Rethinking the Architecture of O/R Mapping for EMF in terms of LINQ

Miguel Garcia [1] Rakesh Prithiviraj [2]

Institute for Software Systems (STS)
Hamburg University of Technology (TUHH), 21073 Hamburg, Germany
[1] http://www.sts.tu-harburg.de/~mi.garcia
[2] http://www.tuhh.de/~sorp1428/

Abstract: There is a trend in programming language design toward adopting the same query and concurrency management mechanisms that have proven successful for databases, in the form of integrated query languages and transactional memory [HG06]. We focus on the demands placed on Object/Relational Mapping (ORM) in order to support *comprehension queries*, as known from LINQ and Scala. The additional expressive power enables the application of optimization techniques originating in the field of databases (query unnesting, ordering final instead of intermediate results, etc). As an interim step toward the proposed query integration, we describe a translator from a functional query language (comparable to LINQ) into JPQL, the query language of ORM engines following the JPA standard (JSR-317). A summary of related and ongoing work and an outlook of future work conclude this position paper.

1 Introduction

The capabilities supported by object to relational (O/R) mapping engines continue to evolve, and thus our interest in revisiting the design decisions on which O/R solutions for EMF are based. Two advances in this area are (a) *query shipping*, by which statically-typed queries are fully evaluated by the persistent store (thus bypassing extensive main-memory processing); and (b) more expressive query languages (such as LINQ and XQuery) that adopt a functional-style as opposed to the relational-algebra approach of SQL and its JPA counterpart, JPQL [Jav08]. As an example of (b), the ordering of a resultset in LINQ may be chosen by providing a *comparator function* as parameter, with type-safety being checked by the compiler. At runtime, the Abstract Syntax Tree (AST) implied by the surface syntax is translated into the native query language of the target data store by a component contributed by the DBMS vendor.

Interestingly, the implementation of both query shipping and static typing relies on techniques well supported by EMF: models as first-class citizens, ASTs of Domain Specific Languages (DSLs), and AST-to-AST rewriting. We discuss the design of a framework for EMF making use of such techniques, leaving for future work usability issues like IDE support.

The structure of this paper is as follows. Sec. 2 sums up the aspects of LINQ relevant to our design, highlighting differences with JPQL and also with OCL, the existing object-oriented query language for EMF. The integration of compile-time checking of queries and their runtime translation into JPQL is the topic of Sec. 3 (a LINQ metamodel is developed for this purpose). Finally, the two last sections offer an overview of related work (Sec. 4), and discuss conclusions and areas for future work (Sec. 5). Knowledge is assumed from the reader about database query languages and language metamodeling.

2 Review of the LINQ set of technologies

Underneath the syntactic sugar, the semantic foundation of LINQ is *list comprehensions* [JW07], similar to their counterparts in the purely functional Haskell, the dynamic Python and Ruby, the object-functional Scala, and the mostly-functional XQuery[1]. This semantic foundation is public knowledge, however software patents have been filed for other elements of LINQ[2]. In this section we discuss LINQ from a language engineering perspective, starting with a formalization of comprehensions in Sec. 2.1. Sec. 2.2 reviews the conversion an IDE performs from LINQ textual syntax into the building blocks of queries, the so called *Standard Query Operators*[3]. The resulting ASTs are then type-checked and can be visited.

2.1 Semantic foundation: List comprehensions

In the list comprehension $[e \mid e_1 \ldots e_n]$ each e_i is a qualifier, which can either be a generator of the form $v \leftarrow E$, where v is a variable and E is a sequence-valued expression, or a filter p (a boolean valued predicate). Informally, each generator $v \leftarrow E$ sequentially binds variable v to the items in the sequence denoted by E, making it visible in successive qualifiers. A filter evaluating to $true$ results in successive qualifiers (if any) being evaluated under the current bindings, otherwise 'backtracking' takes place. The *head* expression e is evaluated for those bindings that satisfy all the filters, and taken together these values constitute the resulting sequence. For example, the meaning of the following SQL query:

```
select e(x)
from ( select d(y) from E as y where q(y) ) as x
where p(x)
```

is captured by the comprehension $[\, e(x) \mid x \leftarrow [\, d(y) \mid y \leftarrow E \,,\, q(y) \,] \,,\, p(x) \,]$.

[1] XQuery is mostly-functional because it does allow side-effects (when constructing XML elements, which gives them identity). For example [Teu06, p. 100], the query `let $v := <a/> return $v is $v` (evaluating to `true()`) is not equivalent to the "unfolded" expression `<a/> is <a/>` obtained by replacing all occurrences of `$v` by its binding expression `<a/>`. While the former expression constructs a single element node and binds variable `$v` to it, the latter produces two distinct element instances and, hence, returns `false()` as its result.

[2] Search in `http://www.freepatentsonline.com` for "Microsoft AND LINQ"

[3] Standard Query Operators, `http://msdn.microsoft.com/en-us/library/bb397896.aspx`

In tandem with closures (i.e., functions with parameters bound upon evaluation) the notation allows expressing complex queries, albeit not always in a compact manner (we'll see in a moment how LINQ improves on this). For example [JW07], the SQL query `select dept, sum(salary) from employees group by dept` can be expressed in Haskell as:

```
let depts = asSet [ dept | (name, dept, salary) <- employees ]
in [ ( dept, sum [ salary | (name, dept', salary) <- employees,
                            dept == dept']) 
     | dept <- depts ]
```

As can be seen from the examples, comprehensions contain in general nested queries. If evaluated as-is on large datasets, the engine would spend an excessive amount of time in nested loops, a situation that can be overcome with the optimizations described by Grust and Scholl in [GS96]. Optimizations have also been devised to query datasets fully contained in main-memory [WPN06]. Keeping a query language purely functional allows caching function results (*memoization*): whenever a function is invoked, its input values can be used to look up in a dictionary of recorded invocations. If a match is found, the cached result can be reused, as Yanlei Diao reports for an XQuery subset [Dia04].

Ruby is another example of deeper integration of database queries based on comprehensions, as the projects `ActiveRecord`[4] and `Ambition`[5] show. Deeper integration provides for free capabilities that otherwise require code inspection, e.g. impact analysis upon logical schema changes [MER08].

2.2 From textual query syntax to standard-query-operators syntax

A LINQ query in textual syntax (Figure 1) is a sequence of clauses, of which there are nine kinds: `from`, `join`, `join...into`, `let`, `where`, `orderby`, `group...by`, `select`, and `into`. Having the `from` clauses precede the usages of variables they declare allows the IDE to provide Content Assist. Wes Dyer provides insight into the scoping and translation rules from textual syntax into query operators[6]. For example, the query `from x in foo let y = f(x) select h(x, y, z)` *actually stands for* the following code (after inlining the `let`, making explicit transparent identifiers, etc.): `foo.Select(x => new { x, y = f(x) }).Select(t0 => h(t0.x, t0.y))`. It is possible to directly type queries in "method syntax", as well as to visualize the resulting AST[7].

In fact, not every query in method syntax can be expressed in textual syntax (whose design favors the common cases, or "configuration by exception"). In turn, the standard query operators also capture recurrent cases (if expanded, such queries would look just

[4] ActiveRecord, O/R Mapping put on Rails. `http://ar.rubyonrails.org/`

[5] Ruby's Ambition, `http://ambition.rubyforge.org/`

[6] Translation rules, `http://blogs.msdn.com/wesdyer/archive/2006/12/21/comprehending-comprehensions.aspx`

[7] Visual depiction of LINQ ASTs, `http://www.thinqlinq.com/Default/Enable-the-Expression-Tree-Visualizer-in-VS-2008.aspx`

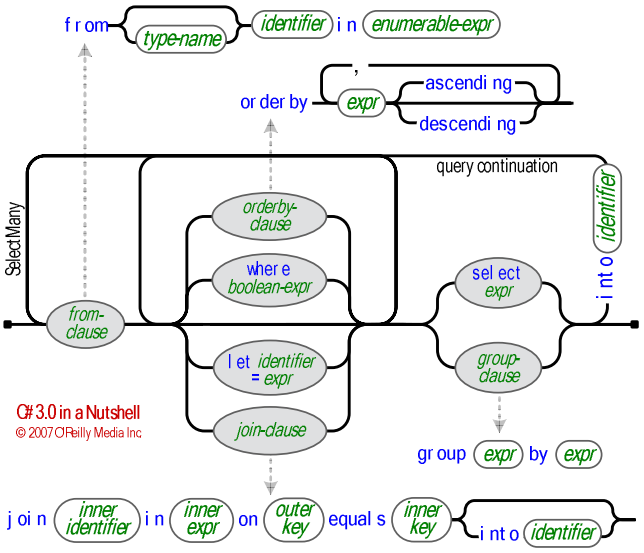

Figure 1: Railroad diagram for the textual syntax of LINQ, reproduced from `http://www.albahari.com/nutshell/linqsyntax.html`

as verbose as the Haskell example above). For example, there are both `OrderBy()` and `OrderByDescending()`, each overloaded as depicted in Listing 1 to optionally accept, besides a closure for key selection, another closure in charge of comparing keys. The first argument stands for the `source` collection, in method syntax such collection appears as receiver of the invocation. Although there are some similarities with OCL (source collections, functional notation) there's also the crucial difference that OCL has no syntax for arbitrary closures.

In the case of collections residing in main-memory, the contract of query methods calls for lazy (on-demand) evaluation. Summing up the technical documentation: "*These methods are implemented by using deferred execution. The immediate return value is an iterator that stores all the information that is required to evaluate the chosen query operator with*

Listing 1: Two sorting operators in LINQ

```
public static IOrderedQueryable<TSource> OrderBy<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector
)

public static IOrderedQueryable<TSource> OrderBy<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> keySelector,
    IComparer<TKey> comparer
)
```

Listing 2: Adding AST nodes to an existing query

```
var query = from ad in carfinderDB.ExpandedAds select ad;
if (minPrice !=null) query = query.Where(ad => ad.Price >= (minPrice));
if (maxPrice != null) query = query.Where(ad => ad.Price <= (maxPrice));
return query.ToList(); // query is composed and executed at this point
```

the given arguments. The query represented by this method is not evaluated (interpreted) until the iterator is enumerated, for example by using foreach". This design guarantees that the minimum amount of work will be performed to obtain the first result, and that some queries on infinite input will be answered: `iterOverAllIntegers.Take(10)` will actually consume only the first 10 items returned by the `iterOverAllIntegers` iterator. The design also enables adding AST nodes to an existing query, e.g. adding conditions on the fly (Listing 2) in a manner resistant to SQL injection attacks.

Admittedly, there are query operators that require consuming the whole input sequence before returning the first result (e.g., `OrderBy()`) while others can stop evaluation as soon as some item is found (e.g., `Any()`, which determines whether an element satisfying a condition exists).

In case collections reside in secondary storage, the semantics are weakened so as to avoid *the big-inhale problem* (i.e., caching the whole input): equality is not defined as reference identity (nor in terms of overridden `equals()`) but as primary key, value equality (several operators rely on equality comparisons, e.g. `Contains()`). When run against an RDBMS, queries operate not on sequences but on multisets: if any operators (`Select()`, `Where()`) are applied after an `OrderBy()` there is no assurance that results will reflect the original order of the source collection. PLINQ, the project focusing on parallel evaluation of LINQ queries, puts it in these terms: "*ordering operators re-establish order, shuffle points shuffle the order*"[8].

3 Query optimization and compilation into JPQL

Before discussing how to compile functional queries into JPQL (Sec. 3.2), it is instructive to see how this process is broken down in the Microsoft implementation (Sec. 3.1). Unlike the situation for JPA engines where the translation JPQL → native SQL is performed by a monolithic component, there are several participants involved in the ADO.NET Entity Framework [Sce09, ABMM07], comprising three phases: (a) standard-query-operators AST → *Entity SQL* [Ent08], (b) *Entity SQL* + mapping definition → an AST consisting of ADO.NET `DbCommand` nodes, and (c) interpretation of the `DbCommand` tree on a particular DBMS. Because of this design, open-source projects announcing an "Entity Framework implementation for XYZ RDBMS" showcase only translation (c), with the C# 3.0 compiler preparing the input for phase (a), and the .NET Entity Framework performing phases (a) and (b) (all of which are closed source).

[8] PLINQ, `http://blogs.msdn.com/pfxteam/archive/2008/06/11/8592301.aspx`

Listing 3: Syntactically different ways to test equality between Strings in LINQ

```
(s1, s2) => s1 == s2;
(s1, s2) => s1.Equals(s2);
(s1, s2) => String.Equals(s1, s2);
(s1, s2) => String.Compare(s1, s2) == 0;
```

3.1 LINQ to native queries, under the hood

Once the C# compiler has converted a LINQ query into method syntax (Sec. 2.2) the next step consists in translating it into the native query language of the target persistent store. This runtime task is delegated to an implementation of a *LINQ provider*, of which there are many. To name a few: (a) LINQ to SQL (targeting SQL Server only and lacking O/R capabilities); (b) LINQ to Entities (targeting any DBMS with an ADO.NET adapter, and supporting O/R mappings); (c) LINQ to XSD [TMB08].

Ideally, every provider should accept any LINQ query and translate it into a "semantically equivalent" (T-SQL, ADO.NET, or XQuery) query. In practice, (a) some standard query operators are not supported by some providers, and (b) the same query, when applied to different data stores with the same data, may evaluate differently. As an example of (b), `sum()` will return zero when evaluated on an array all whose elements are `null`, while the same query on an SQL column containing only `NULL`s will result in `null`. Coming back to the issue of unsupported query operators, each provider documents its own set of gotchas, in the case of LINQ to SQL and LINQ to Entities in [Sho08b] and [Sho08a] resp.

Summing up, while it is true that LINQ queries are statically typed, a LINQ provider may decide not to support all valid LINQ ASTs. For example, the Entity Framework will throw a runtime exception when attempting to translate a query that invokes a user-defined method (even if such method is side-effects-free). As of now, custom LINQ providers are not given a means to extend a "compile-time query validation" extension point to provide IDE-level feedback on these situations.

3.2 Translation into JPQL, under the hood

Usually more than one way exists to indicate the same operation, and a translation should cope with them all. It thus makes sense to normalize the input AST into a canonical form before further processing. For example[9], there are four ways in LINQ to test two Strings for equality, each producing an AST with a different shape (Listing 3).

Also complicating the translation is the fact that JPQL is not *composable*, unlike Entity SQL [Ent08] (*composable* meaning that, whenever a value is expected, a subquery may be used). For example, as per the current Early Draft 1 of JPA 2.0 (JSR 317), sub-

[9] B# .NET Blog, `http://bartdesmet.net/blogs/bart/archive/2008/08/13/expression-tree-normalization-how-many-ways-to-say-string-string.aspx`

queries are restricted to the `WHERE` and `HAVING` clauses only, and not allowed in the `FROM` clause. Moreover, a subquery may not be used in places where a single persistable entity is expected. Additionally, JPQL may allow some predicates only if expressed with certain idioms (e.g., testing whether a subquery result is empty cannot be done with `IS EMPTY` and must be done with `NOT EXISTS` instead).

Given that we are not extending a compiler, the LINQ-like metamodel in our prototype serves the same purpose as the OCL metamodel in the Eclipse MDT implementation of OCL: textual syntax is parsed against an existing Ecore model (to resolve type identifiers to type declarations), and the resulting ASTs are checked for well-formedness before being interpreted or translated. Some usability features can be provided (e.g., *type inference*[10]) while functionality that surfaces in the context of an imperative programming language has no equivalent in our implementation (e.g., *variable capture*[11]).

The new metamodel is kept minimal by relying extensively on that of Ecore. For example, the projection operators `Select()` and `SelectMany()` usually introduce an implicit, unnamed, item type for its result. Using the dynamic capabilities of EMF, such type can be added to the environment for sub-expressions in scope.

Our first prototype LINQ → JPQL translator was based on the design discussed by Matt Warren[12] of a LINQ → T-SQL translator. To make a long story short, our prototype only managed to cover an idiosyncratic subset of LINQ, a situation we attribute to the non-composability of JPQL (subqueries may not show up wherever the values they compute may appear). Instead of building upon that code base, we are following in a second prototype the approach favored by query optimizers like *lambda-DB*[13] (a research OODBMS supporting OQL) and the more recent *Pathfinder*[14] (a purely relational XQuery processor). In a nutshell, the architecture is more demanding because tasks that the ORM engine used to perform for us have to considered too, but in the long term we see no way around if all of LINQ is to be supported. We plan to report on the insights gained from this project.

4 Related Work

The subtopics addressed in this paper (embedding functional queries in Java and O/R mapping engines) have been the focus of several efforts in the programming language and object database communities.

Over time, language designers have repeatedly granted the status of language constructs only to abstractions that previously established themselves as design patterns (garbage

[10] Lambdas and type inference, `http://greenblog.blogsome.com/2008/03/18/c-30-lambdas-and-type-inference/`

[11] Lambdas and variable capture, `http://www.interact-sw.co.uk/iangblog/2008/03/29/linq-range-odd`

[12] LINQ: Building an IQueryable Provider, `http://blogs.msdn.com/mattwar/archive/2008/07/14/linq-building-an-iqueryable-provider-part-xi.aspx`

[13] lambda-DB, `http://lambda.uta.edu/lambda-DB/manual/overview.html`

[14] Pathfinder, `http://www.pathfinder-xquery.org/`

collection, foreach loops, closures[15] in the upcoming Java 7). Before getting there, two possibilities exist to *shoehorn* new abstractions into an existing language: (a) using a preprocessor, and (b) DSL embedding. Both techniques have been applied in Java to support the SQL query language. As an example of (a), Van Wyk [VWKSB07] uses a kind of attribute grammars to embed SQL in Java 1.4. Short of providing IDE-awareness, the resulting extension involves adding: grammar productions, type checking rules, and compilation rewritings (from SQL snippets into the base language). Additions of this kind are more directly supported by extensible compilers [Cle08] than by the JDT.

As for the second approach, DSL embedding, a *type-safe* embedding of SQL in Java 5 is covered by Kabanov [KR08]. In contrast to a full-fledged language extension, no preprocessor is needed but only a library realizing the Builder pattern (Item 2 in *Effective Java*, 2nd ed). While the IDE helps big time with Content Assist, some well-formedness rules (WFRs) of an embedded DSL may not be expressible alone in terms of typing rules of the host language. Still, this approach is attractive because of its automation: an Ecore-based DSL metamodel can be fed as input to `DSL2JDT` [Gar08] to generate the aforementioned Builder. Moreover, additional WFRs (in the form of EMF Validators) can also be specified for the Eclipse JDT to check at compile-time on the ASTs of DSL expressions.

Regarding the impact of functional programming on object persistence, OODBMS vendors are studying the possibility of adopting a functional-style query language, in the context of the upcoming ODMG 4.0 standard[16]. Although object databases have not attained a large market share, the work of the Object Data Management Group (ODMG) has been influential in the design of OO query languages (among others, JPQL). The functional approach to database query languages was pioneered by the Kleisli system [Won00].

5 Conclusions and Future Work

The integration of functional queries in an object-oriented language allows expressing algorithms at a higher level of abstraction, a capability that proves useful in a variety of application domains. For example, nowadays code generation requires coding visitors, performing pattern matching and rewriting, and multi-stage AST transformations, all of which contain special cases of querying. The resulting code could shrink by an order of magnitude if an Ecore-level language along the lines of LINQ were available. Moreover, such language could be taken for granted by authors of domain-specific languages, thus avoiding reinventing the wheel but more importantly avoiding the proliferation of similar capabilities under different names, lowering the *Tower of Babel* barrier when adopting a new DSL.

In order to fully realize the query translation and optimization techniques that functional query languages call for, we see the need for a closer cooperation between the providers of ORM engines and the EMF community. We hope the discussion of design aspects included in this paper helps to advance such roadmap.

[15] Closures in Java 7, official prototype at `http://www.javac.info`

[16] Next Generation ODMG Standard, `http://www.odbms.org/odmg_ng.html`

References

[ABMM07] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *SIGMOD '07: Proc. of the 2007 ACM SIGMOD Intnl Conf on Management of Data*, pages 877–888, New York, NY, USA, 2007. ACM. http://research.microsoft.com/~melnik/pub/adonet-industrial_SIGMOD07.pdf.

[Cle08] Austin Clements. A Comparison of Designs for Extensible and Extension-Oriented Compilers. Master's thesis, Massachusetts Institute of Technology, Feb 2008. http://pdos.csail.mit.edu/xoc/clements-thesis.pdf.

[Dia04] Yanlei Diao. Implementing Memoization in a Streaming XQuery Processor. In *Proc. 2nd Intnl XML Database and XML Technologies Symposium, XSym 2004*, volume 2004 of *LNCS*, pages 35–50, Toronto, Canada, 2004. Springer. http://www.dbis.ethz.ch/research/publications/paper160.pdf.

[Ent08] Entity SQL Reference, 2008. http://msdn.microsoft.com/en-us/library/bb387118.aspx.

[Gar08] Miguel Garcia. Automating the embedding of Domain Specific Languages in Eclipse JDT, 2008. Eclipse Technical Article. http://eclipse.org/articles/article.php?file=Article-AutomatingDSLEmbeddings/index.html.

[GS96] Torsten Grust and Marc H. Scholl. Translating OQL into Monoid Comprehensions—Stuck with Nested Loops? Technical Report 3a/1996, Dept. of Computer and Information Science, Database Research Group, U Konstanz, September 1996.

[HG06] Benjamin Hindman and Dan Grossman. Strong Atomicity for Java Without Virtual-Machine Support. Technical Report 2006-05-01, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, USA, May 2006. http://www.cs.washington.edu/homes/djg/papers/atomjava_tr_may06.pdf.

[Jav08] Java™ Persistence 2.0 Expert Group. JSR 317: Java™ Persistence 2.0. Available at http://jcp.org/en/jsr/detail?id=317, 2008.

[JW07] Simon Peyton Jones and Philip Wadler. Comprehensive Comprehensions. In *Proc. of the ACM SIGPLAN Workshop Haskell '07*, pages 61–72, New York, NY, USA, 2007. ACM Press. http://research.microsoft.com/~simonpj/papers/list-comp/list-comp.pdf.

[KR08] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for Java. In *Proc. of 6th Int. Conf. on Principles and Practice of Programming in Java, PPPJ 2008*, Modena, Italy, Sept. 2008. To appear. http://www.ekabanov.net/kabanov-raudjarv-pppj08.pdf.

[MER08] Andy Maule, Wolfgang Emmerich, and David S. Rosenblum. Impact analysis of database schema changes. In *ICSE '08: Proc. of the 30th Intnl Conf on Software Engineering*, pages 451–460, New York, NY, USA, 2008. ACM. http://www.cs.ucl.ac.uk/staff/w.emmerich/publications/ICSE2008/schemaChangeICSE08/icse08.pdf.

[Sce09] David Sceppa. *Programming the Microsoft ADO.NET Entity Framework*. Number ISBN 978-0735625297. Microsoft Press, 2009. To appear.

[Sho08a] Known Issues and Considerations in LINQ to Entities, 2008. http://msdn.microsoft.com/en-us/library/bb896317.aspx.

[Sho08b] Standard Query Operator Translation (LINQ to SQL), 2008. http://msdn.microsoft.com/en-us/library/bb399342.aspx.

[Teu06] Jens Teubner. *Pathfinder: XQuery Compilation Techniques for Relational Database Targets*. PhD thesis, Technische Universität München, October 2006. http://www-db.in.tum.de/~teubnerj/publications/diss.pdf.

[TMB08] James F. Terwilliger, Sergey Melnik, and Phil A. Bernstein. Language Integrated Querying of XML Data in SQL Server, 2008. Demo in VLDB 2008. To Appear. http://research.microsoft.com/~melnik/pub/xml-mapping-vldb08.pdf.

[VWKSB07] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute Grammar-based Language Extensions for Java. In *European Conference on Object Oriented Programming (ECOOP)*, LNCS. Springer Verlag, July 2007. http://www.umsec.umn.edu/sites/all/files/publications/ext_java.pdf.

[Won00] Limsoon Wong. The functional guts of the Kleisli query system. In *ICFP '00: Proc of the Fifth ACM SIGPLAN Intnl Conf on Functional Programming*, pages 1–10, New York, NY, USA, 2000. ACM. http://www.comp.nus.edu.sg/~wongls/psZ/wls-icfp00-1.ps.

[WPN06] Darren Willis, David J. Pearce, and James Noble. Efficient Object Querying for Java. In Dave Thomas, editor, *ECOOP*, volume 4067 of *LNCS*, pages 28–49. Springer, 2006. http://www.mcs.vuw.ac.nz/~djp/files/WPN_ECOOP06.ps.