

Two-Stage Programming: Compilation and Case Studies

Inauguraldissertation

zur

Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

Romeo A. Dumitrescu

aus Drobeta Turnu-Severin, Rumänien

Basel, 2000

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät auf
Antrag von

Prof. Dr. Helmar Burkhart, Universität Basel, Fakultätsverantwortlicher

Dr. Edgar F. A. Lederer, Universität Basel, Dissertationsleiter

Prof. Dr. Robert Stärk, ETH Zürich, Korreferent

Basel, den 04.07.2000

Prof. Dr. A. Zuberbühler, Dekan

Research Reports in Computer Science

Band 6

Romeo A. Dumitrescu

**Two-Stage Programming:
Compilation and Case Studies**

Shaker Verlag
Aachen 2000

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Dumitrescu, Romeo A.:

Two-Stage Programming: Compilation and Case Studies/

Romeo A. Dumitrescu. Aachen : Shaker, 2000

(Research Reports in Computer Science ; Bd. 6)

Zugl.: Basel, Univ., Diss., 2000

ISBN3-8265-7793-0

Copyright Shaker Verlag 2000

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the publishers.

Printed in Germany.

ISBN 3-8265-7793-0

ISSN 1436-6967

Shaker Verlag GmbH • P.O. BOX 1290 • D-52013 Aachen

Phone: 0049/2407/9596-0 • Telefax: 0049/2407/9596-9

Internet: www.shaker.de • eMail: info@shaker.de

To my wife, my parents, and my grandmother “Maia”

Acknowledgements

I would like to mention here some people who helped me complete this thesis. I apologize to those I have unintentionally omitted.

I thank Prof. Helmar Burkhart for accepting me as a Ph.D. student, giving me the opportunity to work on a topic of my choice, and helping me in critical moments. I also thank Prof. Robert Stärk for accepting to review my thesis.

Special thanks go to Dr. Edgar Lederer, my direct project leader, for being an excellent supervisor who carefully watched my work and invested a lot of time and patience in fruitful discussions which helped me find and chose the best solutions. He was also a very good teammate and teacher from whom I have learned a lot.

I could not have accomplished anything without the love, support, and encouragement of my wife Lavinia. She lovingly accepted my long working days and weekends and gave me real power in the much too little time we spent together. All the time, I have also felt the love of my parents and my grandmother although I was quite far from them.

I am greatly indebted to Lucas Godelmann and Penelope Vounatsou for their sincere friendship, wholeheartedly help, and strong encouragement.

I would like to express my gratitude to Robert Sinclair for his useful hints and advice. He also helped me to set up my new home and get used with a new environment.

Etienne Studer kindly took over a significant part of my assistant duties during the last months when I was writing my thesis.

My colleagues Peter Christen, Niandong Fang, Robert Frank, Guido Häechler, Beat Hörman, Walter Kuhn, Gérard Prétôt, Radu Prodan, Kay Ruchty-Crowley, Christoff Sinn, Gonzalo Travieso, Guoqiang Wei, and Birgit Westermann have created a pleasant work environment. The secretaries Regula Kneubühler-Fricker and Karin Liesenfeld helped me in administrative matters.

Finally, I take this opportunity to thank my high-school teacher Ștefan Bojincă to whom I owe my carrier path. He showed me a computer for the first time and guided my first steps towards studying computer science and research work. I also thank all my professors and assistants from the “Politehnica” University of Timișoara who contributed to my education in computer science, especially Prof. Ioan Jurca who, in addition, offered me the possibility to be his assistant and encouraged me together with Prof. Ștefan Holban and Prof. Vladimir Crețu to pursue Ph.D. studies.

Abstract

Two-Stage Programming (2SP) is a novel, mixed-paradigm approach (functional/imperative) to developing reliable programs based on complete run-time checking of computations with respect to a given specification. A 2SP program consists of a functional specification and an imperative coordination tightly connected to the specification. The coordination maps the specification to an imperative and possibly parallel/distributed program. During run-time, the consistency between the coordination and specification is checked based on their connection. Normal termination of a 2SP program execution implies the correctness of the computed results with respect to the specification, for that execution.

I address in this thesis the impact of 2SP's run-time consistency checks on both program reliability and efficiency.

I describe the basic ideas of the first efficient implementation of 2SP I have developed, and present, based on this implementation, an analysis of two significant case studies (one sequential, the other parallel), which shows that 2SP offers automatic run-time result checking and enhanced debugging support through *early* detection and *precise* location of errors at run-time, for an increase of about one order of magnitude of the program execution time.

I consider that this initial study shows that run-time consistency checks are reasonably efficient, and programming languages can benefit by including a 2SP-like consistency checker mechanism. This is especially useful for ML, well-known as a very robust language, since I show that a 2SP program can be both more reliable and faster than its corresponding ML program.

Keywords Run-time checking, result checking, functional/imperative programming, debugging, coordination language.

Parts of Chapters 2, 3, and 4 are accepted for publication in the *International Journal of Foundations of Computer Science*.

Part of Chapter 3 has been published in the *Proceedings of the 14th ACM Symposium on Applied Computing, 1999*.

This research was supported by the Swiss National Science Foundation by the grants 2100-046932.96/1 and 2000-053690.98/1, and the University of Basel.

Contents

1	Introduction	1
1.1	Research Focus	1
1.2	Contribution	2
2	Two-Stage Programming: Main Features	5
2.1	Specification and Coordination	5
2.2	Consistency Link	6
2.2.1	Consistency Transformer	8
2.2.2	Consistency Checker	8
2.3	Relationship with Program Verification	10
2.4	Value Name Based Input/Output	11
2.4.1	I/O Declarations, Global I/O States	12
2.4.2	I/O Commands, I/O Run-Time Checks.	13
2.4.3	Remarks	16
2.5	Value Name Based Message Passing	17
2.5.1	Send Command	17
2.5.2	Receive Command	18
2.5.3	Send Message Designator	18
2.5.4	Receive Message Designator	19
2.6	Implementation	21
3	Case Studies	23
3.1	Floyd's Algorithm (Sequential)	23
3.1.1	Algorithm's Description	23
3.1.2	C, SML, and 2SP Implementations	24
3.1.3	Comparisons	25
3.1.3.1	Reliability	27
3.1.3.2	Efficiency	29

3.2	Dijkstra's Algorithm (Parallel)	32
3.2.1	Algorithm's Description	32
3.2.2	2SP Implementation	34
3.2.3	Comments on the Implementation	38
3.2.3.1	Reliability	38
3.2.3.2	Efficiency	41
4	Related Work. Comparisons	45
4.1	Correctness/Debugging	45
4.1.1	Program Result-Checking	46
4.1.1.1	Comparison	46
4.1.2	Certification Trail	48
4.1.2.1	Comparison	48
4.1.3	Relative Debugging	48
4.1.3.1	Comparison	48
4.2	Separation of Concerns	49
4.2.1	Aspect-Oriented Programming	49
4.2.1.1	Comparison	50
4.2.2	Parallelism/Coordination	51
4.2.2.1	Comparison	51
4.3	Recursive Function Definitions	51
4.3.1	Comparison	51
5	Implementation: The Compiler Back-End	53
5.1	Compiling 2SP Core	55
5.1.1	2SP Basic Concepts: Overview	55
5.1.1.1	Values	55
5.1.1.2	Functions	55
5.1.1.3	Stores	55
5.1.1.4	Value Names and Named Values	56
5.1.2	Front-End Back-End Interface	56
5.1.2.1	Types	56
5.1.2.2	Basic Abstract Syntax	56
5.1.2.3	Attributes	56
5.1.2.4	Abstract Syntax Tree 2	56
5.1.3	Compilation Overview	64
5.1.3.1	Identifiers	64
5.1.3.2	Tuples	65
5.1.3.3	Types, Variables, Functions	65
5.1.3.4	Stores	66

5.1.3.5	Value Name and Named Value Designators	67
5.1.3.6	Declarations	67
5.1.3.7	Expressions	67
5.1.3.8	Commands	68
5.1.4	New Abstract Representations	68
5.1.4.1	Structure Types '	68
5.1.4.2	Structure Common	69
5.1.5	Environments	72
5.1.6	AST_2 – AST_3 Transformation Subphase	73
5.1.6.1	Abstract Syntax Tree 3	73
5.1.6.2	Notation Conventions	75
5.1.6.3	Identifier Encoding	75
5.1.6.4	Attributed Identifier Transformation	77
5.1.6.5	Pattern Transformation	77
5.1.6.6	Value Name Designator Transformation	78
5.1.6.7	Named Value Designator Transformation	79
5.1.6.8	Expression Transformation	84
5.1.6.9	Declaration and Command Transformations	85
5.1.6.10	Program Transformation	85
5.1.6.11	Expression Qualification	87
5.1.7	AST_3 – AST_4 Transformation Subphase	91
5.1.7.1	Abstract Syntax Tree 4	91
5.1.7.2	Lambda Lifting	91
5.1.7.3	Qualified Expression Elimination	96
5.1.7.4	Iterator Expression Transformation	97
5.1.7.5	Remarks	98
5.1.7.6	Global Information Gathering	98
5.1.8	AST_4 – AST_5 Transformation Subphase	99
5.1.8.1	Abstract Syntax Tree 5	99
5.1.8.2	Notation Conventions	102
5.1.8.3	Declaration Transformation	103
5.1.8.4	Expression Transformation	108
5.1.8.5	Command Transformation	114
5.1.8.6	Generating Intrinsic Function Declarations	118
5.1.9	Code Generation	118
5.1.9.1	Code Generation for Store Types	119
5.1.9.2	Code Generation for Intrinsic Functions	122
5.1.9.3	Code Generation for Declarations	122
5.1.9.4	Code Generation for Commands	123

5.2	Compiling Input/Output	124
5.2.1	AST_2 – AST_3 Transformation	124
5.2.1.1	Abstract Representations	124
5.2.1.2	Input/Output Declaration Transformation	126
5.2.1.3	Command Transformation	126
5.2.2	AST_3 – AST_4 Transformation	128
5.2.2.1	Abstract Representations	129
5.2.2.2	Transformations	130
5.2.3	AST_4 – AST_5 Transformation	133
5.2.3.1	Abstract Representations	133
5.2.3.2	Transformations	134
5.2.4	Code Generation	140
5.2.4.1	Code Generation for Global I/O States .	140
5.2.4.2	Code Generation for I/O Commands . .	141
6	Conclusions	143
6.1	Conclusions	143
6.1.1	Benefits	143
6.1.2	Costs	144
6.2	Future Work	145
6.2.1	Application Suitability/Case Studies	146
6.2.2	Compiler Optimization	146
6.2.3	Language Improvements	147
6.3	Perspective	147
	Bibliography	147
A	Structures Types, BasicAbsSyn, and Attribute	155
A.1	Structure Types	155
A.2	Structure BasicAbsSyn	156
A.3	Structure Attribute	157
B	Structure AbsSyn2	159
C	Structures Types' and Common	163
C.1	Structure Types'	163
C.2	Structure Common	164
D	Structure AbsSyn3	167
E	Structure AbsSyn4	171

F Structure AbsSyn5	175
Index	178
Curriculum Vitae	183

List of Figures

2.1	A simple 2SP program.	6
2.2	A corresponding standard imperative program.	7
2.3	Proof outline.	11
2.4	Interpreting and compiling a 2SP program.	22
3.1	Floyd’s algorithm in C.	24
3.2	Floyd’s algorithm in SML.	25
3.3	Floyd’s algorithm in 2SP.	26
3.4	Value-store associations for computing $d(t, i, j)$	27
3.5	A graph plus its adjacency and all-pairs shortest-path matrices.	27
3.6	2SP specification of the PBSP algorithm.	35
3.7	2SP coordination of the PBSP algorithm (master). Continued in Figure 3.8.	36
3.8	2SP coordination of the PBSP algorithm (slave). Continued from Figure 3.7.	37
5.1	The phases of the compiler back-end.	53
5.2	The subphases of the transformation phase.	54
5.3	Structure <code>Types</code>	57
5.4	Structure <code>BasicAbsSyn</code>	58
5.5	Structure <code>Attribute</code>	59
5.6	Structure <code>AbsSyn2</code> . Continued in Figure 5.7	60
5.7	Structure <code>AbsSyn2</code> . Continued from Figure 5.6	61
5.8	Structure <code>Types’</code>	69
5.9	Structure <code>Common</code>	70
5.10	Basic environment signature.	72
5.11	Layered environment signature.	73
5.12	Structure <code>AbsSyn3</code>	74

5.13	Structure AbsSyn4 . Continued in Figure 5.14.	92
5.14	Structure AbsSyn4 . Continued from Figure 5.13.	93
5.15	Structure AbsSyn5 . Continued in Figure 5.16.	100
5.16	Structure AbsSyn5 . Continued from Figure 5.15.	101