iTask as a new paradigm to building GUI applications – extended abstract –

Steffen Michels, Rinus Plasmeijer, and Peter Achten

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
`S.Michels@student.ru.nl, {rinus, p.achten}@cs.ru.nl`

1 Introduction

The *iTask* system [21, 20] is a *workflow management system* (WFMS) which uses a workflow description language (WDL) embedded in the functional language *Clean* as a combinator library. There are many advantages to this approach. First, the embedding in a *general purpose programming language* adds recursion and arbitrary computations to the WDL. Second, the use of a *functional* language adds higher-order tasks: workflows that can accept and / or compute workflows during execution. Third, the use of *Clean* adds generic programming facilities [4] which is used intensively to abstract away from a lot of boilerplate programming [17]. As a result, the *iTask* WDL is a rich and high level formalism to create workflow applications.

An additional advantage of the *iTask* system is that it is currently implemented using web browser technology. This makes it readily available to any platform that provides a standard web browser. The programmer who uses the *iTask* WDL to create multi-user web-enabled workflows does not have to concern herself about the technological challenges that come with programming web applications: the generic foundations of *iTask* abstract from these concerns.

The key contribution of this paper is that we generalize the application domain of the *iTask* WDL, and show how to adapt the *iTask* combinator library in such a way that it is a new paradigm for programming distributed GUI applications. There are several reasons to substantiate this claim. First, traditional widget-callback GUI paradigms force the programmer to break the program logic in terms of callback functions. In such approaches, the programmer also has to carefully arrange the handling and identification of state to handle the proper 'communication' between subsequent callbacks. In *iTask*, the program is a single, typically recursive, expression that expresses the behavior of the program in terms of user input. Second, in traditional widget-based GUI paradigms, the programmer is responsible for the entire life-cycle of the GUI elements: creation, management, event handling, and destruction all need to be programmed explicitly and carefully. In *iTask*, most of this is handled by means of the generic interface, and the programmer is only bothered with pure functional data types that model the GUI, rather than implement it. Third, past experience with using *iTask* for a non-workflow interactive application [13] turned out to be successful: it was both possible to add a rich new GUI component to the framework and the program structure was a fairly straightforward recursive function.

However, the *iTask* WDL has not been designed and implemented as a general GUI programming language. In this paper we show what is needed to extend it to make it suited for this purpose. The contributions of this paper are:

– We extend the *iTask* system and WDL with a number of missing features to support office-like GUI applications. This concerns infrastructure for menus, model-view-controller abstraction, support for multiple windows and dialogs, rich GUI elements and MDI infrastructure.
– The extensions are orthogonal to the basic *iTask* paradigm. 'Standard' workflow applications are not modified due to the extensions. Vice versa, interactive GUI applications can use the workflow facilities.
– We show how to create a multi-document editing application and also create a first prototype of an entire IDE for the Clean programming language. An additional advantage of using *iTask* is that this IDE is platform independent.

This high level of abstraction comes with the cost that the programmer cannot arbitrarily influence the layout of the user interface. There will also be no general canvas or the possibility to realise highly interactive applications like games. The paradigm is restricted to applications based on entering data using standard controls, like a multi-document text editor or an integrated development environment.

First we shortly summarise our extensions to the *iTask* system and our first experiences with realising complex applications in Section 2. In Section 3 we compare our approach to existing solutions. We draw conclusions in Section 4.

2 Results

One important extension added to the language are menus. They are defined in a similar way as in Clean Event I/O [3]. They are neatly integrated into the existing language since they generate actions the same way buttons do. The structure is defined once for a process, the actual menu is dynamically generated based on the context. Also whether items are enabled can declaratively be defined using predicates.

A very powerful extension for solving the classical *model-view problem* [15] is the concept of *views* on *shared data*. Here *shared variables* which behave like global data accessible by a reference are used. A view on such a variable is actually a realisation of lenses [5]. The programmer only has to define one function for converting the value from the model to the editor domain and another function for changing the model value based on a changed editor value. Actually this concept is the only way parallel tasks can interact, which forces the programmer to realise applications by defining an abstract state which is modified by tasks. This abstract state only models data but **not** the application's control flow. The tasks working on it therefore have a high level of independence.

Multiple windows and dialogs can be expressed by *grouped tasks* running in parallel. One can influence the behaviour (fixed, floating window or modal dialog) of each task. Also new tasks can dynamically be added without interfering with existing ones based on the result of other tasks or by menu commands.

Those concepts are powerful enough to make it possible to define a high-level combinator for realising multiple-document interface applications. Handling the entire application state and providing an encapsulated state for each document is done automatically. It can be used for implementing for instance a multi-file text editor. Also an application like an IDE can be realised, by using specialised types for providing source code input with syntax highlighting and using special tasks for accessing operating system functionality on the server, like writing to the file system and calling processes.

3 Related Work

In contrast to existing functional libraries for generating user interfaces the *iTask* paradigm does not deal with composing widgets to build a user interface directly, but works on a higher level.

Object I/O [1, 2] and *wxHaskell* [16] are examples with a more imperative taste in the sense that the user explicitly has to create user interface elements and update them if the state of the application changes. *Object I/O* has a global application state and also local states for realising encapsulation. For communication between processes sophisticated message passing mechanisms are used. In *wxHaskell* there is the concept of *mutable values* similar to *iTask*'s shared variables. One can wait for such a variable to change and can update widgets accordingly. In *iTasks* sharing data among a subset of tasks inside a process, the entire process or between multiple processes can be done using the same concept of shared data and views.

Haggis [11] also lets the user create widgets explicitly but gives the user a more compositional view on the user interface. Each component is treated as virtual I/O device. Components are repeatedly combined together to build up the entire application. Also a separation between the user interface and the application, which means between the representation and the actual value or interaction with the user, is made. This ensures a higher level of abstraction for the implementation of the program logic. No callback functions are used to handle events but one can wait for a message to be generated by a component. One can for example wait for a button to be pressed or a variable to change. Concurrency is used to make it possible to compose parts of the user interface waiting for messages at the same time.

A more functional approach of defining user interfaces is *Fudgets* [6, 7]. Here *fudgets*, which are *stream processors* and pass messages, are hierarchically combined to build up the application. There are no mutable variables. Sharing data has to be realised by routing messages between components.

Fruit [9, 10] emphasises being built on a formal model even more. The main building blocks here are *signals* which are continuous time-varying values and *signals transformers* using pure functions for mapping signals to other signals. A GUI application is modelled as a signal transformer from a signal including all user inputs to a signal representing a picture. The cost of the formal model used by *Fruit* is that it is very cumbersome to define I/O other than turning the user input into a picture. All I/O operations explicitly have to be added to the input and output signal.

An example where generic programming techniques are used for generating forms is the *iData toolkit* [19]. It supports the creation of interactive web applications consisting of interconnected forms. A mechanism for providing views similar to the approach discussed in this paper is used. It has been shown that a complex application like a *Conference Management System* which also uses destructively updated shared data can be realised using this approach [18]. However the *iData toolkit* is not based on a workflow semantics. In contrast to *iTasks* there the system automatically keeps track of the control flow, with *iData* the programmer has to keep track of the application state.

Two more recent approaches that are also based on functional languages are *Links* [8] and *Hop* [22]. Both languages aim to deal with web programming within a single framework, just as the iData and iTask approach do. iTask has similar capabilities as Links and Hop in terms of client-side processing and thread-creation due to the use of client-side interpreter technology [12]. Furthermore, iTask provides a higher degree of automation due to the intensive use of generic programming techniques.

4 Conclusions

The *iTask* system has been extended to deal with essential features needed for implementing modern user interfaces. It is possible to structure commands using menus, to organise work in different windows and to modify a shared data model by different parallel tasks. Also special types can be added to allow for abstract representations of more complex user interface components. Further it is possible to build complex applications using the extended workflow language, as long as they are mainly based on standard controls filled in by the user and are not too interactive. The extended system is still based on the semantics of workflows and stays abstract and declarative.

The proposed paradigm for implementing graphical user interfaces has a higher level of abstraction than existing solutions with the price that it gives the programmer less control over how the user interface looks like. We think that the paradigm is highly suited for using rapid prototyping techniques and has a low learning curve, because it is based only on few core concepts. Still it is very powerful because all the power of functional programming can be used. Being embedded in a workflow system provides features like dealing with multiple users for free. Finally the program logic is based on a language for which abstract formal semantics can be defined [14], which helps reasoning about applications.

References

1. Peter Achten and Marinus J. Plasmeijer. Interactive functional objects in Clean. In *Implementation of Functional Languages*, pages 304–321, 1997.
2. Peter Achten and Martin Wierich. A tutorial to the Clean Object I/O Library - version 1.2. ftp://ftp.cs.ru.nl/pub/Clean/supported/ObjectIO.1.2/doc/tutorial.pdf, February 2000.
3. P.M. Achten and M.J. Plasmeijer. The Ins and Outs of Concurrent CLEAN I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.
4. Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications.* PhD thesis, Radboud University Nijmegen, 2005.
5. Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM.
6. M. Carlsson and T. Hallgren. FUDGETS - A Graphical User Interface in a Lazy Functional Language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.
7. Magnus Carlsson and Thomas Hallgren. *Fudgets — Purely Functional Processes with applications to Graphical User Interfaces.* PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Gteborg, Sweden, March 1998.
8. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Proceedings of the 5th'06*, volume 4709, CWI, Amsterdam, The Netherlands, 7-10, November 2006. Springer-Verlag.
9. Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.
10. Antony Alexander Courtney. *Modeling user interfaces in a functional language.* PhD thesis, Yale University, New Haven, CT, USA, 2004. Director-Hudak, Paul.
11. Sigbjörn Finne and Simon Peyton Jones. Composing the user interface with Haggis. In *Advanced Functional Programming: Second Interational School, LNCS #1129*, pages 26–30. Springer-Verlag, 1996.
12. Jan Martin Jansen. *Functional Web Applications – Implementation and Use of Client Side Interpreters.* PhD thesis, 8, July 2010. ISBN 978-90-9025436-4.
13. Pieter Koopman, Peter Achten, and Rinus Plasmeijer. Validating specifications for model-based testing. In Hamid Arabnia and Hassan Reza, editors, *Proceedings of the'08*, pages 231–237, Las Vegas, NV, USA, 14-17, July 2008. CSREA Press.
14. Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In *Scholz, S.-B. (ed.), IFL'08 : Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages*, pages 53–64. Hertfordshire, UK : University of Hertfordshire, September 2008.
15. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August-September 1988.
16. Daan Leijen. wxHaskell: a portable and concise gui library for Haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, New York, NY, USA, 2004. ACM.
17. Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for end-users. In *Proceedings 21st Symposium on Implementation and Application of Functional Languages*, September 2009.

18. Rinus Plasmeijer and Peter Achten. A conference management system based on the iData toolkit. In *Implementation and application of functional languages : 18th international symposium, IFL 2006, Budapest, Hungary, September 4-6, 2006 ; revised selected papers*, pages 108–125. Springer Verlag, 2006.
19. Rinus Plasmeijer and Peter Achten. iData for the world wide web - programming interconnected web forms. In *In Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006), volume 3945 of LNCS*, pages 24–26. Springer Verlag, 2006.
20. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An Introduction to iTasks: Defining Interactive Work Flows for the Web. In *Central European Functional Programming School, Revised Selected Lectures, CEFP 2007*, volume 5161 of *LNCS*, pages 1–40, Cluj-Napoca, Romania, June 23-30 2007. Springer.
21. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th'07*, pages 141–152, Freiburg, Germany, 1-3, October 2007. ACM Press.
22. Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Proceedings of the 11th'06*, pages 975–985, Portland, Oregon, USA, 22-26, October 2006.