Exceptional Mite: simple yet flexible non-local exits in a binary-portable VM

Reuben Thomas[*]
University of Glasgow

July 2001

Abstract

Mite is a low-level virtual machine that supports binary portability. Its non-local exit mechanism is compatible with system calling conventions, and is both language and machine neutral, yet allows the straightforward and efficient implementation of a range of exception mechanisms, such as those of C, Java and ML, and other sorts of non-local exit, such as continuations. This makes it a good environment for experimentation in binary-portable mixed-language code generation.

1 Introduction

Most programming languages have some form of non-local exit, that is, a way of exiting a procedure or function without returning to its direct caller. Language features that use non-local exit include exceptions, continuations, co-routines, backtracking and multithreading, and occur in all sorts of language: procedural, functional, logical, object-oriented, message-passing and so on. Hence, any language-neutral virtual machine (VM) must support non-local exits. At first glance, it seems that no special support is necessary; after all, most processors lack explicit instructions for non-local exit. However, in order to provide efficient machine-independent access to native calling conventions, a VM must abstract the stack. Normal subroutine call and return instructions are not affected by an abstracted stack, as they already act implicitly on the stack, but in order to provide non-local exit, new primitives are needed.

Most existing VM systems provide some non-local exit mechanism, but they tend to be limited: either language-specific (JVM), OS-specific (.NET), low-performance (Cintcode) or not binary-portable (`C--`). Mite attempts to provide a mechanism that is at the same time simple to use, simple to implement, efficient, language-neutral and supports system calling conventions. It does not attempt to support optimal code generation, nor to support every possible type of non-local exit, but rather aims to be an 80–20 solution: 80% of the functionality with 20% of the complexity.

The rest of this paper is organized as follows. Section 2 discusses what functionality is required for efficient language and machine-neutral non-local exit. Section 3 gives an introduction to Mite's architecture, and section 4 discusses difficulties in providing non-local exit in this context; section 5 describes Mite's non-local exits mechanism. Section 6 shows how C, Java and ML exceptions can be encoded in Mite, and section 7 discusses continuations and co-routines. Section 8 considers related work, and section 9 concludes.

[*]`rrt@sc3d.org`

2 Required functionality

Although non-local exit constructs vary widely between languages, one basic property ensures that most can be easily and efficiently implemented: it should be possible to make a non-local exit to any point in any routine in the current call chain in a single operation. This means that:

- An arbitrary number of stack frames can be unwound in one go, making non-local exit efficient
- A non-local exit site doesn't need to be a call site, allowing exception handlers to be separate from the main flow of control
- Conversely, the same code *can* be used both as the target of a non-local exit, and as normal code, which eases the implementation of some constructs, such as C's `setjmp` (a function whose return value may be generated by a non-local exit) and Java's `finally` blocks (which may be reached either by an exception being raised, or by simply falling through from a `try` block)
- A non-local exit can be made to the currently executing function, allowing actions to be restarted, and exceptions to be handled by function in which they are raised[1]

For simplicity, some more esoteric functionality is not required:

Arbitrary return type Non-local exits often want to pass values, but it is tricky to arrange to pass arbitrary types; however, provided a pointer can be passed, arbitrary values can be passed by reference, though not on the stack.[2]

Multiple stack support For simplicity, Mite does not support the use of multiple stacks, as required for multithreading. Library or OS support can still be used to perform these functions, though that will affect portability.

Asynchronous exceptions Also for simplicity, there is no support for asynchronous exceptions; however, since it is natural to implement Mite fully re-entrantly, it automatically supports most external asynchronous exception mechanisms.

Integration with OS non-local exit As there is no standard for OS support of non-local exits, portability and simplicity dictate that Mite not support such mechanisms either.

3 Mite

Mite [10] is a simple register-based virtual machine with a RISC-like load-store architecture and three-operand data-processing instructions. While simple enough that its instruction set could be described as "idealized", it is designed to allow binary-portable code generation, is a good target for optimizing compilers, so that its translator can be simple and fast, while still producing good native code, and has a precise definition. It is intended to be implemented by just-in-time translation.

Mite abstracts both the underlying machine's physical registers and the system stack as a single entity, the virtual register stack. An indefinitely large number of

[1] Of course, if it is known in advance that the handler for an exception is in the same function, a simple branch rather than a non-local exit can be used; however, it is often not known statically where an exception will be handled.

[2] In the special case of tail calls, Mite *does* of course allow arbitrary tuples to be passed, just as in a normal function call; see section 7.

virtual registers is allowed, which correspond to abstract stack locations. Virtual registers are statically created and destroyed by directives inserted in the instruction stream; notionally, this corresponds to growing and shrinking the virtual stack frame by register-sized amounts. A new stack frame is started on entry to a function, and only virtual registers in the current function are directly accessible.

To give a flavour of Mite's assembly code, the following annotated example contains a recursive function to calculate factorials, and a sample call of it. It is followed by a more detailed explanation.

```
f.main               declare a function with no arguments
NEW                  declare a new register to hold the argument to
                     fact
MOV 2,#10            load the constant 10
CALLF .fact,1,[1]    call fact with 1 argument, expecting 1 return
                     value
RETF 1,[2]           return the result of fact from main
KILL                 kill the argument register
KILL                 kill main's return chunk
NEW                  declare a single parameter
f.fact               declare a subroutine
NEW                  declare a register . . .
MOV 3,#1              . . . to hold the constant 1
NEW                  declare a register to hold the result
SUB 4,1,3            subtract 1 from the argument
BEQ .exit            exit if zero
CALLF .fact,1,[1]    otherwise recurse
MUL 1,1,4            multiply the argument by the result
.exit                declare a branch label
KILL                 kill the result register
KILL                 kill the constant 1 register
RETF 2,[1]           return the result
KILL                 kill the argument register
KILL                 kill the return chunk
```

Data-processing instructions have their destination as the first operand. Constants are only allowed in the `MOV` instruction. Two types of label are used above: branch labels, which denote a branch target, and function labels, which denote a function entry point and are prefixed with '`f`'.

A register is introduced with `NEW`, and destroyed with `KILL`. Registers are created and destroyed in stack order, so a register cannot be killed until all the registers created after it have also been killed. Registers are numbered according to stack position, counting from the bottom; the bottom-most register is number 1. `NEW` and `KILL` are static directives, as mentioned above, so are not executed each time they are reached; hence, the current size of the stack frame (or equivalently, the virtual register file) at any point in the program can be calculated statically by simply counting all the `NEW`s and `KILL`s up to the desired point.

The number of parameters to a function is indicated by the number of virtual registers active at the function entry label (since registers cannot be shared between stack frames); all parameters are assumed to be word-sized (as are registers).[3] A

[3] Actually, arbitrary-sized "chunks" can also be declared, which are not held in machine registers, but must be accessed indirectly; for simplicity these are not treated further in the present discussion, with the exception of return chunks.

function label also implicitly declares a “return chunk” as an implicit extra argument, which holds the caller and callee-saved information specified by the system calling convention; this is passed to the `RETF` instruction as its first argument.

Note that so far the stack has not been mentioned explicitly at all, though it is affected implicitly by virtual register spilling and function call and return.

4 Difficulties with providing non-local exit in Mite

Non-local exit involves unwinding the system stack. In native code this is generally achieved by simply setting the stack pointer to a previously saved value, then branching to the return address, but in Mite it must be handled rather more delicately.

Most of the difficulties arise from Mite’s virtual register stack, and in particular the mapping from virtual to machine registers, which in general varies within a program. There are three main problems:

- The physical to virtual register binding active at the return site must be restored correctly.
- Unlike a call site, which knows the return type of the function called, a non-local return site may be reached from anywhere, with different types of return value.
- The stack pointer must be reset correctly by a non-local exit, a tricky operation when the virtual register stack is taken into account.

When a function returns, the return site’s physical register binding must be restored, and any return values written to the correct registers and memory locations. The same things must be done at a non-local exit. However, while a function return targets a call site, which can restore caller-saved registers and store results, a non-local exit instruction goes to an arbitrary point in the code. Hence, some way of indicating a potential non-local return site is needed, so that native code can be inserted to deal with the result and reset the virtual to physical register binding.

This is easily achieved by assuming a standard register binding at such points, with just the return value mapped into a register, and all other virtual registers spilt to the stack. Hence, code that falls through or makes a normal branch to such a point must spill all but one of the virtual registers to the stack, and make sure that the virtual register which would have held the return code (had the destination been reached by a non-local exit) is mapped to the correct physical register. It is most natural for a non-local exit to pass its result in a register rather than on the stack, since it alters the stack pointer as part of its operation.

The virtual registers that are active at a non-local return site must also be correctly spilled when that site is actually reached by a non-local exit. To see how this is done, let the routine that executes the non-local exit be called the ‘thrower’, and that to which it returns the ‘handler’. The two possible situations are illustrated in figures 1 and 2: either the thrower is the same routine as the handler, or it is deeper in the call chain. Clearly, the registers live at the return site must have been spilt by the time the non-local exit is performed. If the thrower is the same as the handler, this is easy: the non-local exit itself can perform the necessary spilling. If the thrower is not the same as the handler, it is not possible to wait until the non-local exit to perform the spilling, as the system cannot retrospectively spill the necessary registers (it may not even know where their current values reside!). Instead, the spilling must be performed at the function call that leaves the handler.

The second problem, dealing with the return type of a non-local exit, is much simpler. At a normal function return site, the return type is known; non-local return

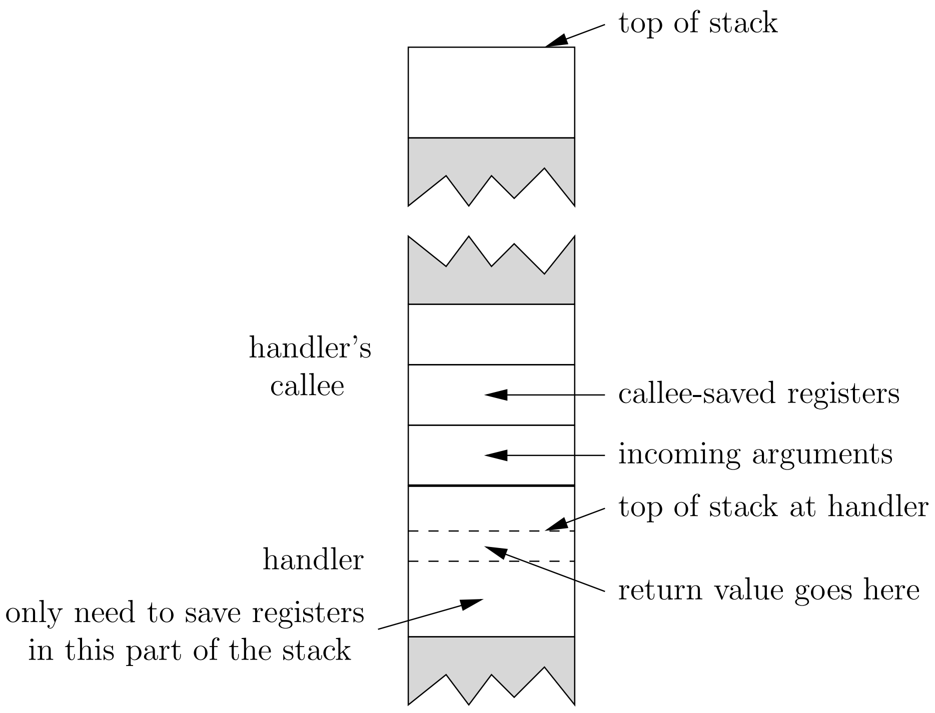

Figure 1: Throwing between stack frames

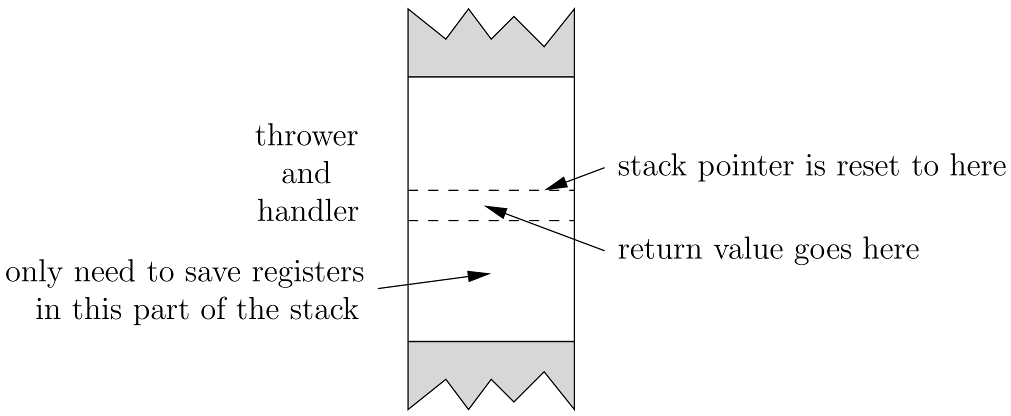
thrower and handler
stack pointer is reset to here
return value goes here
only need to save registers in this part of the stack

Figure 2: Throwing within a stack frame

sites, however, may be reached from anywhere. Mite's solution is to fix the return type of non-local exits to be a single register value. If more than a single value needs to be passed, the value can be a pointer, and the actual result placed in the heap, or deeper in the stack.

The third problem is to ensure that the stack pointer is correctly reset. A function call implicitly saves the current value of the stack pointer, to be restored by the corresponding return. Obviously this is not possible for a non-local exit, where the return site is not known, nor who will return to it. Hence saving and restoring the stack pointer for non-local return must be done manually. Unfortunately, it is not possible just to read the stack pointer's value and then write it back later. First, making the stack pointer directly readable and writable would involve devising rules for its use in portable code, which would be hard to get right, as its use would have to be heavily restricted. Secondly, how would one calculate the value of the stack pointer needed at the handler label from elsewhere in the function? Depending on the implementation, the stack pointer may vary during a function as virtual registers and chunks are created and destroyed. Hence, a special instruction is needed to obtain the correct value of the stack pointer, by allowing Mite's translator, with its knowledge of the generated code, to calculate it.

For similar reasons, a special non-local exit instruction is needed. It may seem to be merely an abbreviation for:

```
                            return value in 1
MOV SP, 2                   set SP
BAL 3                       branch to return site
```

where `SP` is the stack pointer, but it covers up some nasty surprises: what if having executed the first instruction it turns out that register 3, which holds the branch target, is currently spilt? It has to be reloaded from its spill location, probably on the stack, but that is no longer accessible, as `SP` has already been reset to its value at the return site.

5 Non-local exits in Mite

Mite provides non-local exits by means of two instructions, `CATCH` and `THROW`, a modifier for the `CALLF` and `THROW` instructions, `SYNC`, and a new type of label, the handler label (prefixed with '`h`').

`CATCH` r,l saves the value of the stack pointer at handler label l in register r. Later, while the subroutine in which `CATCH` was executed is still live, `THROW` l,r,v returns control to the handler at l. The value v overwrites the top-most register that is active at the handler. The value of l (which may be either a label or a register holding the value of a label) in the `THROW` instruction must be the same as in the `CATCH` that yielded the value of r.

When `THROW` is executed, the stack's state is changed to that of the handler label. Since this may not be the same as at the corresponding `CATCH` instruction, only those stack items which are live at both the `CATCH` and the handler label have a defined value. Moreover, since the values of registers that are cached in physical registers may be lost when a `THROW` is executed, all registers are assumed to be held in memory at a handler, and the `SYNC` modifier is provided to save registers to memory. It has one operand, a handler label, which is used to decide which registers need to be saved. `SYNC` may be attached to a `CALLF` or `THROW` instruction, and is only needed when the handler is in the same subroutine or function as the instruction being `SYNC`ed. Whether it is used on `THROW` or `CALLF`, `SYNC` is optional: for `THROW` it need only be used when the handler could be the same as the thrower; for `CALLF`, when the callee (or any more deeply nested callee) could `THROW` to the caller. `SYNC` effectively enforces a caller-saves calling convention; indeed, it has no effect if the system calling convention is purely caller-saves.[4]

The following code demonstrates the use of `CATCH` and `THROW`. The code from `.hand` onwards is run three times: first on entry to `main`, then by the first `THROW`, which throws from one point in `main` to another, and finally by the second `THROW`, which causes a non-local exit from the subroutine `sub`. The result is that the four words at `.store` are changed from their initial contents of four zeros to the sequence 0, 1, 2, 3. Note that both `THROW` and `CALLF` must have `SYNC .hand` added, so that the virtual registers live at the handler are sure to have the correct values when it is reached.

```
f.main
NEW
DEF 2, .store                address of results
NEW
MOV 3, #1                    first value to store
h.hand
NEW                          address offset
NEW                          shift
MOV 5, ashift                shift to turn bytes into words
SL 4, 3, 5                   make address offset
KILL
ST_a 3, [2, 4]
MOV 4, #1
ADD 3, 3, 4                  next value to throw
NEW
CATCH 5, .hand               get catch value
NEW
MOV 6, .hand                 address to throw to
MOV 4, #2                    second value to store
SUB , 3, 4                   decide next destination
BEQ .same                    based on previous result
MOV 4, #3                    third value to store
SUB , 3, 4
BEQ .sub
RETF 1, []
.same
MOV 4, #2
THROW 6, 5, 4 SYNC .hand     throw to same function
.sub
MOV 4, #3
CALLF .sub, 3, [] SYNC       this call won't return
.hand
KILL                         kill remaining items
KILL
KILL
NEW                          create parameters
NEW
NEW
f.sub
```

[4] `SYNC` is similar to `C--`'s `cuts to` annotation. This is used at a call site to specify variables that are live at other points in the procedure, which may be reached by a non-local return from the call about to be made. `SYNC` only allows one such point to be specified.

```
THROW 3, 2, 1
KILL
KILL
KILL
d.store
SPACEZ_a 4                         words to hold results
```

6 Encoding exceptions

The commonest use of non-local exit in most programming languages is for exceptions. This section shows how three common languages' exception styles can be encoded using the mechanisms introduced above.

One shortcoming of all three encodings presented here is that they will not in general interwork with other implementations of the same mechanisms. How then do Mite's function call and return manage to interwork with native code? Function call and return are rather simpler mechanisms, and most system calling conventions are simple, and designed to cater to the needs of C-like and Pascal-like languages. They are designed to be fast, since function calls are common in compiled code. The same is not true of exceptions, whose form varies widely between languages, and since they are generally considered to occur infrequently, need not be as efficient.

This situation does not prevent portable Mite-encoded exceptions being used solely within Mite code, however. It may also be possible, with a little knowledge of how Mite's translator works, to generate Mite code to handle exceptions in a system-specific way.[5] Finally, where exceptions are implemented through system calls, as with Windows's Structured Exceptions [8] and some implementations of `setjmp` and `longjmp`, Mite can use these like any other compiled code.

6.1 C exceptions

C uses the `setjmp` and `longjmp` macros [1] to implement non-local exit. A call to `setjmp` is translated as follows:

```
                  register 1 will hold the result of setjmp
                  register 2 points to the jmp_buf
NEW               scratch register
NEW               constant
MOV 4, #0+1       one-word offset
CATCH 3, .hand
ST_a 3, [2]       store the address in the jmp_buf
MOV 2, .hand      get the address of the handler
ST_a 3, [2, 4]    store the handler address in the jmp_buf
KILL              kill registers that are no longer needed
KILL
KILL              the top stack item is now register 1
MOV 1, #0         set result of setjmp to 0
h.hand            the point reached by longjmp
```

[5]Mite is designed for graceful degradation of portability: it is possible, for example, to write word-length dependent code and endianness-dependent code that is otherwise still binary portable, or even processor-independent device drivers. Similarly, it may sometimes be possible and advantageous to generate Mite code rather than native code even for system-dependent instruction sequences such as those involved in exception handling.

When control reaches `.hand`, register 1 contains either 0, if the code was entered at the top, or the `longjmp` value, if the handler was reached by a `THROW` instruction (which overwrites the top-most stack item with the throw value). The call to `longjmp` is implemented as:

```
                    register 1 points to the jmp_buf
                    register 2 contains the return value
NEW                 register to hold the stack state
NEW                 constant
DEF 4, #a           one-word offset
LD_a 3, [1]         get stack state
LD_a 1, [1, 4]      get handler address
THROW 1, 3, 2       perform the THROW
```

This causes the handler to be reached with the given return value. Since all registers are spilled at a handler, the `jmp_buf` need contain no registers. That makes this portable implementation less efficient than many system-specific implementations, but unless `longjmp` is used frequently this is unlikely to be a problem.

There is a further subtlety: to ensure that the stack state is consistent when a `longjmp` is executed, all `CALL`s and `THROW`s in a function that calls `setjmp` must be followed by `SYNC .hand`.

6.2 Java exceptions

Exceptions in Java work as follows: a code block guarded by `try` can raise an exception, which is an object whose type is a sub-class of `Exception`. A `try` block is followed by a number of `catch` blocks, each of which has an associated exception type. The first whose type is a super-class of that of the exception object is executed. After the `catch` blocks there may be a `finally` block, which is always executed, whether the `try` block terminates normally, or with a `return` or `break`, or by an exception. This applies even if a further exception is raised in one of the `catch` blocks. Exceptions may be raised anywhere by `throw`, which is given the exception object. This is often created at the same time:

```
throw new MyExceptionClass("we made a booboo");
```

is a common idiom.

Since user-supplied exception classes can add extra instance variables and methods, exceptions are naturally value-passing.

As exceptions have a special syntax in Java, the implementation is more straightforward than that for C. The `try` block starts with a `CATCH`, and all method calls and `throw`s inside it are `SYNC`ed. The first `catch` block is preceded by a handler label, whose address is used as the current innermost handler. When an exception is thrown to this handler, it determines which `catch` block to run, according to the type of the exception, and then branches to it. Each `catch` block ends with a branch to the end of the last such block, where the `finally` block occurs, if any. If no suitable exception value is found, the exception must be re-`thrown` to the next innermost handler.

Any `return`s, `break`s or `continue`s within the `try` block must also cause the `finally` block to be run before the appropriate action is performed. Thus it might be best to translate the `finally` block as a subroutine, or alternatively to pass it a continuation address. Since an exception may be raised inside a `catch` block, an extra handler must be installed for the duration of the `catch` blocks, which causes the `finally` block to be executed before the exception is re-raised.

The addresses of handlers can be passed to `THROW` sites in a number of ways. The currently active handler could be passed as an implicit parameter to every method call, or the handler chain could be kept as a linked list on the stack. It would also be possible to have a separate handler stack. Most conventional compilation methods are applicable to Mite.

Note that although Mite's `THROW` instruction only allows a single register to be passed, rather than the compound values allowed in Java, no run-time penalty is incurred by forcing the exception value to be passed by reference, since it is a Java object, and must in any case be allocated on the heap.

6.3 ML exceptions

In ML, exceptions are datatype constructors, and may thus pass arbitrary datatypes. An exception e is raised with `raise` e. An exception causes immediate termination of expression evaluation, and the value of the expression becomes the exception value. Exceptions thus propagate outwards like any other result, except that they prevent any further evaluation.

An exception handler is a guard on an expression of the form

E `handle` P_1 `=>` E_1 `| . . . |` P_n `=>` E_n

where E is the guarded expression, the P_i are patterns whose top-level constructor is an exception, and the E_i are expressions. There is no equivalent of `finally` in ML.

When an exception value is propagated into an expression that has a handler, the exception value is matched against each clause in the handler; if a match is found, the corresponding handler expression is evaluated, and its value becomes the value of the expression. Otherwise, the exception value becomes the value of the whole expression, just as if there were no handler.

The implementation is similar to the Java case. Since exceptions are propagated until they reach a handler, intervening unguarded expressions can be ignored, and exceptions can be `THROW`n straight to the next innermost handler, just as in Java. When a handler is reached, the exception is dispatched by ML pattern matching rather than according to the Java class hierarchy, but this does not affect the implementation per se.

Unlike the Java case, since ML exceptions need not be constructed on the heap, there is a potential speed penalty in having to place them there, rather than simply treating them as return values. On the other hand, if an exception has to be propagated through several handlers before being handled, it may well be quicker to allocate space for it on the heap than have to copy it between stack frames once per handler.

7 Continuations and co-routines

As well as exceptions, other forms of non-local exit are widely used; however, since they are not simply and efficiently expressible in terms of returning to a point in the current call chain, more specialised support is required.

Continuations are equivalent to tail calls. The `TCALLF` instruction is just like `CALL`, but removes the current stack frame before making the call, so that the return will be made to the current function's caller.

Implementing co-routines is trickier: if they are forbidden to make calls, a group of co-routines can be compiled into a single function. Otherwise support for multiple stacks is required, which could either be emulated (rather inefficiently) in portable code, or make use of (possibly non-portable) external libraries.

8 Related work

There is a wide variety of approaches to non-local exits in current VM systems. Some simply do not allow them, such as the dynamic code generation systems VCODE [3] and Lightning [2], which are designed for run-time code generation, rather than as compiler targets. In such a system, non-local exit is only possible via library support. Cintcode [4], a BCPL-oriented VM, does not abstract the stack, which can therefore be manipulated directly to obtain non-local exit; however, Cintcode has an abstract memory model, and does not allow interworking with native code. The Java virtual machine [5] has specific support for Java exceptions, but, as one might expect from its Java-oriented design, does not cater for other types of non-local exit. The .NET VM [6] offers flexible exceptions integrated with Windows's Structured Exception Handling [8], but this is a heavyweight and OS-specific framework: processing an exception involves two passes over the entire stack, and the creation and traversal of complex data structures. Finally, `C--` [7], an intermediate language specifically designed as a compiler target, supports a wide range of non-local exits [9], but its interface is complex and, as yet, unimplemented. Also, it does not directly implement all the mechanisms it supports, but merely provides hooks for a run-time system to do so, and it does not support the generation of binary-portable code.

9 Conclusion

Most languages support some form of non-local exit, but few VM systems allow such features to be implemented simply, efficiently and portably. Mite provides direct support for the most common types of non-local exit, in particular exceptions of various sorts and continuations, and allows other non-local exit mechanisms to be supported, either less efficiently, or less portably; importantly, this trade-off is made by the programmer, not forced by Mite's design. This helps to make Mite a good environment for experimentation in binary portable mixed-language programming, at a level where efficiency aspects have not been abstracted away and can still be fully explored.

References

[1] American National Standards Institute, "ANS X3.159-1989: Programming Languages—C," (1989).

[2] Bonzini, P., *Using and porting GNU lightning* (2000), `ftp://alpha.gnu.org/gnu/`.

[3] Engler, D. R., VCODE*: A retargetable, extensible, very fast dynamic code generation system*, in: *Proceedings of the 23rd Annual ACM Conference on Programming Language Design and Implementation*, 1996, `http://www.pdos.lcs.mit.edu/~engler/`.

[4] Jobson, C. and J. Richards, "BCPL for the BBC Microcomputer," (1983).

[5] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1999, second edition.

[6] *.NET framework SDK technology preview*, `http://msdn.microsoft.com/downloads/`.

[7] Peyton Jones, S., T. Nordia and D. Oliva, *`C--`: A portable assembly language*, in: *Proceedings of the 1997 Workshop on Implementing Functional Languages*, 1997.

[8] Pietrek, M., *A crash course on the depths of Win32 structured exception handling* (1997), `http://www.microsoft.com/msj/0197/exception/exception.htm`.

[9] Ramsey, N. and S. Peyton Jones, *A single intermediate language that supports multiple implementations of exceptions*, in: *Proceedings of PLDI '00*, 2000.

[10] Thomas, R., "Mite: a basis for ubiquitous virtual machines," Ph.D. thesis, University of Cambridge Computer Laboratory (2000), `http://sc3d.org/rrt/`.