The Movitz development platform

As of December 2003

Frode V. Fjeld

frodef@cs.uit.no

July 10, 2014

$Id: movitz.tex,v 1.1 2004/01/13 11:05:04 ffjeld Exp $

1 Introduction

Movitz consists of a Common Lisp compiler[1], a run-time environment, a library of operating system-related functionality, and debugging and monitoring tools. Movitz is, among other things, an attempt to bring exploratory programming and easy prototyping and development to embedded and kernel-level programming.

This document provides a technical overview of the Movitz platform, and is expected to evolve with the software project.

2 The Movitz run-time environment

The Movitz run-time environment is the core software components and the regime of conventions that are in effect on the target machine. It is designed for the x86 PC architecture, running in 32-bit protected mode. It is self-sufficient, in the sense that it relies on no software services other than those implemented as part of Movitz itself, and aims to support kernel-level software. The primary design goal for the run-time environment is to be as efficient and flexible as possible, without imposing unnecessary burdens or restrictions on any direction of application-level design that might be built over it. To this end, it defines a "run-time context" that holds all computational state that does not fit naturally on the control stack, and a flexible and efficient procedure call protocol that conforms to the Common Lisp specification.

---

[1]Movitz implements a growing subset of the ANSI Common Lisp standard (X3.226-1994), a hypertext version of which is on-line at http://www.lispworks.com/reference/HyperSpec/.

2.1 Run-time context

The "run-time-context" is the most central data structure of the Movitz run-time environment. The CPU's **edi** register is designated to always point to this structure. The same pointer value also constitutes the NIL value, so that this often-used lisp value is always immediately available from the **edi** register. Various aspects of the run-time environment, as outlined in the following sections, are located in the run-time-context, and that way they are readily available through the **edi** register. A visual representation of the run-time-context is shown in figure 5 on page 20.

2.1.1 Primitive functions

Primitive functions are assembly-level sub-routines that implement certain often-used functionalities. These functions do *not* follow neither the standard function-call protocol (section 2.4) nor function result protocol (section 2.5). Consequently, the compiler must know how to invoke each primitive function. This knowledge is typically embedded in a compiler-macro[2]. For example, the CAR function has a compiler-macro that expands into assembly-code that calls the primitive function "fast-car", which implements the CAR operation. Also, lookup of dynamic variables is performed by a primitive function.

The performance gain from primitive functions, as compared to normal functions, stems from the immediate availability of their code-vector through **edi** (rather than having to go through a symbol and funobj), and the freedom to use a tailor-made function-call protocol.

2.1.2 Global constants

Certain lisp objects are of such a nature that they are assigned a slot in the "run-time-context", in order to be readily available to code anywhere. These include e.g. the symbol T, and the unbound value (which must be compared against at every reference to a potentially unbound slot).

2.1.3 Dynamic environment

The dynamic environment consists of dynamic variable bindings, active catch tags and unwind-protections. These are implemented in an (intertwined) linked list structure (i.e. not a standard lisp list), that is located on the stack. This list's head is kept in the "run-time-context". Thus, the dynamic environment is captured entirely by the stack and the "run-time-context". Hence, the basic Movitz run-time environment adds very little (if any) overhead to context-switching mechanisms built on top of it.

---

[2]Section 3.2.2.1 of the Hyperspec describes compiler-macros.

2.2   Multiple run-time contexts

Any form of multi-tasking (threads, processes, or SMP) requires the ability to change between execution contexts. This involves, at least, switching the stack and the "run-time-context". However, a new value cannot simply be loaded into **edi**, since this register is assumed to hold the NIL value, whose identity must be preserved. This means that the value in **edi** must be truly constant, even across threads, processes, and CPUs.

The x86 architecture includes a segmented memory architecture. The compiler can ensure that every reference to the run-time-context is using a designated segment register. Thus the switching between different run-time-contexts can be implemented by loading the proper value into this segment register. The downside to this strategy is that overriding the segment for a particular instruction incurs a certain performance penalty each time[3].

A similar approach would be to utilize a different part of the x86 memory management subsystem, namely the paging virtual memory system. The idea is conceptually the same as that of the segment register mechanism: Switching the run-time-context by mapping different physical memory pages to the global run-time-context memory address range. This mechanism has at least two downsides, however: First, there is a substantial performance overhead involved in changing the virtual memory layout. Second, dealing with virtual memory is complex, and might interfere with kernel applications use of virtual memory. On the other hand, in some applications a context switch might involve a remapping of virtual memory remapping anyway, such that integrating a switch of run-time-context can be easily integrated, at little extra cost.

An alternative strategy for implementing multiple execution contexts would be to install a run-time-context by simply copying it into the "active" run-time-context memory area. This copying operation may be substantially optimized by noting that most of the elements of the run-time-context will indeed be constant, and expectedly only a handful of elements are actually required to capture the dynamic context. This scheme has the advantage of being conceptually very simple, and doesn't make use of any "exotic" CPU features. However, in a multi-CPU environment where true concurrency is involved, each CPU will still require a private constant block. This can be accomplished using either the segment register or virtual memory, as described above.

2.3   Function objects

Much like the run-time-context constitutes the global (or at least thread-wide) running context, each function has a local object, which we call a "funobj", that holds contextual information local to the function. The current function's funobj is always loaded in the **esi** register (We say about the use of this register

---

[3]The exact quantification of this penalty is unclear and specific to the CPU model. The *Pentium 4 Optimization Reference Manual* states simply: "*Do not use many segment registers*", so there may in fact be no associated performance penalty. But at the very least there is the cost of 1 extra byte in the instruction encoding for overriding the segment register.

in section 2.4). The funobj includes a reference to the function's code-vector (actually, several code-vectors, as explained in section 2.4.2), and the function's name, if any. A code-vector is a vector with element-type (UNSIGNED-BYTE 8) whose contents is the machine-code that implements the function. A function's name is typically a symbol, but can also be a list such as (SETF MY-ACCESSOR) or (METHOD MY-GF (MY-CLASS T)).

The funobj is also a variable-sized, vector-like container for various objects that the function's code may reference. Most importantly, this includes every non-immediate value (i.e. heap objects) that the function references explicitly, such as strings and symbols. This traditional lisp compilation technique facilitates the work of the garbage collector, which otherwise would have to parse the function's machine code in order to find and modify its heap references. For funobjs that are lexical closures, the code references closed-over variables[4] indirectly through slots in the funobj's vector. Finally, the funobj vector may contain so-called "jumpers". These are raw instruction pointers (i.e. not lisp values) that point to somewhere inside the code-vector, typically corresponding to assembly-level labels. Jumpers are used to implement computed gotos, and require support by the garbage collector because these pointers must be updated whenever the code-vector is moved to another address.

2.3.1 Local functions

Local functions are created with operators such as LAMBDA, FLET, or LABELS. Local functions are represented by the same funobj data-type as normal functions. However, while normal functions have the same extent (life-time) as any other lisp object, there are four types of extent for local closures, listed here in exclusive-to-inclusive and inexpensive-to-expensive order:

Null extent An unused local function has null extent, because it will never be created.

Lexical extent When a local function is known to only be *called*, meaning it is never captured with the FUNCTION operator and passed of to somewhere outside the current lexical scope, it has lexical extent.

Dynamic extent The local functions that are known to exist only in the dynamic scope of its defining form, have dynamic extent.[5]

Indefinite extent Otherwise, the default is for local functions to have indefinite extent, just like any other normal, heap-allocated lisp object.

Because lexical-extent closures need never exist as lisp objects, the compiler is free to utilize more efficient representations than funobjs. Also, the standard function-call protocol (section 2.4) does not have to be obeyed. For example,

---

[4]These are termed "borrowed bindings" when seen from the viewpoint of the closure that references the bindings, or "lended bindings" when seen from the viewpoint of whoever establish such bindings.

[5]See the CLHS entry for the DYNAMIC-EXTENT declaration.

lexical-extent closures could be inlined, or implemented as assembly-level sub-routines.

2.4 Function-call protocol

The Movitz function-call protocol implements the full semantics of Common Lisp function calls. This includes the dynamic, late-binding behavior that enables the redefinition of functions at run-time, and the ability to pass and receive an arbitrary number of arguments.

All registers are caller-save, with two exceptions: **edi** is constant, so it does not need to be saved nor restored. Secondly, **esi** (which always points to the current function's funobj) is saved by the caller as it enters its stack-frame, but it is the responsibility of the callee to restore it.

Table 1 sketches the details of the function-call protocol. The syntax **(ebp-4)** denotes the contents of the address that is contained in the register **ebp**, offset by minus four.

Register	Meaning
ecx	Number of arguments, non-linear.
esi	Callee's function object
(ebp-4)	Caller's function object
eax	First argument
ebx	Second argument
(esp)	Return address
(esp-4)	Third argument
(esp-8)	Fourth argument, etc.
edx	Callee's name (a symbol), when available.

Table 1: Register usage during function calls.

The **edx** register is not a strictly required part of the function-call protocol, but it will often contain the function name, a fact that is exploited for debugging purposes. The Movitz compiler extends function lambda-lists with the &EDX keyword, which names a variable that will be bound to the value of **edx** as the function is called.

Figure 1 shows the stack layout right after a function-call has occurred. The stack is depicted as growing downwards. The two first arguments, a_0 and a_1, are not required to be present on the stack, as they are assigned to the registers **eax** and **ebx**, respectively. However, in practice (because the compiler must observe Lisp's left-to-right evaluation rule) they will sometimes be present above a_2, a situation that can be detected and exploited in debugging.

2.4.1 Non-linear encoding of ecx

The register **ecx** holds the number of arguments during a function call (with a few exceptions, as explained in the next sections). With the x86 instruction-set,

$$
\begin{array}{|c|l|}
\hline
\vdots & \\
\hline
\mathbf{ebp}_\uparrow & \leftarrow (\mathbf{ebp}_f) \\
\hline
\mathrm{funobj}_f & \leftarrow (\mathbf{ebp}_f - 4) \\
\hline
local0_f & \leftarrow (\mathbf{ebp}_f - 8) \\
\hline
local1_f & \cdots \\
\hline
\vdots & \\
\hline
a_2 & \leftarrow (\mathbf{ebp}_g + 4(n-1)) \\
\hline
a_3 & \leftarrow (\mathbf{ebp}_g + 4(n-2)) \\
\hline
\vdots & \cdots \\
\hline
a_{n-2} & \leftarrow (\mathbf{ebp}_g + 12) \\
\hline
a_{n-1} & \leftarrow (\mathbf{ebp}_g + 8) \\
\hline
\mathbf{eip}_f & \\
\hline
\mathbf{ebp}_f & \leftarrow (\mathbf{ebp}_g) \\
\hline
\mathrm{funobj}_g & \leftarrow (\mathbf{ebp}_g - 4) \\
\hline
local0_g & \cdots \\
\hline
\vdots & \\
\hline
\end{array}
$$

Figure 1: A map of the stack, including f and g's stack-frames, after f has called g with the n arguments $a_0 - a_{n-1}$. $\mathbf{eip}_f$ denotes the return-address, whereas $\mathbf{ebp}_\uparrow$ denotes the stack-frame pointer of the function that called f.

it is relatively expensive, in terms of code-size, to load a 32-bit immediate value into a register. It is much cheaper to load an eight-bit value into the lower eight bits of a register. Whereas the vast majority of function calls will be with fewer than 256 arguments, we do not want to impose this as a hard limit for CALL-ARGUMENTS-LIMIT. Our best-of-both-worlds solution is to employ a non-linear coding of the argument count in **ecx**.

The non-linear encoding separates the argument-counts into two ranges: The small range and the big range. The small range fits in eight bits, and is loaded into the lower eight bits of **ecx** (which with x86 are directly accessible as the **cl** register), with complete disregard of the upper 24 bits. For big-range argument-counts, an escape value (255) is loaded in the lower eight bits of **ecx**, and the actual count in the upper 24 bits. Because the escape value is greater than any other small-range value, this encoding preserves the values' ordering, such that a single eight-bit compare is sufficient to establish whether the call-count is equal to or below some number within the small range.

n	**ecx** bits 8–31	**ecx** bits 0-7
0–127	don't care	n
128–2^{24}	n	255

Figure 2: The non-linear encoding of argument-count n in **ecx**.

The penalty incurred by this scheme is that any function that can accept more arguments than what fits in the small range, must spend a few instructions to normalize **ecx**. This includes any function with an &REST parameter.

2.4.2 Optimized function-calls

Bot static and dynamic examination of programs shows that more than 90% of all function-calls involve 1, 2, or 3 arguments. Therefore, these cases have been specially optimized with modified function-call protocols.

Lambda-list	%1	%2	%3
(x)	✓		
(x &optional y)	✓	✓	
(x y)		✓	
(x y &optional z)		✓	✓
(x y z)			✓

Table 2: The combinations of lambda-lists and call-counts that are currently being automatically optimized by the compiler. NB: After a compiler re-write, the lambda-lists involving `&optional` are no longer actually optimized. These optimizations will hopefully be added again later.

Every function's funobj contains a reference to that function's code-vector. This code-vector holds the machine-code that implements the function, obeying the function-call protocol as described in section 2.4. However, if the argument-count is 1, 2, or 3, each of these cases have a separate code-vector entry-point associated with them. The function-call protocol for these entry-points is identical to the normal protocol, except that **ecx** is not used, since the argument-count is implied. Not only does this save the caller the overhead of loading **ecx**, but the callee may now also save at least one compare and branch operation, since there is no need to examine **ecx**.

There is also support for manual implementation of optimization of the dispatch on a function's number of arguments. This is provided by the special operator NUMARGS-CASE, which may only appear as the first form in a function. This operator allows the programmer to associate completely independent procedures with each of the function's four different entry-points. Still, although the procedures are independent, they are eventually merged into the same code-vector and the same funobj, so that apart from the separate entry-pointers, such functions look exactly the same as any other function to the run-time system. This feature is intended for "under-the-hood" optimizations, not user-level programs.

For functions that don't implement such optimized code-vector entry-points (that is, functions whose lambda-list the compiler has not been taught how to optimize, and do not employ NUMARGS-CASE), these special-case code-vectors will point to global code-stubs that load **ecx** with the correct value, and then jump to the function's normal entry-point. One of these code-stubs is shown

```
(defun find-if (predicate sequence &key from-end (start 0)
                                        end (key 'identity))
  (numargs-case
   (2 (predicate sequence)
      #| Code for the simplest case of finding an element
         matching <predicate> in <sequence>. |#)
   (t (predicate sequence &key from-end (start 0)
                               end (key 'identity))
      #| Code for the full-fledged find-if. |#)))
```

Figure 3: An example use of NUMARGS-CASE taken from the implementation of the FIND-IF function. Here, the entry-point for two arguments may skip all keyword parsing, and assume the default values for the keyword arguments when performing the search.

in figure 4. Note that while this re-direction certainly represents a performance overhead, it is something that is expected to happen relatively rarely.

```
(define-primitive-function trampoline-funcall%2op ()
  "Call a function with 2 arguments"
  (with-inline-assembly (:returns :nothing)
    (:movb 2 :cl)
    (:jmp (:esi -6))))
```

Figure 4: The function-call trampoline stub function for functions that don't implement the two-arguments optimized protocol. This code is located in the file losp/cl/run-time.lisp.

2.5   Function result protocol

Functions may return any number of values. However, the majority of functions return one value, and—even more importantly—most function calls are evaluated for their primary value exclusively[6]. The function result protocol takes these facts into consideration. The CPU's carry flag (i.e. the **CF** bit in the **eflags** register) is used to signal whether anything other than precisely one value is being returned. Whenever **CF** is set, **ecx** holds the number of values returned. When **CF** is cleared, a single value in **eax** is implied. A function's primary value is always returned in **eax**. That is, even when zero values are returned, **eax** is loaded with NIL. This way, a function-call's primary value is always found in **eax**, without further ado on the caller's part.

---

[6]Common Lisp defines the primary value of an form to be the first value it returns, or, if it returns zero values, the value NIL.

n	**CF**	**eax**	**ebx**	**ecx**	**edi**
0	1	NIL	-	0	-
1	0	v_0	-	-	-
2	1	v_0	v_1	2	-
$3, 4, 5, \ldots$	1	v_0	v_1	n	$(\mathbf{edi}+z) \to v_2$
					$(\mathbf{edi}+z+4) \to v_3$
					$(\mathbf{edi}+z+8) \to v_4$
					$\vdots$

Table 3: Register and carry-flag usage in the function result protocol for returning the n values v_0 to v_{n-1}. The offset of the "values" section of the run-time-context is denoted by z.

As can be seen from table 3, the register **ebx** is used to hold the function's secondary value when two or more values are being returned. From the ternary value onwards, the values are placed in thread-local storage, which is to say a designated section of the run-time-context.

2.6 Dynamic typing

Common Lisp is a dynamically typed language, meaning that type is attached to values rather than variables, and that any value's type is available at run-time. Movitz implements the full dynamic semantics of Common Lisp's type system, as documented in this section.

The traditional lisp approach of assigning a few bits of the machine word is employed. Movitz uses the three least significant bits of x86's 32-bit words to encode type information, according to table 4.

Tag	*Tag$_2$*	*Type*
0	000	even fixnum
1	001	cons
2	010	character
3	011	*not assigned*
4	100	odd fixnum
5	101	null
6	110	other heap object
7	111	symbol

Table 4: Type tags.

The tag value 3 is not being used, and is available for use e.g. by a scheme for garbage collection. The fixnum and character types are immediates, meaning their value is encoded in the 32-bit word per se. For the non-immediate types, the word is a pointer to the memory location that holds the object's information.

2.7 Memory management

2.7.1 Dynamic memory allocation

The one major shortcoming of the current run-time environment, is the lack of decent memory management, and in particular a garbage collector. One reason why this has been put off, is the fact that memory management is an area that is expected to overlap with many kernel applications. That is, a kernel design might include concepts and mechanisms that put specific requirements on the run-time environment's memory management strategy.

We expect to address this problem in the following way: Memory management will nominally be left out of the core Movitz run-time environment. However, the hooks required for implementing memory management modules will be provided, as well as example implementations that will provide basic memory management for applications that do not have special requirements.

Currently, a very simple allocation scheme provides for dynamic memory allocation. A pointer is initialized to the start of the free heap at boot-time, and objects are simply allocated sequentially from this heap. No provision is made for what happens when the heap runs out of space, or any form of garbage collection. The ROOM function reports how far the heap allocation-pointer has proceeded.

3 The compiler

The Movitz compiler targets the x86 instruction architecture, and the Movitz run-time environment. The compiler is implemented in Common Lisp, and has not been designed to be retargetable to other CPU architectures.

3.1 Movitz images

The Movitz compiler is a cross-compiler. While the compiler qua lisp program lives in the normal lisp world of whatever Common Lisp implementation it is running, it targets a virtual lisp world that is a symbolic representation of a native Movitz lisp world. This is called a Movitz "image". The current Movitz image is bound to the special variable *IMAGE*. This object defines all the aspects of the target lisp world, including references to all its objects, such as symbols, functions, lists, etc.

There are several kinds of images. The compiler usually works with a particular kind, called a "symbolic image", which is implemented purely as a lisp data-structure. Other kinds of images are mostly used for inspection and debugging. This includes "file images" (interfacing finalized bootable kernel images in a file), and—more interestingly—"bochs images" that interfaces (via the unix procfs mechanism) a Bochs PC emulator[7] running a Movitz image.

One important aspect of any image is that it defines an address space. When an object is loaded into a symbolic image ("interned" is the term being used),

---

[7]http://bochs.sourceforge.net

it is given an address in this address space. The address-space mapping is two-way, meaning that either an object's address, or the object at a given address can be requested. Symbolic images simulate the address-space using two hash-tables, whereas file images translate addresses to file positions, and bochs images interface the emulated machine's address-space.

Symbolic images may be "dumped", a process that transforms the symbolic representation to binary form and saves it to a bootable kernel image. The compiler can work incrementally with a symbolic image, so that individual functions may be recompiled (or new functions added) before the symbolic image is dumped again for a new test-run. Also, functions and other objects may be transported from one image to another, for example from a symbolic image to an image running on a machine or emulator, thus providing true dynamic compilation. This has in fact been tried, using a simple (IPv6 UDP over Ethernet) network protocol for communication.

3.2   Compiler internals

This section provides an introduction to the implementation details of the Movitz compiler. It is expected to be useful primarily to anyone who wishes to study the source-code for the compiler, and is not essential for those who wish to just use the Movitz system.

The Movitz compiler ultimately produces x86 assembly code, suitable for input to an assembler that generates machine-code. However, the compiler mostly works with what we will call "extended assembly", which is normal x86 assembly extended with a few pseudo-instructions that facilitate symbolic references and code analysis at later stages in the compiler.

The compiler is implemented in three conceptually separated modules: The compiler protocol, the core compiler, and the special operator compilers, each detailed below.

3.2.1   The compiler protocol

The compiler protocol is meta-code that is concerned with managing the information flow in the compiler[8]. That is, it defines a scheme for passing the current (downstream) compilation context (the form to be compiled, the current lexical environment, etc.) around. Similarly, there is a scheme for the (upstream) return values, which consists of compiled code and its meta-data, such as the types and whereabouts of the results from the code. These two schemes are primarily what is referred to as the "compiler protocol". The compiler protocol is supported by macros for defining (`define-compiler` and `define-special-operator`) and calling compiler functions (`compiler-call`), and for returning and binding their return values (`compiler-values` and `compiler-values-bind`, respectively).

One important aspect of the compiler protocol is the downstream "result-mode" and upstream "returns" values. Whenever a form is compiled, a result-

---

[8]Source-code file `compiler-protocol.lisp`.

mode parameter must be provided. This expresses a wish for where the result of the compiled code should end up, and must be one of a finite set of result-mode designators that the compiler knows about, for example denoting a machine register, a stack push, a lexical binding, or the full-fledged function result protocol described in section 2.5. The result-mode may also indicate that the form's result is to be ignored, and compiled for side-effects only. However, the result-mode is merely a *request*, and the actual code-producing function is at liberty to ignore it. Every code-producing function must also return a "returns" parameter, which declares where the code actually produces a value. A special "nothing" value indicates that the code produces no values anywhere.

The result-mode and return-mode designators are overlapping, but not equal sets. One special result-mode designator, "function", is used also to recognize when the compiler context is at the function's tail position. This piece of information is useful for various optimizations. For example, there is no need to restore the stack-pointer after a function call if we are about to exit the current stack-frame anyway. The transformation of (recursive) tail-calls to jumps is another optimization technique that requires discovery of the tail position, but this is not yet implemented. "Function" is not a legal return-mode designator, but is normally matched by the "multiple-values" designator, which denotes the function result protocol that is described in section 2.5.

In addition to specifying what amounts to storage slots for lisp values, more abstract modes can be specified both upstream and downstream. Particularly useful are the boolean modes, because it is quite common that forms are evaluated not for their value per se, but rather for their boolean status[9]. There are two downstream boolean modes: "branch on true" and "branch on false", both of which take an assembly label as parameter. For example, the function that produces code for the NIL object[10] knows how to deal with the "branch on false" result-mode: Generate an unconditional branch to that label. Upstream, code-producers may specify that the code is boolean true given a certain x86 **eflags** status, such as "true when **ZF**=1". This translates directly to a corresponding branch instruction, and is thus easily glued with a "branch on true" or "branch on false" request.

In general, the result/return-mode part of the compiler protocol determines to a large extent the quality of the code produced by the compiler. For example, a (hypothetical) very primitive, stack-based compiler would know only one result-mode, namely "push", and one return-mode, "pop", and thus produce rather inefficient code. On the other hand, a very clever compiler would know about most every kind of source and destination, every combination of how to move values from one to the other, and all this while taking type and control information from the compilation environment into account. While the Movitz compiler is currently considerably smarter than a simple push-and-pop compiler, it is still expected to improve in this area.

---

[9]Common Lisp specifies that NIL is boolean false, while any other value is boolean true.
[10]That function is `compile-self-evaluating`.

3.2.2 The core compiler

The core compiler implements the basic compilation control and code-generation[11]. The functions here can be roughly divided into four categories: First, dispatching compilers that examine the form or other aspects of the current compilation context, and forwards this to a more specialized function. Second, there are code-producing functions that actually produce assembly-code. These two categories of functions both adhere to the compiler protocol. Third, there are code-producing helper functions, usually with a name on the form `make-compiled-<foo>`. These functions do not follow the compiler protocol. Finally, there are miscellaneous helper functions.

One central function that fall into the "miscellaneous helper functions" category, is the function `make-result-and-returns-glue`. This function is used to match a compiler-protocol result-mode (as desired by the compiler caller) value with a returns mode (as provided by the code producer). That is, this function knows which modes are compatible, which are not, and how to generate the glue code that is required for a particular match. Another interesting function in this category, is `optimize-code`, which implements an assembly-level peephole optimizer.

There are two functions which are normally used to invoke the compiler on a form: COMPILE-FORM and COMPILE-FORM-UNPROTECTED. They differ in that the former is guaranteed to honor the requested result-mode (by adding some glue code if necessary), whereas the latter is not.

<div style="float:right; font-variant:small-caps; font-size:small">COMPILE-FORM
COMPILE-FORM-
UNPROTECTED</div>

The compilation of a function's symbolic definition to assembly code is performed in two passes, where the first pass produces extended assembly code. The second pass translates the extended assembly code to a code-vector and finalizes the funobj and other objects that are part of the function's representation.

3.2.3 The compiler's first pass

For each function, the following occurs:

1. Compile initialization code for "complicated" function arguments. That is, all arguments that are not required arguments. This code checks whether the argument is provided, and if not arranges for the default init-form to be evaluated.

<div style="float:right; font-variant:small-caps; font-size:small">MAKE-FUNCTION-
ARGUMENTS-INIT</div>

2. The function's body forms are compiled, with COMPILE-FORM, to extended assembly code.

This process is recursive, in the sense that whenever a local function is created (i.e. with LAMBDA, FLET, or LABELS), the first-pass compiler is applied to the local function. Hence, the result of the first-pass compilation is—conceptually—a tree of functions.

---

[11]Source-code file `compiler.lisp`.

The functions that are involved in the first-pass compilation process, are (or should be) strictly functional. That is, they should not side-effect the funobj or other such objects they receive from "upstream" in the compiler. This is because it should be safe to apply any of the compiler functions[12] to the same form more than once (with different parameters), so as to chose the better code produced. This would not work properly if one of the compilations that in the end wasn't chosen to be included modified the funobj or lexical environment.

3.2.4 The compiler's second pass

1. The first-pass input is analyzed for its use of local functions in order to determine their type and extent. Local functions that don't close over any aspect of the lexical environment become simply "local funobjs" that behave just like any other self-evaluating object. Proper local closures, however, require more complex handling, depending on their extent, as described in section 2.3.1.

2. The code produced so far is analyzed for its use of lexical variables. Local variables are assigned slots in the function's stack-frame. Non-local lexical variables (i.e. closed-over variables that are "borrowed¨ from some lexically enclosing function) are recognized and remembered. Also, the sets of non-immediate objects and jumper tables used by the code are determined.

   FUNOBJ-ASSIGN-BINDINGS

   CODE-CONSTANTS-AND-JUMPERS

3. The funobj's tail vector is laid out, in the following sequential pattern (see section 2.3 for details about the contents of funobjs):

   FINALIZE-FUNOBJ

   (a) Jumper tables.
   (b) Slots for borrowed bindings.
   (c) Constants.

   The number of jumpers is encoded in the funobj, so that the run-time system (i.e. the garbage collector) may recognize the jumpers as such.

4. At this point, all the symbolic references in the extended assembly-code produced so far should be resolved. This extended assembly code is now translated to actual assembly code.

   FINALIZE-CODE

3.2.5 Special forms

The compilation of special operators is dispatched to the function that is responsible for compiling that operator. These functions adhere to the compiler protocol. Also, they are registered in a special name-space such that the compiler can automatically dispatch to the correct special operator compiler when it encounters a special form. The macro DEFINE-SPECIAL-OPERATOR is thus a thin layer over DEFINE-COMPILER that registers the function in this name-space.

---

[12]I.e. the operators defined with DEFINE-COMPILER.

The special operators defined in Common Lisp are located in a separate file from other Movitz-specific special operators[13]. The special operator compiler functions work in precisely the same manner as the core compiler. They are only distinguishable from the core compiler in the way they are automatically hooked into the compiler's form dispatch.

Among the Movitz-specific special operators, probably the most interesting is WITH-INLINE-ASSEMBLY. This operator provides a short-cut for inserting assembly code directly in a lisp form. This is particularly useful in conjunction with compiler-macros. Arguably, a substantial part of the Movitz compiler consists of compiler-macros that expand to assembly code by way of this operator. The WITH-INLINE-ASSEMBLY special operator takes declarative parameters that enables the compiler to glue the provided assembly code with the current compilation context, according to the compiler protocol. Note that the assembly code injected with this operator is not considered "foreign" code. That is, the assembly code must observe all the rules of the Movitz run-time system.

3.3 The file compiler and the boot process

The file compiler is still rather primitive, and has not been implemented with much regard to the standard COMPILE-FILE semantics. Multi-file systems are currently defined by way of REQUIRE and PROVIDE forms, which also have extended semantics wrt. the standard.

Basically, the file compiler makes each file into a function, with a name computed from the file's pathname. That function's forms consists of the top-level forms in the file. Also, the function is registered as a file function with the current image (section 3.1). When the image is dumped, a function named TOPLEVEL-FUNCTION is automatically constructed, which simply calls each of the file functions in turn, according to their priority (the PROVIDE form is extended to allow a file priority to be specified). The dump process arranges for TOPLEVEL-FUNCTION to be called when the image is booted.

4 Libraries

The Movitz system includes many lines of library code, some of which are briefly described here. By "library" we mean a collection of related functions and macros.

4.1 The Common Lisp library

The Common Lisp language defines an extensive library of functions and macros. This library is implemented in lisp itself, resorting occasionally to inline assembly in order to achieve improved efficiency or to express something that is otherwise inexpressible. The work on implementing the Common Lisp library is not completed. However, what has been implemented tries hard to comply

---

[13]In files `special-operators-cl.lisp` and `special-operators.lisp`, respectively.

with the ANSI specification. We document here the most obvious shortcomings that currently exists. This can also be regarded as an (incomplete) todo-list for further development.

4.1.1 Numbers

Of Common Lisp's many numeric system classes, only fixnums of 30 bits have been implemented so far. This means that only integer numbers in the range $-(2^{29})$ to $(2^{29} - 1)$ can currently be represented as lisp values. Overflows are detected, and cause an exception, for which there is currently no remedy as there are no bignums. We expect to implement at least bignums and ratios, and possibly some floating-point type as well.

4.1.2 Sequences

Common Lisp defines numerous sequence operators, and many of these also have rather complex definitions. That is, these operators must accept both lists and vector objects, operate forwards or backwards (via the FROM-END keyword), etc. Implementing all this is a major undertaking, and while most operators are currently included, many of these are limited: Some functions only work with lists or vectors, and some do not accept FROM-END.

4.1.3 Eval

The current EVAL implementation has one major shortcoming, in that it does not support macros. Currently, macro definitions are not packaged into dumped images. However, certain macros are implemented as special operators, including the SETF operator, so that functions named (SETF <FOO>) still work[14]. Additionally, EVAL does not implement every special operator. In short, EVAL is currently mostly useful for executing function calls interactively.

4.1.4 CLOS

The Common Lisp Object System is supported, including working DEFCLASS, DEFGENERIC, and DEFMETHOD macros. This is work in progress, but we intend to include strong support for CLOS, and do not regard it as an after-thought addition to the system, but rather a central part of it. Device drivers and much other OS functionality is expected to make extensive use of the powerful abstraction mechanisms provided by CLOS.

Movitz' CLOS implementation is based on the Closette implementation from the book "The Art of the Metaobject Protocol", by Kiczales, Rivières, and Bobrow.

More details about Movitz' CLOS implementation are provided in section 4.2.

---

[14]The cross-compiler implements SETF fully, however.

4.1.5 File-system related functions

Because Movitz targets a bare-bones kernel environment, there is no file-system, and thus no related functions, such as OPEN and CLOSE, or any pathname functionality. This may be implemented as part of a particular kernel application.

4.1.6 Conditions

Condition objects are implemented atop CLOS. Condition signaling is fully implemented.

4.1.7 Reader

The current reader, while adhering to the most important aspects of Common Lisp standard syntax, is incomplete in that it is "hard-coded". That is, it does not take *READ-TABLE* or other reader customization context into account, with the sole exception of *READ-BASE*.

4.2 CLOS details

4.2.1 Method dispatch

The manner in which method dispatch is implemented affects heavily the performance of any CLOS implementation. When a generic function is called, control is transferred to the function that has been set with the MOP operator SET-FUNCALLABLE-INSTANCE-FUNCTION. That function must transfer control to the appropriate method function.

Normally (that is, currently always) generic functions are initialized to INITIAL-DISCRIMINATING-FUNCTION. This function will inspect the generic function's methods and number of required arguments, and based on this information it will install a new funcallable-instance-function, and invoking this by calling the generic function anew with the same arguments. This new discriminating function will perform method lookup according to the CLOS specification, and (typically) cache the result for later invocations.

The information that INITIAL-DISCRIMINATING-FUNCTION seeks for a generic function is a "specializer bitmap", often referred to as "map" in the relevant code. The bitmap is an integer where bit 0 is set if any method specializes on argument 0, and so on. Thus, if the specializer-bitmap is 0, the generic function has precisely one method that specializes on no arguments, in which case that method's method-function is used as the generic function's discriminating-function (although there's no actual discriminating going on) . A map of `#b1` means that all methods specialize on the first argument (if any), and the map `#b101` means all methods specialize on the first and/or third arguments (if any), and so on.

The system is now in a position to install discriminating functions that are specialized to different specializer-maps. This helps performance in two important respects. Firstly, any zero-bit in the specializer-map means that

argument can be disregarded in the cache lookup process, and for which the dispatcher doesn't have to perform CLASS-OF (or even more, if eql-specializers are involved), or one compare for each entry in the cache lookup. Secondly, and perhaps most importantly, each argument that can be disregarded in the lookup might drastically reduce the size of the cache. To see this, consider the case of slot writers: They take two arguments, and are typically only specialized on the second one (the object). If the first argument (the new value for the slot) is not disregarded, one entry will have to be saved in the cache for *each combination* of object and value classes, which is clearly going to hurt lookup performance and increase memory footprint.

All discriminator functions currently cache method dispatches using (approximately) assoc-lists. This is the most efficient approach when the number of entries in the caches are below some number (that is expectedly in the area of 15–40). When bigger systems happen, it might be worthwhile to implement hash-table caching or some other caching strategy approaching $O(1)$ lookup.

4.3 Miscellaneous libraries

4.3.1 Read-line

The read-line library provides a mechanism for entering text interactively. Simple line-editing key-strokes are supported, and a history buffer.

4.3.2 IPv6 networking

A library of functions related to implementing the IPv6 networking protocols. Most of the functions are packet accessors, that provides read/write access to for example a packet's destination field. Packets are represented by vectors. There are also functions for managing IPv6 addresses.

A very minimal IPv6 stack is also implemented, mostly as an example of how to use the accessors. This stack can perform rudimentary neighborhood discovery, reply to ping requests, and receive UDP packets.

TCP (or anything else really interesting) is not implemented yet.

4.3.3 IPv4 networking

There is now also an IPv4 library, similar to the IPv6 one.

4.4 x86 CPU library

This library contains functionality related to interfacing certain aspects of the x86 CPU. This includes a low-level memory interface, and hardware I/O-ports for interfacing hardware devices. Also, functionality for determining the brand and model of CPU currently running, and access to special CPU features such as the performance counters is available.

4.5   x86 PC drivers library

This library contains drivers to various parts of the de-facto PC standard, and for different peripheral hardware devices. Currently included are a drivers for a simple text-mode screen driver, the PC/AT keyboard, the NE2000 network interface card, and the PC interrupt and timer controllers.

5   Debugging and monitoring tools

5.1   Image browser

A graphical browser tool has been implemented, using the CLIM GUI system. This enables the user to inspect an image at a low detail level. The tool displays a graphical representation of the image's lisp objects, and any references it contains may be opened or closed by clicking with the mouse. A screen-shot can be seen in figure 5.

The browser tool requires only the most fundamental aspects of images to be supported (section 3.1 introduces images). This means that the browser can attach to all the existing forms of images, even including Movitz running in a Bochs emulator. However, currently the emulator is suspended for the duration of a browser session. Integrating and synchronizing the browser seamlessly with a truly running image is a topic to be researched.

5.2   Bochs emulator interface

As previously mentioned, one of the image implementations interfaces a running Bochs emulator, using the Unix procfs IPC mechanism. In addition to providing access to the emulated address space, the entire state of the emulated CPU is also made available. This enables for example the developer to obtain a stack-trace of the running program. This has proved to be a valuable debugging tool.

6   Summary

We have introduced the Movitz system, and described some of its most important concepts, some technical details, and some of its current shortcomings with respect to the Common Lisp standard.

Figure 6 shows a screen-shot of the Bochs PC emulator running a Movitz image. The image has just been booted (the minus signs in parens are the boot-loaders progress indicator), and a short interactive session showing off some of Movitz' features is displayed.
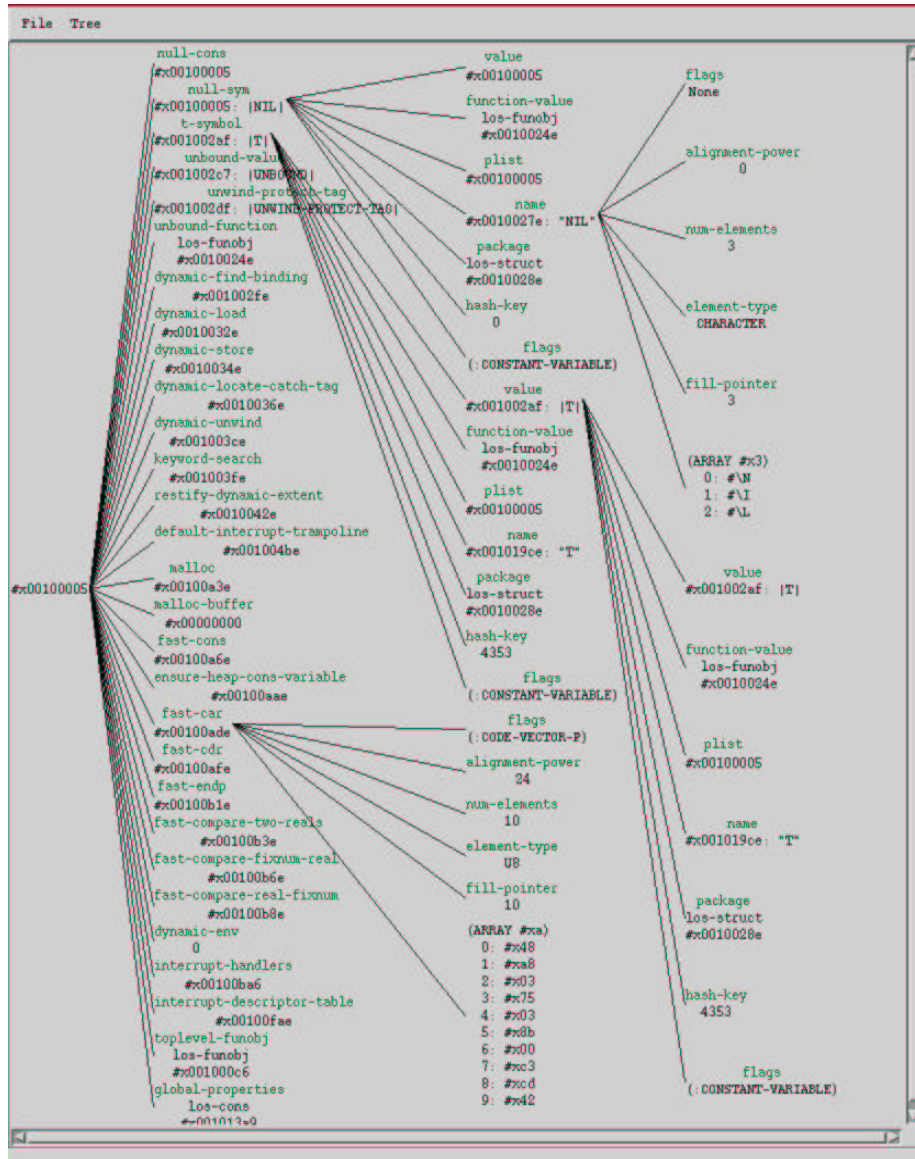
Figure 5: A screen-shot of the Movitz image browser tool. The root object to the left is the global "run-time-context", a.k.a the NIL object.

Figure 6: A screen-shot of Bochs running Movitz.

Contents

Symbol index