The Morpho Virtual Machine

Snorri Agnarsson

December 16, 2010

Contents

1 Introduction

The purpose of the Morpho virtual machine is to run Morpho executables that have been compiled by a Morpho compiler. The system is designed to run on top of Java and Morpho executables are sequences (arrays) of Morpho instructions. Each Morpho instruction is a Java object[1].

The Morpho compiler reads Morpho program text and compiles it into either Morpho module files or executable Morpho programs (Morpho executables). Both of these contain Morpho instructions in serialized form. Future versions of the Morpho compiler may use other intermediate forms for Morpho unit files, in order to facilitate optimization.

A running Morpho virtual machine has one array of Morpho instructions and some associated information (virtual registers) as described below.

The Morpho instruction set is easily extendable by creating new subclasses of instructions.

The run-time values in Morpho are Java object references, including the null value.

Among the design goals for the Morpho instruction set are the following:

- Support stackless semantics in order to make possible high-level features.
- Support block structure, nested lexical scopes, functions and closures as first class values.
- Support small-grained multi-threading where each thread has minimal memory footprint.
- Support even smaller-grained fibers (also known as coroutines).

2 Registers

A running Morpho Virtual Machine (Morpho VM) has the following (virtual) registers.

AC Accumulator. Contains results from evaluating expressions.

AR Current activation record. See below.

PC Program counter. Integer address of current instruction inside the instruction array.

[1]Future versions may support running on top of something other than Java, e.g. C# or even C++.

3 Environments

An environment is implemented as a Java object of type Environment. An environment contains the parameters and local variables of a particular activation of a function. An environment is an array of Object values that can be indexed by any non-negative integer. In addition, an environment contains a reference to the lexically enclosing environment. This reference will be null if the environment is at the top lexical level.

4 Activation Records

An activation record is implemented as a Java object of type ActivationRecord. An activation record can be viewed as an array of Object values which can be indexed by any integer value. An index can be positive, negative or zero. In addition to the indexed positions, each activation record contains the following modifiable values:

environment This refers to an object of type Environment. This contains the parameters and local variables of the current function, as well as a reference to the lexically enclosing environment. When accessing a position in the activation record using a non-negative index, we are really accessing positions in the environment referred to by the activation record.

return_pointer This is an integer value that is the continuation program counter (*return pointer*) to use when returning from the current function. On returning from the function the program counter (PC) will get this value.

control_link This is the continuation activation record (*control link*, dynamic link) to use when returning from the current function. This is a reference to an activation record that will become the current activation record (AR) on returning.

exception_continuation This is either null or a value that specifies the continuation to use in the case of an exception. This is used to implement try/catch/throw functionality. See the operations GetHook (5.12) and SetHook (5.27).

function_name This is either null or a string that is intended to be the name of the currently running function. It is the responsibility of the compiler writer to generate code to set this appropriately. See the Enter operation (5.9).

file_name This is either null or a string that is intended to be the name of the source file contining the definition of the currently running function. It is the responsibility of the compiler writer to generate code to set this appropriately. See the Enter operation (5.9).

line_number This is either zero or an integer that is intended to be the starting position of the function definition in the source file contining the definition of the currently running function. It is the responsibility of the compiler writer to generate code to set this appropriately. See the Enter operation (5.9).

When accessing positions in the activation record (see, for example, the Fetch and Store operations), the indexing scheme is that parameters and local variables have indexes $0, 1, \ldots$ etc.[2], while temporary variables have keys $-1, -2, \ldots$ etc.[3]

Each activation record is considered to be extendable to any size in both directions (negative and positive indexes). Fetching a value from a position corresponding to a previously non-existent index gives a null value. Storing a value into a previously un-accessed position creates that position.

5 Operations

5.1 Become

Purpose: Call a top-level function tail-recursively.

Usage: (Become target ar)

Pre: target is an integer, the relative address of the first instruction in the function to call[4], ar denotes a position in the current activation record that contains the activation record to use.

Post: Control has been passed to target, the activation record referred to by ar is now the current activation record. The return pointer (return_pointer) and control link (control_link) in that activation record have been set to the same values as in the current activation record.

[2]With these non-negative indexes we are, as described above, really accessing the environment inside the activation record.

[3]Temporary variables mostly contain activation records under construction, but may contain other temporary values that are not considered to be lexically accessible variables of the current activation record.

[4]Normally, in Morpho assembly code the target would be denoted by a label and the assembler would take care of computing the corresponding integer value.

5.2 BecomeClosure

Purpose: Call a closure tail-recursively.

Usage: (BecomeClosure n ar [lev])

Pre: Before this instruction is executed, the accumulator must contain a reference to a closure that takes the same number of parameters as n specifies. n is the integer count of arguments that the closure takes (for run-time error detection purposes), ar is an integer (usually negative) that specifies a variable that contains the activation record for the call, lev is a non-negative integer that specifies how many steps up the control chain to go in order to find the activation record that will receive the eventual result of the call. If lev is zero then the result is returned to the caller of the current activation, and so on. If lev is missing then that is equivalent to lev=0.

Post: Control has been passed to the function denoted by the closure in the accumulator, the activation record referred to by ar is now the current activation record. The return pointer (return_pointer) and control link (control_link) in that activation record have been set according to the return level specified by lev.

5.3 BecomeDispatch

Purpose: Dispatch a message tail-recursively to an object.

Usage: (BecomeDispatch key ar [lev [name]])

Pre: The accumulator must contain a reference to a Morpho object (not the same as a Java object), key is an integer value that is a message, ar is an integer (usually negative) that specifies a variable that contains the activation record for the call, lev is a non-negative integer that specifies how many steps up the control chain to go in order to find the activation record that will receive the eventual result of the call. If lev is zero then the result is returned to the caller of the current activation, and so on. If lev is missing then that is equivalent to lev=0, name is a string that is the name of the message (only used for debugging purposes in the case of an error), if name is missing then that is equivalent to name="[unknown]".

Post: Control has been passed to the method denoted by the message according to the object in the accumulator, the activation record referred to by ar is now the current activation record. The return pointer (return_pointer) and control link (control_link) in that activation record have been set according to the return level specified by lev.

5.4 Call

Purpose: Call a top-level function non-tail-recursively.

Usage: (Call target ar)

Pre: target is an integer, the relative address of the first instruction in the function to call, ar refers to a position in the current activation record that contains the activation record to use.

Post: Control has been passed to the relative position denoted by target. The activation record referred to by ar is now the current activation record. The return pointer (return_pointer) in that record has been set to refer to the instruction following the Call instruction, the control link (control_link) has been set to the activation record that was current before the call.

The position denoted by ar has been cleared and now contains null[5].

Assuming that the address of this instruction inside the VM instruction array is PC (the contents of the program counter register) the value of the program counter after execution of this instruction will be PC+target+1. Similar holds for all other relative call/become/goto instructions.

5.5 CallClosure

Purpose: Call a closure.

Usage: (CallClosure n ar)

Pre: n is the integer count of arguments that the closure takes (for run-time error detection purposes), ar is an integer (usually negative) that specifies a variable that contains the activation record for the call. Before this instruction is executed, the accumulator must contain a reference to a closure that takes the same number of parameters as n specifies.

Post: Control has been passed to the function denoted by the closure in the accumulator, the activation record referred to by ar is now the current activation record. The return pointer (return_pointer) and control link (control_link) in that activation record has been set to refer to the next operation after this operation. The control link has been set to refer to the activation record that was current before this operation.

[5]Clearing the position denoted by ar is done to facilitate using that position later for constructing future activation record for other calls. Otherwise we would need to execute separate instructions to clear the position or allocate new activation records into it.

5.6 Case

Purpose: Specify a case in a switch statement.

Usage: (Case val target)

Note: This is not an executable operation but is used to specify a target location for a switch statement. See also the description of the Switch operation (5.34). In order for the operation to be valid, val is a string, an integer or a character value, target is an integer that specifies a relative target location corresponding to the given value.

5.7 Deref

Purpose: Dereference a pointer.

Usage: (Deref)

Pre: The accumulator must contain a pointer.

Post: The accumulator now contains the value from the variable that the accumulator referred to.

5.8 Dispatch

Purpose: Dispatch a message to an object.

Usage: (Dispatch key ar [name])

Pre: The accumulator must contain a Morpho object (not the same as a Java object), key is an integer denoting a message, ar refers to a position in the current activation record that contains the activation record to use, name is a string that is the name of the message, if present (only used for debugging purposes in case of an error). If name is omitted then that is equivalent to name="[unknown]".

Post: Control has been passed to the method denoted by the message key according to the object in the accumulator. The activation record referred to by ar is now the current activation record. The return pointer (return_pointer) in that record has been set to refer to the instruction following the Call instruction, the control link (control_link) has been set to the activation record that was current before the call.

The position denoted by ar has been cleared and now contains null[6].

5.9 Enter

Purpose: Register debug information about a function.

Usage: (Enter fname file line)

Pre: fname and file are strings, line is an integer.

Post: The effect of the operation is to register the three arguments into the current activation record. This information is then available for use in case of a runtime error.

5.10 Fetch

Purpose: Fetch the value of a variable into the accumulator.

Usage: (Fetch [lev] pos)

Pre: lev is a non-negative integer, pos is an integer. If lev is greater than zero then pos must not be negative. If lev is omitted then that is equivalent to lev=0.

Post: The accumulator contains the value from position pos in the activation record in lexical level lev from the current activation record. The current activation record is considered to be in level 0. Note that if pos is negative (in which case lev must be zero) we are accessing temporary variables in the current activation record, whereas if pos is non-negative we are accessing parameters or local variables in the lexical level denoted by lev.

5.11 FetchGlobal

Purpose: Fetch a value from a global variable.

Usage: (FetchGlobal n t)

Pre: n is an integer, t is a character, one of 'm', 'p' or 'f'.

[6]Clearing the position denoted by ar is done to facilitate using that position later for constructing future activation record for other calls. Otherwise we would need to execute separate instructions to clear the position or allocate new activation records into it.

Post: The accumulator now contains the value in the global variable indexed by n and t. If t is 'm' this is a machine global variable, so that all fibers and tasks share the same variable. If t is 'p' then this is a task global variable so that each task has a separate copy of the variable whereas separate fibers in the same task share the same variable. If t is 'f' then each separate fiber has a separate copy of the variable.

5.12 GetHook

Purpose: Get the current exception hook (exception_continuation).

Usage: (GetHook lev)

Pre: lev is a non-negative integer.

Post: After the operation the accumulator contains the current exception hook in the activation record at position lev in the control chain. If lev is zero then this is the current activation record, if lev is 1 then the next activation record in the control chain, and so forth.

5.13 Go

Purpose: Go to another instruction.

Usage: (Go target [lev])

Pre: target is an integer, the relative address of the instruction to go to, lev is a non-negative integer. If lev is omitted then that is equivalent to using zero as lev.

Post: Control has been passed to target. At the same time lev activation records have been dropped from the control chain. If lev is zero then the control chain is unchanged. Dropping records from the control chain is useful when control passes to upper lexical scopes, when the lower scopes are implemented as function activations.

5.14 GoFalse

Purpose: Conditional goto.

Usage: (GoFalse target)

Pre: target is an integer, the relative address of the instruction to go to.

Post: Control has been passed to target if and only if the accumulator contains the boolean value false.

5.15 GoTrue

Purpose: Conditional goto.

Usage: (GoTrue target)

Pre: target is an integer, the relative address of the instruction to go to.

Post: Control has been passed to target if and only if the accumulator contains the boolean value true.

5.16 Link

Purpose: Set up the access link in a new activation record.

Usage: (Link ar n)

Pre: ar refers to a position in the current activation record that contains an activation record. n is a non-negative integer.

Post: The access link of the environment of the activation record denoted by ar has been set to refer to the n-th environment counting out lexical scopes from the current environment (n=0 implies that we use the current environment as the linked-to environment, etc.).

5.17 MakeClosure

Purpose: Create a closure.

Usage: (MakeClosure target lev nparams)

Pre: target is an integer that is the relative address of a function, lev is an integer, either non-negative or -1, nparams is the integer number of parameters that the function takes. If lev is -1 then the function is a global function, i.e. it does not refer to any local variables in any outer scope. If lev is non-negative then it is the number of steps to go through access links (starting from the current activation record) to get to the activation record that the access link in the created closure should refer to.

Post: After the operation the accumulator contains reference to a new closure that refers to the given function.

5.18 MakeFiberRef

Purpose: Make a pointer to a global (fiber) variable.

Usage: (MakeFiberRef pos)

Pre: pos is any integer.

Post: The accumulator contains a pointer to the unique global variable in the current fiber denoted by pos.

5.19 MakeMachineRef

Purpose: Make a pointer to a global (machine) variable.

Usage: (MakeMachineRef pos)

Pre: pos is any integer.

Post: The accumulator contains a pointer to the unique global variable in the current machine denoted by pos.

5.20 MakeObject

Purpose: Create a Morpho object.

Usage: (MakeObject target count)

Pre: target is an integer that is the relative address of the virtual method table for the object. The virtual method table is a sequence of count Method operations that define the table.

Post: The accumulator contains a reference to the new object. The object has the current environment as an access link, making it possible to use the current local variables as instance variables.

5.21 MakeTaskRef

Purpose: Make a pointer to a global (task) variable.

Usage: (MakeTaskRef pos)

Pre: pos is any integer.

Post: The accumulator contains a pointer to the unique global variable in the current task denoted by pos.

5.22 MakeLocalRef

Purpose: Make a pointer to a local variable.

Usage: (MakeLocalRef [lev] pos)

Pre: pos is any integer, lev is a non-negative integer. If lev is omitted then that is equivalent to lev=0.

Post: The accumulator contains a pointer to the local variable in position pos in lexical level lev from the current activation record.

5.23 MakeVal

Purpose: Put a given value into the accumulator.

Usage: (MakeVal v)

Pre: v is an integer (Integer, Long or BigInteger), a floating point number (Double), a string, a character, a boolean (true or false) or null.

Post: The accumulator now contains n.

5.24 Method

Purpose: Define a method in a Morpho object.

Usage: (Method key target)

Note: This is not an executable operation. key is an integer that stands for a message, target is an integer that is the relative address of the method corresponding to the message. See the description of the MakeObject operation (5.20).

5.25 Not

Purpose: Logically negate the value in the accumulator.

Usage: (Not)

Post: The accumulator now contains true if its previous value was either null or false, otherwise it now contains false. In other words, the only false values for the purposes of this operation are false and null.

5.26 Return

Purpose: Return from call.

Usage: (Return [lev])

Pre: lev is a non-negative integer. If lev is omitted then that is equivalent to lev=0.

Post: Control has been passed to the position given by the return pointer (return_pointer) in the activation record at control level lev, where level 0 denotes the current activation record and successive levels are found by following the control links (control_link). The current activation record has been also been changed to be the one referred to by control link at control level lev.

5.27 SetHook

Purpose: Set the current exception hook (exception_continuation).

Usage: (SetHook lev)

Pre: The accumulator must refer to an object of type FiberState.

Post: The given FiberState object is now the current exception hook.

5.28 Store

Purpose: Store the value in the accumulator into a variable.

Usage: (Store [lev] pos)

Pre: lev is a non-negative integer, pos is an integer. If pos is negative then lev must be zero, as in the Fetch operation. If lev is omitted then that is equivalent to lev=0.

Post: The value in the accumulator has been copied into position pos in the activation record (or environment) in level lev from the current activation record. The current activation record is considered to be in level 0. See also the Fetch operation (5.10).

5.29 StoreArgAcc

Purpose: Store the value in the accumulator into a variable (argument) in another activation record that is under construction.

Usage: (StoreArgAcc ar pos)

Pre: ar and pos are integers, ar (usually a negative number) refers to a variable in the current activation record that contains an activation record or null.

Post: The value in the accumulator has been copied into position pos in the activation record referred to by the variable denoted by ar in the current activation record. If the variable denoted by ar contained null before the operation was executed then a new activation record was allocated there by the operation.

5.30 StoreArgVal

Purpose: Store a given value into a variable in another activation record.

Usage: (StoreArgVal ar pos v)

Pre: ar and pos are integers, v is a value, as in the MakeVal operation, ar refers to a variable in the current activation record that contains an activation record or null.

Post: The value v was copied into position pos in the activation record referred to by the variable denoted by ar in the current activation record. If the variable denoted by ar contained null then a new activation record was allocated there.

5.31 StoreArgVar

Purpose: Store a value from a variable into a variable in another activation record.

Usage: (StoreArgVar ar pos lev2 pos2)

Pre: ar and pos are integers, lev2 and pos2 are integers that together denote a variable, ar refers to a variable in the current activation record that contains an activation record or null.

Post: The value from the variable denoted by (lev2,pos2) was copied into position pos in the activation record referred to by the variable denoted by ar in the current activation record. If the variable denoted by ar contained null then a new activation record was allocated there.

5.32 StoreGlobal

Purpose: Store a value into a global variable.

Usage: (StoreGlobal n t)

Pre: n is an integer, t is a character, one of 'm', 'p' or 'f'.

Post: The value in the accumulator has been stored in the global variable indexed by n and t. If t is 'm' this is a machine global variable, so that all fibers and tasks share the same variable. If t is 'p' then this is a task global variable so that each task has a separate copy of the variable whereas separate fibers in the same task share the same variable. If t is 'f' then each separate fiber has a separate copy of the variable.

5.33 StoreRef

Purpose: Store a value through a pointer reference.

Usage: (StoreRef [lev] pos)

Pre: lev is non-negative (if missing then that is equivalent to lev=0), pos is an integer that denotes a local variable at lexical level lev. If pos is negative then lev must be zero.

The local variable in question must contain a pointer to some variable.

Post: The value in the accumulator has been stored in the variable denoted by the pointer in the local variable denoted by lev and pos.

5.34 Switch

Purpose: Make it easy to implement switch statements (switch expressions).

Usage: (Switch targets count default_target)

Pre: targets is the relative address of a sequence of count Case operations defining the different possible cases for the switch. default_target is the relative address of the position to go to in the case when none of the given cases apply.

Post: Execution continues from the address found by searching through the possible cases and going to the address specified by the case matching the value in the accumulator. If no case matches execution continues from the address specified by default_target.

6 Example

Consider the following Morpho module containing a top-level function:

```
"test.mmod" =
{{
fibo =
    fun(n)
    {
        if( n<=2 )
        {
            return 1;
        }
        else
        {
            return fibo(n-1)+fibo(n-2);
        };
    };
}};
```

It might be translated into the following Morpho assembly code:

```
"test.mmod" =
{{
#"fibo[1]" =
  [
    (StoreArgVar -1 0 0)      ;;; n
    (StoreArgVal -1 1 2)      ;;; 2
    (Call #"<=[2]" -1)        ;;; n<=2
    (GoFalse _1)
    (MakeVal 1)
    (Return)
_1:
    (StoreArgVar -3 0 0)      ;;; n
    (StoreArgVal -3 1 1)      ;;; 1
    (Call #"-[2]" -3)         ;;; n-1
    (StoreArgAcc -2 0)
    (Call #"fibo[1]" -2)      ;;; fibo(n-1)
    (StoreArgAcc -1 0)
    (StoreArgVar -3 0 0)      ;;; n
    (StoreArgVal -3 1 2)      ;;; 2
```

```
    (Call #"-[2]" -3)          ;;; n-2
    (StoreArgAcc -2 0)
    (Call #"fibo[1]" -2)       ;;; fibo(n-2)
    (StoreArgAcc -1 1)
    (Become #"+[2]" -1)        ;;; fibo(n-1)+fibo(n-2)
  ];
}};
```