Teaching Task Oriented Programming

Pieter Koopman and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
{pieter,rinus}@cs.ru.nl

Abstract. On the Composability, Comprehensibility, Correctness winter school there will be two tutorials about Task Oriented Programming and concrete systems based on this paradigm. The iTask system offers a web-based interface for humans to see their tasks and share their progress with these tasks. The mTask system applies the same concepts to specify the tasks executed by microprocessors. In this contribution, we justify the decisions made at the teacher training in Amsterdam about what and how these topics will be presented at the winter school. Due to the limited time and diverse background of the audience we will focus on the practical usage of the paradigm. There is barely time to address the challenges and beauty of the construction of these systems.

Keywords: Task Oriented Programming · Domain Specific Language · Microprocessor · Code Generation · Simulation · Generic Programming · Monad

1 Introduction

Task Oriented Programming, TOP, is a style of programming centered around the concept of tasks as executed by humans and machines. These tasks are specified by ordinary functions in a functional programming language. In all our examples we will use Clean [6]. The semantics of the tasks is quite different from plain function evaluation. A task is evaluated over and over again until it produces a stable value, or its result is not needed anymore. Intermediate task results can be observed by other tasks. Tasks can be composed by task combinators.

At the Composability, Comprehensibility, Correctness winter school in Kosice, there will be two sessions on TOP. These are entitled *Why "Task Oriented Programming" matters* and *Functional Programming of Devices*. Both sessions will consist of a lecture and practical work of the attendees. In this paper, we motivate the decisions about content and organization of these sessions after the discussions at the teacher training.

2 Audience

The Composability, Comprehensibility, Correctness winter school is for BSc, MSc, PhD students as well as for teachers. Discussions at the teacher training

revealed that the experience in functional programming is diverse, both in the amount of experience as language-wise. The languages used range from pure and lazy languages like Haskell and Clean to Erlang, Scheme and Scala.

This implies that we cannot assume a solid common functional programming experience. Only a part of the audience will be familiar with concepts like strong typing, higher order functions, type constructor classes, Monads and generics. Although these topics are the building blocks of TOP, we cannot assume that they are known by all participants.

3 Task Oriented Programming

Task Oriented Programming is based on a small number of domain specific primitive tasks. These primitive tasks typically interact with the environment, e.g., humans executing part of the tasks, or hardware interacting with the physical world. Task combinators are used to compose task from smaller tasks.

Tasks can communicate via their results as well as via Shared Data Sources, SDSs. Such an SDS contains typed data that can be accessed via primitives like `get` and `set`. These primitives act on the task state to ensure referential transparency.

In order to reuse datatypes and computations of an existing language, a TOP system is often constructed as a Domain Specific Language, DSL, embedded in an existing (functional) programming language.

4 The iTask System

The iTask system was the first implementation of TOP [5]. It is a DSL embedded in the functional programming language Clean. The iTask system facilitates interaction with human worker by the type driven generation of web-pages. These pages are displayed in one of the existing browsers. The page provides information of the current tasks for a user. The user can interact with the iTask system by filling out forms and pushing buttons.

4.1 Used Techniques

The iTask system passes a state around very similar to a state monad. The operators to `return`, bind (>>=), and sequence (>>|) are very similar to the well known monadic versions [4, 7]. This requires higher-order functions and user-defined infix operators. To reuse the operator symbol for different Monads they are defined as type constructor classes.

The task combinators are all higher order functions, typically user-defined infix operators, manipulating task results and the global task state. Specific for TOP is that the tasks produce intermediate results that can be observed while the tasks are repeated over and over again until they produce a stable result or their result is no longer used. This requires higher order functions, laziness and automatic garbage collection.

The iTask system generates a web-server that is used by the human workers to find their task. Like all web-servers, this requires serialization and deserialization of the state to store and retrieve the current state. By filling out web-forms for arbitrary algebraic data types users indicate their progress with the tasks. All of these features are implemented using generic programming.

To implement the tasks monitoring the value of an SDS efficiently there is a hidden publish-subscribe system for each SDS that activates tasks using this SDS when its value is updated.

Apart from these properties, the iTask system uses many additional techniques. For instance the execution of task parts in an interpreter running in the browser to ensure a fast response of the system for highly interactive tasks like a drawing.

4.2 Teaching at the Winter School

Any teaching of programming requires practical programming experience using the educated techniques to master them. This holds also for TOP. As a consequence, we divide the four hours available for *Why "Task Oriented Programming" matters* into two parts of nearly identical size.

In the first part, we will outline the concept of TOP using the iTask system. Given the background of the majority of the students, we have to skip nearly all details about the implementation of the system and we have to focus on the use of the library. This library is actually a shallow embedded DSL for TOP. In the lecture, we use this a set of primitives without spending much time to explain its architecture and implementation.

For the practical work, we will split the existing basic example project into a set of small independent TOP projects. The assignments will consist of small variations of these projects to experience the flavor of task oriented programming.

The more experienced functional programmers can skip most of the basic exercises and jump directly to more advanced assignment. This way, we will be able to adapt to the individual skills of each participant.

5 The mTask System

Microprocessors are computer systems with very limited computing capabilities. They have typically a rather low clock rate and severe memory restrictions, like a few KB of memory to store the data of a running program. These cheap processors are the driving force of many elements in the Internet of Things, IoT. In such microprocessor systems one typically has to monitor several input ports as well as to control some outputs based on these observations. Due to the hardware restrictions, there is typically no operating system offering support.

The TOP paradigm provides lightweight threads that are very well suited to monitor and coordinate the progress of such well defined simple tasks. These tasks can run at their own speed while combinators and shared data sources are

used to coordinate them. Running the iTask system on the IoT devices would enable us to construct programs that are partially executed on a web-server as well as on the IoT devices. The limitations of microprocessors make it impossible to run a full-fledged iTask program on IoT devices.

To approximate the ideal solution we have developed the mTask system [3, 2]. This is a multiview shallowly embedded DSL that can be used as part of the iTask system. It supports the TOP paradigm including the same task results as the iTask system, task combinators and shared data sources. By construction this DSL has no higher order functions and no recursive datatypes. Due to the restrictions imposed in this DSL, mTask programs can be compiled to code that executes on microprocessors. Despite these restrictions, the DSL is very suitable to specify the tasks to be executed on IoT devices easily and very concisely.

5.1 Used Techniques

The mTask system is a multiview shallowly embedded DSL based on type constructor classes [1]. Each instance of these classes defines an interpretation, called a *view*, of a program constructed from these primitives. Typical views implement pretty printing, code generation for microprocessors, and simulation of the mTask programs as an iTask program.

The DSL is extendable by construction in order to reuse existing libraries for peripherals like temperature sensors, displays and servo motors. It is simple to add such a library as a language primitive to the mTask system by introducing a new type constructor class and the required instances.

To make the mTask implementation portable to many different microprocessors and to make the reuse of existing C++ libraries easily, the code generation view produces C++ code for the Arduino platform instead of native machine code for some specific processor. The avr-gcc compiler inside the Arduino platform can translate the generated C++ code and the libraries used to native code for many different microprocessors.

5.2 Teaching at the Winter School

Being able to execute high-level TOP programs on a tiny microprocessor interacting with peripherals is appealing for a tutorial in the winter school. However, executing a mTask program on an actual microprocessor requires much of the students; they must compose an mTask program inside the iTask system, execute the iTask program to obtain C++ code, feed this C++ code to the Arduino IDE, connect the Arduino IDE to the microprocessor and select the right options, upload the compiled program to the microprocessor, and finally run it. All of these steps are quite easy, but the entire process produces only a result when every step is done correctly. Since the generated program will run on a microprocessor without operating system and very limited input/output peripherals debugging such a program is challenging. After ample discussion, this sequence of steps was deemed to be too ambitious for the given time and audience.

Fortunately, the simulator view of the mTask system offers an alternative that is much easier to use. This view transforms the mTask program to an ordinary iTask program. The simulator offers a step-by-step execution of the mTask program. It displays a trace of the last step of the last executed task and the state of all peripherals and shared data sources. The clock, the value of the SDSs, as well as the state of the peripherals can be changed interactively to control the execution and investigate various scenarios. This makes the practical work of this tutorial a direct successor of the practical work of the previous iTask tutorial.

It was decided to schedule these tutorials on one day with the iTask lecture and associated practical work in the morning and the mTask tutorial in the afternoon. This way, the mTask session can directly build on the knowledge and skills gained in the iTask session. The understanding of TOP that stated in the morning will be deepened in the afternoon.

6 Conclusion

For both tutorials on TOP there are many more interesting topics than can be covered in the given time for the audience of the winter school. In the sessions, we will focus on understanding and using the TOP paradigm by relative simple examples. Slightly advanced examples will be used to illustrate the capabilities of this approach. In the practical work, we will focus on illustrative exercises that are mostly variants of examples used in the tutorial. For the advanced participants there will be some challenging assignments as well as the opportunity to discuss aspects of the systems in depth.

Acknowledgement

This paper is part of the Intellectual Output O2 of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No. 2017-1-SK01-KA203-035402: "Focusing Education on Composability, Comprehensibility and Correctness of Working Software". The information and views set out in this paper are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

References

1. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19(5), 509–543 (Sep 2009), http://dx.doi.org/10.1017/S0956796809007205
2. Koopman, P., Lubbers, M., Plasmeijer, R.: A task-based dsl for microcomputers. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. pp. 4:1–4:11. RWDSL2018, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3183895.3183902

3. Koopman, P., Plasmeijer, R.: A shallow embedded type safe extendable DSL for the Arduino. In: Revised Selected Papers of the 16th International Symposium on Trends in Functional Programming - Volume 9547. pp. 104–123. TFP 2015, Springer-Verlag New York, Inc., New York, NY, USA (2016), http://dx.doi.org/10.1007/978-3-319-39110-6_6
4. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 71–84. POPL '93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/158511.158524
5. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the ICFP'07. pp. 141–152. ACM, Freiburg, Germany (2007)
6. Plasmeijer, R., van Eekelen, M., van Groningen, J.: Clean language report (version 2.2) (2011), http://clean.cs.ru.nl/Documentation
7. Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. pp. 61–78. LFP '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/91556.91592