Masami Hagiya
Philip Wadler (Eds.)

LNCS 3945

Functional and Logic Programming

8th International Symposium, FLOPS 2006
Fuji-Susono, Japan, April 2006
Proceedings

Springer

TP311.1-53
F979
2006

Masami Hagiya Philip Wadler (Eds.)

Functional and Logic Programming

8th International Symposium, FLOPS 2006
Fuji-Susono, Japan, April 24-26, 2006
Proceedings

Springer

Volume Editors

Masami Hagiya
University of Tokyo
and NTT Communication Science Laboratories
Department of Computer Science
Graduate School of Information Science and Technology
E-mail: hagiya@is.s.u-tokyo.ac.jp

Philip Wadler
University of Edinburgh
Department of Informatics
James Clerk Maxwell Building, The King's Buildings
Mayfield Road, Edinburgh EH9 3JZ, UK
E-mail: wadler@inf.ed.ac.uk

Library of Congress Control Number: 2006923563

CR Subject Classification (1998): D.1.6, D.1, D.3, F.3, I.2.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-33438-6 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-33438-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper   SPIN: 11737414   06/3142   5 4 3 2 1 0

¥410.00元

Preface

This volume contains the proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006), held in Fuji-Susono, Japan, April 24–26, 2006 at the Fuji Institute of Education and Training.

FLOPS is a forum for research on all issues concerning functional programming and logic programming. In particular it aims to stimulate the cross-fertilization as well as the integration of the two paradigms. The previous FLOPS meetings took place in Fuji-Susono (1995), Shonan (1996), Kyoto (1998), Tsukuba (1999), Tokyo (2001), Aizu (2002) and Nara (2004). The proceedings of FLOPS 1999, FLOPS 2001, FLOPS 2002 and FLOPS 2004 were published by Springer in the *Lecture Notes in Computer Science* series, as volumes 1722, 2024, 2441 and 2998, respectively.

In response to the call for papers, 51 papers were submitted. Each paper was reviewed by at least three Program Committee members with the help of expert external reviewers. The Program Committee meeting was conducted electronically for a period of 2 weeks in December 2005 and January 2006. After careful and thorough discussion, the Program Committee selected 17 papers (33%) for presentation at the conference. In addition to the 17 contributed papers, the symposium included talks by two invited speakers: Guy Steele (Sun Microsystems Laboratories) and Peter Van Roy (Université Catholique de Louvain).

On behalf of the Program Committee, we would like to thank the invited speakers who agreed to give talks and contribute papers, and all those who submitted papers to FLOPS 2006. As Program Chairs, we would like to sincerely thank all the members of the FLOPS 2006 Program Committee for their excellent job, and all the external reviewers for their invaluable contribution. The support of our sponsors is gratefully acknowledged. We are indebted to the Japan Society for Software Science and Technology (JSSST), the Association of Logic Programming (ALP), and the Asian Association for Foundation of Software (AAFS). Finally we would like to thank members of the Local Arrangements Committee, in particular Yoshihiko Kakutani, for their invaluable support throughout the preparation and organization of the symposium.

February 2006

Masami Hagiya
Philip Wadler
Program Co-chairs
FLOPS 2006

Symposium Organization

Program Chairs

Philip Wadler	Edinburgh, UK
Masami Hagiya	Tokyo, Japan

Program Committee

Vincent Danos	Paris, France
Jacques Garrigue	Nagoya, Japan
Manuel Hermenegildo	New Mexico, USA & Madrid, Spain
Gabrielle Keller	UNSW, Sydney, Australia
Michael Rusinowitch	INRIA Lorraine, France
Konstantinos Sagonas	Uppsala, Sweden
Ken Satoh	NII, Tokyo, Japan
Peter Selinger	Dalhousie, Canada
Eijiro Sumii	Tohoku, Japan
Naoyuki Tamura	Kobe, Japan
Peter Thiemann	Freiburg, Germany
David Warren	Stony Brook, USA

Local Arrangements Chair

Masami Hagiya	Tokyo, Japan

Referees

Tatsuya Abe
Amal Ahmed
Kenichi Asai
Demis Ballis
Maria Garcia de la Banda
Bruno Blanchet
Daniel Cabeza
Venanzio Capretta
Olga Caprotti
Francois Charoy
Ezra Cooper
Markus Degen
Rachid Echahed
Carl Christian Frederiksen
Naoki Fukuta
Martin Gasbichler
Samir Genaim
Michael Hanus
Ralf Hinze
Hiroshi Hosobe
Haruo Hosoya
Zhenjiang Hu
Atsushi Igarashi
Koji Kagawa
Yoshihiko Kakutani
Dominique Larchey
Pedro López
Ugo Dal Lago
Toshiyuki Maeda
Julio Mariño
Yasuhiko Minamide
Jean-Yves Moyen
Susana Muñoz
Keiko Nakata
Jorge Navas
Matthias Neubauer
Tobias Nipkow
Susumu Nishimura
Shin-ya Nishizaki
Martin Odersky
Yoshihiro Oyama
Mikael Pettersson
David Pichardie
Benjamin Pierce
Paola Quaglia
Christophe Ringeissen
Don Sannella
Ganesh Sittampalam
Yasuyuki Tahara
Yoshiaki Takata
Yasuyuki Tsukada
Satoshi Tojo
Akihiko Tozawa
Rafael del Vado
German Vidal
Dimitrios Vytiniotis
Hironori Washizaki
Stefan Wehr
Stephanie Weirich
Jeremy Yallop
Akihiro Yamamoto
Mitsuharu Yamamoto

Table of Contents

Invited Papers

Parallel Programming and Parallel Abstractions in Fortress
Guy L. Steele 1

Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place
Peter Van Roy 2

Refereed Papers

Data Types

"Scrap Your Boilerplate" Reloaded
Ralf Hinze, Andres Löh, Bruno C.d.S. Oliveira 13

Ruler: Programming Type Rules
Atze Dijkstra, S. Doaitse Swierstra 30

A Framework for Extended Algebraic Data Types
Martin Sulzmann, Jeremy Wazny, Peter J. Stuckey 47

FP Extensions

Lock Free Data Structures Using STM in Haskell
Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, Satnam Singh 65

XML Type Checking Using High-Level Tree Transducer
Akihiko Tozawa 81

Type Theory

A Computational Approach to Pocklington Certificates in Type Theory
Benjamin Grégoire, Laurent Théry, Benjamin Werner 97

Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant
Gilles Barthe, Julien Forest, David Pichardie, Vlad Rusu 114

LP Extensions

Soutei, a Logic-Based Trust-Management System
Andrew Pimlott, Oleg Kiselyov 130

A Syntactic Approach to Combining Functional Notation, Lazy Evaluation, and Higher-Order in LP Systems
Amadeo Casas, Daniel Cabeza, Manuel V. Hermenegildo 146

Analysis

Resource Analysis by Sup-interpretation
Jean-Yves Marion, Romain Péchoux 163

Lazy Set-Sharing Analysis
Xuan Li, Andy King, Lunjin Lu 177

Size-Change Termination and Bound Analysis
James Avery 192

Contracts

Typed Contracts for Functional Programming
Ralf Hinze, Johan Jeuring, Andres Löh 208

Contracts as Pairs of Projections
Robert Bruce Findler, Matthias Blume 226

Web and GUI

iData for the World Wide Web – Programming Interconnected Web Forms
Rinus Plasmeijer, Peter Achten 242

Crossing State Lines: Adapting Object-Oriented Frameworks to Functional Reactive Languages
Daniel Ignatoff, Gregory H. Cooper, Shriram Krishnamurthi 259

WASH Server Pages
Peter Thiemann 277

Author Index 295

Parallel Programming and Parallel Abstractions in Fortress

Guy L. Steele

Sun Microsystems Laboratories

Abstract. The Programming Language Research Group at Sun Microsystems Laboratories seeks to apply lessons learned from the Java (TM) Programming Language to the next generation of programming languages. The Java language supports platform-independent parallel programming with explicit multithreading and explicit locks. As part of the DARPA program for High Productivity Computing Systems, we are developing Fortress, a language intended to support large-scale scientific computation. One of the design principles is that parallelism be encouraged everywhere (for example, it is intentionally just a little bit harder to write a sequential loop than a parallel loop). Another is to have rich mechanisms for encapsulation and abstraction; the idea is to have a fairly complicated language for library writers that enables them to write libraries that present a relatively simple set of interfaces to the application programmer. We will discuss ideas for using a rich polymorphic type system to organize multithreading and data distribution on large parallel machines. The net result is similar in some ways to data distribution facilities in other languages such as HPF and Chapel, but more open-ended, because in Fortress the facilities are defined by user-replaceable libraries rather than wired into the compiler.

Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place

Peter Van Roy

Université catholique de Louvain,
B-1348 Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be
http://www.info.ucl.ac.be/people/cvvanroy.html

Abstract. What will a definitive programming language look like? By *definitive language* I mean a programming language that gives good solutions at its level of abstraction, allowing computer science researchers to move on and work at higher levels. Given the evolution of computer science as a field with a rising level of abstraction, it is my belief that a small set of definitive languages will eventually exist. But how can we learn something about this set, considering that many basic questions about languages have not yet been settled? In this paper, I give some tentative conclusions about one definitive language. I present four case studies of substantial research projects that tackle important problems in four quite different areas: fault-tolerant programming, secure distributed programming, network-transparent distributed programming, and teaching programming as a unified discipline. All four projects had to think about language design. In this paper, I summarize the reasons why each project designed the language it did. It turns out that all four languages have a common structure. They can be seen as layered, with the following four layers in this order: a strict functional core, then deterministic concurrency, then message-passing concurrency, and finally shared-state concurrency (usually with transactions). This confirms the importance of functional programming and message passing as important defaults; however, global mutable state is also seen as an essential ingredient.

1 Introduction

This paper presents a surprising example of convergence in language design.[1] I will present four different research projects that were undertaken to solve four very different problems. The solutions achieved by all four projects are significant contributions to each of their respective areas. The four projects are interesting to us because they all considered language design as a key factor to achieve success. The surprise is that the four projects ended up using languages that have very similar structures.

[1] This paper was written to accompany an invited talk at FLOPS 2006 and is intended to stimulate discussion.

This paper is structured as follows. Section 1.1 briefly presents each of the four projects and Section 1.2 sketches their common solution. Then Sections 2 to 5 present each of the four projects in more detail to motivate why the common solution is a good solution for it. Finally, Section 6 concludes the paper by recapitulating the common solution and making some conclusions on why it is important for functional and logic programming.

Given the similar structure of the four languages, I consider that their common structure deserves to be carefully examined. The common structure may turn out to be the heart of one possible *definitive* programming language, i.e., a programming language that gives good solutions at its level of abstraction, so that computer science researchers can move on and work at higher levels. My view is that the evolution of programming languages will follow a similar course as the evolution of parsing algorithms. In the 1970s, compiler courses were often built around a study of parsing algorithms. Today, parsing is well understood for most practical purposes and when designing a new compiler it is straightforward to pick a parsing algorithm from a set of "good enough" or "definitive" algorithms. Today's compiler courses are built around higher level topics such as dataflow analysis, type systems, and language design. For programming languages the evolution toward a definitive set may be slower than for parsing algorithms because languages are harder to judge objectively than algorithms.

1.1 The Four Projects

The four projects are the following:[2]

- Programming highly available embedded systems for telecommunications (Section 2). This project was undertaken by Joe Armstrong and his colleagues at the Ericsson Computer Science Laboratory. This work started in 1986. The Erlang language was designed and a first efficient and stable implementation was completed in 1991. Erlang and its current environment, the OTP (Open Telecom Platform) system, are being used successfully in commercial systems by Ericsson and other companies.
- Programming secure distributed systems with multiple users and multiple security domains (Section 3). This project was undertaken over many years by different institutions. It started with Carl Hewitt's Actor model and led via concurrent logic programming to the E language designed by Doug Barnes, Mark Miller, and their colleagues. Predecessors of E have been used to implement various multiuser virtual environments.
- Making network-transparent distributed programming practical (Section 4). This project started in 1995 with the realization that the well-factored design of the Oz language, first developed by Gert Smolka and his students in 1991 as an outgrowth of the ACCLAIM project, was a good starting point for making network transparent distribution practical. This resulted in the Mozart Programming System, whose first release was in 1999.

[2] Many people were involved in each project; because of space limitations only a few are mentioned here.

- Teaching programming as a unified discipline covering all popular programming paradigms (Section 5). This project started in 1999 with the realization by the author and Seif Haridi that Oz is well-suited to teaching programming because it covers many programming concepts, it has a simple semantics, and it has an efficient implementation. A textbook published in 2004 "reconstructs" the Oz design according to a principled approach. This book is the basis of programming courses now being taught at more than a dozen universities worldwide.

1.2 The Layered Language Structure

In all four research projects, the programming language has a layered structure. In its most general form, the language has four layers. This section briefly presents the four layers and mentions how they are realized in the four projects. The rest of the paper motivates the layered structure for each project in more detail. The layers are the following:

- The inner layer is a strict functional language. All four projects start with this layer.
- The second layer adds deterministic concurrency. Deterministic concurrency is sometimes called declarative or dataflow concurrency. It has the property that it cannot have race conditions. This form of concurrency is as simple to reason in as functional programming. In Oz it is realized with single-assignment variables and dataflow synchronization. Because Oz implements these variables as logic variables, this layer in Oz is also a logic language. In E it is realized by a form of concurrent programming called *event-loop concurrency*: inside a process all objects share a single thread. This means that execution inside a process is deterministic. The Erlang project skips this layer.
- The third layer adds asynchronous message passing. This leads to a simple message-passing model in which concurrent entities send messages asynchronously. All four projects have this layer. In E, this layer is used for communication between processes (deterministic concurrency is used for communication inside a single process).
- The fourth layer adds global mutable state.[3] Three of the four projects have global mutable state as a final layer, provided for different reasons, but always with the understanding that it is not used as often as the other layers. In the Erlang project, the mutable state is provided as a persistent database with a transactional interface. In the network transparency project, the mutable state is provided as an object store with a transactional interface and as a family of distributed protocols that is used to guarantee coherence of state across the distributed system. These protocols are expensive but they are sometimes necessary. In the teaching programming project, mutable state is used to make programs modular. The E project skips this layer.

[3] By *global*, I mean that the mutable state has a scope that is as large as necessary, not that it necessarily covers the whole program.

This layered structure has an influence on program design. In all four projects, the starting point is the functional inner layer, complemented by the message-passing layer which is just as important. In three of the four projects, the final layer (global mutable state) is less used than the others, but it provides a critical functionality that cannot be eliminated.

Note that the network-transparent distribution project and the teaching programming project were undertaken by many of the same people and started with the same programming language. Both projects were undertaken because we had reasons to believe Oz would be an adequate starting point. Each project had to adapt the Oz language to get a good solution. In the final analysis, both projects give good reasons why their solutions are appropriate, as explained in Sections 4 and 5.

2 Fault-Tolerant Programming

The Erlang programming language and system is designed for building high availability telecommunications systems. Erlang was designed at the Ericsson Computer Science Laboratory [5, 4]. Erlang is designed explicitly to support programs that tolerate both software and hardware faults. Note that software faults are unavoidable: studies have shown that even with extensive testing, software still has bugs. Any system with high availability must therefore have a way to tolerate faults due to software bugs. Erlang has been used to build commercial systems of very high availability [8]. The most successful of these systems is the AXD 301 ATM switch, which contains around 1 million lines of Erlang, a similar amount of C/C++ code, and a small amount of Java [29].

An Erlang program consists of a (possibly very large) number of processes. An Erlang process is a lightweight entity with its own memory space. A process is programmed with a strict functional language. Each process has a unique identity, which is a constant that can be stored in data structures and in messages. Processes communicate by sending asynchronous messages to other processes. A process receives messages in its mailbox, and it can extract messages from the mailbox with pattern matching. Note that a process can do dynamic code change by receiving a new function in a message and installing it as the new process definition. We conclude that this structure gives the Erlang language two layers: a functional layer for programming processes, and a message-passing layer for allowing them to communicate.

To support fault tolerance, two processes can be linked together. When one process fails, for example because of a software error, then the other fails as well. Each process has a supervisor bit. If a process is set to supervisor mode, then it does not fail when a linked process fails, but it receives a message generated by the run-time system. This allows the application to recover from the failure. Erlang is well-suited to implement software fault tolerance because of process isolation and process linking.

Erlang also has a database called Mnesia. The database stores consistent snapshots of critical program data. When processes fail, their supervisors can use the database to recover and continue execution. The database provides a transactional interface to shared data. The database is an essential part of Erlang programs. It can therefore be considered as a third layer of the Erlang language. This third layer, mutable state with a transactional interface, implements a form of shared-state concurrency [26].

Because Erlang processes do not share data, they can be implemented over a distributed system without any changes in the program. This makes distributed programming in Erlang straightforward. Using process linking and supervisors, Erlang programs can also recover from hardware failures, i.e., partial failures of the distributed system.

3 Secure Distributed Programming

The E programming language and system is designed for building secure distributed systems [21, 19]. The E language consists of objects (functions that share an encapsulated state) hosted in secure processes called *vats* that communicate through a secure message-passing protocol based on encryption. Within the language, security is provided by implementing all language references (including object references) as capabilities. A *capability* is an unforgeable reference that combines two properties that cannot be separated: it designates a language entity and it provides permission to perform a well-defined set of operations on the entity. The only way to perform an operation is to have a capability for that operation.

Capabilities are passed between language entities according to well-defined rules. The primary rule is that the only way to get a capability is that an entity to which you already have a capability passes you the capability ("connectivity begets connectivity"). A system based on capabilities can support the Principle of Least Authority (POLA): give each entity just enough authority to carry out its work. In systems based on POLA the destructive abilities of malicious programs such as viruses largely go away. Unfortunately, current programming languages and operating systems only have weak support for POLA. This is why projects such as E and KeyKOS (see below) are so important [25].

Inside a vat, there is a single thread of execution and all objects take turns executing in this thread. Objects send other objects asynchronous messages that are queued for execution. Objects execute a method when they receive a message. This is a form of deterministic concurrency that is called *event-loop concurrency*. Single threading within a vat is done to ensure that concurrency introduces no security problems due to the nondeterminism of interleaving execution. Event-loop concurrency works well for secure programs; a model based on shared-state concurrency is much harder to program with [20]. Between two or more vats, execution is done according to a general message-passing model.

In a system such as E that is based on capabilities, there is no *ambient authority*, i.e., a program does not have the ability to perform an operation just because it is executing in a certain context. This is very different from most other systems. For example, in Unix a program has all the authority of the user that executes it. The lack of ambient authority does not mean that E necessarily does not have global mutable state. For example, there could be a capability