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

Deeper into MetaLinks

1.1 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)
```

1.2 MetaLink options

- Describe the options system

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

1.3 More complex examples

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