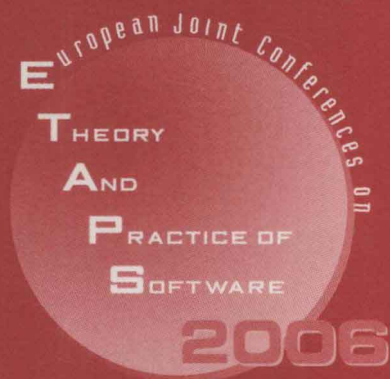


Peter Sestoft (Ed.)

LNCS 3924

Programming Languages and Systems

15th European Symposium on Programming, ESOP 2006
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2006
Vienna, Austria, March 2006, Proceedings



 Springer

Peter Sestoft (Ed.)

Programming Languages and Systems

15th European Symposium on Programming, ESOP 2006
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2006
Vienna, Austria, March 27-28, 2006
Proceedings

Volume Editor

Peter Sestoft

IT University of Copenhagen

Rued Langgaardsvej 7, 2300 Copenhagen S, Denmark

E-mail: sestoft@itu.dk

Library of Congress Control Number: 2006922219

CR Subject Classification (1998): D.3, D.1, D.2, F.3, F.4, E.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743

ISBN-10 3-540-33095-X Springer Berlin Heidelberg New York

ISBN-13 978-3-540-33095-0 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, **reprinting**, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2006

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper SPIN: 11693024 06/3142 5 4 3 2 1 0

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

New York University, NY, USA

Doug Tygar

University of California, Berkeley, CA, USA

Moshe Y. Vardi

Rice University, Houston, TX, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Foreword

ETAPS 2006 was the ninth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (CC, ESOP, FASE, FOSSACS, TACAS), 18 satellite workshops (ACCAT, AVIS, CMCS, COCV, DCC, EAAI, FESCA, FRCSS, GT-VMT, LDTA, MBT, QAPL, SC, SLAP, SPIN, TERMGRAPH, WITS and WRLA), two tutorials, and seven invited lectures (not including those that were specific to the satellite events). We received over 550 submissions to the five conferences this year, giving an overall acceptance rate of 23%, with acceptance rates below 30% for each conference. Congratulations to all the authors who made it to the final programme! I hope that most of the other authors still found a way of participating in this exciting event and I hope you will continue submitting.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate Program Committee and proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for “unifying” talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2006 was organized by the Vienna University of Technology, in co-operation with:

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST);
- Institute for Computer Languages, Vienna;
- Austrian Computing Society;
- The *Bürgermeister der Bundeshauptstadt Wien*;
- Vienna Convention Bureau;
- Intel.

The organizing team comprised:

Chair:	Jens Knoop
Local Arrangements:	Anton Ertl
Publicity:	Joost-Pieter Katoen
Satellite Events:	Andreas Krall
Industrial Liaison:	Eva Kühn
Liaison with City of Vienna:	Ulrich Neumerkel
Tutorials Chair, Website:	Franz Puntigam
Website:	Fabian Schmied
Local Organization, Workshops Proceedings:	Markus Schordan

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Perdita Stevens (Edinburgh, Chair), Luca Aceto (Aalborg and Reykjavík), Rastislav Bodík (Berkeley), Maura Cerioli (Genova), Matt Dwyer (Nebraska), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Roberto Gorrieri (Bologna), Reiko Heckel (Leicester), Michael Huth (London), Joost-Pieter Katoen (Aachen), Paul Klint (Amsterdam), Jens Knoop (Vienna), Shriram Krishnamurthi (Brown), Kim Larsen (Aalborg), Tiziana Margaria (Göttingen), Ugo Montanari (Pisa), Rocco de Nicola (Florence), Hanne Riis Nielson (Copenhagen), Jens Palsberg (UCLA), Mooly Sagiv (Tel-Aviv), João Saraiva (Minho), Don Sannella (Edinburgh), Vladimiro Sassone (Southampton), Helmut Seidl (Munich), Peter Sestoft (Copenhagen), Andreas Zeller (Saarbrücken).

I would like to express my sincere gratitude to all of these people and organizations, the Program Committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, the many reviewers, and Springer for agreeing to publish the ETAPS proceedings. Finally, I would like to thank the Organizing Chair of ETAPS 2006, Jens Knoop, for arranging for us to have ETAPS in the beautiful city of Vienna.

Edinburgh, January 2006

Perdita Stevens
ETAPS Steering Committee Chair

Preface

This volume contains 21 papers presented at ESOP 2006, the annual European Symposium on Programming, in Vienna, Austria, 27–28 March 2006. The first ESOP was organized in 1986 by Bernard Robinet and Reinhard Wilhelm in Saarbrücken, so this marks the 20th anniversary of ESOP, but is the 15th symposium, since the symposia were initially held biannually. On occasion of the anniversary we are particularly happy that Reinhard Wilhelm agreed to join this year's program committee.

The goal of ESOP has always been to bridge the gap between theory and practice, and the conferences continue to be devoted to addressing fundamental issues in the specification, analysis, and implementation of programming languages and systems.

The volume begins with a summary of Sophia Drossopoulou's ESOP invited talk, continues with the contributed ESOP papers, and ends with the abstract of Benjamin Pierce's ETAPS joint invited talk. The 21 ESOP papers were selected by the program committee from 87 full paper submissions, each reviewed by three or more reviewers, with four being the typical number. The reviews were done by the program committee and 143 additional referees, listed here. The accepted papers were selected during a two-week electronic discussion within the program committee.

Thanks go to the authors, the members of the program committee and the external referees for their excellent work, to the ESOP steering committee chair Hanne Riis Nielson, the ETAPS steering committee chair Perdita Stevens and the ETAPS 2006 local organization chair Jens Knoop for providing infrastructure and gentle reminders, and finally to the Online Conference System maintainer Martin Karrusseit for fixing server problems and adding desirable functionality.

Copenhagen, January 2006

Peter Sestoft

Organization

Program Chair

Peter Sestoft
Royal Veterinary and Agricultural University (KVL)
and IT University Copenhagen, Denmark

Program Committee

Anindya Banerjee	Kansas State University, USA
Anton Ertl	Technische Universitt Wien, Austria
David Warren	Stony Brook University, USA
Didier Rmy	INRIA Rocquencourt, France
Erik Meijer	Microsoft Corporation, USA
Eugenio Moggi	University of Genova, Italy
German Vidal	Technical University of Valencia, Spain
Giuseppe Castagna	cole Normale Suprieure, France
Joe Wells	Heriot-Watt University, UK
Kostis Sagonas	Uppsala University, Sweden
Michele Bugliesi	University of Venice, Italy
Mooly Sagiv	Tel-Aviv University, Israel
Nick Benton	Microsoft Research, UK
Peter O'Hearn	Queen Mary, University of London, UK
Peter Sestoft (chair)	KVL and IT University Copenhagen, Denmark
Peter Stuckey	Melbourne University, Australia
Peter Thiemann	Freiburg University, Germany
Pieter Hartel	Twente University, Netherlands
Reinhard Wilhelm	Saarland University, Germany
Stephanie Weirich	University of Pennsylvania, USA
Susan Eisenbach	Imperial College London, UK
Todd Veldhuizen	Indiana University, USA
Ulrik Pagh Schultz	University of Southern Denmark, Denmark

Additional Referees

E. Albert	C. Anderson	S. Barker
T. Amtoft	J. Avery	J. Bauer
D. Ancona	B. Aydemir	J. Berdine

- | | | |
|-----------------|------------------|-------------------|
| G. Bierman | W. Heaven | B. Pierce |
| L. Birkedal | T. Hildebrandt | A. Pitts |
| V. Bono | T. Hirschowitz | A. Podelski |
| A. Bossi | J. Jaffar | F. Pottier |
| B. Brassel | N.D. Jones | G. Puebla |
| R. Brinkman | A. Kennedy | F. Puntigam |
| A. Brogi | D. Kesner | F. Ranzato |
| P. Buchlovsky | J. Knoop | J. Rehof |
| N. Busi | K. Kristoffersen | J. Reineke |
| R. Caballero | G. Lagorio | S. Rossi |
| N. Cameron | J. Lawall | C. Russo |
| V. Capretta | J. Lee | R. Rydhof Hansen |
| L. Cardelli | S. Lengrand | C. Sadler |
| J. Cederquist | M. Lenisa | A. Saptawijaya |
| R. Chatley | X. Leroy | A. Schmitt |
| S. Chong | P. Levy | C. Schrmann |
| A. Compagnoni | P. Li | C. Segura |
| B. Cook | H.H. Lvengreen | P. Sewell |
| R. Corin | M. Maffei | J. Silva |
| A. Cortesi | S. Maffeis | C. Skalka |
| S. Crafa | M. Maher | M. Smith |
| K. Crary | P. Maier | P. Sobocinski |
| D. Cunningham | H. Makhholm | Z. Somogyi |
| M. Czenko | Y. Mandelbaum | H. Sndergaard |
| F. Damiani | R. Manevich | A. Stoughton |
| R. Davies | L. Maranget | E. Sumii |
| J. den Hartog | C. McBride | D. Syme |
| D. Distefano | I. Mijajlovic | S. Tse |
| J. Doumen | A. Myers | D. Varacca |
| D. Dreyer | A. Mller | A. Villanueva |
| S. Drossopoulou | R. Mller Jensen | D. Vytiniotis |
| G. Dufay | S. Nanz | B. Wachter |
| N. Dulay | M. Neubauer | P. Wadler |
| M. Elsmann | U. Neumerkel | D. Walker |
| E. Ernst | S. Nishimura | G. Washburn |
| S. Escobar | B. Nordstrm | A. Wasowski |
| S. Fagorzi | L. Ong | H. Xi |
| T. Field | R. Pagh | H. Yang |
| A. Filinski | N.S. Papaspyrou | G. Yorsh |
| J. Foster | J. Parrow | F. Zappa Nardelli |
| C. Fournet | A. Petrounias | U. Zarfary |
| A. Francalanza | M. Pettersson | S. Zdancwic |
| R. Garcia | S. Peyton Jones | E. Zucca |
| P. Giannini | A. Phillips | A. Zych |
| D. Gorla | I. Phillips | |

Table of Contents

Types for Hierarchic Shapes (Summary) <i>Sophia Drossopoulou, Dave Clarke, James Noble</i>	1
Linear Regions Are All You Need <i>Matthew Fluet, Greg Morrisett, Amal Ahmed</i>	7
Type-Based Amortised Heap-Space Analysis <i>Martin Hofmann, Steffen Jost</i>	22
Haskell Is Not Not ML <i>Ben Rudiak-Gould, Alan Mycroft, Simon Peyton Jones</i>	38
Coinductive Big-Step Operational Semantics <i>Xavier Leroy</i>	54
Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types <i>Amal Ahmed</i>	69
Approaches to Polymorphism in Classical Sequent Calculus <i>Alexander J. Summers, Steffen van Bakel</i>	84
Pure Pattern Calculus <i>Barry Jay, Delia Kesner</i>	100
A Verification Methodology for Model Fields <i>K. Rustan M. Leino, Peter Müller</i>	115
ILC: A Foundation for Automated Reasoning About Pointer Programs <i>Limin Jia, David Walker</i>	131
Bisimulations for Untyped Imperative Objects <i>Vasileios Koutavas, Mitchell Wand</i>	146
A Typed Assembly Language for Confidentiality <i>Dachuan Yu, Nayeem Islam</i>	162
Flow Locks: Towards a Core Calculus for Dynamic Flow Policies <i>Niklas Broberg, David Sands</i>	180
A Basic Contract Language for Web Services <i>Samuele Carpineti, Cosimo Laneve</i>	197

Types for Dynamic Reconfiguration

João Costa Seco, Luís Caires 214

Size-Change Termination Analysis in k -Bits

Michael Codish, Vitaly Lagoon, Peter Schachte, Peter J. Stuckey 230

Path Optimization in Programs and Its Application to Debugging

Akash Lal, Junghee Lim, Marina Polishchuk, Ben Liblit 246

Inference of User-Defined Type Qualifiers and Qualifier Rules

Brian Chin, Shane Markstrum, Todd Millstein, Jens Palsberg 264

Assertion Checking over Combined Abstraction of Linear Arithmetic
and Uninterpreted Functions

Sumit Gulwani, Ashish Tiwari 279

Embedding Dynamic Dataflow in a Call-by-Value Language

Gregory H. Cooper, Shriram Krishnamurthi 294

Polymorphic Type Inference for the JNI

Michael Furr, Jeffrey S. Foster 309

Type Safety of Generics for the .NET Common Language Runtime

Nicu G. Fruja 325

The Weird World of Bi-directional Programming

Benjamin C. Pierce 342

Author Index 343

Types for Hierarchic Shapes^{*}

(Summary)

Sophia Drossopoulou¹, Dave Clarke², and James Noble³

¹ Imperial College London, UK

² CWI, Amsterdam, The Netherlands

³ Victoria University of Wellington, Wellington, NZ

Abstract. Heap entities tend to contain complex references to each other. To manage this complexity, types which express shapes and hierarchies have been suggested. We survey type systems which describe such hierarchic shapes, how these types are used for reasoning about programs, and applications in concurrent programming.

Most imperative programs create and manipulate heap entities (objects, or records) which contain references to each other forming intricate topologies. This creates complexity, and makes programs difficult to understand and manipulate. Programmers, on the other hand, tend to think in terms of shapes, categorizations and hierarchies.

Thus, in the last decade, types describing shapes and hierarchies have been proposed to express programming intuitions, to support verification, and for synchronization and optimizations. We will discuss types for hierarchic shapes in terms of object oriented programming, because, even though the ideas are applicable to any imperative language, most of the related research was conducted in the context of object oriented languages.

1 Types for Hierarchic Shapes

Information hiding [28] was suggested as early as the 1970s, as a means to make programs more robust and easy to understand. Mechanisms that achieve information hiding by restricting the visibility of names, *e.g.*, private/protected annotations, are useful but insufficient. They prevent the *name* of an entity from being *used* outside a class or package, but do not prevent a *reference* to an entity from being *leaked* out of a structure [26].

To prevent such leaking of references, type systems have been suggested which give guarantees about the topology of the object graph, *i.e.*, about which objects may access which other objects.

Ownership types [15] introduce the concept of an object *owning* its nested objects; an *ownership context* is the set of objects with a given common owner.

^{*} Slides available from slurp.doc.ic.ac.uk/pubs.html#esop06.

Objects have a unique owner, thus ownership contexts are organized hierarchically into a tree structure. Furthermore, the owner controls access to the owned objects, because an object may only be accessed by its direct owner, or by objects (possibly indirectly) owned by the former object's owner. Therefore, owners are *dominators* [15], where o_1 *dominates* o_2 , if any path from the “outside” (or “root” of the object graph) to o_2 goes through o_1 .

Ownership types can thus be used to characterize the runtime structure of object graphs. Analysis of the heaps of programs has demonstrated that indeed, object graphs tend to have structure: In [29] analysis of heap dumps for a corpus of programs demonstrated that the average nesting (ownership) depth of objects is 5. In [30] heap dumps for 60 object graphs from 35 programs demonstrated that the number of incoming and outgoing references follow a *power law*, whereby the *log* of the number of objects with k references is proportional to *log* of k , thus challenging the common perception that oriented programs are built out of layers of homogeneous components.

The owners as dominators approach, also known as *deep ownership*, gives very strong encapsulation properties which are natural in containers and nested structures [13]. The approach has been used in program visualization [25].

On the other hand, deep ownership makes coding some popular structures, notably iterators, rather cumbersome. To alleviate this, *shallow ownership* has given up on the notion of owners as dominators. In [10] inner classes have privileged access to the objects enclosed by the corresponding outer class object; *i.e.*, objects of an inner class may refer to objects owned by their outer class objects. In [14, 2] objects on the stack are allowed to break deep ownership, and to refer to the inside of an ownership context. A more refined approach [1] decouples the encapsulation policy from the ownership mechanism, by allowing multiple ownership *domains* (contexts in our terminology) per object, and by allowing the programmer to specify permitted aliasing between pairs of contexts.

Ownership types usually cannot easily handle change of owner, except for *externally* unique objects, *i.e.*, for objects for which references from the outside are unique [17].

The type of an object describes the owner of the object itself as well as the owners of the fields of the object; because these may be distinct, types are parameterized by *ownership parameters* which will be instantiated by objects enclosing the current object. This requires all types to be annotated by a number of ownership parameters.

Universes [22] suggest a more lightweight approach, whereby references to owned objects, or references to objects with the same owner may be used for modifications, and references to any other objects are readonly. Thus, universe type systems do not require ownership parameters, and instead only distinguish between **rep** for owned, **peer** for same owner, and **readonly** annotations. Types are coarser: **readonly** are readonly references which may point into any context.

Confined types, on the other hand, introduce the concept of classes confined to their defining package, and guarantee that instances of a confined class are only accessible by instances of classes from the same package; thus, they are only

manipulated by code belonging to the same package as the class [8]. The annotations required for confined types are simple, and the object graph structure is simpler in the sense that the ownership contexts represent the packages, and thus are statically known.

1.1 Hierarchic Shapes for Program Verification

The decomposition of heaps into disjoint sets of objects allows these objects to be treated together for the purposes of verification. Central issues in the context of program verification are that an object’s properties may depend on other objects’ properties, that objects’ invariants need to be temporarily broken and later re-established, and the treatment of abstraction layers, *e.g.*, when a `Set` is implemented in terms of a `List`. The notion of ownership is primarily related to the dependence of objects’ properties rather than the topology of object graphs.

Universes were developed with the aim to support *modular* program verification; in [24] universe types were applied to JML for the description of frame properties, where `modifies` clauses of method specifications define which objects may be modified. Modularity is achieved by a form of “underspecification” of the semantics, allowing method calls to modify objects *outside* the ownership context of the receiver without being mentioned in the relevant `modifies`-clause.

In [6] a methodology for program specification and verification is proposed, whereby an object’s invariants may depend on (possibly indirectly) owned objects. The state space of programs is enriched to express whether an object’s validity holds (*i.e.*, whether its invariant holds); there is support for explicitly altering an object validity, and explicit ownership transfer. Subclassing means that an object’s invariant may hold at the level of different superclasses of the given object. This approach is refined and adapted to universes in [21], and is implemented in Boogie, and further extended in [7] to allow invariants to be expressed over shared state.

However, the necessity to explicitly manipulate an object’s validity increases the overhead of verification; therefore, [23] defines implicitly in which execution states an object’s invariants must hold, based on an ownership model which is enforced by the type system.

Representation independence, which means that a class can safely be replaced by another “equivalent” class provided it is encapsulated, *i.e.*, its internal representation is owned by instances of that class, is proven in [4]. In [5] the approach is extended to deal with shared state, recursive methods and callbacks, and the application to program equivalence.

In a more fundamental approach, [20] develops a logic for reasoning about mutable data structures whereby the spatial conjunction operator $*$ splits the heap into two disjoint parts, usually one representing the part necessary for some execution, and the other representing the rest. In [19] the conjunction $*$ is used to separate the internal resources of a module from those accessed by its client, to support verification in the context of information hiding. Work in [27] introduces *abstract predicates*, which are treated atomically outside a data structure, but whose definition may be used within the data structure, thus

supporting reasoning about modules, ADTs and classes. In these approaches the heap is split afresh in each verification step; there is no hierarchy in that the heap is just split into two parts. The approaches are very flexible, but do not yet handle issues around the dependency of objects' properties and breaking/re-establishing of objects' invariants.

Using a simpler methodology, rather than attempt full-fledged verification, [14] describes read-write effects of methods in terms of the ownership contexts, and uses these to determine when method calls are *independent*, *i.e.*, their execution does affect each other. In [31] the approach is extended to describe read-effects of predicates, and to infer when some execution does not affect the validity of some predicate.

1.2 Applications of Hierarchic Shapes

Hierarchic shapes have successfully been applied in concurrent programming, garbage collection, and in deployment time checks of architectural invariants.

Guava [3] introduces additional type rules to Java which control synchronization by distinguishing between objects which can be shared across threads, and those which cannot. The former are monitors, and the latter are either thread-local, or encapsulated within a monitor.

In [11] race-free programs are obtained though an extension of ownership types, where an object may be owned not only by another object (as in the classical system) but also by the object itself, or by a thread (to express objects local to threads). By acquiring the lock at the root of an ownership tree, a thread acquires exclusive access to all the members of that tree. In [9] the approach is extended to prevent deadlocks, by requiring a partial order among all locks, and statically checking that threads holding more than one lock acquire them in descending order.

In real-time Java, timely reclamation of memory is achieved through *scoped types* [32, 12]. Scopes correspond to ownership contexts, in that they contain objects, are hierarchically organized into a tree, and outer scopes may not hold references to objects within more deeply nested inner scopes. When a thread working in scope S_1 enters scope S_2 , then S_1 becomes the owner of S_2 . When a thread enters a scope it is dynamically checked that it originated in its owner scope, thus guaranteeing nesting of scopes into a tree hierarchy. Scopes are released upon thread exit.

In [16] the architectural integrity constraints of the Enterprise Java Beans architecture, which require beans to be confined within their wrappers, are enforced through a lightweight confinement model and a deployment checker.

1.3 Inference of Hierarchic Shapes

The various systems for hierarchic shapes impose an extra burden of annotation to programmers, as they require each appearance of a class in a type description to be annotated by ownership parameters or restrictions such as `rep`.

Kacheck/J [18] is a tool which infers which classes are confined within a package in the sense of [8]. Applied on a corpus of 46,000 classes, it could deduce that around 25% of package scoped classes are confined.

In [2] an algorithm to infer ownership types is developed and successfully applied to 408 classes of the Java standard library. However, inferred types often contain too many ownership parameters, so precision needs to be improved.

2 Conclusions

Hierarchic shapes have successfully been used for program visualization and verification, in concurrent programming, garbage collection, and for architectural integrity constraints. Hierarchic shapes come in different flavours, and differ in whether they support change of owner, whether ownership implies restrictions on aliasing (through deep or shallow ownership) or dependence of properties, whether the ownership contexts correspond to objects, classes or packages, whether the number of ownership contexts is statically or dynamically known, whether ownership is checked statically or dynamically, how many annotations are required, and whether inference is supported.

Further work is required to combine the different uses of the shapes, to develop more lightweight yet powerful systems, to develop better inference tools to alleviate the process of annotating programs, to combine shape types with new trends in program development (most notably with aspect oriented programming), and finally to combine the ease of use offered by types with the flexibility offered by full-fledged verification as in separation logic.

References

1. Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating aliasing Policy from Mechanism. In *ECOOP*, 2004.
2. Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, November 2002.
3. David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA*, 2000.
4. Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *JACM*, 2005.
5. Anindya Banerjee and David A. Naumann. State based ownership, reentrance and encapsulation. In *ECOOP*, 2005.
6. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology*, 2004.
7. Mike Barnett and David A. Naumann. Friends Need a Bit More: Maintaining Invariants Over Shared State. In *Mathematics of Program Construction*, 2004.
8. Boris Bokowski and Jan Vitek. Confined Types. In *OOPSLA*, 1999.
9. Chandrasekar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.
10. Chandrasekar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.

11. Chandrasekar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2002.
12. Chandrasekar Boyapati, Alexandru Salcianu, William Beebe, and Martin Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *PLDI*, June 2003.
13. Gustaf Cele and Sebastian Stureborg. Ownership Types in Practice. Technical Report TR-02-02, Stockholm University, 2002.
14. David Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, 2002.
15. David Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.
16. David Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: Deployment-time confinement checking. In *OOPSLA*, 2003.
17. David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, 2003.
18. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. In *OOPSLA*, 2001.
19. Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL*, 2004.
20. Samin Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, 2000.
21. K. Rustan M. Leino and Peter Müller. Object Invariants in Dynamic Contexts. In *ECOOP*, 2004.
22. Peter Müller and Arnd Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In *Programming Languages and Fundamentals of Programming*, 1999.
23. Peter Müller, Arnd Poetzsch-Heffter, and Gary Leavens. Modular Invariants for Layered Object Structures. Technical Report 424, ETH Zürich, 2004. *further development under way*.
24. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular Specification of Frame Properties in JML. *Concurrency and Computation Practice and Experience*, 2003.
25. James Noble. Visualising Objects: Abstraction, Encapsulation, Aliasing and Ownership. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Software Visualisation, State of the Art Survey*, pages 58–72. LNCS 2269, 2002.
26. James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *ECOOP*, 1998.
27. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2004.
28. David Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACS*, 1972.
29. Alex Potanin and James Noble. Checking ownership and confinement properties. In *Formal Techniques for Java-like Programs*, 2002.
30. Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in OO programs. *Commun. ACM*, 2005.
31. Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. In *IWACO*. 2003.
32. Tian Zhao, James Noble, and Jan Vitek. Scoped Types for Real-time Java. In *RTSS*, 2004.