Lazy Causal Programming for Service Oriented Architectures

Remi.Douence@mines-nantes.fr
Ascola (Mines Nantes)

February 23, 2015

Abstract

Lazy evaluation is an important software engineering technique. It enables to define general (hence reusable) software components, then to compose them in order to get efficient applications. Lazy evaluation is not an optimization technique, it "*only*" makes programs simpler and more reusable. Laziness is quite natural in functional languages. Oddly enough, its quite different for imperative languages. Although this technique is well-know and widely spread (for instance, buffered file access, game trees incrementally built, and web pages loaded on demand are laziness in disguise), in imperative languages, the programmer is responsible for manually programming laziness. This is tedious and this generates bugs. This PhD thesis will provide language mechanisms and tools to support lazy evaluation in an imperative context in general, and in a service-oriented context in particular.

keywords:

Imperative programming, laziness, dependency, semantics, static and dynamic analysis, optimization, distribution, causality, service-oriented architecture.

Introduction

Lazy evaluation is an important software engineering technique [Hug89]. Laziness is back with a vengeance, and although it is not supported (this is the issue) in mainstream languages, a google search for "laziness" and a programming language returns hundred of thousands results: in particular blogs (e.g., C++ [Mil14], Java [SPI14], Python [Nai13]) that explain how to encode it (at your own risk). This PhD thesis proposes to extend preliminary results [DT14] in order to provide supports for laziness in SOAs.

Context and problem

Laziness is central to software engineering. It does not help to get more efficient programs. It helps to write simpler programs by reusing more code [Hug89]. For instance, the function that computes the minimum of a collection can be defined as the composition of a sorting function with the head function that returns the first element of a collection:

```
minimum collection = head (sort collection)
```

However, this definition is quite inefficient because the sort function makes numerous useless computations in this specialized context of use. Lazy evaluation makes such a definition efficient by performing only the computation required by head. Simple definitions that reuse much code are totally inefficient with strict evaluation but they become realistic with lazy evaluation.

Laziness is quite natural in functional languages (lack of side effect makes dependencies explicit and easy to reorder computation). Laziness is not supported in imperative languages (side effects make dependencies implicit and very hard to reorder computation). However, laziness is quite useful in an imperative context. For instance, a file is not read immediately but it is read block by block with a buffer, a game tree is alternatively built and explored, a web page is incrementally downloaded. Laziness exists at the software architecture level too (e.g., batch *versus* pipe-filter, push *versus* pull [SG96]).

Problems and opportunities

In an imperative context, the programmer is responsible for manually programming laziness. This is tedious, so programs are not as lazy as the could be. This is dangerous, because programs can be too lazy and compute wrong results.

So, programmers do not use much laziness and general software components (they are so general they require laziness to get efficient as exemplified above with the sorting function in the context of head).

In a distributed context the problem is the same (but the notion of sequence of computation and dependency rely on causality [Lam78]). Service-oritented architectures rely on service composition (i.e., orchestration and choreography). Here again, for efficiency concerns services are not as general as they could be.

If dependencies between blocks/services could be declared (or inferred) an algorithm [DLL09, DT14] could perform lazy evaluation by delaying or forcing cascade evaluation of blocks/services.

Moreover, once computation are reified (in order to delay them) the algorithm could introduce dynamic optimizations [GLJ93]. For instance, when a collection must be sorted twice, once is enough: the two delayed blocks can be replaced by a single one.

Workplan

Objectives

The objective of this work is to propose tool support for lazy programming in a service-oriented context. This tool will rely on effect annotations for each service and it will perform the dynamic (or static) composition. It will also support dynamic optimizations. This tool will promote software component reuse by decoupling the composition (generalization) from the efficiency (specialization).

Steps

The work consists in:

- studying laziness in functional languages,
- identifying laziness opportunities in established (for instance well known open source) imperative and service oriented applications,
- reviewing proposals (if any) and programming techniques for laziness programming in imperative/distributed context,
- providing support to specify dependencies in an imperative/distributed context,
- providing support to postponed, force and cascade evaluations in an imperative/distributed context,
- applying this support to replace ad-hoc laziness by general laziness in the applications mentioned above,
- providing support to dynamically optimize sequences of delayed computations, and
- introducing dynamic optimizations.

Applicants

Skills

Applicants must be passionate about programming languages, both from a practical and theoretical point of view. They should be able to answer questions about [DT14].

References

[DLL09] Rémi Douence, Xavier Lorca, and Nicolas Loriant. Lazy composition of representations in java. In Alexandre Bergel and Johan Fabry, editors, *Software Composition*, volume 5634 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2009.

[DT14] Rémi Douence and Nicolas Tabareau. Lazier Imperative Programming. In *Principles and Practice of Declarative Programming (PPDP)*, Canterbury, Royaume-Uni, September 2014.

[GLJ93] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

[Hug89] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Mil14] Bartosz Milewski. Getting lazy with c++. *Blog*, April 2014.

[Nai13] Vineet Naik. Python generators and being lazy. *Blog*, March 2013.

[SG96] Mary Shaw and David Garlan. *Software architecture - perspectives on an emerging discipline*. Prentice Hall, 1996.

[SPI14] SPIRiT_1984. Will java 8 have lazy evaluation? *Stackoverflow*, February 2014.