Abstract

In this work, we present examples for metaprogramming using `thisContext` and mixins and their implementation in Squeak/Pharo, both of which are dialects of Smalltalk.

1 Reflection and Metaprogramming in Smalltalk

Metaprogramming is a program's ability to reason about itself. In an abstract sense, this means that programs can be treated as ordinary data and, therefore, be analyzed and changed at runtime. In this section, we show an example for Smalltalk's `thisContext`, which is a mechanism to control the bytecode interpreter.

1.1 Stack Machine Model

Smalltalk source code is compiled to bytecode by a compiler written in Smalltalk itself. The bytecode is then executed by a virtual machine. Smalltalk's bytecode follows the stack machine model. When a message send should be performed, the compiler generates bytecode that pushes the receiver and all arguments onto the stack, followed by a `send:` instruction with the selector.

1.2 Accessing Stack Frames

Smalltalk provides a special keyword `thisContext` evaluating to an object representing the current stack frame. That object has methods for accessing and modifying stack variables (e.g., local variables), modifying the program counter and jumping to an arbitrary address within the executing method, and accessing the sender's stack frame. `thisContext` is an instance of class `MethodContext` which lets programmers effectively control the bytecode interpreter within the guest language. The compiler translates `thisContext` to a bytecode instruction that generates an instance of `MethodContext` and pushes it onto the stack.

Protocol The following list gives an overview of some interesting methods provided by `MethodContext`[1].

- `jump:`: Modifies the program counter to perform a relative jump in the bytecode of the current method.
- `method`: Returns the `CompiledMethod` object of the executing method. This object contains meta information about the method and its bytecode.
- `tempAt:`: Accesses a temporary variable of this stack frame using its index.
- `tempAt:put:`: Sets a temporary variable of this stack frame using its index.
- `pc`: Returns the current program counter, an index into the bytecode of the method of this stack frame.
- `pc:`: Sets the current program counter.
- `push:`: Pushes a value onto the stack, effectively increasing the size of this stack frame.
- `receiver`: Returns the receiver of this stack frame.
- `return:`: Causes the current stack frame to return with a certain value.
- `sender`: Returns the stack frame below this frame.
- `swapReceiver:`: Sets the receiver of this stack frame.

[1] Some methods are stored in a superclass of `MethodContext`, but we only mention `MethodContext` in this work.

Optimizations In the presence of a just-in-time compiler, a virtual machine might never generate a `MethodContext` object unless the programmer uses `thisContext`. Some of the methods shown below first try to use a primitive in the virtual machine (for performance reasons) and execute the shown code only as a fallback, in case the virtual machine does not provide an implementation for the primitive. Nevertheless, the programmer can change these methods at any time to force that the Smalltalk code is executed instead of the implementation in the virtual machine.

1.3 Example: Exceptions Implemented in the Guest Language

The mechanism for traversing the method stack and controlling the bytecode interpreter can be used to implement exception handling in the guest language, such that the underlying virtual machine does not have to be aware of that mechanism. The benefit of this implementation approach is that more functionality can be implemented in high-level Smalltalk code, resulting in a *smaller* virtual machine. This is not only a *cleaner* approch with respect to architectural design[2], but also us to use the guest language as a playground for new language features, since it is no longer necessary to recompile the virtual machine or to even restart the running Smalltalk system for a language modification to take effect.

Exception Handling Exceptions are raised in Smalltalk by sending the message `signal` to an exception object. An exception is caught by the first exception handler contained in a stack frame. The message `on:do:` can be sent to a block closure containing the code throwing an exception with the type of exception and an exception handler block as arguments.

```
HttpRequest»titechWebsiteContent
    [ ↑ self httpGet: 'http://www.titech.ac.jp/' ]
       on: Exception
       do: [ :e | ↑ '<b>Unable to send HTTP GET request' ]

HttpRequest»httpGet: aURL
    "..."
    timedOut ifTrue: [ Exception new signal ].
    "..."
```

If we run the execute `titechWebsiteContent` in the previous code snippet and the network host cannot be reached, `httpGet:` will signal (throw) an exception which will be handled by the exception handler block in the first method.

Implementation of Exception Handling This paragraph describes the implementation of exception handling in Squeak/Smalltalk using `thisContext`. The following source code snippets are taken from a Squeak 5.0 image and simplified. A number of subtle details are omitted such as checking if an exception handler should handle a certain type of exception.

The method `Exception»signal` uses the method `nestHandlerContext` to find a stack frame with an exception handler. To detect such a stack frame, the method `on:do:` starts with a primitive call. That primitive is not implemented in the virtual machine, i.e., when that method is executed, the primitive fails immediately and the code after the primitive statement is executed. The primitive only acts as a method marker (annotation). Note that in method `signal` the keyword `thisContext` refers to the stack frame executing the `signal` method. There is a stack frame calling that method on the stack below that frame, and we assume that there is also a stack frame containing an exception handler somewhere below that frame.

```
Exception»signal
```

[2] In the best case, we would like to have all functionality implemented in the guest language itself.

```
    "Ask ContextHandlers in the sender chain to handle this signal.  The default is to execute
        and return my defaultAction."
    ↑ thisContext nextHandlerContext handleSignal: self

BlockClosure»on: exception do: handlerAction
    "Evaluate the receiver in the scope of an exception handler."
    <primitive:  199> "just a marker, fail and execute the following"
    ↑ self value
```

The method `nextHandlerContext` finds the next stack frame containing an exception handler by iterating through all stack frames until one stack frame is marked, i.e., it contains a primitive call with number 199. A different approach could check if the method of a stack frame is the compiled method object `BlockClosure»on:do:`, but checking the primitive number might be more efficient.

```
MethodContext»nextHandlerContext
    "Return the next handler marked context, returning nil if there is none.  Search starts with
        self and proceeds up to nil."
    | ctx |
    ctx := self.
        [ ctx isHandlerContext ifTrue:[ ↑ ctx ].
        (ctx := ctx sender) == nil ] whileFalse.
    ↑ nil

MethodContext»isHandlerContext
    "Is this a context for a method that is marked?"
    ↑ method primitive = 199
```

Once a stack frame handling exceptions was found, the handler must be executed and the method containing the handler must return. Notice that the `return:` message send in the following source code snippet is not a regular method return statement but a method defined on class `MethodContext`. It causes that method to return with a certain result, regardless of where the program counter points to. This automatically terminates all stack frames on top of that frame. The method `tempAt:` is used to retrieve the second temporary variable, which is the second argument to `on:do:` (exception handler block). The method `cull:` tries to execute the exception handler block with the exception object as argument or without any arguments in case the block does not take any arguments.

```
MethodContext»handleSignal: exception
    self return: ((self tempAt: 2) cull: exception)
```

Implementation without Metaprogramming Exception handling is typically implemented in the virtual machine/interpreter. Stack frames can be marked as exception handlers by setting a flag similarly to the primitive call in the example above. Raising an exception translates to a primitive call or a special bytecode instruction, upon which the virtual machine traverses the stack of method frames until it finds one that is marked. This mechanism is very similar to the mechanism described above. Smalltalk is special in a sense that stack frames are guest language object and accessible and modifyable in the guest language.

It is not obvious how to implement exception handling in the guest language without using metaprogramming. One very tedious approach would have every method return a tuple of the actual return value and an optional exception object. As soon as a method call returns, the sender first checks if the tuple contains an exception object and, if so, returns immediately with that exception object as well. Otherwise, it proceeds with proceeds with the execution, possibly using the actual return value of the called method. This mechanism requires modifying every return statement and every method call, but it might be possible to do this transformation automatically using macros.

2 Mixins in Smalltalk

A mixin is an abstract subclass that can be applied to multiple (super)classes. Mixins are typically used to share methods that are common to multiple classes, such that the source code does not have to be duplicated. Most Smalltalk dialects do not support mixins out of the box[3], but it is easy to implement rudimentary mixin functionality in Squeak. The last part of this section describes how to do that.

2.1 Protocol

Classes are defined in Squeak using a message send to the superclass. The following snippet defines a subclass of `Object`.

```
Object subclass: #NewClass
   instanceVariableNames: 'foo bar'
   classVariableNames: 'qux'
```

The following listing shows how to apply three mixins during class definition. A mixin is an ordinary class but not meant to be instantiated.

```
Object subclass: #NewClass
   instanceVariableNames: 'foo bar'
   classVariableNames: 'qux'
   mixins: { Mixin1. Mixin2. Mixin3 }
```

2.2 Example: `Comparable` Mixin

In this example, we assume that an application needs two classes `Time` and `Money`. Since our application should be deployed in an international environment, class `Time` must be aware of time zones and class `Money` should support multiple currencies. For that reason, we do not want to use temporal and numeric classes provided by the execution environment.

The following listing shows how these two classes are defined.

```
Object subclass: #Time
   instanceVariableNames: 'hour minute second timeZone'
   classVariableNames: ”.

Object subclass: #Money
   instanceVariableNames: 'amount currency'
   classVariableNames: ”.
```

A frequent operation in our application involves comparing instances of `Time` and instances of `Money`. Therefore, both classes should understand the methods `<`, `<=`, `>`, `>=`, `=`, and (inequality). We first present an implementation without mixins and then an implementation with mixins.

Without Mixins The following source code snippest shows how to implement `Time` without mixins. `Time` has a method `gmtTime` which returns the time in seconds according to the GMT time zone. Methods for comparing two `Time` instances compare this value.

```
Time»gmtTime
   ↑ (self hour * 3600 + self minute * 60 + self second - self timeZone gmtDifference * 3600) ↩
      \\ 86400
```

[3] Newspeak is similar to Smalltalk and supports Mixins.

```

Time»< other
   ↑ self gmtTime < other gmtTime

Time»<= other
   ↑ self gmtTime <= other gmtTime

Time»> other
   ↑ self gmtTime > other gmtTime

Time»>= other
   ↑ self gmtTime >= other gmtTime

Time»= other
   ↑ self gmtTime = other gmtTime

Time»∼ other
   ↑ self gmtTime ~ other gmtTime
```

The following source code snippest shows how to implement `Money` without mixins. `Money` has a method `toUSD` which returns the amount in US dollars according to current exchange rate. Methods for comparing two `Money` instances compare this value. Note that methods for comparing instances are similar in `Time` and `Money`. In the next paragraph, we will get rid of this code duplication using mixins.

```
Money»toUSD
   ↑ self amount * (WebRequest queryRate: self currency to: 'USD').

Money»< other
   ↑ self toUSD < other toUSD

Money»<= other
   ↑ self toUSD <= other toUSD

Money»> other
   ↑ self toUSD > other toUSD

Money»>= other
   ↑ self toUSD >= other toUSD

Money»= other
   ↑ self toUSD = other toUSD

Money»∼ other
   ↑ self toUSD ~ other toUSD
```

With Mixins We first define a mixin `Comparable` providing methods for comparing instances of any kind of class, given that the class provides implementations for `=` and `>`. Based on these two methods, the remaining for methods can be implemented as follows.

```
Object subclass: #Comparable
   instanceVariableNames: ''
```

```
   classVariableNames: ''.

Comparable»= other
   self subclassResponsibility.

Comparable»> other
   self subclassResponsibility.

Comparable»< other
   ↑ (self = other | (self > other)) not

Comparable»<= other
   ↑ (self > other) not

Comparable»>= other
   ↑ self = other | (self > other)

Comparable»~ other
   ↑ (self = other) not
```

We now define `Time` and `Money` as subclasses of `Object` with the mixin `Comparable`. This means that these classes are subclasses of a mixin application of `Comparable`, which is a subclass of `Object`. Notice that we only have to implement the methods `>` and `=` along with the converter methods `toUSD` and `gmtTime`, removing the code duplication partly.

```
Object subclass: #Time
   instanceVariableNames: 'hour minute second timeZone'
   classVariableNames: ''
   mixins: { Comparable }.

Object subclass: #Money
   instanceVariableNames: 'amount currency'
   classVariableNames: ''
   mixins: { Comparable }.

Time»> other
   ↑ self gmtTime > other gmtTime

Time»= other
   ↑ self gmtTime = other gmtTime

Money»> other
   ↑ self toUSD > other toUSD

Money»= other
   ↑ self toUSD = other toUSD
```

2.3 Implementation of Mixins

The following listing describes how mixins can be implemented in Squeak using metaprogramming. We provide a new method that takes an additional collection of mixins during subclassing. For every mixin, the algorithm generates a new subclass with the instance/class variables of the mixin and adds the methods of the mixin to that subclass. This is possible because Smalltalk allows generating new classes and adding new methods from source code while a program is running. Class-side methods and instance variables are added to the meta class object, which can be obtained by sending the message `class` to the class object[4].

```
Class class»subclass: name instanceVariableNames: instVarNames classVariableNames: classVars
    mixins: mixinClasses
  | result |
  result := self
  mixinClasses withIndexDo: [ :cls :idx |
    result := result subclass: name, '_', idx asString
      instanceVariableNames: cls instVarNames
      classVariableNames: cls classVarNames.
    cls methodDict do: [ :sel :meth | result compile: meth getSource ].
    cls class methodDict do: [ :sel :meth | result class compile: meth getSource ] ].
  ↑ result subclass: name
    instanceVariableNames: instVarNames
    classVariableNames: classVars
```

[4]For that reason we write `Classname class»methodName` to denote class-side methods.