Gavin Bierman
Christoph Koch (Eds.)

Database Programming Languages

LNCS 3774

10th International Symposium, DBPL 2005
Trondheim, Norway, August 2005
Revised Selected Papers

Gavin Bierman Christoph Koch (Eds.)

Database Programming Languages

10th International Symposium, DBPL 2005
Trondheim, Norway, August 28-29, 2005
Revised Selected Papers

Springer

Volume Editors

Gavin Bierman
Microsoft Research
JJ Thomson Avenue, Cambridge CB3 0FB, UK
E-mail: gmb@microsoft.com

Christoph Koch
Universität des Saarlandes
Lehrstuhl für Informationssysteme
Postfach 15 11 50, 66041 Saarbrücken, Germany
E-mail: koch@infosys.uni-sb.de

Library of Congress Control Number: 2005937142

CR Subject Classification (1998): H.2, H.3, E.2, D.3.3, H.4

ISSN 0302-9743
ISBN-10 3-540-30951-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-30951-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11601524 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 3774

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Preface

The 10th International Symposium on Database Programming Languages, DBPL 2005, was held in Trondheim, Norway in August 2005. DBPL 2005 was one of 11 meetings to be co-located with VLDB (the International Conference on Very Large Data Bases).

DBPL continues to present the very best work at the intersection of database and programming language research. DBPL 2005 accepted 17 papers out of a total of 63 submissions; an acceptance rate of 27%. Every submission was reviewed by at least three members of the program committee. In addition, the program committee sought the opinions of 51 additional referees, selected because of their expertise on particular topics. The final selection of papers was made during the last week of June. All authors of accepted papers submitted corrected versions, which were collected in an informal proceedings and distributed to the attendees of DBPL 2005. As is traditional for DBPL, this volume was produced after the meeting and authors were able to make improvements to their papers following discussions and feedback at the meeting.

The invited lecture at DBPL 2005 was given by Giuseppe Castagna entitled "Patterns and Types for Querying XML Documents"; an extended version of the lecture appears in this volume. Given the topic of this invited lecture, we invited all attendees of the Third International XML Database Symposium (XSym 2005), also co-located with VLDB, to attend. Continuing this collaboration, we organized with the co-chairs of XSym 2005 a shared panel session to close both meetings. The invited panel discussed "Whither XML, c. 2005?" and consisted of experts on various aspects of XML: Gavin Bierman (Microsoft Research), Peter Buneman (University of Edinburgh), Dana Florescu (Oracle), H.V. Jagadish (University of Michigan) and Jayavel Shanmugasundaram (Cornell University). We are grateful to the panel and the audience for a stimulating and good-humored discussion.

We owe thanks to a large number of people for making DBPL 2005 such a great success. First, we are grateful to the hard work and diligence of the 21 distinguished researchers who served on the program committee. We also thank Peter Buneman, Georg Lausen and Dan Suciu, who offered us much assistance and sound counsel. Svein Erik Bratsberg provided flawless local organization. Chani Johnson gave us much help in mastering the subtleties of the Microsoft Research Conference Management Tool. It was a great pleasure to organize a shared panel and invited lecture with Ela Hunt and Zachary Ives; the co-chairs of XSym 2005. Finally, we acknowledge the generous financial support of Microsoft Research.

September 2005 Gavin Bierman and Christoph Koch

Organization

Program Co-chairs

Gavin Bierman	Microsoft Research Cambridge, UK
Christoph Koch	University of Saarland, Germany

Program Committee

Marcelo Arenas	University of Toronto, Canada
Omar Benjelloun	Stanford University, USA
Sara Cohen	Technion, Israel
James Cheney	University of Edinburgh, UK
Alin Deutsch	University of California, San Diego, USA
Alain Frisch	INRIA Rocquencourt, France
Philippa Gardner	Imperial College, London, UK
Giorgio Ghelli	University of Pisa, Italy
Torsten Grust	University of Konstanz, Germany
Jan Hidders	University of Antwerp, Belgium
Haruo Hosoya	Tokyo University, Japan
Sergey Melnik	Microsoft Research, USA
Tova Milo	Tel Aviv University, Israel
Gerome Miklau	University of Washington, USA
Frank Neven	University of Limburg, Belgium
Alexandra Poulovassilis	Birkbeck College, London, UK
Francesco Scarcello	University of Calabria, Italy
Michael Schwartzbach	BRICS, Denmark
Alan Schmitt	INRIA Rhône-Alpes, France
Nicole Schweikardt	Humboldt University, Berlin, Germany
David Toman	University of Waterloo, Canada

Additional Referees

Fabrizio Angiulli	William Cook
Alessandro Artale	Giovanni Conforti
Pablo Barcelo	Thierry Coupaye
Leo Bertossi	Nick Crasswell
José Blakeley	Wlodzimierz Drabent
Claus Brabrand	Wolfgang Faber
Gilad Bracha	Nate Foster
Cristiano Calcagno	Eric Fusy
Dario Colazzo	Vladimir Gapeyev

Gianluigi Greco
Kenji Hashimoto
Zhenjiang Hu
Giovambattista Ianni
Kazuhiro Inaba
Shinya Kawanaka
Christian Kirkegaard
Leonid Libkin
Andrei Lopatenko
Ioana Manolescu
Paolo Manghi
Wim Martens
Elio Masciari
Anders Møller
Keisuke Nakano
Nathaniel Nystrom
Atsushi Ohori
Dan Olteanu
Vanessa de Paula Braganholo
Andrea Pugliese
Mukund Raghavachari
Carlo Sartiani
Stefanie Scherzinger
Helmut Seidl
Jérôme Siméon
Cristina Sirangelo
Keishi Tajima
Jens Teubner
Stijn Vansummeren
Roel Vercammen
Philip Wadler
Geoffrey Washburn
Grant Weddell

Sponsoring Institution

Microsoft Research

Lecture Notes in Computer Science

For information about Vols. 1–3735

please contact your bookseller or Springer

Vol. 3837: K. Cho, P. Jacquet (Eds.), Technologies for Advanced Heterogeneous Networks. IX, 307 pages. 2005.

Vol. 3835: G. Sutcliffe, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. XIV, 744 pages. 2005. (Subseries LNAI).

Vol. 3833: K.-J. Li, C. Vangenot (Eds.), Web and Wireless Geographical Information Systems. XI, 309 pages. 2005.

Vol. 3829: P. Pettersson, W. Yi (Eds.), Formal Modeling and Analysis of Timed Systems. IX, 305 pages. 2005.

Vol. 3828: X. Deng, Y. Ye (Eds.), Internet and Network Economics. XVII, 1106 pages. 2005.

Vol. 3826: B. Benatallah, F. Casati, P. Traverso (Eds.), Service-Oriented Computing - ICSOC 2005. XVIII, 597 pages. 2005.

Vol. 3824: L.T. Yang, M. Amamiya, Z. Liu, M. Guo, F.J. Rammig (Eds.), Embedded and Ubiquitous Computing. XXIII, 1204 pages. 2005.

Vol. 3823: T. Enokido, L. Yan, B. Xiao, D. Kim, Y. Dai, L.T. Yang (Eds.), Embedded and Ubiquitous Computing. XXXII, 1317 pages. 2005.

Vol. 3822: D. Feng, D. Lin, M. Yung (Eds.), Information Security and Cryptology. XII, 420 pages. 2005.

Vol. 3821: R. Ramanujam, S. Sen (Eds.), FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science. XIV, 566 pages. 2005.

Vol. 3820: L.T. Yang, X. Zhou, W. Zhao, Z. Wu, Y. Zhu, M. Lin (Eds.), Embedded Software and Systems. XXVIII, 779 pages. 2005.

Vol. 3818: S. Grumbach, L. Sui, V. Vianu (Eds.), Advances in Computer Science – ASIAN 2005. XIII, 294 pages. 2005.

Vol. 3815: E.A. Fox, E.J. Neuhold, P. Premsmit, V. Wuwongse (Eds.), Digital Libraries: Implementing Strategies and Sharing Experiences. XVII, 529 pages. 2005.

Vol. 3814: M. Maybury, O. Stock, W. Wahlster (Eds.), Intelligent Technologies for Interactive Entertainment. XV, 342 pages. 2005. (Subseries LNAI).

Vol. 3810: Y.G. Desmedt, H. Wang, Y. Mu, Y. Li (Eds.), Cryptology and Network Security. XI, 349 pages. 2005.

Vol. 3809: S. Zhang, R. Jarvis (Eds.), AI 2005: Advances in Artificial Intelligence. XXVII, 1344 pages. 2005. (Subseries LNAI).

Vol. 3808: C. Bento, A. Cardoso, G. Dias (Eds.), Progress in Artificial Intelligence. XVIII, 704 pages. 2005. (Subseries LNAI).

Vol. 3807: M. Dean, Y. Guo, W. Jun, R. Kaschek, S. Krishnaswamy, Z. Pan, Q.Z. Sheng (Eds.), Web Information Systems Engineering – WISE 2005 Workshops. XV, 275 pages. 2005.

Vol. 3806: A.H. H. Ngu, M. Kitsuregawa, E.J. Neuhold, J.-Y. Chung, Q.Z. Sheng (Eds.), Web Information Systems Engineering – WISE 2005. XXI, 771 pages. 2005.

Vol. 3805: G. Subsol (Ed.), Virtual Storytelling. XII, 289 pages. 2005.

Vol. 3804: G. Bebis, R. Boyle, D. Koracin, B. Parvin (Eds.), Advances in Visual Computing. XX, 755 pages. 2005.

Vol. 3803: S. Jajodia, C. Mazumdar (Eds.), Information Systems Security. XI, 342 pages. 2005.

Vol. 3802: Y. Hao, J. Liu, Y. Wang, Y.-m. Cheung, H. Yin, L. Jiao, J. Ma, Y.-C. Jiao (Eds.), Computational Intelligence and Security, Part II. XLII, 1166 pages. 2005. (Subseries LNAI).

Vol. 3801: Y. Hao, J. Liu, Y. Wang, Y.-m. Cheung, H. Yin, L. Jiao, J. Ma, Y.-C. Jiao (Eds.), Computational Intelligence and Security, Part I. XLI, 1122 pages. 2005. (Subseries LNAI).

Vol. 3799: M. A. Rodríguez, I.F. Cruz, S. Levashkin, M.J. Egenhofer (Eds.), GeoSpatial Semantics. X, 259 pages. 2005.

Vol. 3798: A. Dearle, S. Eisenbach (Eds.), Component Deployment. X, 197 pages. 2005.

Vol. 3797: S. Maitra, C. E. V. Madhavan, R. Venkatesan (Eds.), Progress in Cryptology - INDOCRYPT 2005. XIV, 417 pages. 2005.

Vol. 3796: N.P. Smart (Ed.), Cryptography and Coding. XI, 461 pages. 2005.

Vol. 3795: H. Zhuge, G.C. Fox (Eds.), Grid and Cooperative Computing - GCC 2005. XXI, 1203 pages. 2005.

Vol. 3794: X. Jia, J. Wu, Y. He (Eds.), Mobile Ad-hoc and Sensor Networks. XX, 1136 pages. 2005.

Vol. 3793: T. Conte, N. Navarro, W.-m.W. Hwu, M. Valero, T. Ungerer (Eds.), High Performance Embedded Architectures and Compilers. XIII, 317 pages. 2005.

Vol. 3792: I. Richardson, P. Abrahamsson, R. Messnarz (Eds.), Software Process Improvement. VIII, 215 pages. 2005.

Vol. 3791: A. Adi, S. Stoutenburg, S. Tabet (Eds.), Rules and Rule Markup Languages for the Semantic Web. X, 225 pages. 2005.

Vol. 3790: G. Alonso (Ed.), Middleware 2005. XIII, 443 pages. 2005.

Vol. 3789: A. Gelbukh, Á. de Albornoz, H. Terashima-Marín (Eds.), MICAI 2005: Advances in Artificial Intelligence. XXVI, 1198 pages. 2005. (Subseries LNAI).

Vol. 3788: B. Roy (Ed.), Advances in Cryptology - ASIACRYPT 2005. XIV, 703 pages. 2005.

Vol. 3785: K.-K. Lau, R. Banach (Eds.), Formal Methods and Software Engineering. XIV, 496 pages. 2005.

Vol. 3784: J. Tao, T. Tan, R.W. Picard (Eds.), Affective Computing and Intelligent Interaction. XIX, 1008 pages. 2005.

Vol. 3783: S. Qing, W. Mao, J. Lopez, G. Wang (Eds.), Information and Communications Security. XIV, 492 pages. 2005.

Vol. 3781: S.Z. Li, Z. Sun, T. Tan, S. Pankanti, G. Chollet, D. Zhang (Eds.), Advances in Biometric Person Authentication. XI, 250 pages. 2005.

Vol. 3780: K. Yi (Ed.), Programming Languages and Systems. XI, 435 pages. 2005.

Vol. 3779: H. Jin, D. Reed, W. Jiang (Eds.), Network and Parallel Computing. XV, 513 pages. 2005.

Vol. 3778: C. Atkinson, C. Bunse, H.-G. Gross, C. Peper (Eds.), Component-Based Software Development for Embedded Systems. VIII, 345 pages. 2005.

Vol. 3777: O.B. Lupanov, O.M. Kasim-Zade, A.V. Chaskin, K. Steinhöfel (Eds.), Stochastic Algorithms: Foundations and Applications. VIII, 239 pages. 2005.

Vol. 3775: J. Schönwälder, J. Serrat (Eds.), Ambient Networks. XIII, 281 pages. 2005.

Vol. 3774: G. Bierman, C. Koch (Eds.), Database Programming Languages. X, 295 pages. 2005.

Vol. 3773: A. Sanfeliu, M.L. Cortés (Eds.), Progress in Pattern Recognition, Image Analysis and Applications. XX, 1094 pages. 2005.

Vol. 3772: M. Consens, G. Navarro (Eds.), String Processing and Information Retrieval. XIV, 406 pages. 2005.

Vol. 3771: J.M.T. Romijn, G.P. Smith, J. van de Pol (Eds.), Integrated Formal Methods. XI, 407 pages. 2005.

Vol. 3770: J. Akoka, S.W. Liddle, I.-Y. Song, M. Bertolotto, I. Comyn-Wattiau, W.-J. van den Heuvel, M. Kolp, J. Trujillo, C. Kop, H.C. Mayr (Eds.), Perspectives in Conceptual Modeling. XXII, 476 pages. 2005.

Vol. 3769: D.A. Bader, M. Parashar, V. Sridhar, V.K. Prasanna (Eds.), High Performance Computing – HiPC 2003. XXVIII, 550 pages. 2005.

Vol. 3768: Y.-S. Ho, H.J. Kim (Eds.), Advances in Multimedia Information Processing - PCM 2005, Part II. XXVIII, 1088 pages. 2005.

Vol. 3767: Y.-S. Ho, H.J. Kim (Eds.), Advances in Multimedia Information Processing - PCM 2005, Part I. XXVIII, 1022 pages. 2005.

Vol. 3766: N. Sebe, M.S. Lew, T.S. Huang (Eds.), Computer Vision in Human-Computer Interaction. X, 231 pages. 2005.

Vol. 3765: Y. Liu, T. Jiang, C. Zhang (Eds.), Computer Vision for Biomedical Image Applications. X, 563 pages. 2005.

Vol. 3764: S. Tixeuil, T. Herman (Eds.), Self-Stabilizing Systems. VIII, 229 pages. 2005.

Vol. 3762: R. Meersman, Z. Tari, P. Herrero (Eds.), On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops. XXXI, 1228 pages. 2005.

Vol. 3761: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, Part II. XXVII, 653 pages. 2005.

Vol. 3760: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, Part I. XXVII, 921 pages. 2005.

Vol. 3759: G. Chen, Y. Pan, M. Guo, J. Lu (Eds.), Parallel and Distributed Processing and Applications - ISPA 2005 Workshops. XIII, 669 pages. 2005.

Vol. 3758: Y. Pan, D.-x. Chen, M. Guo, J. Cao, J.J. Dongarra (Eds.), Parallel and Distributed Processing and Applications. XXIII, 1162 pages. 2005.

Vol. 3757: A. Rangarajan, B. Vemuri, A.L. Yuille (Eds.), Energy Minimization Methods in Computer Vision and Pattern Recognition. XII, 666 pages. 2005.

Vol. 3756: J. Cao, W. Nejdl, M. Xu (Eds.), Advanced Parallel Processing Technologies. XIV, 526 pages. 2005.

Vol. 3754: J. Dalmau Royo, G. Hasegawa (Eds.), Management of Multimedia Networks and Services. XII, 384 pages. 2005.

Vol. 3753: O.F. Olsen, L.M.J. Florack, A. Kuijper (Eds.), Deep Structure, Singularities, and Computer Vision. X, 259 pages. 2005.

Vol. 3752: N. Paragios, O. Faugeras, T. Chan, C. Schnörr (Eds.), Variational, Geometric, and Level Set Methods in Computer Vision. XI, 369 pages. 2005.

Vol. 3751: T. Magedanz, E.R. M. Madeira, P. Dini (Eds.), Operations and Management in IP-Based Networks. X, 213 pages. 2005.

Vol. 3750: J.S. Duncan, G. Gerig (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2005, Part II. XL, 1018 pages. 2005.

Vol. 3749: J.S. Duncan, G. Gerig (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2005, Part I. XXXIX, 942 pages. 2005.

Vol. 3748: A. Hartman, D. Kreische (Eds.), Model Driven Architecture – Foundations and Applications. IX, 349 pages. 2005.

Vol. 3747: C.A. Maziero, J.G. Silva, A.M.S. Andrade, F.M.d. Assis Silva (Eds.), Dependable Computing. XV, 267 pages. 2005.

Vol. 3746: P. Bozanis, E.N. Houstis (Eds.), Advances in Informatics. XIX, 879 pages. 2005.

Vol. 3745: J.L. Oliveira, V. Maojo, F. Martín-Sánchez, A.S. Pereira (Eds.), Biological and Medical Data Analysis. XII, 422 pages. 2005. (Subseries LNBI).

Vol. 3744: T. Magedanz, A. Karmouch, S. Pierre, I. Venieris (Eds.), Mobility Aware Technologies and Applications. XIV, 418 pages. 2005.

Vol. 3742: J. Akiyama, M. Kano, X. Tan (Eds.), Discrete and Computational Geometry. VIII, 213 pages. 2005.

Vol. 3740: T. Srikanthan, J. Xue, C.-H. Chang (Eds.), Advances in Computer Systems Architecture. XVII, 833 pages. 2005.

Vol. 3739: W. Fan, Z. Wu, J. Yang (Eds.), Advances in Web-Age Information Management. XXIV, 930 pages. 2005.

Vol. 3738: V.R. Syrotiuk, E. Chávez (Eds.), Ad-Hoc, Mobile, and Wireless Networks. XI, 360 pages. 2005.

Vol. 3737: C. Priami, E. Merelli, P. Gonzalez, A. Omicini (Eds.), Transactions on Computational Systems Biology III. VII, 169 pages. 2005. (Subseries LNBI).

Table of Contents

Patterns and Types for Querying XML Documents

Giuseppe Castagna

CNRS, École Normale Supérieure de Paris, France

Abstract. Among various proposals for primitives for deconstructing XML data two approaches seem to clearly stem from practice: path expressions, widely adopted by the database community, and regular expression patterns, mainly developed and studied in the programming language community. We think that the two approaches are complementary and should be both integrated in languages for XML, and we see in that an opportunity of collaboration between the two communities. With this aim, we give a presentation of regular expression patterns and the type systems they are tightly coupled with. Although this article advocates a construction promoted by the programming language community, we will try to stress some characteristics that the database community, we hope, may find interesting.

1 Introduction

Working on XML trees requires at least two different kinds of language primitives: (*i*) deconstruction/extraction primitives (usually called patterns or templates) that pinpoint and capture subparts of the XML data, and (*ii*) iteration primitives, that iterate over XML trees the process of extraction and transformation of data.

Concerning iteration primitives, there are many quite disparate proposals: in this category one can find such different primitives as the FLWR (i.e., for-let-where-return) expressions of XQuery [7], the `filter` primitive of XDuce [40, 39], the `xtransform` primitive of CDuce [4], the `iterate` primitive of Xtatic [31], the select-from-where of Cω [6] and CQL [5], the select-where of Lorel [1] and loto-ql [51], while for other languages, for instance XSLT [22], the iterator is hard-coded in the semantics itself of the language.

Concerning deconstructing primitives, instead, the situation looks clearer since, among various proposals (see the related work section later on), two different and complementary solutions clearly stem from practice: path expressions (usually XPath paths [21], but also the "dot" navigations of Cω or Lorel [1], caterpillar expressions [12] and their "looping" extension [33]) and regular expression patterns [41].

Path expressions are navigational primitives that pinpoint where to capture data subcomponents. XML path expressions (and those of Cω and Lorel in particular) closely resemble the homonimic primitives used by OQL [23] in the context of OODB query languages, with the difference that instead of sets of objects they return sets or sequences of XML elements: more precisely all elements that can be reached by following the paths at issue. These primitives are at the basis of standard languages such as XSLT and XQuery.

More recently, a new kind of deconstruction primitive was proposed: regular expression patterns [41], which extends by regular expressions the pattern matching primitive

G. Bierman and C. Koch (Eds.): DBPL 2005, LNCS 3774, pp. 1–26, 2005.

as popularised by functional languages such as ML and Haskell. Regular expression patterns were first introduced in the XDuce programming language and are becoming more and more popular, since they are being adopted by such quite different languages as ℂDuce [4] (a general purpose extension of the XDuce language) and its query language ℂQL [5], Xtatic [31] (an extension of C#), Scala [54] (a general purpose Java-like object-oriented language that compiles to Java bytecode), XHaskell [45] as well as the extension of Haskell proposed by Broberg *et al.* [11].

The two kinds of primitives are not antagonist, but rather orthogonal and complementary. Path expressions implement a "vertical" exploration of data as they capture elements that may be at different depths, while patterns perform a "horizontal" exploration of data since they are able to perform finer grained decomposition on sequences of elements. The two kinds of primitives are quite useful and they complement each other nicely. Therefore, it would seem natural to integrate both of them in a query or programming language for XML. In spite of this and of several theoretical works on the topic (see the related work section), we are aware of just two running languages in which both primitives are embedded (and, yet, loosely coupled): in ℂQL [5] it is possible to write select-from-where expressions, where regular expression patterns are applied in the from clause to sequences that are returned by XPath-like expressions (see the example at the end of Section 2.3); Gapeyev and Pierce [32] show how it is possible to use regular expression patterns with an all-matches semantics to encode a subset of XPath and use this encoding to add XPath to the Xtatic programming language.

The reason for the lack of study of the integration of these two primitives may be due to the fact that each of them is adopted by a different community: regular patterns are almost confined to the programming language community while XPath expressions are pervasive in the database community.

The goal of this lecture is to give a brief presentation of the regular pattern expressions style together with the type system they are tightly coupled with, that is the *semantic subtyping*-based type systems [19, 29]. We are not promoting the use of these to the detriment of path expressions, since we think that the two approaches should be integrated in the same language and we see in that a great opportunity of collaboration between the database and the programming languages communities. Since the author belongs to latter, this lecture tries to describe the pattern approach addressing some points that, we hope, should be of interest to the database community as well. In particular, after a general overview of regular expression patterns and types (Section 2) in which we show how to embed patterns in a select-from-where expression, we discuss several usages of these semantic subtyping based patterns/types (henceforward, we will often call them "semantic patterns/types"): how to use these patterns and types to give informative error messages (Section 3.2), to dig out errors that are out of reach of previous type checker technologies (Section 3.3) and how the static information they give can be used to define very efficient and highly optimised runtimes (Section 3.4); we show that these patterns permit new logical query optimisations (Section 3.5) and can be used as building blocks to allow the programmer to fine-grainedly define new iterators on data (Section 3.6); finally, the techniques developed for the semantic patterns and types can be used to define optimal data pruning and other optimisation techniques (Section 3.7–3.8)

Related Work. In this work we focus on data extraction primitives coming from the *practice* of programming and query languages manipulating XML data. Thus, we restrict our attention to the primitives included in full-featured languages with a stable community of users. There are however many other proposals in the literature for deconstructing, extracting, and querying XML data.

First and foremost there are all the languages developed from logics for unranked trees whose yardstick in term of expressiveness is the Monadic Second Order Logic. The list here would be too long and we invite the interested reader to consult the excellent overview by Leonid Libkin on the subject [44]. In this area we want to single out the work on composition of monadic queries in [26], since it looks as a promising step toward the integration of path and pattern primitives we are promoting in this work: we will say more about it in the conclusion. A second work that we want to distinguish is Neven and Schwentick's ETL [49], where regular expressions over logical formulæ allow both horizontal and vertical exploration of data; but, as the authors themselves remark, the gap with a usable pattern language is very important, especially if one wants to define non-unary queries typical of Hosoya's regular expressions patterns.

Based on logics also are the query languages developed on or inspired to Ambient Logic, a modal logic that can express spatial properties on unordered trees, as well as to other spatial logics. The result is a very interesting mix of path-like and pattern-like primitives (cf. the dot notation and the spatial formulæ with capture variables that can be found in TQL) [24, 13, 16, 14, 15, 17].

In the query language research, we want to signal the work of Papakonstantinou and Vianu [51] where the loto-ql query language is introduced. In loto-ql it is possible to write `select` x `where` p, where p is a pattern in the form of tree which uses regular expressions to navigate both horizontally and vertically in the input tree, and provides bindings of x.

2 A Brief Introduction to Patterns and Types for XML

In this section we give a short survey of patterns and types for XML. We start with a presentation of pattern matching as it can be found in functional languages (Section 2.1), followed by a description of "semantic" types and of pattern-based query primitives (Section 2.2); a description of regular expression patterns for XML (Section 2.3) and their formal definition (Section 2.4) follow, and few comments on iterators (Section 2.5) close the section. Since we introduce early in this section new concepts and notations that will be used in the rest of the article, we advise also the knowledgeable reader to consult it.

2.1 Pattern Matching in Functional Languages

Pattern matching is used in functional languages as a convenient way to capture subparts of non-functional[1] values, by binding them to some variables. For instance, imagine that

[1] We intend *non-functional* in a strict sense. So non-functional values are integer and boolean constants, pair of values, record of values, etc., but not λ-abstractions. Similarly a non-functional type is any type that is not an arrow type.

e is an expression denoting a pair and that we want to bind to x and y respectively to the first and second projection of e, so as to use them in some expression e'. Without patterns this is usually done by two `let` expressions:

```
let x = first(e) in
let y = second(e) in e'
```

With patterns this can be obtained by a single let expression:

```
let (x,y) = e in e'
```

The pattern `(x,y)` simply reproduces the form of the expected result of e and variables indicate the parts of the value that are to be captured: the value returned by e is *matched* against the pattern and the result of this matching is a *substitution*; in the specific case, it is the substitution that assigns the first projection of (the result of) e to x and the second one to y.

If we are not interested in capturing all the parts that compose the result of e, then we can use the wildcard "`_`" in correspondence of the parts we want to discard. For instance, in order to capture just the first projection of e, we can use the following pattern:

```
let (x,_) = e in ...
```

which returns the substitution that assigns the result of $first(e)$ to x. In general, a pattern has the form of a value in which some sub-occurrences are replaced by variables (these correspond to parts that are to be captured) and other are replaced by "`_`" (these correspond to parts that are to be discarded). A value is then matched against a pattern and if they both have the same structure, then the matching operation returns the substitution of the pattern variables by the corresponding occurrences of the value. If they do not have the same structure the matching operation *fails*. Since a pattern may fail—and here resides the power of pattern matching—it is interesting to try on the same value several different patterns. This is usually done with a `match` expression, where several patterns, separated by `|`, are tried in succession (according to a so-called "first match" policy). For instance:

```
match e with
  | (_,_) -> true
  | _ -> false
```

first checks whether e returns a pair in which case it returns `true`, otherwise it returns `false`. Note that, in some sense, matching is not very different from a type case. Actually, if we carefully define the syntax of our types, in particular if we use the same syntax for constructing types and their values, then the `match` operation *becomes* a type case: let us write (s,t) for the product type of the types s and t (instead of the more common $s \times t$ or $s * t$ notations) and use the wildcard "`_`" to denote the super-type of all types (instead of the more common Top, $\mathbb{1}$, or $\top$ symbols), then the match expression above is indeed a type case (if the result of e is in the product type `(_,_)` —the type of all products—, then return true else if it is of type top—all values have this type—, then return false). We will see the advantages of such a notation later on, for the time

being just notice that with such a syntactic convention for types and values, a pattern is a (non-functional) type in which some variables may appear.

Remark 1. *A* pattern *is just a non-functional type where some occurrences may be variables.*

The matching operation is very useful in the definition of functions, as it allows the programmer to define them by cases on the input. For instance, imagine that we encode lists recursively *à la* lisp, that is, either by a nil element for the empty list, or by pairs in which the left projection is the head and the right projection the tail of the list. With our syntax for products and top this corresponds to the recursive definition `List = 'nil | (_,List)`: a list is either `'nil` (we use a back-quote to denote constants so to syntactically distinguish them in patterns from variables) or the product of any type and a list. We can now write a tail recursive function[2] that computes the length of a list[3]

```
fun length ((List,Int) -> Int)
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

which is declared (see Footnote 3 for notation) to be of type `(List,Int) -> Int`, that is, it takes a pair composed of a list and an integer and returns an integer. More precisely, it takes the list of elements still to be counted and the number of elements already counted (thus `length(a,0)` computes the length of the list `a`). If the list is `'nil`, then the function returns the integer captured by the pattern variable `n`, otherwise it discards the head of the list (by using a wildcard) and performs a recursive call on the tail, captured in `t`, and on `n+1`. Note that, as shown by the use of 'nil in the first pattern, patterns can also specify values. When a pattern contains a value v, then it matches only values in which the value v occurs in the same position. Remark 1 is still valid even in the case that values occur in patterns, since we can still consider a pattern as a type with variables: it suffices to consider a value as being the denotation of the singleton type that contains that value.

---

[2] A function is *tail recursive* if all recursive calls in its definition occur at the end of its execution flow (more precisely, it is tail recursive if the result of every call is equal to result of its recursive calls): this allows the compiler to optimise the execution of such functions, since it then becomes useless to save and restore the state of recursive calls since the result will be pushed on the top of the stack by the last recursive call.

[3] We use two different syntaxes for functions. The usual notation is standard: for instance, the identity function on integers will be written as `fun id(x:Int):Int = x`. But if we want to feed the arguments of a function directly to a pattern matching, then the name of the function will be immediately followed by the type of the function itself. In this notation the identity for integers is rather written as `fun id(Int->Int) x -> x`. This is the case for the function length that follows, which could be equivalently defined as

```
fun length (x :(List,Int)):Int =
match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

2.2 Union, Intersection, and Difference Types

In order to type-check `match` expressions, the type-checker must compute unions, intersections, and differences (or, equivalently, negations) of types: let us denote these operations by `|` for the union, `&` for the intersection, and `\` for the difference. The reason why the type-checker needs to compute them can be better understood if we consider a type as a set of values, more precisely as the set of values that have that type: $t = \{v \mid v$ value of type $t\}$[4]. For instance, the product of the singleton type `'nil` and of the type `Int`, denoted by `('nil,Int)`, will be the set of all pairs in which the first element is the constant `'nil` and the second element is an integer. Notice that we already implicitly did such an hypothesis at the end of the previous section, when we considered a singleton type as a type *containing* just one value.

As we did for types, it is possible to associate also patterns to sets of values (actually, to types). Specifically, we associate to a pattern p the type $\lbag p \rbag$ defined as the set of values for which the pattern does not fail: $\lbag p \rbag = \{v \mid v$ matches pattern $p\}$. Since we use the same syntax for type constructors and value constructors, it results quite straightforward to compute $\lbag p \rbag$: it is the type obtained from p by substituting "`_`" for all occurrences of variables: the occurrences of values are now interpreted as the corresponding singleton types.

Let us check whether the function `length` has the type `(List,Int)` $\rightarrow$ `Int` it declares to have. The function is formed by two branches, each one corresponding to a different pattern. To know the type of the first branch we need to know the set of values (i.e., the type) that can be bound to n; the branch at issue will be selected and executed only for values that are arguments of the function—so that are in `(List,Int)`—and that are accepted by the pattern of the branch—so that are in $\lbag$`('nil,n)`$\rbag$ which by definition is equal to `('nil,_)`—. Thus, these are the values in the intersection `(List,Int)&('nil,_)`. By distributing the intersection on products and noticing that `List&'nil`= `'nil` and `Int&_`= `Int`, we deduce that the branch is executed for values in `('nil,Int)` and thus n is (bound to values) of type `Int`. The second branch returns a result of type `Int` (the result type declared for the function) provided that the recursive call is well-typed. In order to verify it, we need once more to compute the set of values for which the branch will be executed. These are the arguments of the function, minus the values accepted by the first branch, and intersected with the set of values accepted by the pattern of second branch, that is: `((List,Int)/('nil,_)) & ((_,_),_)`. Again, it is easy to see that this type is equal to `((_,List),Int)` and deduce that variable t is of type `List` and the variable n is of type `Int`: since the arguments have the expected types, then the application of the recursive call is well typed. The type of the result of the whole function is the union of the types of the two branches: since both return integers the union is integer. Finally, notice also that the match is *exhaustive*, that is, for every possible value that can be fed to the match, there exists at least one pattern that matches it. This holds true because the set of all arguments of the the function (that is, its domain) is contained in the union of the types accepted by the patterns.

[4] Formally, we are not defining the types, we are giving their semantics. So a type "is interpreted as" or "denotes" a set of values. We prefer not to enter in such a distinction here. See [19] for a more formal introduction about these types.