Reflectivity

??Marcus Denker and ??Steven Costiou and ??Vincent Aranega

October 1, 2018

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Introduction

- why we need reflection and...s
- ...what is reflection
- introduce Reflectivity
- Explain how the book is structured and how to read it

CHAPTER 2

MetaLinks

2.1 What is a MetaLink?

A trivial example

Basics

Metalinks are annotations of AST nodes that can be executed at runtime to provide behavioral variations or code instrumentation. Such an annotated AST node is expanded, compiled and executed on the fly. A link can be defined by instantiating a MetaLink object:

```
metalink := MetaLink new.
```

The link needs to be configured with a *metaobject* and a selector. When the link is be executed, the configured selector is sent to the *metaobject*, which executes the corresponding behavior. The following example configures the link to send the *#now* message to the *Halt* class:

```
metalink metaObject: Halt.
metalink selector: #now.
```

The link now needs to be installed, for example on the method *#add:* of *OrderedCollection*. It will not be installed on the method itself, but on its AST node. The *link:* message has to be sent to this particular AST node on which we want to install the link, and takes the link object as an input parameter:

```
method := OrderedCollection lookupSelector: #add:.
node := method ast.
node link: metalink
```

The system will now halt each time the *add:* message is sent to an *OrderedCollection*. Note that the *add:* method is shared in the class hierarchy of collections in Pharo. It means that any instance of a class that shares the same version of method on which the link has been installed will halt upon reception of an *add:* message.

The link can later be uninstalled so that it will have no more effect:

```
metalink uninstall
```

Removing or modifying a method with annonated nodes will automatically uninstall all present links from the AST of this method (*i.e.* the method node and all its children).

Reifying arguments

In the previous example, the *add:* method takes a parameter. We might want to access that parameter when the link executes so that we can use it. It is done by providing the *#arguments* reification to the link. Each time the *add:* message is sent to a collection, the metaobject bloc will execute and will take as an input parameter the original argument provided to the *add:* method:

```
metalink := MetaLink new.
metalink metaObject: [:arg| Transcript show: arg printString].
metalink selector: #value:.
metalink arguments: #(#arguments).
(OrderedCollection lookupSelector: #add:) ast link: metalink
```

The following code will print *'Hello World'* in the *Transcript*:

```
OrderedCollection new add: 'Hello World'
```

Controlling the execution position

For a given node, metalinks can be put at different positions:

- *#before*: the metalink is executed before the execution of the node
- *#instead*: the metalink is executed instead the node
- *#after*: the metalink is executed after the execution of the node

For example, we could configure our link to print the parameter after the *add:* message has returned:

```
metalink control: #after
```

```
Not all the nodes provide all the positions. For example, literals don't provide onError and onSuccess positions.
```

Activation and deactivation

A link can be activated, deactivated. Conditions

Composing links

Put links on the same node, and discuss the execution order and also the #instead case.

2.2 Basic examples

2.3 Instance specific MetaLinks

Since *Pharo 7*, metalinks can be installed on specific objects. When installing a link on a node, instead of affecting all instances of the class in which the node is located, only a given object will be touched.

The entities we manipulate do not change:

- AST nodes
- instances of MetaLink

What changes is the way to install a link on a node. When linking to a node, we will now specify for which object the metalink will be active. In the following code, only the `date` object will be affected, and any other instance of `Date` will not have the link installed on the *#asDate* method node:

```
date := Date new.
link := MetaLink new.

node := (Date lookupSelector: #asDate) ast.
node link: link forObject: date
```

To the contrary of *standard* links, only AST nodes can be targetted by instance specific links, *i.e.* instances of any subclass of `RBProgramNode`. That means that using this interface, we cannot put instance specific links on slots or variables. They can however be put on any kind of AST node.

Installing an object specific link has also the effect to migrate the object to an anonymous subclass of its original class. This anonymous class can be viewed while inspecting this particular object or by introspection tools. The object however keeps all its attributes and behaviors, and is different only by the links installed on its nodes.

Putting multiple instance specific links is transparent, and only one anonymous subclass will be generated for the same object. *Standard* links are conserved, so it is possible, for example on a given method node, to have metalinks active for all instance of the object's class, and metalinks only active for this one object.

The standard interface *#removeLink:* cannot be used to remove instance specific links: it would remove all standard links on the node but not the object-specific ones. Removing an instance specific can be done by using the following interface instead:

To do Need to check that in the code, and write a test about it!

```
node removeLink: link fromObject: date
```

However, uninstalling a link through the *#uninstall* interface will always work. In that case, the MetaLink will be removed from every node it has been installed on:

```
link uninstall
```

Once the last instance-specific link is removed from the object, then it silently migrates back to its original class.

It is worth noting that for every (un)install operation, the nodes are the one we can recover from the system classes. We never have to worry about the object changing its class by an anonymous one, as metalinks are able to resolve their targets during their (un)install operations.

2.4 The MetaLink API

Navigating the AST to find a particular node on which install a link can be tedious and error-prone. Furthermore, it does not seem natural that all linking operations are performed on AST nodes while in reality what we meant is to install a link that will be active for a particular object, or for all instances of a given class.

As a more practical way of installing links, object and classes provide a helper interface that will find AST nodes and install metalinks on it without any user intervention.

This interface is common to objects and classes. In the following, the MetaLink API is described and explained, aswell as its immediate effects.

MetaLink interface

The MetaLink interface can be used either on classes or on their instances. The difference in effect is that links will be active for all instances of a class, if called on a class, or only for a specific object.

The `entity` variable in the code snippets below is either a class or an instance of a class.

Direct linking to a given AST node

Giving an AST node, one can ask a class or an object to install a link on it. It is technically the same as using the *#link:* interface on AST nodes, but expressed in a more *object-centric* way.

```
anAstNode := ...get an ast node somewhere...
entity link: metalink toAST: anAstNode.
```

Direct linking to set of AST nodes

It is the same interface as above, except that the linking is made on a group of node. The link is installed for `entity` on each of these nodes.

```
entity link: metalink toNodes: aSetOfNodes
```

Linking to a method name

The link will be installed on the method described by the selector given as parameter. The system will actually lookup for the first method in the class hierarchy of the entity that implements this method, and will raise an error if none is found. While it poses no particular problem if `entity` is an object, it is a bit more subtle for classes. For a given class `C`, if the method is actually implemented higher in its class hierarchy for example in one of its superclasses `C0`, then the link will be installed on the method found in `C0`. It means that all instances of subclasses of `C0`, including `C`, will be affected by the link.

```
entity link: metalink toMethodNamed: #asDate
```

Linking to slots and variables

This interface is available for classes and for individual objects. It provides helper methods to install metalinks on class variables, slots and temporary variables. Variables and slots are always referenced by their name. More informations are usually required to input by the user, for example the method name in which the wanted temporary variable is located.

Installing a metalink on a slot or a variable will actually install the link on all ast nodes that either read or write the target slot or variable.

In the following, all code snippets use the class `ReflectivityExamples` as an example: the entity on which we install metalinks can be either the class `ReflectivityExamples` and links will affect all instances of this class, or an instance of `ReflectivityExamples` in which case the link will only affect this object.

Linking to class variables

#ClassVar is the name of the class variable on which we want to install the link.

```
ReflectivityExamples link: metalink toClassVariableNamed: #ClassVar.
ReflectivityExamples new link: metalink toClassVariableNamed:
    #ClassVar.
```

Linking to slots

#ivar is the name of the target slot on which we want to install the link.

```
ReflectivityExamples link: metalink toSlotNamed: #ivar.
ReflectivityExamples new link: metalink toSlotNamed: #ivar.
```

Linking to temporary variables

#temp is the name of the temporary variable object in the method *#exampleAssignment*. We always have to provide the method name in which the temporary is located.

```
ReflectivityExamples link: metalink toTemporaryNamed: #temp
    inMethod: #exampleAssignment.
ReflectivityExamples new link: metalink toTemporaryNamed: #temp
    inMethod: #exampleAssignment.
```

API options

An option can be specified when installing on a variable, so that assignments nodes or reads nodes of this variable can be specifically targeted. The option is specified through the *#option:* keyword for each interface, respectively:

```
ReflectivityExamples link: metalink toClassVariableNamed: #ClassVar
    option: #read.
ReflectivityExamples link: metalink toSlotNamed: #ivar option:
    #write.
ReflectivityExamples link: metalink toTemporary: #temp option: #all.
```

The option value can be one of the following:

- #read will install the permalinks on all read nodes of the target entity
- #write will install the permalinks on all assignement nodes of the target entity
- #all both read and assignment nodes will be targeted

When the option is not specified when calling an interface, it is equivalent to the #all option, *i.e.* it will be installed on all nodes referencing the target variable or slot.

2.5 PermaLinks

A `PermaLink` is a metalink that is persistent. It must be installed through a dedicated interface, and only target specific nodes which are slots, class variables and temporary variables read and assignment nodes. A permalink is unique, but is bound to a `MetaLink` that can have multiple permalinks associated.

When installed on one of these kind of nodes, permalinks ensure that they are always reinstalled when a method with such nodes is added or modified. It is meant to be permanent in the system. For example a permalink installed on all read nodes of a slot will automatically be installed on a new method with read nodes of the same slot. It will also be reinstalled on a modified method if any read node still exists for this slot (even if the source code has radically changed).

Permalinks are removed from all of their nodes when the user calls the *#uninstall* method of the bound metalink.

Although permalinks can be per-object or for all instances of a class, they are compatible with all other metalinks. Removing all other metalinks from a method (for example) will not remove permalinks, unless its link is explicitely uninstalled.

PermaLink API

The following example illustrates how to install permalinks on temporary and class variables and on a slot. The interface is similar to the metalink API, at the difference that it uses the *#permaLink:* keyword instead of *#link:*. In the code snippet below, permalinks are installed on an instance of the `ReflectivityExamples` class for a temporary variable and a class variable, while it is installed on the slot *#ivar* for all instances of `ReflectivityExamples`.

The *#option:* parameter defines the persistent nature of the permalink. It is by default installed on all user nodes of the target entity if it is not specified, or at the discretion of the user on read or write nodes of the target entity.

```
|example|
example := ReflectivityExamples new.

"PermaLink on all user nodes of the temporary named #temp in
    #exampleAssignment"
example permaLink: metalink toTemporaryNamed: #temp inMethod:
    #exampleAssignment.

"PermaLink on all read nodes of the class variable named #classVar"
example permaLink: metalink toClassVariableNamed: #ClassVar option:
    #read.

"PermaLink on all assignment nodes of the slot named #ivar"
ReflectivityExamples permaLink: metalink toSlotNamed: #ivar option:
    #write
```

PermaLinks options

A permalink can specify one option that defines on which nodes it will be permanently installed. Options are the same as the MetaLink API options.

CHAPTER 3

Deeper into MetaLinks

3.1 Reflectivity reifications

Often when a link is executed, we need information from the base level at the meta level. These informations can be reified by passing the link reification arguments. Reifications of concepts from the base level are modeled by subclasses of the `RFReification` abstract class.

Each time we request a link to reify base level informations, we need to provide it with a reification key that will be used to recover the proper objects at runtime. This is done using the *#arguments:* message:

```
link := MetaLink new.
link arguments: #(...keywords...)
```

The keywords are symbols, and each one represents a given reification. Not all nodes can support all the possibilities, for example it makes no sense to ask for an argument reification on a variable node. Asking for a reification that is not available for a given node will raise an error at compile time. It can cause program interruptions if metalinks are dynamically created and installed at runtime.

In the following, all available reifications are listed and described. For each possibility, you will find the corresponding keyword to use to ask for the reification and the kind of nodes it can affect. A short example to demonstrate the usage will also be explained for each reification. We assume that a metalink has been previously instantiated so that we can use it in our code snippets. For the sake of simplicity, the metaobject will always be represented as a block with the reified entities and the *#value:* selector to be sent to this block.

RFArgumentsReification

I reify arguments from the original node on which the link is installed. When the link is executed, i will provide an array with the arguments (if any) in the same order as the original node arguments were passed.

Reification key: #arguments

Available for:

- `RBMessageNode`
- `RBMethodNode`
- `RBBlockNode`

In the example below, `args` is an array with the arguments values of the original node:

```
link metaObject: [:args| ...].
link arguments: #(#arguments)
```

RFClassReification

I reify the class of the object in which i am executing.

Reification key: #class

Available for:

- `RBProgramNode`
- `LiteralVariable`
- `Slot`

If the link in the code below is installed on a method node of the class `Date`, or a method node of an instance of `Date`, then at execution time `class` will be the `Date` class:

```
link metaObject: [:class| ...].
link arguments: #(#class)
```

RFReceiverReification

I reify the receiver of a Message or a method

Reification key: #receiver

Available for:

- `RBMethodNode`
- `RBMessageNode`

Imagine the following code, and that `node` is the message node representing the *#printString* message send to the created instance of `Date`:

```
Date today printString.
```

In the example below, `rcv` is the instance of `Date` created by the code *Date today*, which is the receiver of the *#printString* message:

```
link metaObject: [:rcv| ...].
link arguments: #(#receiver)
```

RFEntityReification

I stand for the structural entity that the link is installed on. It is *#node* for AST nodes or *#variable* for variables.

Reification key: #entity

Available for:

- `RBProgramNode`
- `LiteralVariable`
- `Slot`

RFLinkReification

I reify the link itself.

Reification key: #link

Available for:

- `RBProgramNode`
- `LiteralVariable`
- `Slot`

RFNameReification

I reify the name of variables. I will reifiy either the variable name in case or variables or slots, or the variable of an assignment.

Reification key: #name

Available for:

- `RBVariableNode`
- `RBAssignmentNode`
- `LiteralVariable`
- `Slot`

RFNewValueReification

I reify each new value of links put on assignments or on temporary variable nodes. That means that the link will execute each time there is a new value stored in the temporary or through an assignment.

Reification key: #newValue

Available for:

- `RBVariableNode`
- `RBAssignmentNode`

In the code below, `value` represent the new value stored in the variable:

```
link metaObject: [:value| ...].
link arguments: #(#newValue)
```

RFNodeReification

I am the node on which the link is installed on.

Reification key: #node

Available for:

- `RBProgramNode`

```
link metaObject: [:node| ...].
link arguments: #(#node)
```

RFObjectReification

Using the ObjectRefification, one can pass a pointer to the object where the link is installed in.

Reification key: #object

Available for:

- `RBProgramNode`
- `LiteralVariable`
- `Slot`

If the link is installed on the `Date` class, then `object` below is the current instance of `Date` for which the link is executing:

```
link metaObject: [:object| ...].
link arguments: #(#object)
```

RFSelectorReification

I am the selector of a message send or method. The reified entity is a symbol.

Reification key: #selector

Available for:

- `RBMessageNode`
- `RBMethodNode`

RFSenderReification

I reify the sender for message sends and methods. The sender is always the `ReflectiveMethod` that triggered the link execution.

Reification key: #sender

Available for:

- `RBMessageNode`
- `RBMethodNode`

In the example below, the `sender` entity is the `ReflectiveMethod` associated to the *#asDate* compiled method of class `Date`:

```
|link node|
node := (Date lookupSelector: #asDate) ast.
link := MetaLink new.
link metaObject: [:sender| ...].
link selector: #value:.
link arguments:#(#sender).
node link: link.
Date today asDate
```

To check... it is a bit troubling to speak about the ReflectiveMethod here, because there were no mention of it before... Maybe we should introduce it... ?

RFThisContextReification

I can be used to pass the context to the meta object.

Reification key: #context

Available for:

- `RBProgramNode`
- `LiteralVariable`
- `Slot`

RFValueReification

I am the value of a variable read or assignment.

Reification key: #value

Available for:

- `RBVariableNode`
- `RBAssignmentNode`
- `RBReturnNode`
- `RBMessageNode`
- `LiteralVariable`
- `Slot`

In the code below, `value` represent the value of the variable read or variable assignment:

```
link metaObject: [:value| ...].
link arguments: #(#value)
```

RFVariableReification

I reify a variable, that can be of kind:

- GlobaVar (or one of its subclass)
- Slot for istance variables

Reification key: #variable

Available for:

- `RBVariableNode`
- `LiteralVariable`
- `Slot`

Does not work for temps ?

In the code below, `var` represents the variable on which the link is installed. It is not a value but an object (a slot, a variable or a variable node).

```
link metaObject: [:var| ...].
link arguments: #(#variable)
```

3.2 MetaLink options

- Describe the options system

- All the options, their meaning, their usage
- optionInlineMetaObject
- optionInlineCondition
- optionCompileOnLinkInstallation
- optionOneShot
- optionMetalevel
- optionDisabledLink
- option argsAsArray

3.3 More complex examples

- Using multiple reifications
- Multiple links on the same node
- ...

The Reflectivity version we refer to is the following: https://github.com/StevenCostiou/Re
dev

CHAPTER 4

MetaLinks in action

CHAPTER 5

MetaLinks trick and tips