (),\ \mathit{depth} - n\rangle) \\
\mathit{push}\ n\ = \\
\quad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow \langle (),\ \mathit{depth} + n\rangle)
\end{array}
$$

The *depth* function returns the current stack depth.

$$
\begin{array}{l}
\mathit{depth}\ = \\
\quad M\ (\backslash\langle \mathit{env},\ \mathit{depth}\rangle \rightarrow \langle \mathit{depth},\ \mathit{depth}\rangle\ )
\end{array}
$$

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

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

Note that the *offset* of a local variable is the difference between the current stack depth and the stack depth at which the variable was bound (with the *bind* function). This well

known trick allows us to dereference all local variables relative to the current stack pointer and removes the need for a seperate *base pointer*, which is still used in some C compilers to aid debuggers.

2.5.2 The algorithm

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

```
resolves (Param(x) : instrs) =
    do{ push 1; bind x (resolves instrs) }
resolves (Var(x) : instrs) =
    bind x (do{ resolves instrs })
resolves (instr : instrs) =
    do{ is ← resolve instr
        iss ← resolves instrs
        return (is ++ iss) }
```

Individual instructions are resolved by the *resolve* function.

```
resolve PushVar(x) =
    do{ ofs ← offset x;
        push 1;
        return [PushVar(ofs)] }
resolve PackAp(x, n) =
    do{ ofs ← offset x;
        pop n;
        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;
```

$is' \leftarrow resolves\ is;$
$d1 \leftarrow depth;$
let $m\ =\ d1 - d0 - n$
$pop\ m;$
$return\ (is'\ +\!\!+\ [\mathsf{Slide}(n, m)])\ \}$

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

$resolveAlt\ \langle t,\ is\rangle\ =$
  do$\{\ is' \leftarrow resolves\ is;\ return\ \langle t,\ is\rangle\ \}$

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

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

2.6 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 2.4.3, the rewrite rules are an important optimization 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.

$\mathsf{MatchCon}(alt_1, ..., alt_n); instrs \quad \Rightarrow$
  $\mathsf{MatchCon}(alt_1; instrs\ , ...,\ alt_n; instrs)$
$\mathsf{MatchInt}(alt_1, ..., alt_n); instrs \quad \Rightarrow$
  $\mathsf{MatchInt}(alt_1; instrs\ , ...,\ alt_n; instrs)$

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.

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

If an application node is entered immediately after building it, we can safely enter the

application directly without building the application node at all! The last rule moves the Slide instruction up in order to prevent space leaks while calling external functions.

$$\begin{aligned}
&\mathsf{NewAp}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\
&\quad \mathsf{Slide}(n, m); \mathsf{Enter} \\
&\mathsf{NewNap}(n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\
&\quad \mathsf{Slide}(n, m); \mathsf{Enter} \\
&\mathsf{Call}(prim, n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\
&\quad \mathsf{Slide}(n, m); \mathsf{Call}(prim, n); \mathsf{Enter}
\end{aligned}$$

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

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

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

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

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.

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

Commonly, a constructor or literal is returned. The LVM has the special ReturnCon and ReturnInt instructions that can potentially execute without extra heap allocation implied by building a new constructor. Instead of building a new constructor that is immediately entered, the constructor is kept on the stack. This is the 'return in registers' convention as described in the STG machine paper (Peyton Jones, 1992).

$$\begin{aligned}
&\mathsf{NewCon}(t, n); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\
&\quad \mathsf{Slide}(n, m); \mathsf{ReturnCon}(t, n) \\
&\mathsf{PushInt}(i); \mathsf{Slide}(1, m); \mathsf{Enter} \quad \Rightarrow \\
&\quad \mathsf{Slide}(0, m); \mathsf{ReturnInt}(i)
\end{aligned}$$

An LVM interpreter can test a variable cheaply 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.

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

Many times we can merge slides, arising from instructions pushed into an alternative.

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

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.

```
Call(prim, n)      | prim = instr instr :: t_1 → ... → t_n → t  ⇒
    instr
NewAp(n)           | n ≤ 1 ⇒
    –
Slide(n, 0)        ⇒
    –
```

2.7 Code generation

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

```
codegens is =
    concat (map codegen is)
codegen Eval(is) =
    let is' = codegens is
    in [PushCont(size is')] ++ is'
codegen 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*.

2.8 More optimization: superfluous stack movements

An important optimization is to reduce the number of superfluous stack movements. Due to the close relation of the low-level LVM language with the LVM instruction set, it is possible to perform this optimization on the language level instead of 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 (Peyton Jones and Marlow, 2002) 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 never duplicate either work or code. Trivial expressions consist of:

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

2.8.1 Inlining strict bindings

We look again at the example program *discriminant* from section 2.4.6:

$discriminant\ a\ b\ c =$ `let!` $ac = a * c$ `in`
`let!` $ac4 = 4 * ac$ `in`
`let!` $b2 = b * b$ `in` $b2 + ac4$

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)

$\mathsf{PushVar}(c); \mathsf{PushVar}(a); \mathsf{MulInt};$
$\mathsf{Var}(ac);$
$\mathsf{PushInt}(4); \mathsf{MulInt};$
...

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 – off 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:

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

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.

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

The $pure(e)$ predicate ensures that the expression has no side effect and the $total(e)$ predicate ensures that the expression can not fail or loop. These conditions can probably only be approximated in practice but works for many common primitive expressions like comparison and bitwise operations, but not for operations that can raise an exception, 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 already handled in the inliner. The last condition, is formalized with the *firstuse* predicate.

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

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.

$$firstuse\ x\ e = first\ x\ False\ e$$

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

3 The abstract machine

The state of the LVM is determined by the current instructions *is*, the stack *st*, and the heap *hp*.

The instruction sequence *is* consists of instructions and arguments. The empty sequence is written as [] and an initial instruction is written as $\mathsf{Instr}(x, y) : is$ where x and y are its arguments. 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 can also contain stack *markers*. These markers take up 2 stack slots. Normally a marker is associated with the value next on the stack. A marker with its value is called a *frame*. There exist three kinds of frames:

upd : p	An update frame. Update the heap value pointed to by p with the value on top of the stack.
cont : is	A continuation frame. Continue with the instructions is with the evaluated value on top of the stack.
catch : p	A catch frame. Continue at the exception handler p when an exception occurs.

The heap *hp* is a dynamic 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 a pointer p with value x is written as $hp \bullet [p \mapsto x]$.

Heap values are tagged. There exist six kinds of heap values:

$\mathbf{instr}(is)$	A sequence of instruction 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 values 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 instructions that reference a global are fixed by the runtime loader to contain the proper heap pointer. For example, if function f has index i, then the initial heap contains $f_i \mapsto \mathbf{instr}(code(i))$ and all instructions that reference function f are updated: $\mathsf{PushCode}(i) \Rightarrow \mathsf{PushCode}(f_i)$. The special value *inv* points to an invalid block of size 0: $inv \mapsto \mathbf{inv}_0$

The semantics of the LVM is given by state transitions (Plotkin, 1981). The initial state of the machine consists of the instructions of the function *main* with an empty stack and an initial heap.

3.1 Basic instructions

We first introduce a minimal set of instructions that support a minimal subset of the LVM language. New instructions are added incrementally with each new language feature. The

minimal subset of the LVM language is called LVM-min and consists 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 local variable (or parameter), PushCode pushes an index 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. In a higher-order language that allows partial applications, it is necessary to use this calling convention. 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. 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 than f is called. This is actually the expression *const id* which 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 on top of the other argument. This leads to a lot of complexity 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, we can not determine without a whole-program analysis whether the parameter f in the *apply* function is partially applied or not. 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 stop with a funtional value as a result.

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

1. termination with a functional value.

Just like partial applications, 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.

	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 the LVM-min is a *tail* call and there is no need to push a return address. It is necessary however to remove any local variables and parameters 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 slided out, they are treated by the argument check as if they are extra parameters!

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 2.4 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 enough arguments. This special case can be optimized with the EnterCode instruction. If a known function is called with enough arguments, the argument check of the called function can be skipped. This is called the 'direct entry point' in the STG machine. The EnterCode instruction performs this optimization and enters a known function with enough 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 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	$p : st$	$hp \circ [p \mapsto \mathbf{nap}(x_1, ..., x_n)]$

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

3.3 Sharing

Allthough the NewNap instruction delays the evaluation of an expression, it isn't 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 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 – indeed, the only whnf values at this moment are functional values and the updateable application node is overwritten with a non-updateable one.

	Code	Stack	Heap
	NewAp(n) : *is*	$x_1 : ... : x_n : st$	hp
$\Longrightarrow$	*is*	$p : st$	$hp \circ [p \mapsto \mathbf{ap}(x_1, ..., x_n)]$
	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
$n > m$	ArgChk(n) : *is*	$f : x_1 : ... : x_m : \mathbf{upd} : p : st$	hp
$\Longrightarrow$	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 fairly expensive. Previously, the number of arguments on the stack was equal to the depth of the stack, but now it seems the 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 2 stack slots: one slot is the real marker while the second is just 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 simply 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:

$$main \;\mapsto\; \mathbf{ap}(main')$$

$main' \mapsto \mathbf{instr}(\mathsf{ArgChk}(0); ...)$

3.4 Recursive values

Recursive values are constructed in two steps: dummy values are allocated first and later initialized, allowing 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.

	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 values, it behaves like the Return instruction. 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. 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.

	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.

3.6 Strict evaluation

Before describing how 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 evaluation of the right-hand side is done, execution is resumed 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 (Peyton Jones, 1992) 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 too few arguments, which is checked by looking at the top frame. If only a plain return address is pushed instead of a frame, the number arguments can't be determined! 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 never reaches this configuration. Indeed, the STG machine has special **seq** frames to support the polymorphic *seq* function of Haskell. This function can be expressed directly in the 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 constructors.

	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. The 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.

	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.

3.8 Synchronous exceptions

Any robust programming language needs to handle exceptional situations. The LVM instruction set should support exception handling at a fundamental level for two reasons. The first reason is efficiency – since exceptional situations are, well exceptional, normal execution shouldn't be penalized. Another reason is that 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 frame on the stack. When an exception is raised, execution is continued at the exception handler. When no exception is raised, the frame is simply ignored by other instructions that look for stack frames, i.e. ArgChk and Return.

	Code	Stack	Heap
	$\mathsf{Catch} : is$	$h : st$	hp
$\Longrightarrow$	is	$\mathbf{catch} : h : st$	hp
$n > m$	$\mathsf{ArgChk}(n) : is$	$x_1 : ... : x_m : \mathbf{catch} : h : st$	hp
$\Longrightarrow$	$\mathsf{ArgChk}(n) : is$	$x_1 : ... : x_m : st$	hp
	$\mathsf{Return} : is$	$x : \mathbf{catch} : h : st$	hp
$\Longrightarrow$	$\mathsf{Return} : is$	$x : st$	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. 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	$hp \bullet [p \mapsto \mathbf{raise}(x)]$
	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.

When the Raise instruction encounters an update frame it updates the value with a **raise** block – indeed, if a value raises an exception it will always raise that exception 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

4 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. This section only gives an overview of all instructions, appendix B gives the precise operational semantics for each instruction.

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 5).

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[1].
b – rec_{32}, bytes record index.

4.1 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.

4.2 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.

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

(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.

4.3 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.

4.4 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).

4.5 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 (see appendix E).
(29) ModInt Euclidean modulus (see appendix E).
(30) QuotInt Truncated Quotient (see appendix E).
(31) RemInt Truncated Remainder (see appendix E).
(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.

4.6 Comparison

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

4.7 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.

4.8 Optimized stack

(53) PushVar0 $\Rightarrow$ PushVar(0)
(54) PushVar1 $\Rightarrow$ PushVar(1)
(55) PushVar2 $\Rightarrow$ PushVar(2)
(56) PushVar3 $\Rightarrow$ 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($n4$)

4.9 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)

4.10 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($c, 3$)

4.11 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($c, 0$).

4.12 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)

4.13 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.

4.14 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.

5 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.

5.1 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.

5.2 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.

5.3 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.

5.4 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**;**
}

5.5 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**;**
}

5.6 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**;**
}

5.7 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;
   ...           ...;
}
```

5.7.1 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;
   ...         ...;
}
```

5.7.2 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*

5.8 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**;**
... ...**;**
}

A Assessment

We have implemented a LVM interpreter on top of the O'Caml runtime system, 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 enriched lambda expressions into LVM files using the compilation rules described in section 2.4. 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 Haskell Light compiler.

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.

```
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)
```

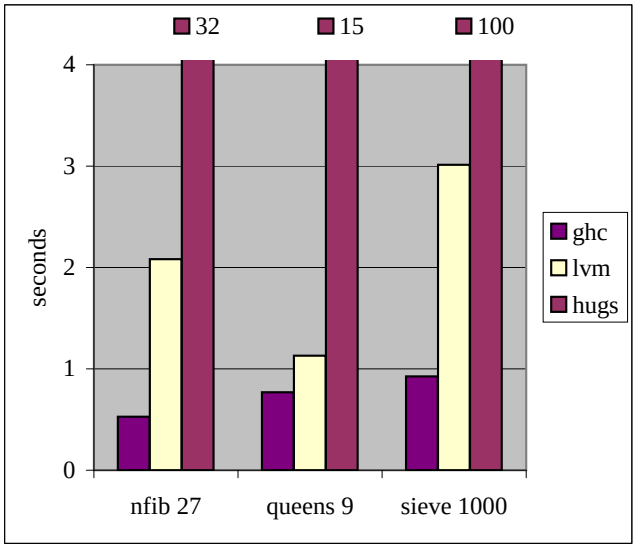

Figure 2: Benchmarks

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 A 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. Off 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
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 A 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

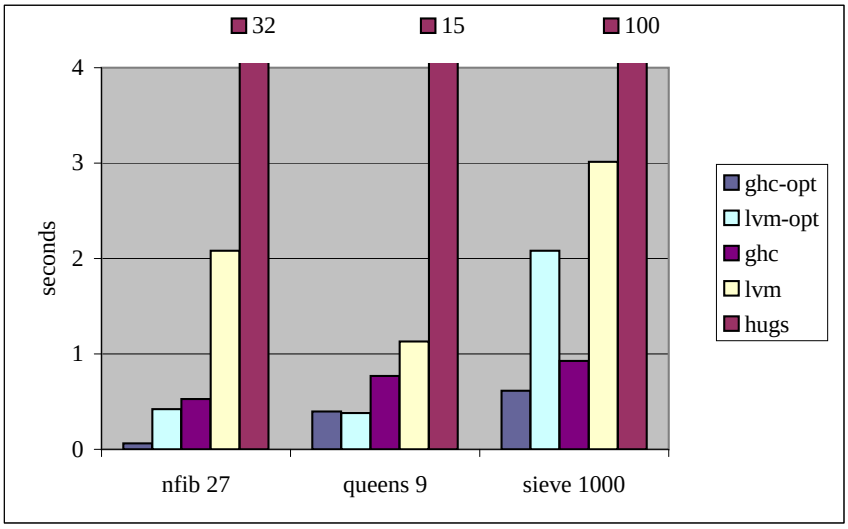

Figure 3: Benchmarks

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 why the *queens* program performs so well, it might be that we have been too smart in strictifying the program 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.

B Instruction reference

B.1 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

C Primitive operations

C.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
```

C.2 Bytes

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

C.3 IO

```
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
};
```

D Floating point

```
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 );
```

E Division and modulus for computer scientists

There exist many definitions of the **div** and **mod** functions in computer science literature and programming languages. Boute (Boute, 1992) describes most of these and discusses their mathematical properties in depth. We shall therefore only briefly review the most common definitions and the rare, but mathematically elegant, *Euclidean* division. We also give an algorithm for the Euclidean **div** and **mod** functions and prove it correct with respect to Euclid's theorem.

E.1 Common definitions

Most common definitions are based on the following mathematical definition. For any two real numbers D (dividend) and d (divisor) with $d \neq 0$, there exists a pair of numbers q (quotient) and r (remainder) that satisfy the following basic conditions of division:

(1) $q \in \mathbb{Z}$ (the quotient is an integer)
(2) $D = d \cdot q + r$ (division rule)
(3) $|r| < |d|$

We only consider functions **div** and **mod** that satisfy the following equalities:

$q = D \,\mathbf{div}\, d$
$r = D \,\mathbf{mod}\, d$

The above conditions don't enforce a *unique* pair of numbers q and r. When **div** and **mod** are defined as functions, one has to choose a particular pair q and r that satisfy these conditions. It is this choice that causes the different definitions found in literature and programming languages.

Note that the definitions for division and modulus in Pascal and Algol68 fail to satisfy even the basic division conditions for negative numbers. The four most common definitions that satisfy these conditions are **div**-dominant and use the same basic structure.

$q = D \,\mathbf{div}\, d = f(D/d)$
$r = D \,\mathbf{mod}\, d = D - d \cdot q$

Note that due to the definition of r, condition (2) is automatically satisfied by these definitions. Each definition is instantiated by choosing a proper function f:

$q = \mathit{trunc}(D/d)$ (T-division)
$q = \lfloor D/d \rfloor$ (F-division)
$q = \mathit{round}(D/d)$ (R-division)
$q = \lceil D/d \rceil$ (C-division)

The first definition truncates the quotient and effectively rounds towards zero. The sign of the modulus is always the same as the sign of the dividend. Truncated division is used by virtually all modern processors and is adopted by the ISO C99 standard. Since the behaviour of the ANSI C functions / and % is unspecified, most compilers use the processor provided division instructions, and thus implicitly use truncated division anyway. The Haskell functions *quot* and *rem* use T-division, just as the integer / and *rem* functions of Ada (Tucker Taft and Duff (eds.), 1997). The Ada *mod* function however fails to satisfy the basic division conditions.

F-division floors the quotient and effectively rounds toward negative infinity. This definition is described by Knuth (Knuth, 1972) and is used by Oberon (Wirth, 1988) and Haskell (Peyton Jones and Hughes (eds.), 1998). Note that the sign of the modulus is always the same as the sign of the divisor. F-division is also a sign-preserving division (Boute, 1992), i.e. given the signs of the quotient and remainder, we can give the signs of the dividend and divisor. Floored division can be expressed in terms of truncated division.

ALGORITHM F:

$$
\begin{aligned}
&q_F = q_T - I \\
&r_F = r_T + I \cdot d \\
&\text{where} \\
&\quad I = \textbf{if}\ \mathit{signum}(r_T) = -\mathit{signum}(d)\ \textbf{then}\ 1\ \textbf{else}\ 0
\end{aligned}
$$

The round- and ceiling-division are rare but both are available in Common Lisp (Steele Jr., 1990). The $\textbf{mod}_R$ function corresponds with the *REM* function of the IEEE floating-point arithmetic standard (Cody et al., 1984).

E.2 Euclidean division

Boute (Boute, 1992) describes another definition that satisfies the basic division conditions. The *Euclidean* or E-definition defines a **mod**-dominant division in terms of Euclid's theorem – for any real numbers D and d with $d \neq 0$, there exists a *unique* pair of numbers q and r that satisfy the following conditions:

$$
\begin{aligned}
&(a)\ \ q \in \mathbb{Z} \\
&(b)\ \ D = d \cdot q + r \\
&(c)\ \ 0 \leq r < |d|
\end{aligned}
$$

Note that these conditions are a superset of the basic division conditions. The Euclidean conditions garantee a unique pair of numbers and don't leave any choice in the definition the **div** and **mod** functions. Euclidean division satisfies two simple equations for negative divisors.

$$
\begin{aligned}
D\ \textbf{div}_E\ (-d) &= -(D\ \textbf{div}_E\ d) \\
D\ \textbf{mod}_E\ (-d) &= D\ \textbf{mod}_E\ d
\end{aligned}
$$

Euclidean division can also be expressed efficiently in terms of C99 truncated division. The proof of this algorithm is given in section E.5.

ALGORITHM E:

$$
\begin{aligned}
&q_E = q_T - I \\
&r_E = r_T + I \cdot d \\
&\text{where} \\
&\quad I = \textbf{if}\ r_T \geq 0\ \textbf{then}\ 0\ \textbf{else if}\ d > 0\ \textbf{then}\ 1\ \textbf{else}\ -1
\end{aligned}
$$

Boute argues that Euclidean division is superior to the other ones in terms of regularity and useful mathematical properties, allthough floored division, promoted by Knuth, is also a good definition. Despite its widespread use, truncated division is shown to be inferior to the other definitions.

An interesting mathematical property that is only satisfied by Euclidean division is the shift-rule. A compiler can use this to optimize divisions by a power of two into an arithmetical

shift or a bitwise-and operation

$$\begin{aligned} D\ \mathbf{div}_E\ (2^n) &= D\ \mathbf{asr}\ n \\ D\ \mathbf{mod}_E\ (2^n) &= D\ \mathbf{and}\ (2^{n-1}) \end{aligned}$$

Take for example the expression, $(-1)\ \mathbf{div}\ (-2)$. With T- and F-division this equals 0 but with E-division this equals 1, and indeed:

$$(-1)\ \mathbf{div}_E\ (-2) = -((-1)\ \mathbf{div}_E\ 2^1) = -((-1)\ \mathbf{asr}\ 1) = 1$$

The LVM implements Euclidean division through the DivInt and ModInt instructions. For completeness, truncated division is also supported by the QuotInt and RemInt instructions.

E.3 Comparision of T-, F- and E-division

The following table compares results of the different division definitions for some inputs.

(D, d)	(q_T, r_T)	(q_F, r_F)	(q_E, r_E)
$(+8, +3)$	$(+2, +2)$	$(+2, +2)$	$(+2, +2)$
$(+8, -3)$	$(-2, +2)$	$(-3, -1)$	$(-2, +2)$
$(-8, +3)$	$(-2, -2)$	$(-3, +1)$	$(-3, +1)$
$(-8, -3)$	$(+2, -2)$	$(+2, -2)$	$(+3, +1)$
$(+1, +2)$	$(\ 0, +1)$	$(\ 0, +1)$	$(\ 0, +1)$
$(+1, -2)$	$(\ 0, +1)$	$(-1, -1)$	$(\ 0, +1)$
$(-1, +2)$	$(\ 0, -1)$	$(-1, +1)$	$(-1, +1)$
$(-1, -2)$	$(\ 0, -1)$	$(\ 0, -1)$	$(+1, +1)$

E.4 C sources for algorithm E and F

This section implements C functions for floored- and Euclidean division in terms of truncated division, assuming that the C functions / and % use truncated division. Note that any decent C compiler optimizes a division followed by a modulus into a single division/modulus instruction.

```
/* Euclidean division */
long divE( long D, long d )
{
  long q = D/d;
  long r = D%d;
  if (r < 0) {
     if (d > 0) q = q-1;
           else q = q+1;
  }
  return q;
}

long modE( long D, long d )
{
  long r = D%d;
```

```
  if (r < 0) {
    if (d > 0) r = r + d;
          else r = r - d;
  }
  return r;
}

/* Floored division */
long divF( long D, long d )
{
  long q = D/d;
  long r = D%d;
  if ((r > 0 && d < 0) || (r < 0 && d > 0)) q = q-1;
  return q;
}

long modF( long D, long d )
{
  long r = D%d;
  if ((r > 0 && d < 0) || (r < 0 && d > 0)) r = r+d;
  return r;
}
```

E.5 Proof of correctness of algorithm E

We prove that algorithm E is correct with respect to Euclid's theorem. First we establish that T-division satisfies the basic division conditions. The first two conditions follow directly from the T-definition.

condition (1) :
$q_T = trunc(D/d) \in \mathbb{Z} \quad \square$

condition (2) :
$r_T = D - d \cdot q_T \equiv D = r_T + d \cdot q_T \quad \square$

condition (3) :
$|r_T| = \quad \{def\}$
$|D - d \cdot q_T| = \quad \{def\}$
$|D - d \cdot trunc(D/d)| = \quad \{math\}$
$|d \cdot (D/d - trunc(D/d))| < \quad \{|D/d - trunc(D/d)| < 1\}$
$|d| \quad \square$

Any division that satisfies Euclid's conditions also satisfies the basic division conditions since these are a subset of Euclid's conditions. Given the properties of T-division, we can now prove that algorithm E is correct with respect to Euclid's theorem.

condition (*a*) :
$q_E = q_T - I \in \quad \{(1) \wedge I \in \mathbb{Z}\}$
$\mathbb{Z} \quad \square$

condition (*b*) :
$D = \quad \{(2)\}$
$d \cdot q_T - r_T = \quad \{math\}$

$$
\begin{array}{ll}
d \cdot q_T - d \cdot I + r_T + d \cdot I = & \{math\} \\
d \cdot (q_T - I) + (r_T + I \cdot d) = & \{def\} \\
d \cdot q_E + r_E \quad \square &
\end{array}
$$

condition (c) :

$$
\begin{array}{ll}
r_E = & \{def\} \\
r_T + I \cdot d = & \{math\}
\end{array}
$$

if $(r_T \geq 0)$ **then** $I = 0$

$$
\begin{array}{ll}
r_T \Rightarrow & \{(3) \wedge r_T \geq 0\} \\
0 \leq r_E < |d| &
\end{array}
$$

if $(r_T < 0 \wedge d < 0)$ **then** $I = -1$

$$
\begin{array}{ll}
r_T - d \Rightarrow & \{(3) \wedge r_T < 0 \wedge d < 0\} \\
0 \leq r_E < |d| &
\end{array}
$$

if $(r_T < 0 \wedge d > 0)$ **then** $I = 1$

$$
\begin{array}{ll}
r_T + d \Rightarrow & \{(3) \wedge r_T < 0 \wedge d > 0\} \\
0 \leq r_E < |d| \quad \square &
\end{array}
$$

References

Raymond T. Boute. *The Euclidean definition of the functions div and mod*. In ACM Transactions on Programming Languages and Systems (TOPLAS), 14(2):127–144, New York, NY, USA, April 1992. ACM press.

W. J. Cody et al. *A proposed radix- and word-length-independent standard for floating-point arithmetic*. In IEEE Micro, 4(4):86–100, August 1984.

Graham Hutton and Erik Meijer. *Monadic parser combinators*. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps.

Thomas Johnsson. *Lambda lifting: transforming programs to recursive equations*. In Functional programming languages and computer architecture (FPCA), New York, NY, USA, September 1985. Springer-Verlag.
http://www.citeseer.nj.nec.com/johnsson85lambda.html.

Donald. E. Knuth. *The Art of Computer Programming, Vol 1, Fundamental Algorithms*. Addison-Wesley, 1972.

Simon Peyton Jones and John Hughes (eds.). *Report on the language Haskell'98*, February 1998. http://www.haskell.org/report.

Simon Peyton Jones and Simon Marlow. *Secrets of the Glasgow Haskell Compiler inliner*. submitted to the Journal of Functional Programming, 2002.
http://www.research.microsoft.com/~simonpj/Papers/papers.html#compiler.

Simon Peyton Jones, Will Partain, and Andre Santos. *Let-floating: moving bindings to give faster programs*. In International Conference on Functional Programming (ICFP'96), May 1996. http://www.research.microsoft.com/~simonpj/Papers/papers.html#compiler.

Simon Peyton Jones. *Implementing non-strict languages on stock hardware: The Spineless Tagless G-machine*. Journal of Functional Programming, 2(2):127–202, April 1992.
http://www.research.microsoft.com/~simonpj/Papers/papers.html#compiler.

Gordon D. Plotkin. *A structural approach to operational semantics*. Technical Report DAIMI FN-19, Computer Science department, Aarhus University, September 1981.

Guy L. Steele Jr. *Common LISP: The Language, 2nd edition*. Digital Press, Woburn, MA, USA, 1990. ISBN 1-55558-041-6.

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

Niklaus Wirth. *The programming language Oberon*. Software Practice and Experience, 19(9), 1988. The Oberon language report. http://www.oberon.ethz.ch.