Multicore Programming
The SLY3 Programming Language

Pablo Inostroza Valdera
<pinostro@vub.ac.be>

23 June 2011

1 Introduction

SLY3 is a Smalltalk dialect that features two concepts in order to deal with parallelism: *ensembles*, which are collections of objects that can receive parallel messages, and modifiers, which decorate messages in order to modify their parallel behavior (*adverbs*) or to specify how to handle reductions (*gerunds*). In this article we analyze SLY3's programming model.

SLY3 is a language being developed in the context of the Renaissance Project[1], at IBM. With SLY3, their authors David Ungar and Sam Adams explore abstractions for nondeterministic parallel programming. SLY3[2] is the experimental successor of LY, a javascript-like interpreted language that was implemented on top of Smalltalk [UA10]. The main idea behind the design of SLY3 is that multicore programming is commonly addressed by restricting the nondeterminism parts and trying to enforce strict determinism. As an example of this, Ungar and Adams refer to the actors paradigm. While the actor model certainly adds some order to concurrent computations, the truth is that some nondeterminism still persists because the global order in which messages arrive to an actor is nondeterministic. They proposed that instead of trying to restrict the nondeterminism, it should be embraced. The LY language and, later on, the SLY3 language are steps towards that direction. In this article we discuss the latter.

2 Ensembles, or How to Represent A Flock of Birds

SLY3's key elements were conceived by observing the nature. Think of a flock of birds, an ant colony or a school of fishes. These are all examples for a multitude of independent agents, acting in a nondeterministic way, whose aggregated behavior is more complex than that of the individuals. In other words, a more complex behavior *emerges*. The idea of a whole formed from independent units

[1] http://soft.vub.ac.be/~smarr/renaissance/
[2] As the 3 in Sly3 indicates, Sly itself underwent already three iterations of design.

is key to SLY3, and it has been reflected in the concept of an *ensemble*. An ensemble is a collection of objects, subject of receiving messages as a whole. In the following example, we create an ensemble and execute an operation on it:

```
numbers := Sly3Ensemble with: {1. 2. 3. 4. 5}.
squaredNumbers := numbers squared.
```

This code yields: `%{1. 4. 9. 16. 25}`. [3]

Notice how the ensemble received a message as a whole, but it acted on each individual in order to produce a new ensemble with all of the elements squared. It is not surprising to know that the default behavior of this *mapping* is parallel, i. e., a different thread of execution is triggered for each individual computation. The standard execution strategy in this this simple example is as follows:

- The message was delegated to members of the ensemble, creating a parallel process for the evaluation of each of the messages. We can say that each member was treated **individually** and was executing its own *task*.
- For generating the final result, a new ensemble was created from the results of all the tasks of the previous step. We can say that we are **ensembling** together all the individual results.

Notice that we have put in bold letters two words: individually and ensembling. We have done so, because both words characterize a strategy for handling a parallel computation. Actually, this is the default strategy in SLY3, but the programming model was designed to be able to customize these behaviors.

3 Adverbs and Gerunds, or How to Modulate Parallel Behavior

The basic model of message handling for an ensemble is:

1. The ensemble that receives the message is treated in accordance to its *adverb*. A receiver's adverb precedes the message name. In the absence of an explicit one, the default adverb for receiver, namely `individualLY`, is used [4]. Adverbs' names always end in *LY*.
2. If there are operands, they have to be treated in accordance with their *adverbs*. An operand adverb follows the message name. In the absence of an explicit one, the default adverb for operands, namely `wholLY`, is used.
3. The operation is executed according to an overall strategy, also determined by an *adverb*.

[3]Recall that the curly braces represent an array in Smalltalk. In SLY3, curly braces preceded by `%` denote an ensemble.

[4]We elaborate on the meaning of `individualLY` in subsequent paragraphs.

Adverb	Informal description
`wholLY`	just pass me along as a whole ensemble
`individualLY`	create a process/thread for each of my members
`collectionLY`	treat me as a collection instead of an ensemble
`duplicativeLY`	copy each of my members, to produce a new ensemble
`roundLY`	create a process for each combination of the receiver's members and my members
`randomLY`	chose `n` of my members randomly, `n` is a parameter of this adverb
`valueLY`	take a block and apply it to each of my members

Table 1: Some of the principal *adverbs* in SLY3.

4. The result of the operation is reduced in accordance to a *gerund*. In the absence of an explicit one, the default gerund, namely `ensemblING`, is used. Gerunds' names always end in *ING*.

Regarding the point 3, it suffices to mention that currently there are only three overall strategies, namely, *serially*, *plainly* and the default one, which we can call *parallelly*[5]. *Serially* forces serial executions, so no processes/threads are spawned to compute the results. This can be useful either for semantic reasons or when the cost of parallel execution is to high, i. e., when we are below the sequential threshold. The adverb *plainly* can be used to turn off ensemble behavior, and therefore, to treat the ensemble as a collection (e. g., we could ask for the size of an ensemble by using plainly, and this would not imply sending a message to each of its members). For the sake of brevity, we are not going to discuss the overall strategies further, since they are not relevant for grasping the essence of the SLY3 programming model.

The syntax of SLY3 is cumbersome, so next we present some examples in order to provide a more clear idea of how to express things in this language. Let's analyze them in the light of three of the steps presented before (1, 2 and 4; 3 is excluded since are not going to discuss the *overall strategy*). In order to guide our analysis, figure 1 presents a table with brief descriptions of some of the main adverbs in SLY3, and their impact on the message evaluation process. We have not included a similar table for gerunds, because their names are in general more self descriptive.

Example 1

```
%{true. true. false. true} andING
```

- In this example, there are no adverbs specified, therefore, the *receiver's* adverb is the default one, i. e., *individually*. This implies that the pro-

[5]Note that the metaphor of the adverb is taken to an extreme in SLY3. Many words that do not sound like adverbs are *adverbialized* in order to fit in the conceptual framework.

cessing of the members is done independently (i. e. in different Smalltalk processes).

- There are no arguments, therefore, no argument's adverb.
- There is a gerund `andING` that acts on the result of the operation reducing the independent results coming from different processes. As its name indicates, `andING` performs logical *and* over the members of the resulting ensemble. Notice that there no actual message specified here, thus, `andING` is applied to unchanged set of members of the ensemble.
- Therefore, the result of this operation is: `false`.

Example 2

```
%{1. 2. 3} plusRoundLY: %{10. 20. 30}
```

- Here are no adverbs specified for the receiver, therefore, the *receiver's* adverb is the default one, i. e., *individually*. This implies that the processing of the members is done independently (i. e. in different processes/threads).
- `roundLY` is an argument to the operand (`%10. 20. 30`). This adverb says: create a process/thread for each combination of the receiver's member and the argument's member; acting as a cartesian product.
- There is no gerund, which means that the default one (`ensemblING`) is used.
- Therefore, the result of this operation is: `%{11. 12. 13. 21. 22. 23. 31. 32. 33}`.

Example 3

```
%{1. 2. 3} valueLY: [ :x | x * 10]
```

- `valueLY` is an argument of the receiver, therefore, the receiver has to be mapped to a new ensemble using the given block as the *mapper*.
- There are no arguments, therefore, no argument's adverb. Recall that the block is an argument to the adverb. As we can see, the problems of Sly3's syntax become evident in these cases.
- Therefore, the result of this operation is: `%{10. 20. 30}`.

4 An Overview of Sly3's Implementation

In order to have a comprehensive view of the spectrum of the current *custom behaviors* available for direct use, in figure 1 we show the hierarchy of gerunds and adverbs that we can find in the current image of SLY3. Notice that this forms a restricted vocabulary that we can use as a basis of more complex strategies, in a LEGO-like way. In fact, we can see this hierarchy as a custom domain-specific language suited for the description of common strategies for handling parallelism.

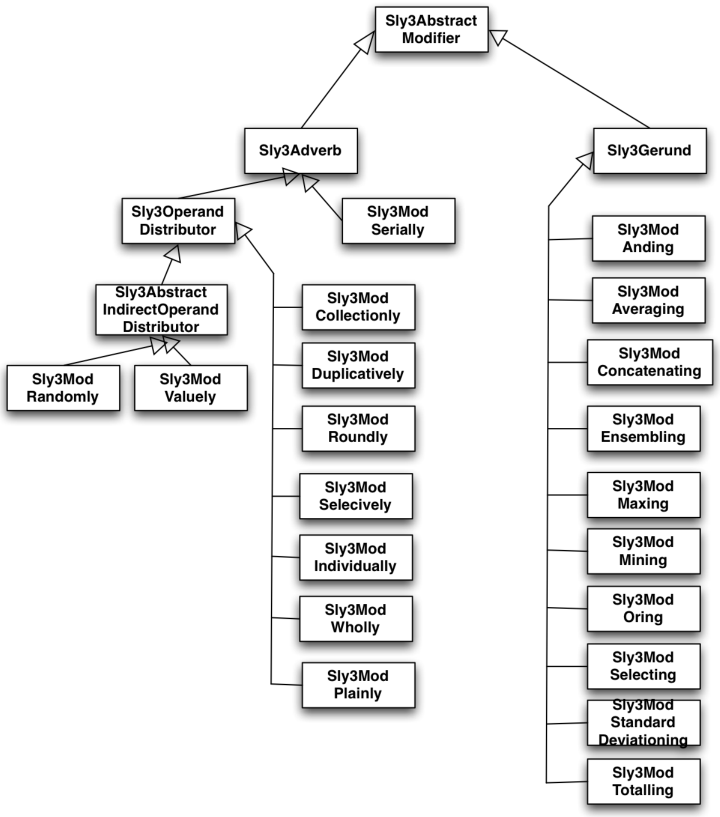

Figure 1: The hierarchy of modifiers of SLY3.

Based on what we have discussed so far, we can characterize SLY3 as:

- An Object-oriented language, descendant of Smalltalk, which incorporates ensembles as first-class entities to model parallelism.
- It features parallelism by default. As soon as a message is sent to an ensemble, it is assumed that the processing will be spawned as independent

tasks over the members of this ensemble.

- Implicit parallelism. The processes that handle messages for ensembles are spawned *under the hood*, and programmers interact without having to switch to a *parallel mindset*.
- The way the receiver and the operands of a message are treated is specified by a set of adverbs.
- The way the result is reduced is specified by a set of gerunds.
- The messages are not reified as for instance in the case of *Communicating Sequential Processes* or *actors*.
- Communication is assumed to be synchronous.
- The processes are not reified as in the case of actor-based computing.

5 A Note on the Evaluation Process

The SLY3 language is implemented as an extension of a Smalltalk Virtual Machine, the RoarVM. The current strategy is that whenever a message to an ensemble is detected, a custom message dispatcher is launched in order to handle it. Thus, the message is parsed and evaluated at runtime[6]. The message is analyzed by partitioning the receiver according to its adverb, and pairing it with the arguments according to arguments' adverbs. Then, the actual operation is executed in whichever set of operands resulted from the previous processing. The final result is reduced according to the gerund. When we use the expression *according to*, it is worth to highlight that the actual behavior is encoded in the classes of the gerunds an adverbs.

Below we give as example the implementation of the `totallING` gerund. In the image it is the only method of the class `Sly3ModTotalling`. Notice, that it nothing more but the implementation of a simple summing policy.

```
; method of the class Sly3ModTotalling
reduceForEvaluator: ev
        "Sum the elements in results and return the result.
        Assumes these elements understand +."

        | result results |
        results := ev result.
        results isEmpty ifTrue:
       [self error:
         'No members, how should we handle?  proceed for nil'.
       ^nil].
        result := 0.
        results do:[:each | result := each + result].
        ev result: result
```

[6]There is current work on optimizing these processes by using a caching strategy

Below we give the code for the two relevant method of the `Sly3ModCollectionly` class that implements the `collectionLY` adverb in Sly3. Although it is necessary to deeply understand the behavior of the process of evaluating a message in order to understand this code, at least we can have an intuition that what is happening at some point is that a collection is created and returned (see line 17). Eventually, this is how the creation of a collection from an ensemble (which is what `collectionLY` does) is implemented.

```
 1  ; methods of the class Sly3ModCollectionly
 2  amendOperandTuples: operandTuplesIncludingThisOne
 3          | ens |
 4          ens := (operandTuplesIncludingThisOne collect: [:ot | ot last]).
 5          ^ operandTuplesIncludingThisOne collect: [:ot |
 6                  ot copy removeLast; addLast: ens; yourself]
 7
 8
 9  extendOperandTuples: operandTuplesSoFar
10             operand: operand
11             membersOrNil: operandMembers
12          | mems |
13          mems := operandMembers ifNil: [operand]
14                                 ifNotNil: [operandMembers].
15          ^ operandTuplesSoFar
16              ifEmpty: [OrderedCollection with:
17                          (OrderedCollection with: mems)]
18              ifNotEmpty: [ operandTuplesSoFar collect:
19                              [:tuple | tuple copy addLast: mems; yourself]]
```

Therefore, each modifier in SLY3 is self descriptive and fully determines the impact that their presence provokes on the evaluation process.

6 Implementing MapReduce in SLY3

In order to show an example of how to use SLY3, we have implemented a *naive* MapReduce framework[7]. It is worth to stress that the conciseness of the code that we eventually obtain is a consequence of both using SLY3 and the application of an Object-oriented approach. The framework relies on ensembles that receive messages. By doing that, the messages are implicitly sent in parallel, so we do not have to specify this in the code. We next discuss the core excerpts of the code.

Imagine two well-known applications of MapReduce. First, we want to count the number of occurrences of words in a set of documents. Another application is to know in which documents some words are found. In order to do that, the MapReduce paradigm requires us to provide the problem-specific knowledge by means of a map procedure and a reduce procedure.

Since SLY3 being an object-oriented language, we have modeled the MapReduce problem-independent behavior as a class `MapReduce`. Any particular imple-

[7]Take a look at the course notes for an Erlang implementation of the same examples.

mentation has to subclass this class and override two abstract methods, namely, `mapForKey:value:` and `reduceForKey:values:`.

Facing the problem of counting words in a document, below are the methods of the class `CountWordsMapReduce` that implement this behavior in SLY3[8].

```
1   "Specific methods of the class CountWordsMapReduce"
2   mapForKey: ignore value: fileName
3           | words |
4           words :=  self consult: fileName.
5           ^ words valueLY: [:y| {y. 1}]
6
7   reduceForKey: word values: counts
8           ^ {word. counts totalING}
9
10  consult: file
11          ^ files at: file
```

Facing the problem of indexing the texts using the words that they contain, below are the methods of the class `TextIndexingMapReduce` that implement this behavior in SLY3.

```
1   "Specific methods of the class TextIndexingMapReduce"
2   mapForKey:  ignore value: fileName
3           |words |
4           words :=   self consult: fileName.
5           ^ words valueLY: [:word| {word. fileName}]
6
7   reduceForKey: word  values: names
8           ^ {word. names members asSet}
9
10  consult: file
11          ^ files at: file
```

Notice that in both cases, it suffices to use combinations of the existing adverbs to achieve the desired functionality (see lines 5, 8 of the first listing and line 5 in the second one). Moreover let's now analyze the class `MapReduce`, that implements all the *machinery* of our naive MapReduce implementation.

```
1   "Specific methods of the class MapReduce"
2   dictionaryToList: dict
3           |lst|
4           lst := OrderedCollection new.
5           dict associationsDo:
6             [:assoc| lst add:{assoc key.
7                     Sly3Ensemble
8                       withMembersFrom: assoc value}].
9           ^ Sly3Ensemble withMembersFrom: lst
10
11  do: input
12          |dict  nonFlatValues flatValues intermediateValues|
13          nonFlatValues:= input valueLY:  [ :pair |
14       self mapForKey: (pair at: 1) value: (pair at: 2)].
```

[8]Notice that for sakes of simplicity, we have modeled a filesystem as a `Dictionary` whose keys are strings that represent file names and whose values are ensembles corresponding to the set of words in a file.

```
15          flatValues := nonFlatValues members
16        inject:  (OrderedCollection new)
17        into: [:acc :cur |
18                acc addAll:
19                      cur members asOrderedCollection.
20                acc].
21          dict := self groupValuesByKey: flatValues.
22          intermediateValues:= self dictionaryToList: dict.
23          ˆ intermediateValues valueLY:
24         [ :pair | self reduceForKey: (pair at: 1)
25                        values: (pair at: 2) ]
26
27  groupValuesByKey: keyValues
28  | dict|
29  dict := Dictionary new.
30  keyValues do:[:keyValue |
31          | key value |
32          key := keyValue at: 1.
33          value := keyValue at: 2.
34          (dict includesKey: key)
35        ifTrue:
36          [(dict at:key) add: value]
37        ifFalse:
38          [dict at: key
39                put: (OrderedCollection newFrom: {value})] .
40          ].
41  ˆ dict
42
43  mapForKey: key value: value
44          self subclassResponsibility.
45
46  reduceForKey: key values: values
47          self subclassResponsibility.
```

As it is possible to see in lines 9, 13 or 23 of the above code, the framework itself also profits from the fact that the data structures that are being passed are ensembles. By having ensembles as the main element being passed around, it is possible to have implicit parallel behavior, which is the key of the MapReduce approach. Evidently, in this case we are discussing a very limited version of MapReduce meant to be used on a single computer, but it is still useful to show how the implicit parallelism in SLY3 works.

7 Conclusions

In the spectrum of language abstractions for dealing with parallelism and concurrency, SLY3 is a language that has opted for a conservative style, in the sense that it is a specific element of the language (an ensemble) the one that is subject of parallelism. On the other hand, the stress has been put on giving a set of tools to programmers, enabling them to combine these tools. For customized strategies, it is the impression of the author of this article that the level of granularity is too coarse-grained. Although it is conceivable to extend the set of adverbs and gerunds by extending the hierarchy aforementioned, this implies to understand the SLY3 implementation in very detail, something that we most likely do not expect from application programmers. It is possible to think of a meta-programming framework that could *inject* custom functionalities in the language, lowering the complexity for extending the set of primitives.

Regarding the approach to parallelism, it is implicit and synchronous. Programmers are not aware of how computations are spawned or scheduled, and the only means they have to initiate parallel computations is by using ensembles.

As regards to the syntax, SLY3 uses decorated keyword based messages to model the way the message is handled by the virtual machine. Although it is conceivable that this schema can be improved, there are some problems originated from the fact that adverbs can decorate receivers, arguments, and the overall strategy. If we add to this complexity the gerunds, it is clear that it is difficult to express these multidimensional facets using a textual syntax.

To sum up, SLY3 is a language that incorporates in its design a series of patterns to deal with parallelism, but the mechanism for extending the set of parallel primitives could be reconsidered, in order to reduce the complexity of extending SLY3. However, the MapReduce example has shown that it is possible to concisely express complex patterns of interactions in SLY3, as soon as these patterns fall within the give set of SLY3's primitives.

References

[UA10] David Ungar and Sam S. Adams. Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 19–26, New York, NY, USA, 2010. ACM.