A Deductive Database with Datalog and SQL Query Languages

Fernando Sáenz-Pérez[1], Rafael Caballero[2], and Yolanda García-Ruiz[2,*]

Grupo de Programación Declarativa (GPD)
[1]Dept. Ingeniería del Software e Inteligencia Artificial
[2]Dept. Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Spain

Abstract. This paper introduces Datalog Educational System (DES), a deductive database which supports both Datalog and SQL as query languages. Since its inception, this system is targeted to educational purposes rather to develop an efficient, competitive system with respect to other existing systems. As distinguishing features, it is free, open-source, multiplatform, interactive, portable, GUI-enabled, implemented following ISO-Prolog and supports extensions to pure Datalog in the form of stratified negation, strong constraints, types, metapredicates, and duplicates. Also, test case generation for SQL views and declarative debugging for Datalog programs and SQL views are supported. SQL statements, following ISO standard, are compiled to Datalog programs and solved by its inference engine. Nonetheless, ODBC connections are also supported, which enables access to external DBMSs and benefit from their solving performance, persistency and scalability.

Keywords: Deductive Databases, Relational Databases, Datalog, SQL, DES.

1 Introduction

We have witnessed recently the advent of new interest on deductive databases and emerging companies promoting deductive technologies. Datalog, as a deductive query language, has been extensively studied and is gaining a renowned interest thanks to their application to ontologies [5], semantic web [7], social networks [16], policy languages [2], and even for optimization [9]. In addition, current companies as LogicBlox, Exeura, Semmle, and Lixto embody Datalog-based deductive database technologies in the solutions they develop.

This paper presents the Datalog Educational System (DES), which born from the need to teach deductive concepts to postgraduate students. As by that time there was no open-source, free, multiplatform, and interactive system, we decided to start this project. It was first released in 2004 and since then, many releases have been published including features as:

* These authors have been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502).

- Tabling-based deductive engine implementing stratified negation.
- Datalog and SQL query languages sharing the same database.
- Nulls and outer join operations.
- Duplicates and duplicate elimination.
- Text-based interactive system with commands for changing and examinin its state, logging, batch execution, and many more.
- Source-level tracers and declarative debuggers for both Datalog and SQL.
- Strong constraints including types, primary and foreign keys, functional de pendencies, and user-defined constraints.
- ODBC connections to seamlessly access external databases.

DES has been developed to be used via an interactive command shell. Noneth less, more appealing environments are available. On the one hand, DES has bee plugged to the multi-platform, Java-based IDE ACIDE [17]. It features synta colouring, project management, interactive console with edition and history, co figurable buttons for commands, and more. On the other hand, Markus Trisl contributed with an Emacs environment.

The system is implemented on top of Prolog and it can be run from a state-o the-art Prolog interpreter (currently, last versions of Ciao, GNU Prolog, SW Prolog and SICStus Prolog) on any OS supported by such Prolog interpret (i.e., almost any HW/SW platform). Portable executables (i.e., they do not ne installation and can be run from any directory they are stored) has been al provided for Windows, Linux, and Mac OS X.

Datalog as supported by DES mainly follows Prolog ISO standard [10], whi SQL follows SQL:2008 ISO standard [11]. DES provides several metapredica as well, some of them are included to add support for SQL operations:

- *Negation.* `not(`*`Goal`*`)` computes the negation of ***Goal*** by means of negat as failure (closed world assumption (CWA) [20]). ***Goal*** is located at a hig strata than the predicate it occurs [20]).
- *Aggregates.* `group_by(`*`Relation,Grouping_Variables,Condition`*`)` crea groups from ***Relation*** w.r.t. ***Grouping_Variables*** and compute aggreg data with ***Condition***, which can include expressions with aggregate functi such as `sum(`*`Variable`*`)`, which returns the running sum of ***Relation*** w. ***Variable***. If no grouping is needed or it is left to be done automatically, gregate predicates are also available, as `sum(`*`Relation,Variable,Resul`*
- *Duplicate elimination.* Metapredicate `distinct(`*`Relation`*`)` computes tinct tuples of ***Relation*** when duplicates are enabled (with the comm `/duplicates on`). Also, `distinct(`*`Projecting_Variables,Relation`*`)` forms the same but finding distinct tuples for its first argument, which subset of variables of the second.
- *Outer join operations.* Nulls and null-related operations coming from database community are allowed, as, e.g.: `lj(`L,R,C`)`, which computes left outer join of relations L and R w.r.t. the join condition C [18].

As a running example, we consider a flight database, which can be defined ternatively from either SQL or Datalog:

```
% SQL:
CREATE TABLE flight(origin STRING,destination STRING,duration INT);
INSERT INTO flight VALUES ('Madrid','Paris',90);
INSERT INTO flight VALUES ('Paris','Oslo',100);
INSERT INTO flight VALUES ('Madrid','London',110);
% Datalog:
:-type(flight(origin:string,destination:string,duration:int)).
flight('Madrid','Paris',90).
flight('Paris','Oslo',100).
flight('Madrid','London',110).
```

Each SQL statement above can be interactively introduced at the system prompt or stored in a file and processed with the command `/process` *`FileName`*. The Datalog program can be also stored in a file and consulted with the command `/consult` *`FileName`*. For inserting tuples (or rules, in general) in Datalog, the command `/assert` ***`Rule`*** is provided. Whilst types are mandatory in table definition, in Datalog they are optionally declared with the assertion `:-type(`*`Relation`*`,[`*`ColumnType`*`])`, where its second argument is a list of column names and its types (*`ColumnName`*`:`*`Type`*). Once one of the programs above has been consulted, queries can be submitted from the system prompt, as:

```
DES> SELECT destination FROM flight WHERE origin='Madrid'
answer(flight.destination) -> { answer('London'), answer('Paris') }
DES> flight('Madrid',Destination,Duration)
{ flight('Madrid','London',110), flight('Madrid','Paris',90) }
```

However, while the first query returns the tuples of `flight` projected by the argument `destination`, the second does not. To get a similar output relation, *temporary* Datalog views are provided, which allow defining both the projection of columns and renaming of relations:

```
DES> dest(Destination) :- flight('Madrid',Destination,Duration)
{ dest('London'), dest('Paris') }
```

For the consulted program, a predicate dependency graph (PDG) [20] is built, which relates each predicate in the program with all the predicates used to compute its meaning. From this graph, a stratification [20] is computed, if it exists, so that negation and aggregate predicates are not involved in cycles. For solving a query, a subgraph restricted to the predicates occurring in the query is computed, so that only the meaning of relevant predicates are computed, following a top-down driven approach rather than a bottom-up. Even when a given PDG is non-stratifiable, it is possible that the subgraph for a given query could be as long as this subgraph does not involve such offending cycles.

2 System Architecture

Figure 1 barely depicts the system architecture of DES. Datalog programs are stored in an in-memory Prolog database. Datalog queries are solved by the deductive engine relying on a cache to store results from fixpoint computations. These are computed by using a tabling technique [6], which finds its roots in dynamic programming. The data structure holding these results is called extension table (ET). Displaying the results for a Datalog query amounts to inspect ET for entries matching the query after its solving. In turn, SQL views and tables are stored in two alternative repositories.

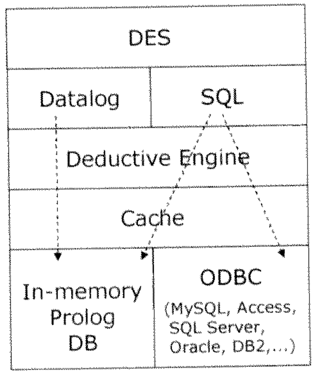

Fig. 1. System Architecture

First alternative for dealing with SQL statements is to use the very same Prolog DB. For this, views are translated into Datalog programs and tables into predicates consisting only of facts. SQL row-returning statements are compiled to and executed as Datalog programs (basics can be found in [20]), and relational metadata for DDL statements are kept. For solving such a query, it is compiled to a Datalog program including the relation `answer`$/n$ with as many arguments as expected from the SQL statement. Then, this program is stored in the Prolog DB, and the Datalog query `answer`$(X_1, \ldots, X_n)$, where $X_i : i \in \{1, \ldots, n\}$ are n fresh variables, is submitted. Results are cached in ET and displayed eventually from this table. After its execution, this Datalog program is removed. On the contrary, if a data definition statement for a view is submitted, its translated program and metadata do persist. This allows Datalog programs to seamlessly use views created at the SQL side (also tables since predicates are used to implement them). The other way round is also possible if types are declared for predicates (further work may include an automatic type assertion via type inferencing). In order to maintain consistency, the cache is cleared whenever the database is updated via asserting Datalog rules, creating SQL views or modifying base tables via `INSERT`, `DELETE` or `UPDATE` SQL statements.

Second alternative is to use the ODBC bridge to access external databases therefore taking advantage from their solving performance, persistency and scalability. Submitting a `CREATE VIEW` SQL statement amounts to forward it to the external database through the ODBC connection. This statement is processe by the external database and operation success is returned to DES, which doe not use the Prolog DB for SQL statements anymore. A SQL row-returning statement is also submitted through the bridge, which returns result tuples that ar cached by DES. Datalog programs and queries can refer to SQL data becaus the bridge provides a view to external data sources as Datalog predicates. How ever, in this second alternative, the other way round is not possible yet, as th external data engine is not aware of the deductive data.

3 Source-to-Source Program Transformations

Asserting a Datalog rule involves several steps. First, parsing builds a syntacti tree for valid rules. If errors are found, an exception is raised with error locatio and source data. Otherwise, a preprocessing stage is performed, consisting of:

- Simplify successive applications of `not(Goal)` to avoid more strata than strictly needed. As well, applications of this predicate to comparison built-ins (as `=`, `<`, ...) are translated to the complemented versions (`\=`, `>=`, ..., resp.) Finally, a compound goal is also allowed and it is defined as the body of a rule for a brand new predicate where its arguments are the relevant variables in *Goal*. The single argument of `not` is replaced by a straight call to this predicate (e.g., the goal `not((p(X),q(X)))` is translated into `not('$p1'(X))` and the rule `'$p1'(X):-p(X),q(X)` is added to the program).
- Aggregate predicates can include compound goals as aggregate relations. These relations are computed before the aggregation itself (sum, average, etc.) following a similar, stratified approach as for computing negation. Therefore, compound goals are also translated as for negation.
- Since disjunctive bodies are allowed, a rule containing a disjunction is transformed to as many rules as needed (e.g., `p(X):-q(X);r(X)` is translated into `p(X):-'$p1'(X)` and the rules `'$p1'(X):-q(X)` and `'$p1'(X):-r(X)` are added to the program).
- Program simplification can be enabled with the command `/simplify on`, which amounts to remove true goals, unify variables, simplify Boolean conditions, and evaluate arithmetic expressions.
- Program transformation for safe rules can also be enabled with the command `/safe on`, which reorder goals in a rule if this rule is unsafe (Section 4).
- Finally, outer joins are translated, first, in order to be solved without resorting to metapredicates as described in [18] and, second, so that relations to be joined are not compound by adding as many predicates as needed (e.g., `lj(rj(p(X),q(Y),X>Y),r(Z),Z>=X)` would be translated into `lj('$p1'(X,Y)),r(Z),Z>=X)` and rule `'$p1'(X,Y):-rj(p(X),q(Y),X>Y)` added to the database).

4 Compile-Time Analyses

DES conducts compile-time analysis to detect unsafe rules. Classic safety [20,22] refers to built-in predicates that can be source of infinite relations, in contrast to user-defined predicates, which are always finite. For instance, the rule `less(X,Y):-X<Y` is unsafe since the built-in `<` can be source of infinite data (its meaning must include any pair such that its first argument is less than its second one). Negation requires its argument to have no unsafe variables, i.e., those which are not bound by a former data provider (as a call to a user-defined predicate). Built-in `X is` *Expr* evaluates the arithmetical expression *Expr*, so that *Expr* is also demanded to be ground and, thus, all its variables must be safe.

Another source of unsafety, departing from the classical notion, resides in metapredicates as `distinct/2` and aggregates. A set variable is any variable occurring in a metapredicate such that it is not bound by the metapredicate. For instance, `Y` in the goal `distinct([X],t(X,Y))` is a set variable, as well as in `group_by(t(X,Y),[X],C=count)`. Because computing a goal follows SLD order, if a set variable is used after the metapredicate, as in `distinct([X],t(X,Y)), p(Y)`, then this is an unsafe goal as in the call to `distinct`, `Y` is not bound, and all tuples in `t/2` are considered for computing its outcome. Swapping both subgoals yields a safe goal. So data providers for set variables are only allowed before their use in such metapredicates.

Along program transformation, unsafe rules can be automatically generated, as in the translations of outer joins. However, they are safe because of their use: unsafe arguments of such rules are always given as input in goals. So, mode information for predicates is handled throughout program compilations to detect truly unsafe rules, avoiding to raise warnings about system generated rules.

The analysis allows deciding whether a rule is safe and, if so, it is transformed by reordering the goals in order to make it computable. An error is raised when a rule or query is actually unsafe (e.g., the rule `p(X):-X<Y` is unsafe because of `Y`), whereas a warning is issued if the rule *might* be safely computed (e.g., `less(1,2)` can be safely computed since its arguments are ground).

5 Strong Constraints

Consistency constraints over data are known as strong integrity constraints in the deductive database area. Examples of such integrity constraints in the relational field are primary keys and foreign keys, to name a few. As well, constraints in deductive systems as DLV [14] or XSB [19] implementing stable model [8] and well-founded model semantics [21], respectively, are otherwise understood as model filters. In these cases, since a database can have several models, only those fulfilling constraints are included in the answer, therefore discarding *unfeasible* models from the answer. In DES, instead, we focus on integrity constraints as understood in the relational field in order to provide a means to detect inconsistent data with respect to user requirements, including types, primary and foreign keys, functional dependencies, and user-defined constraints.

As an example of constraint, in addition to the type constraint in Section 1, let's consider `:-pk(flight,[origin,destination])`, which defines the column pair in the list to be a primary key for `flight`. Also, assuming:

```
connected(O,D,T)  :- flight(O,D,T).
connected(O,D,T)  :- flight(O,A,T1),connected(A,D,T2),T is T1+T2.
```

Then, `:-group_by(connected(O,D,T),[O,D],S=sum(T)),S>=300` is a user constraint limiting the duration from an origin to a destination to be less than 300 minutes. Notice that, as usual in the deductive field and contrary to the norm in SQL, an integrity constraint specifies *unfeasible* values rather than feasible.

6 Tracing and Debugging

In contrast to imperative programming languages, deductive and relational database query languages feature solving procedures which are far from the query languages itself. Whilst one can trace an imperative program by following each statement as it is executed, along with the program state, this is not feasible in declarative (high abstraction) languages as Datalog and SQL.

Similarly, SQL represents a true declarative language which is even farthest from its computation procedure than Prolog. Indeed, the execution plan for a query include transformations considering data statistics to enhance performance. These query plans are composed of primitive relational operations (such as Cartesian product) and specialized operations for which efficient algorithms have been developed, containing in general references to index usage.

Therefore, instead of following a more imperative approach to tracing, here we focus on a (naïve) declarative approach which only take into account the outcomes at some program points. This way, the user can inspect each point and decide whether its outcome is correct or not. This approach will allow examining the syntactical graph of a query, which possibly depends on other views or predicates (SQL or Datalog, resp.) In the case of Datalog queries, this graph contains the nodes and edges in the dependency graph restricted to the query, ignoring other nodes which do not take part in its computation. In the case of SQL, the graph shows the dependencies between a view and its data sources (in the `FROM` clause). Available commands for tracing Datalog and SQL are `/trace_datalog` *`Goal`* and `/trace_sql` *`View`*, respectively.

Algorithmic debugging is also applied to both Datalog and SQL, following [3] and [4], respectively. Similar to how tracing traverses the dependency graph, the debugger in addition prunes paths in the graph by asking the user about validity of its nodes. Available commands for enabling this kind of debugging are `/debug_datalog` *`Goal`* and `/debug_sql` *`View`*.

7 Conclusions

This paper has presented DES, a deductive system used in many universities (`http://des.sourceforge.net/des_facts`) for which its downloading statistics (`http://des.sourceforge.net/statistics`) reveal it as a live project (a new release is expected every two or three months). Statistical numbers show a notable increasing number of downloads, amounting up to more than 1,500 downloads a month, more than 35,000 downloads since 2004.

Features that, as a whole, distinguish DES from other existing systems as DLV [14], XSB [19], bddbddb [13], LDL++ [1], ConceptBase [12], and .QL [15] include null support and outer join operations, duplicates, strong constraints, full-fledged arithmetic, multi-platform, interactiveness, multi-language support, freeness, and open-sourcing, among others. In particular, no one supports outer join operations and full support for duplicates (not only for base relations as LDL++ but also for any rule).

References

1. Arni, F., Ong, K., Tsur, S., Wang, H., Zaniolo, C.: The Deductive Database System LDL++. TPLP 3(1), 61–94 (2003)
2. Becker, M., Fournet, C., Gordon, A.: Design and Semantics of a Decentralized Authorization Language. In: CSF 2007: Proceedings of the 20th IEEE Computer Security Foundations Symposium, pp. 3–15. IEEE, Washington, DC, USA (2007)
3. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Theoretical Framework for the Declarative Debugging of Datalog Programs. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 143–159. Springer, Heidelberg (2008)
4. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Algorithmic Debugging of SQL Views. In: Ershov Informatics Conference (PSI 2011). Springer, Heidelberg (2011)
5. Calì, A., Gottlob, G., Lukasiewicz, T.: Datalog±: a unified approach to ontologies and integrity constraints. In: ICDT 2009: Proceedings of the 12th International Conference on Database Theory, pp. 14–30. ACM, New York (2009)
6. Dietrich, S.W.: Extension tables: Memo relations in logic programming. In: IEEE Symp. on Logic Programming, pp. 264–272 (1987)
7. Fikes, R., Hayes, P.J., Horrocks, I.: OWL-QL - a language for deductive query answering on the Semantic Web. J. Web Sem. 2(1), 19–29 (2004)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080. MIT Press (1988)
9. Greco, S., Trubitsyna, I., Zumpano, E.: NP Datalog: A Logic Language for NP Search and Optimization Queries. In: International Database Engineering and Applications Symposium, pp. 344–353 (2005)
10. ISO/IEC. ISO/IEC 13211-2: Prolog Standard (2000)
11. ISO/IEC. SQL:2008 9075(1-4,9-11,13,14) Standard (2008)
12. Jarke, M., Jeusfeld, M.A., Quix, C. (eds.): ConceptBase V7.1 User Manual. Technical report. RWTH Aachen (April 2008)
13. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Li, C. (ed.) Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 1–12. ACM (2005)
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Tran. on Computational Logic 7(3), 499–562 (2006)
15. Ramalingam, G., Visser, E. (eds.): Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation. ACM (2007)
16. Ronen, R., Shmueli, O.: Evaluating very large Datalog queries on social networks. In: EDBT 2009: Proceedings of the 12th International Conference on Extending Database Technology, pp. 577–587. ACM, New York (2009)
17. Sáenz-Pérez, F.: ACIDE: An Integrated Development Environment Configurable for LaTeX. The PracTeX Journal 3 (2007)
18. Sáenz-Pérez, F.: Outer joins in a deductive database system. In: XI Jornadas sobre Programación y Lenguajes, PROLE, pp. 126–140 (2011)
19. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: SIGMOD 1994: Proc. of the 1994 ACM SIGMOD International Conference on Management of Data, pp. 442–453. ACM, New York (1994)
20. Ullman, J.D.: Database and Knowledge-Base Systems, vol. I (Classical Database Systems) and II (The New Technologies). Computer Science Press (1988)
21. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 619–649 (1991)
22. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann (1997)

Volume Editor

Hongseok Yang
University of Oxford
Department of Computer Science
Parks Road
Oxford OX1 3QD, UK
E-mail: hongseok.yang@cs.ox.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-25317-1 e-ISBN 978-3-642-25318-8
DOI 10.1007/978-3-642-25318-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011941029

CR Subject Classification (1998): D.3, D.2, F.3, D.4, D.1, F.4.1, C.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at APLAS 2011, the 9th Asian Symposium on Programming Languages and Systems held during December 5–7, 2011 in Kenting, Taiwan. The symposium was sponsored by the Asian Association for Foundation of Software (AAFS), Academia Sinica (Taiwan), and National Taiwan University. We are grateful for the administrative support from the Institute of Information Science, Academia Sinica (Taiwan), and the Department of Information Management and the Yen Tjing Ling Industrial Research Institute at National Taiwan University.

APLAS is a premier forum for the discussion of a wide range of topics related to programming languages and systems. Although it is based in Asia, APLAS has been an international forum that serves the worldwide programming language community. The past APLAS symposiums were successfully held in Shanghai (2010), Seoul (2009), Bangalore (2008), Singapore (2007), Sydney (2006), Tsukuba (2005), Taipei (2004) and Beijing (2003) after three informal workshops. Proceedings of the past symposiums were published in Springer Verlag's LNCS series as volumes 6461, 5904, 5356, 4807, 4279, 3780, 3302, and 2895 respectively.

Following the initiative from the previous year, APLAS 2011 solicited submissions in two categories, *regular research papers* and *system and tool presentations*. There were 64 submissions from 22 countries (62 regular research papers and 2 system and tool presentations). Each submission was reviewed by at least 2, and on average 3.2, Program Committee members with the help of external reviewers. The Program Committee meeting was conducted electronically over a period of two weeks in August 2011. The Program Committee decided to accept 22 regular research papers (35%) and 1 system and tool presentation (50%). Among the 22 accepted papers, there was one whose initial verdict was conditional acceptance. To have their paper accepted, the authors were requested to address specific concerns raised by the Program Committee. The revised version of the paper was checked by the Program Committee, before it was finally accepted. I would like to thank all the Program Committee members for their hard work in reviewing papers, participating in online discussions, volunteering to shepherd submissions, and sometimes finding and fixing technical errors in submissions. I also want to thank all the external reviewers for their invaluable contributions.

In addition to contributed papers, this volume contains the full paper or the extended abstracts of of four distinguished invited speakers: Nikolaj Bjørner (Microsoft Research Redmond), Ranjit Jhala (University of California, San Diego), Peter O'Hearn (Queen Mary University of London) and Sriram Rajamani (Microsoft Research India). I would like to thank all of these speakers for accepting our invitation and contributing papers or abstracts.

The General Chair, Tyng-Ruey Chuang, helped me so much from the very beginning while I prepared the technical program of APLAS 2011. I am truly grateful for his support and guidance, and also for making our symposium in Kenting possible and enjoyable. I would like to thank Shin-Cheng Mu for local arrangements, and Yih-Kuen Tsay for serving as the Finance Chair. Noam Rinetzky was an excellent Publicity Chair for APLAS 2011, who had several creative ideas for increasing the awareness of the conference, and Mike Dodds worked very hard to organize the poster session. I greatly appreciate their help and efforts. EasyChair made the handling of submissions and the productions of the proceedings extremely smooth and efficient. Finally, I would like to thank the members of the AAFS Executive Committee for their advice. In particular, Kazunori Ueda, the Program Chair of APLAS 2010, gave me invaluable tips and suggestions, without which I would not have been able to prepare the technical program and the proceedings of APLAS 2011.

September 2011 Hongseok Yang

Organization

Program Committee

Lars Birkedal	IT University of Copenhagen, Denmark
James Brotherston	Imperial College London, UK
Kung Chen	National Chengchi University, Taiwan
Wenguang Chen	Tsinghua University, China
Wei-Ngan Chin	National University of Singapore, Singapore
Javier Esparza	Technische Universität München, Germany
Xinyu Feng	University of Science and Technology of China, China
Jerome Feret	INRIA, France
Matthew Fluet	Rochester Institute of Technology, USA
Rajiv Gupta	Univeristy of California Riverside, USA
Masahito Hasegawa	Kyoto University, Japan
Radha Jagadeesan	DePaul University, USA
Naoki Kobayashi	Tohoku University, Japan
Daniel Kroning	University of Oxford, UK
Rupak Majumdar	Max Planck Institute for Software Systems, Germany
Andrzej Murawski	University of Leicester, UK
Paulo Oliva	Queen Mary University of London, UK
Doron Peled	Bar Ilan University, Israel
Sukyoung Ryu	KAIST, Korea
Sriram Sankaranarayanan	University of Colorado, USA
Armando Solar-Lezama	MIT, USA
Sam Staton	University of Cambridge, UK
Viktor Vafeiadis	Max Planck Institute for Software Systems, Germany
Kapil Vaswani	Microsoft Research, India
Martin Vechev	ETH Zurich, Switzerland
Peng Wu	IBM T.J. Watson Research Center, USA
Hongseok Yang	University of Oxford, UK
Pen-Chung Yew	Academia Sinica, Taiwan

Additional Reviewers

Asada, Kazuyuki	Berger, Ulrich	Charan K, Sai
Bengtson, Jesper	Bocchino, Robert	Charlton, Nathaniel
Berdine, Josh	Camporesi, Ferdinanda	Chen, Yu-Fang
Berger, Martin	Chang, Bor-Yuh Evan	Chu, Duc Hiep

Costea, Andreea
Daniele, Varacca
Debois, Soren
Dimoulas, Christos
Fu, Ming
Gaiser, Andreas
Gherghina, Cristian
Guha, Arjun
Gupta, Ashutosh
Haack, Christian
Haller, Leopold
Hancock, Peter
Hardekopf, Ben
Hoshino, Naohiko
Jiang, Xinyu
Katsumata, Shin-Ya
Kern, Christian
Khoo, Siau-Cheng
Klin, Bartek
Krishnaswami, Neelakantan
Landsberg, David
Lee, Jenq-Kuen
Liang, Hongjin
Lin, Changhui
Longley, John
Maietti, Maria Emilia
Mehnert, Hannes
Mogelberg, Rasmus
Mu, Shin-Cheng
Nakata, Keiko
Nanevski, Aleks
Nordlander, Johan
Petter, Michael
Piérard, Adrien
Rauchwerger, Lawrence
Rival, Xavier
Romanel, Alessandro
Rossberg, Andreas
Sanchez, Alejandro
Sangiorgi, Davide
Schwinghammer, Jan
Schwoon, Stefan
Seghir, Mohamed Nassim
Sharma, Asankhaya
Sobocinski, Pawel
Strassburger, Lutz
Streicher, Thomas
Suenaga, Kohei
Tan, Li
Tautschnig, Michael
Wang, Meng
Wang, Yan
Weng, Shu-Chun
Xu, Na
Xue, Jingling
Yang, Jean

Table of Contents