A software system should be declarative except where it interacts with the real world

Peter Van Roy*
Université catholique de Louvain

May 30, 2018

Abstract

We propose a system design principle that explains how to use declarative programming (logic and functional) together with imperative programming. The advantages of declarative programming are well known; they include ease of analysis, verification, testing, optimization, maintenance, upgrading, and distributed implementation. We do not elaborate on the advantages here, but rather focus on what part of the software system should be written declaratively. We observe that declarative programming cannot interact directly with the real world while remaining declarative, since it does not support common real-world concepts such as physical time and named state. It is important to distinguish reasoning about the real world from interacting with the real world: declarative programming can do the first but not the second. Other programming paradigms that support these concepts must be used, such as imperative programming (which contains named state). To optimize the system design, we propose that real-world concepts should only be used where they cannot be avoided, namely where the system interfaces with the real world. It follows that a software system should be built completely declaratively except possibly where it interfaces with the real world. We motivate this principle with examples from our research and we outline a formal argument to justify it.

1 Introduction

The interplay between declarative (e.g., pure functional or logic) programming and imperative programming (which uses named mutable state) has long been a subject of debate in software design. Important questions are which paradigm to use and when to use it; and when and how to use the paradigms together. The book [1] presents these paradigms in a uniform framework, each with its kernel language, and carefully explains what each can and cannot do. This shows that there is no one paradigm that is uniformly better than the others. Large programs will typically use different paradigms in different parts, just as building a house requires multiple skills such as masonry, carpentry, plumbing, and electricity. But determining which paradigm to use where is left unanswered. This position paper gives a design principle that answers this question:

> A software system should be built completely with declarative programming except where it interacts with the real world.

Section 2 gives two examples to motivate this principle. Section 3 defines what we mean by *interaction with the real world* and gives a formal argument to support the principle. Section 4 discusses some ramifications of the principle, and Section 5 presents a brief conclusion.

2 Motivation

2.1 Example 1: client/server

Consider a client/server application in its simplest form: two clients communicating with one server. Each client sends requests to the server and receives replies. To satisfy liveness of each client, the server must accept each incoming request and reply to it within a reasonable delay. The server's handling of a client request should not be impeded because of what the other client does or does not do. The order of the server's handling of requests cannot be determined in advance, because it depends on the precise timing of the requests.

2.2 Example 2: convergent computation (CRDTs and Lasp)

We give a more substantial example taken from our research into synchronization-free programming for distributed systems, namely the Lasp programming system [2]. A Lasp program consists of a dataflow graph connecting data structures with functional and logical operations (similar to SQL operations). The data structures and operations are both designed to do *convergent computation*: each computation step adds information monotonically. In fact, the data structures are CRDTs (Conflict-free Replicated Data Type) [3]. To the programmer, Lasp executes as a functional/logic program with a dataflow semantics, similar to the deterministic dataflow of Chapter 4 in [1].

*This work is partially funded by the LightKone project in the European Union Horizon 2020 Framework Programme under grant agreement 732505.

The data structures are replicated and information is periodically disseminated between replicas. CRDTs are designed so that the replicas are always converging to the correct result. This is extremely resilient. Changing the timing of the dissemination messages has no effect on correctness. Dropped, delayed, or reordered messages have no effect on correctness. The only possible effect is to slow down convergence. Node crashes have two possible results: either the crash has no effect except for slowing down convergence, or some information disappears completely.

Lasp provides a functional semantics with a highly resilient distributed implementation based on weak synchronization. A Lasp program needs stronger synchronization than periodic dissemination only when talking to external clients (i.e., real-world interaction).

3 Formal definition

We have given two examples that show the principle in action. We now outline a formal argument to justify the principle and to define it precisely. For brevity, we use the lambda calculus, but any other declarative execution model that performs deduction from an initial input could be used instead.

3.1 Functional programming

Consider the lambda calculus with a concurrent evaluation strategy. Given an initial lambda expression, we can reduce it to its normal form. Any reduction order will lead to the same normal form, which is known as the Church-Rosser property. The reduction may take many steps. At any step in the reduction, there may be several different positions in the lambda expression that can be reduced next. By adding a scheduler to determine how to choose the position reduced at each step, we can define a concurrent form of functional programming very similar to the deterministic dataflow model given in Chapter 4 of [1]. In this form, we can define deterministic concurrent agents that communicate through streams, similar to Kahn networks [4].

3.2 Real-world interaction requires extending this model with time

Real-world interaction requires that the computation take into account input coming from the real world. If there are several inputs, they need to be handled in some order. This order is not known in advance; it becomes known during the reduction. We are led to a sequence of inputs, arriving one by one during the reduction process.

If all inputs would be known in advance, then they could be considered part of the initial expression, and the system would be purely declarative. However, a key property of the real world is that they are not known in advance. They arrive during the reduction process because reduction steps take nonzero time. The arrival order is determined by the precise timing with respect to the reduction process. The order can affect the result. For example, if the computation builds a list, the order of its elements can depend on the timing. We conclude that the new concept that must be added to deterministic dataflow to allow interaction with the real world is *time*.

It may be that an expression is not reducible until an input arrives, in which case we say the reduction is suspended. When the input arrives, a reduction step becomes possible. When this step is taken, we say the reduction *synchronizes* with the input. On the other hand, the expression may be reducible at several positions, and an arriving input creates another position where a reduction is possible. In this case, the system is active but can accept new input during execution.

4 Discussion

We claim that the design principle holds generally whenever a system interacts with the real world. The real-world property "time" can appear in different guises to the system, e.g., as nondeterminism (see above), as physical wall-clock time (hardware clock), or as partial failure.

Interaction with the real world happens not just at the API, but everywhere that the real world has to be taken into account. For example, MapReduce handles a straggler (slow node) by speculatively running a copy of its task on another node. Detecting the slowness of a straggler depends on time, which is an interaction with the real world.

5 Conclusions

This position paper presents a software design principle that is a result of the author's study of the differences between declarative and imperative programming for system building. We are working on a full paper with a detailed formal justification of the principle and its application to synchronization-free programming.

References

[1] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[2] Christopher Meiklejohn and Peter Van Roy. Lasp: A Language for Distributed, Coordination-Free Programming. 17th International Symposium on Principles and Practice of Declarative Programming (PPDP), Siena, Italy, July 2015.

[3] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *Conflict-free Replicated Data Types.* INRIA Research Report RR-7687, July 2011.

[4] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. IFIP Congress, pp. 471–475, 1974.