From: L. Peter Deutsch, Mark S. Miller
To: Smalltalk implementors group
Subject: Proposal for 1987 Smalltalk Standard meta-level interfaces to contexts and processes

This is a working document proposing a complete definition of the standard protocols for meta-level access to execution contexts and processes for the 1987 version of the Smalltalk-80 Virtual Machine. The proposal includes a clean separation of object-level and meta-level aspects of Smalltalk consistent with both security and meta-interpretive definition, and without a significant change in the style of the language or environment. Comments are solicited. For changes in successive versions, consult the version history.

This document is the second of a pair specifying meta-level interfaces for the 1987 Smalltalk standard. The first document proposes a similar definition for meta-level access to object state and to executable code. You should read the other document first: you are likely to find the concepts in this one unfamiliar and confusing if you don't.

Acknowledgements: see the first document.

Version history:

[3] ** in progress **: first version (derived from exec87.text.2): major contributions from Mark Miller, providing a first attempt at consistency with a secure meta-interpretive story, also a first spec for transparent forwarding and non-methodical objects.

[M-]{
BlockClosure
==========

A BlockClosure conceptually contains just two pieces of information: a home context, used for accessing outer-scope variables and for non-local returns, and a method to execute when the closure is applied to arguments. An implementation may choose to represent this information in some other form, e.g. it may normally represent the method using the Smalltalk-80 device of embedding the block code inside its home method, but the logical view must be as described. In particular, the block must appear to have its own ExecutableMethod, conceptually distinct from the ExecutableMethod for the enclosing method. The ExecutableMethod for a block responds to the messages described above in the following way:

- The source code is the source code for just the interior of the block, not the entire enclosing method.
- The query messages refer only to the code within the block. The visible variables are precisely those visible from within the block, i.e. all those visible from the method, plus the argument and temporary variables of the block and any enclosing blocks.
- The message valueForReceiver:arguments: invokes the block. The receiver (first) argument must be an appropriate BlockClosure.

When we say the block must "appear to have" its own ExecutableMethod, we do not rule out (indeed, we encourage) implementations in which the block's "ExecutableMethod" is just a mediator object that implements ExecutableMethod protocol by dynamically extracting information from the enclosing method's ExecutableMethod. This option illustrates the flexibility of Smalltalk's totally object-oriented approach to system design.

Accessing

<ExecutableMethod> := <BlockClosure> method

Return the ExecutableMethod for the block. As explained above, this must be a full-fledged method in terms of its external protocol, conceptually distinct from the ExecutableMethod for the enclosing method.

<ExecutionContext> := <BlockClosure> outerScope

Return the next outer scope of the block.

<ExecutionContext> := <BlockClosure> remoteReturnContext

Return the context to which control would return if the block code did an ^ return. In current Smalltalk-80 terminology, this would be the sender context of the block's home.

If we decide to adopt a continuation model for control, a BlockClosure actually has two different 'homes': one that holds the outer-scope variables, and one that represents the continuation for ^ returns. For this reason, as well as the conceptual distinction, we have treated these as though they were different objects. [M?]{ Even if we don't adopt a continuation model, these two are different. The outerScope could be the containing BlockExecutionContext, whereas the remoteReturnContext would still be a MethodExecutionContext.}
}

[M+]{
MetaClosure
==========

A MetaClosure is a metaObject for either type of closure: BlockClosures or NonMethodicalObjects. Note that closures themselves are not considered meta-level objects.

A Closure conceptually contains just two pieces of information: a home context, used for accessing outer-scope variables and for non-local returns, and a method to execute when the closure is applied to arguments. An implementation may choose to represent this information in some other form, e.g. it may normally represent the method using the Smalltalk-80 device of embedding the closure code inside its home method, but the logical view must be as described. In particular, the closure must appear to have its own ExecutableMethod, conceptually distinct from the ExecutableMethod for the enclosing method. The ExecutableMethod for a closure responds to the messages described above in the following way:

- The source code is the source code for just the interior of the closure, not the entire enclosing method.
- The query messages refer only to the code within the closure. The visible variables are precisely those visible from within the closure, i.e. all those visible from the method, plus the argument and temporary variables of the closure and any enclosing Contexts.

- The message valueForMetaReceiver:arguments: invokes the closure. The receiver (first) argument must be an appropriate MetaClosure.

When we say the closure must "appear to have" its own ExecutableMethod, we do not rule out (indeed, we encourage) implementations in which the closure's "ExecutableMethod" is just a mediator object that implements ExecutableMethod protocol by dynamically extracting information from the enclosing method's ExecutableMethod. This option illustrates the flexibility of Smalltalk's totally object-oriented approach to system design.

Accessing

<ExecutableMethod> := <MetaClosure> method

Return the ExecutableMethod for the closure. As explained above, this must be a full-fledged method in terms of its external protocol, conceptually distinct from the ExecutableMethod for the enclosing method.

<Dictionary> := <MetaClosure> outerScope

A closure may refer freely to variables defined in its lexically enclosing defining context. The closure must retain, as part of its state, the bindings of these variables created by the activations that resulted in the creation of this closure. The dictionary refered to above represents these bindings, through it the values of any of these variables may be retrieved or changed. Note that this dictionary is only obligated to represent binding of variables which are actually refered to freely in the method of the closure (although it may represent all lexically apparent binding if it wishes). We allow it this freedom in order to not interfere with some important optimizations associated with closure in the Scheme world. (It may even be necessary to only guarantee access if the original closure method uses the variable as an rValue (evaluated for its value), and only guarantee the ability to change the variable if the original method uses it as an lValue (target of assignment))

Note that the dictionary remembers the "bindings" of the variables, not simply their values. This means that if two closures are created by the same activation and refer to the variable, that changes to its value made by one are visible to the other.

[M?]{
I don't know why we need access to a remoteReturnContext, but if we do, we should return a continuation.

<Continuation> := <MetaClosure> remoteReturnContext

A continuation is not a meta-level construct, and yields no meta-level view. This makes continuations very different from executionContexts. Until and unless we define the mechanics of continuations, I suggest we completely leave this message out.
}

MetaNonMethodicalObject is a subclass of MetaClosure, and is the class for all metaObjects of nonMethodicalObjects. There is no new protocol defined by this class, it is introduced so it can override the "methodForMessage:" and "argsForMessage:" methods as follows:

```
<MetaNonMethodicalObject> methodForMessage: aMessage
    ^ self method

<MetaNonMethodicalObject> argsForMessage: aMessage
    ^ Array with: aMessage
}
```

[M+]{
BlockClosure
==========
}

Invocation

<Object> := <BlockClosure> value
<Object> := <BlockClosure> value: <Object>
<Object> := <BlockClosure> value: <Object> value: <Object>
<Object> := <BlockClosure> value: <Object> value: <Object> value: <Object>
<Object> := <BlockClosure> value: <Object> value: <Object> value: <Object> value: <Object>
(etc.)
<Object> := <BlockClosure> valueWithArguments: <Array>
Invoke the closure with the indicated arguments. Any given implementation may choose to

provide the value:...value: form for more than 4 arguments ad lib. If the wrong number of arguments is supplied, an error occurs.

Control structures

The looping constructs of the language are implemented as messages to BlockClosure; they may also be treated specially by the compiler.

<BlockClosure> whileTrue: <BlockClosure>
Invoke the receiver; as long as the result is true, apply the argument, then apply the receiver again. If the result of invoking the receiver is neither true nor false, an error occurs.

<BlockClosure> whileFalse: <BlockClosure>
Invoke the receiver; as long as the result is false, apply the argument, then apply the receiver again. If the result of invoking the receiver is neither true nor false, an error occurs.

<BlockClosure> whileTrue
Equivalent to <BlockClosure> whileTrue: [].

<BlockClosure> whileFalse
Equivalent to <BlockClosure> whileFalse: [].

<BlockClosure> repeat
Equivalent to [true] whileTrue: <BlockClosure>.

ExecutionContext / MethodContext / BlockContext
==========

We reaffirm our commitment to the Smalltalk object-oriented model of execution by proposing not to compromise the unrestricted object-oriented model of accessing execution state. This does not rule out implementations where some more efficient implementation (such as linear stacks) is used most of the time: we only require that the implementation present the following object-oriented interface.

Accessing

<Dictionary> := <ExecutionContext> localVarDictionary
Return a Dictionary-like object for accessing the local (argument and temporary) variables defined in this context, similar to <Object> instVarDictionary. The dictionary covers only variables defined in the receiver context, not in any of its outer scopes: the caller must step through these explicitly using the outerScope message. The[M-]{ name 'thisContext' is recognized in every context; the} name 'self' is recognized only in MethodContexts, and is read-only (not valid for at:put:). 'super' is a language construct, not a variable name.

<ExecutionContext> := <ExecutionContext> sender
Return the context to which this context would return. For BlockContexts, this is the local (fall-off-the-end) return, not the ^ return.

<ExecutableMethod> := <ExecutionContext> method
Return the method which this context is executing.

[M-]{

<Object> := <ExecutionContext> receiver
Return the receiver of the message that created this context. For BlockContexts, this is the BlockClosure, since the BlockClosure was the receiver of the value or value:* message.

}

[M+]{

<MetaObject> := <ExecutionContext> metaReceiver
Return the metaObject of the receiver of the message that created this context. For BlockContexts, this is the MetaBlockClosure, since the BlockClosure was the receiver of the value or

value:* message.
]

<Integer> := <ExecutionContext> sourcePosition
Return the character position in the source code which corresponds to the current execution point in the context. If the context is about to send a message, the position will indicate the first character of the selector; if the context is about to return, the position will indicate the closing] of a block, or just past the end of a ^ statement, or just past the last character of the method; otherwise, the position indicates the approximate locus of control (for example, just after a .). [M?]{This is specific to the standard source language. This should be stated, as well as what requirements must be met independent of source language.}

Execution

All the messages having to do with execution are messages to the process, not to the context. We propose this because there may be implementation-dependent state (such as saved registers) that individual contexts may not be able to locate or update in general. [M?]{ I don't understand this. Isn't a context specific to a given process? Doesn't it know what process it's specific to? If it does, then it can access this state from its process. Therefore, the decision as to whether to locate some specific functionality with the process or the context should be based on expressiveness.}

[M+]{

Process level meta-interpreter

Processes are considered meta-level objects, not because of concurrency, but simply because they are a means of controlling execution. We would like to see these two issues better separated, but have not yet done so ourselves. (A place to look is the Logix operating system: "computations" are the means for meta-interpretively controlling execution, but a single "computation" may contain zillions of processes.)

A process needs to control execution at the granularity of individual message sends (but not variable accesses). We need to enhance our meta-interpretive story so that the Process object is responsible for carrying out each send operation of the execution it is animating. In order to understand this issue, we present an explanatory implementation for "<Process> step". A running process is one which is sending "step" messages to itself in a loop.

```
<Process> step
    "Note: this code doesn't deal with Semaphores or Waiting"
    | nextStepAction currentContext nextContext |
    nextStepAction _ self nextStepAction.
    currentContext _ self currentContext.
    nextStepAction = #return
            ifTrue: [nextContext _ currentContext sender]
        ifFalse: [nextStepAction = #advance
            ifTrue: [nextContext _ currentContext advance]
        ifFalse: [nextStepAction = #send
            ifTrue: [nextContext _ self stepSend: currentContext]]]
    nextContext isNil
        ifTrue: [state _ #Dead]
        ifFalse: [self currentContext: nextContext]

<Process> stepSend: currentContext
    | reciever metaReciever message method args |
    reciever _ currentContext recieverOfSend.
```

```
        message _ currentContext messageToBeSent.
        currentContext advance.
        metaReciever _ securityArea metaObjectFor: reciever.
            "Note: this implies that a securityArea is
                part of the state of a process"

        metaReciever isNil
            ifTrue: [ "This process doesn't have meta-level
                        permission for the target object, so
                        just send the message normally"
                    ^ reciever :: message]
        method _ metaReciever methodForMessage: message.
        args _ metaReciever argsForMessage: message.
        ^ Context
            sender: currentContext
            metaReceiver: metaReciever
            method: method
            args: args.

    <Context class> sender: sender metaReceiver: metaReceiver method: method args: args
        | params temps localVarDictionary |
        params _ method params.
        temps _ method temps.
        localVarDictionary _ Dictionary new.
        args size = params size ifFalse: ["indicate error" ...]
        1 to: args size do: [ :i | localVarDictionary at: (params at: i) put: (args at: i)].
            "is there any better way to map down two lists in parallel?"
        temps do: [ :temp | localVarDictionary at: temp put: nil ].
        ^ super new
            sender: sender
            metaReceiver: metaReceiver
            method: method
            localVarDictionary: localVarDictionary
            pc: 0

    <Context> advance
        pc _ method pcAfter: pc
            withMetaReceiver: self metaReceiver
            withBindings: self localVarDictionary
        ^ self

    <Context> recieverOfSend
        ^ method evalReceiverOfSendAt: pc
            withMetaReceiver: self metaReceiver
            withBindings: self localVarDictionary

    <Context> messageToBeSent
        ^ method evalMessageAt: pc
            withMetaReceiver: self metaReceiver
            withBindings: self localVarDictionary

    <MetaObject> argsForMessage: message
        ^ message args

    <MetaNonMethodicalObject> argsForMessage: message
        ^ Array with: message
```

```
<Process> nextStepAction
    ^ self currentContext nextStepAction

<Context> nextStepAction
    ^ method stepActionAt: pc
```

This demands that <ExecutableMethod>s respond to the following protocol:

```
<Symbol> := <ExecutableMethod> stepActionAt: <pc>
```

For a given pc, if the above returns #advance or #send, then:

```
<pc> := <ExecutableMethod> pcAfter: <pc>
```

returns the pc of the next block of code after the given pc.

If the above returns #send, then:

```
<Object> evalReceiverOfSendAt: <pc> withMetaReceiver: <MetaObject> withBindings: <Dictionary>
<Object> evalMessageAt: <pc> withMetaReceiver: <MetaObject> withBindings: <Dictionary>
```

This meta-interpreter doesn't deal with sends to 'super'. In order to do so cleanly, we probably need a #sendSuper in addition to the above types of stepAction. It also doesn't deal with the nested evaluation of arguments. Peter informs me that an attempt to do so for Interlisp proved difficult. This should probably wait for a continuation passing story.

This meta-interpreter is not complete, and is probably not consistent. It would be interesting to actually get such a meta-interpreter working. We will then probably understand the semantics of what we are doing much better.

}