

Compiler plugins can handle nested languages: AST-level expansion of LINQ queries for Java

Miguel Garcia

Institute for Software Systems (STS)
Hamburg University of Technology (TUHH), Germany
<http://www.sts.tu-harburg.de/people/mi.garcia>

Abstract. The integration of database and programming languages is made difficult by the different data models and type systems prevalent in each field. Functional-object query languages contribute to bridge this gap by letting software developers write declarative queries without imposing any specific execution strategy. Although some query optimizers support this paradigm, Java provides no means to embed queries in a seamless and typesafe manner. Interestingly, the benefits of such grammar extension (compile-time type inference and checking, user-friendly syntax) can alternatively be achieved with a *compiler plugin* as discussed in this paper for the LINQ query language and two Java compilers (from Sun and Eclipse). A prototype confirms the benefits of the approach by automating at compile-time (a) the parsing of LINQ queries nested in Java, (b) their analysis for well-formedness, and (c) their rewriting into statements to build Abstract Syntax Trees (ASTs). The technique is also applicable to other languages (JPQL, XQuery) which are handled nowadays by a Java compiler as uninterpreted strings, being thus prone to runtime exceptions due to breaches of static semantics.

1 Introduction

The Microsoft project *Language Integrated Query* (LINQ for short) has raised the bar for data access in mainstream programming languages by introducing query-related constructs as first-class citizens. These constructs include relational operations (*e.g.*, projections, selections, joins) as well as the more fundamental functional operations *map*, *filter*, and *flatMap* (which LINQ calls **Select**, **Where**, and **SelectMany**). The underlying semantic foundation, *list comprehensions* [21], makes LINQ amenable to well-known optimizations [4,19] that compute efficient access plans at runtime. Because of this, a Java integration of LINQ does not involve devising new query compilation techniques but applying instead existing scientific knowledge in the context of language and compiler engineering.

This paper addresses just such engineering problem in a portable manner (across Java compilers from different vendors, across different IDEs) by relying on a *compiler plugin* to extend (but not modify) a Java batch compiler. All along, the original syntax of LINQ is supported without extending the Java grammar, and the same error reporting conventions are used as for the host

language. Therefore, our prototype can simply be added to existing toolchains for build automation facilitating real-world adoption. The methodology in question also paves the way for other proofs of concept in the field of OODBs, for example adapting to Java innovative compilation strategies that straddle the database/virtual machine divide [6,23,31].

Our contributions are twofold. First, we give a denotational semantics for LINQ and make explicit the reasoning behind the translation from LINQ into its lower-level, comprehension-style formulation (*Standard Query Operators*, SQO [8]). The LINQ specification glosses over many of the issues involved and lacks a treatment of the confluence of the rewriting process. We cover these aspects, given their importance for a future standardization of LINQ for Java. As second contribution, a technique is presented to realize compile-time program transformations for Java. Unlike other approaches that demand deep knowledge about the internals of a compiler, the proposed technique enables lightweight language embedding as demonstrated for LINQ query expansion.

The structure of this paper is as follows. Background is provided in Sec. 2 on current approaches to language embedding in Java, an issue relevant to the ODBMS, ORM, and RDBMS communities. The syntax and semantics of LINQ are presented in Sec. 3 thus making the paper self-contained. Sec. 4 covers the translation of LINQ into SQO building blocks, as well as aspects of language design (side-effects and variable capture). Adapting our approach to other query languages (XQuery, JPQL) is facilitated by the discussion in Sec. 5 of the implementation of our prototype. Finally, the two last sections offer an overview of related work (Sec. 6) and discuss conclusions and areas for future work (Sec. 7). Knowledge is assumed from the reader about database query languages as well as familiarity with compiler terminology.

A prototype (`LINQExpand4Java`) realizing this approach can be downloaded from <http://www.sts.tu-harburg.de/people/mi.garcia/LINQExpand4Java>

2 Language embedding and Static Semantics

Language extensions, as in the Microsoft implementation of LINQ, require a heavyweight modification of a compiler or the use of a pre-processor. Besides the higher development cost, combining independently developed extensions is impossible, as each front-end rejects all extensions but the one it understands. This explains the renewed interest in *language embedding* for Domain-Specific Languages (DSLs), which fosters an agile approach to language engineering. The original syntax of the host language is kept, while looking for opportunities to express AST building in a visually appealing manner. For example¹:

```
final Sql sql = Select(ARTICLE.NAME, ARTICLE.ARTICLE_NO)
    .from(ARTICLE)
    .where(ARTICLE.OID.in(named("article_oid")))
    .toSql();
```

¹ JEQUEL: SQL embedded in Java, <http://www.jeque1.de/>

The previous example not only resembles SQL but moreover is *typesafe*, *i.e.* the type system of the host language enforces (most) Well-Formedness Rules (WFRs) of the embedded language. Kabanov and Raudj arv [22] provide a comprehensive review of the relevant design patterns (Fluent Interface, Query Builder, reification of the database schema, etc.) Common to all proposals (Native Queries², Criteria API³, etc.) is the limitation that queries are not portable among, say, Java, C#, and Ruby. We reserve the term *nested language* for DSLs that can be reused verbatim across platforms and host languages.

In our context, techniques originally developed for embedded DSLs (EDSLs) are also of interest because *query expansion* takes as input a *nested query* producing statements conforming to an *embedded DSL* (albeit one not intended for direct editing by developers). Thus, the succinctness of nested syntax is combined with the ability to target existing EDSLs.

Both nested and embedded DSLs rely on a facility to import database schema information into the program namespace (*schema awareness*) as an aid to type-checking. This task is made cumbersome by the variety of formats in use today (*i.e.*, the ODMG 3.0 object model [2, Ch. 2] [3], the LINQ Entity Data Model⁴, and the JSR-317 Schema Metamodel [13, p. 12] contained in `persistence.xml`).

Both the nesting and the embedding approach aim at checking at compile time that queries are (a) syntactically correct, (b) well-typed and compliant with the database schema, and (c) robust to cope with *breaking changes* due to schema renamings. Proponents of EDSLs achieve (a) and (b) with library reuse and class generation [22], while (c) is attained by making queries participate in IDE-performed refactorings. However, this IDE functionality cannot be reused for Nested DSLs, their syntax differing radically from that of the host language. Upon breaking schema changes (renamings or others) a compiler plugin signals broken nested queries, for the developer to manually repair. Given our previous work on generators for EDSLs [14] we explored that alternative first, only to realize that encapsulating all well-formedness and schema checks in the compiler plugin (as per the Nested DSL approach) meets all the essential requirements in a modular way, the only shortcomings being the lack of support for refactoring and the fragility resulting from developers tinkering with the generated code.

The code snippet below shows an excerpt of the statements generated by `LINQExpand4Java` before translating into SQO, whose expansion is more verbose.

```
// from entry in contacts select new EmailAddress(entry.name, entry.email)
import static linqtextual.LinqtextualExprBuilder.*; ...
NewExprTraditional newExpr0 = newExprTraditional()
    .fqTypeName("EmailAddress").args(member0, member1).toAST();
SelectClause selClause0 = selectClause().result(newExpr0).toAST();
QueryBody queryBody0 = queryBody().clauses().result(selClause0).toAST();
QueryExpr finalQuery = queryExpr().from(fromClause().var(entry0)
    .inExpr(contacts0).toAST()).body(queryBody0).toAST();
```

² <http://www.db4o.com/about/productinformation/whitepapers>

³ <http://in.relation.to/Bloggers/ATypesafeCriteriaQueryAPIForJPA>

⁴ <http://msdn.microsoft.com/en-us/library/bb387122.aspx>

3 Syntax and Semantics of LINQ

The state of the art of O/R mapping on the platform Microsoft .NET 3.5 is defined by the combination of three technologies: (1) a functional query language, LINQ; (2) a software component (the Entity Framework [1]) in charge of bidirectional, automatically invertible, O/R mapping [25]; and (3) a LINQ-aware Integrated Development Environment (IDE) offering usability features such as syntax completion. The concepts underlying LINQ are however platform-independent and thus our clean-room implementation for Java based on publicly available specifications only.

The textual syntax of LINQ has been designed for readability and not for direct evaluation, which requires a previous translation step. For example, the query `from x in numbers where x>0 select 2*x` actually stands for the following C# code: `numbers.Where(x => x>0).Select(2*x)`. LINQ can query data sources that behave as streams (*i.e.* that support a minimal *open/next/close* iterator interface). In the example, each number `x` is tested (with the predicate given by the lambda expression `x => x>0`) to decide whether to include `2*x` in the result (another stream).

The expressiveness of functional-object query languages [17] calls for optimization techniques to achieve performance competitive with manually tuned “native” queries (in the dialect supported by a particular persistence engine). For read-only queries all the techniques devised to speed up the evaluation of list comprehensions are applicable: deforestation [28, Ch. 7], removal of nested loops [19], join graph isolation [18], and memoization [9], to name a few. Optimizations for the main-memory case have also been devised [4,32,16].

3.1 Syntax

In its simplest form, a LINQ query begins with a `from` clause and ends with either a `select` or `group` clause. In between, zero or more *query body clauses* can be found (`from`, `let`, `where`, `join` or `orderby`). Queries may be nested: the collection over which a `from` variable ranges may itself be a query. A similar effect can be achieved by appending `into variable S2` to a subquery `S1`: with that, `S1` is used as generator for `S2`. The fragment `into variable S2` is called a *query continuation*.

A `join` clause tests for equality the key of an inner-sequence item with that of an outer-sequence item, yielding a pair for each successful match. An `orderby` clause reorders the items of the incoming stream using one or more keys, each with its own sorting direction and comparator function. The ending `select` or `group` clause determines the shape of the result in terms of variables in scope.

The detailed structure of LINQ phrases is captured by the grammar in Table 1 (listing LINQ-proper productions, with *QueryExp* being the entry rule) and in Table 2 (listing other syntactic domains). In order to save space, well-known productions have been omitted (*e.g.*, those for arithmetic expressions). The notation conventions in the grammar follow Turbak and Gifford [29]. Terminals are enumerated (*e.g.* for the syntactic domain *Direction*). Compound syntactic

Table 1. LINQ-related production rules

$$\begin{aligned}
Q \in \text{QueryExp} &::= F_{\text{from}} \quad QB_{\text{qbody}} \\
F \in \text{FromClause} &::= \mathbf{from} \quad T_{\text{type}}^{0..1} \quad V_{\text{var}} \quad \mathbf{in} \quad E_{\text{in}} \\
QB \in \text{QueryBody} &::= B_{\text{qbclauses}}^{0..*} \quad SG_{\text{sel.gby}} \quad QC_{\text{qcont}}^{0..1} \\
B \in \text{BodyClause} &= (\text{FromClause} \cup \text{LetClause} \cup \text{WhereClause} \\
&\quad \cup \text{JoinClause} \cup \text{JoinIntoClause} \cup \text{OrderByClause}) \\
QC \in \text{QueryCont} &::= \mathbf{into} \quad V_{\text{var}} \quad QB_{\text{qbody}} \\
H \in \text{LetClause} &::= \mathbf{let} \quad V_{\text{lhs}} = E_{\text{rhs}} \\
W \in \text{WhereClause} &::= \mathbf{where} \quad E_{\text{booltest}} \\
J \in \text{JoinClause} &::= \mathbf{join} \quad T_{\text{type}}^{0..1} \quad V_{\text{innervar}} \quad \mathbf{in} \quad E_{\text{innerexp}} \\
&\quad \mathbf{on} \quad E_{\text{lhs}} \quad \mathbf{equals} \quad E_{\text{rhs}} \\
K \in \text{JoinIntoClause} &::= J_{\text{jc}} \quad \mathbf{into} \quad V_{\text{result}} \\
O \in \text{OrderByClause} &::= \mathbf{orderby} \quad U_{\text{orderings}}^{1..*} \quad \langle \text{separator}, \rangle \\
U \in \text{Ordering} &::= E_{\text{ord}} \quad \text{Direction}_{\text{dir}} \\
\text{Direction} &\in \quad \{ \mathbf{ascending}, \mathbf{descending} \} \\
S \in \text{SelectClause} &::= \mathbf{select} \quad E_{\text{selexp}} \\
G \in \text{GroupByClause} &::= \mathbf{group} \quad E_{e1} \quad \mathbf{by} \quad E_{e2}
\end{aligned}$$

domains are sets of phrases built out of other phrases. Such domains are annotated with *domain variables*, which are referred from the right-hand-side of productions. References, *e.g.* $QC_{\text{qcont}}^{0..1}$ (which ranges over the *QueryContinuation* domain) are subscripted with a *label* later used to denote particular child nodes in the transformations rules. The superscript of a reference indicates the allowed range of occurrences.

LINQ is mostly implicitly typed: only variables in `from` or `join` clauses may optionally be annotated with type casts. Several ambiguities have to be resolved with arbitrary lookahead (*e.g.* to distinguish between a *JoinClause* and a *JoinIntoClause*) requiring rule priorities or syntactic predicates [27].

3.2 Semantics

The “official” semantics of LINQ is given by translation into query operators [8] whose counterparts in the `Data.List` Haskell library make for a viable denotational semantics [29]. Still, a LINQ-level semantics is useful to determine (a) whether two queries are equivalent, or (b) whether a rewritten SQO formulation is semantically equivalent to the original LINQ query. The denotational

Table 2. Other syntactic domains

$Id, V \in \text{Identifier} = (([a-zA-Z][a-zA-Z0-9]*) - \text{Keyword})$
$SG \in (\text{SelectClause} \cup \text{GroupByClause})$
$E \in \text{Exp} = (\text{QueryExp} \cup \text{ArithExp} \cup \text{BoolExp} \cup \text{UnaryExp}$ $\cup \text{BinaryExp} \cup \text{PrimaryExp} \cup \text{DotSeparated} \cup \dots)$
$EL \in \text{ExpOrLambda} = (\text{Exp} \cup \text{Lambda})$
$P \in \text{PrimaryExp} = (\text{Application} \cup \text{QueryExp} \cup \text{NewExp} \cup \text{PrimitiveLit} \cup \dots)$
$T \in \text{TypeName} ::= Id_{\text{fragments}}^{1..*} \langle \text{separator}; \rangle$
$D \in \text{DotSeparated} ::= P_{\text{pre}} . P_{\text{post}}$
$A \in \text{Application} ::= Id_{\text{head}} \text{Cast}_{\text{cast}}^{0..1} (EL_{\text{args}}^{0..*} \langle \text{separator}; \rangle)$
$L \in \text{Lambda} ::= (Id_{\text{params}}^{0..*} \langle \text{separator}; \rangle) \Rightarrow E_{\text{body}}$

semantics of LINQ given in this section (originally outlined by Wes Dyer⁵) is also necessary to guarantee that each $\text{LINQ} \rightarrow \text{LINQ}$ simplification step (Sec. 4) is semantics preserving. Determining the equivalence of two arbitrary queries is in general undecidable [26, Ch. 8], but the proof is simpler for a transformation affecting a sub-expression in a compositional manner. Automated proof of query equivalence for a functional-object language (Entity SQL) is addressed by Mehra *et al.* [24].

The semantic foundation of LINQ, list comprehensions, is summarized next.

In the list comprehension $[e \mid e_1 \dots e_n]$ 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 filters, and taken together these values constitute the resulting sequence. A *let* expression in a comprehension provides local bindings visible in successive qualifiers (generators, filters, **let** expressions) as well as in the head of the comprehension. For example [21], the SQL query `select dept, sum(salary) from employees group by dept` is expressed in Haskell as:

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

⁵ <http://blogs.msdn.com/wesdyer/archive/2006/12/26/a-model-for-query-interpretation.aspx>

The denotational semantics of LINQ gives meaning to a query in terms of its syntax components. An auxiliary definition and two kinds of valuation functions are needed. A *binding-set* $\mathcal{B} \equiv \{v_1 \mapsto t_1, \dots\}$ is a finite map from non-duplicate variables v_i to values t_i . We write $v_i \mapsto t_i$ as a shorthand for the pair (v_i, t_i) . LINQ forbids declaring a variable whose name would hide another, so a non-ordered map is enough. As usual, an expression E can be evaluated *in the context of* \mathcal{B} by induction on its syntactic structure, with a non-defining occurrence of variable v evaluating to its image t under \mathcal{B} .

The kinds of valuation functions are: (1) $\llbracket Q \rrbracket_{envs}$ denotes the sequence of binding-sets generated by Q (a query body) given the *incoming* sequence of binding-sets $envs$; while (2) $\llbracket E \rrbracket(env)$ denotes the evaluation of E in the context of the single binding-set env . To simplify the formulation of the valuation functions, a query is regarded as a sequence S of body clauses Q , resulting from having desugared query continuations into subqueries (Sec. 4).

The valuation $\llbracket Q \rrbracket_{envs}$ denotes simply the (sub-)query results when Q is a *SelectClause* or a *GroupByClause*:

$$\llbracket \text{select } E_{selexp} \rrbracket_{envs} \stackrel{\text{def}}{=} [\llbracket selexp \rrbracket(env) \mid env \leftarrow envs] \quad (1)$$

Informally speaking, *group result by key* returns a *Grouping*, *i.e.* a finite *ordered* map with entries $key \mapsto cluster$, a cluster being a sequence of results. The valuation of *GroupByClause* involves a left-fold, taking an empty grouping as initial value and progressively adding the valuation of *result* to the cluster given by the valuation of *key*. Using Haskell,

$$\llbracket \text{group } E_{result} \text{ by } E_{key} \rrbracket_{envs} \stackrel{\text{def}}{=} \text{foldl cf [] } envs \quad (2)$$

where *cf*, the combining function, captures the provided result selector and key extractor, has type $Grouping \rightarrow BindingSet \rightarrow Grouping$, and is defined as:

```
cf g bs = let r = result (env) in
           let k = key (env) in
           if hasKey g k then appendToCluster g k r
           else append g [(k, r)]
```

For Q other than *select* or *groupby*, $\llbracket Q \rrbracket_{envs}$ denotes a sequence of binding-sets which constitute the $envs$ in effect for the next clause in S , the first Q in S being evaluated with an empty incoming $envs$.

$$\llbracket \text{from } V_{var} \text{ in } E_{srcSeq} \rrbracket_{envs} \stackrel{\text{def}}{=} [env' \mid env \leftarrow envs, item \leftarrow \llbracket srcSeq \rrbracket(env), \\ \text{let } env' = env \cup \{var \mapsto item\}] \quad (3)$$

$$\llbracket \text{let } V_{var} = E_{exp} \rrbracket_{envs} \stackrel{\text{def}}{=} [env' \mid env \leftarrow envs, \\ \text{let } env' = env \cup \{var \mapsto \llbracket exp \rrbracket(env)\}] \quad (4)$$

$$\llbracket \text{where } E_{test} \rrbracket_{envs} \stackrel{\text{def}}{=} [env \mid env \leftarrow envs, \llbracket test \rrbracket(env)] \quad (5)$$

The valuation of an *OrderByClause* permutes the incoming binding-sets, sorting the sequence $envs$ according to the multi-key given by expressions key_i and sort directions dir_i . In terms of the Haskell function `Data.List.sortBy`,

$$\llbracket \text{orderby } key_1 \text{ dir}_1 \dots key_n \text{ dir}_n \rrbracket envs \stackrel{\text{def}}{=} \text{sortBy comp envs} \quad (6)$$

where `comp` is a comparison function (specific to the given key_i and dir_i , $i = 1 \dots n$) between two binding-sets `bsA` and `bsB`, returning one of `GT`, `EQ`, `LT`. First, $\llbracket key_1 \rrbracket(bsA)$ and $\llbracket key_1 \rrbracket(bsB)$ are compared taking dir_1 into account. If they are not equal that's the outcome of `comp bsA bsB`. Otherwise, $\llbracket key_2 \rrbracket(bsA)$ and $\llbracket key_2 \rrbracket(bsB)$ are compared taking dir_2 into account, and so on. If no `GT` or `LT` is found for $i = 1 \dots n$, `EQ` is returned.

The semantics is defined over a core syntax where explicit type annotations have been desugared into type casts (in `from` and `join` clauses) as discussed in Sec. 4.

$$\begin{aligned} & \llbracket \text{join } V_{innerVar} \text{ in } E_{isrc} \text{ on } E_{outerKey} \text{ equals } E_{innerKey} \rrbracket envs \\ & \stackrel{\text{def}}{=} [ienv \mid env \leftarrow envs, innerItem \leftarrow \llbracket isrc \rrbracket(env), \\ & \quad \text{let } ienv = env \cup \{ innerVar \mapsto innerItem \}, \\ & \quad \llbracket outerKey \rrbracket(env) = \llbracket innerKey \rrbracket(ienv)] \end{aligned} \quad (7)$$

$$\begin{aligned} & \llbracket \text{join } V_{innerVar} \text{ in } E_{isrc} \text{ on } E_{outerKey} \text{ equals } E_{innerKey} \text{ into } V_{resVar} \rrbracket envs \\ & \stackrel{\text{def}}{=} [renv \mid env \leftarrow envs, \\ & \quad \text{let } group = [innerItem \mid innerItem \leftarrow \llbracket isrc \rrbracket(env) \\ & \quad \quad \text{let } ienv = env \cup \{ innerVar \mapsto innerItem \}, \\ & \quad \quad \llbracket outerKey \rrbracket(env) = \llbracket innerKey \rrbracket(ienv)], \\ & \quad \text{let } renv = env \cup \{ resVar \mapsto group \}] \end{aligned} \quad (8)$$

4 Rewriting from LINQ to Query Operators

The translation *LINQ textual syntax* \rightarrow *Standard Query Operators* [7, §7.15.2] is defined in terms of 18 simpler structural transformations. By structural it is meant that they recursively traverse an input AST leaving most nodes unchanged. Surprisingly, the C# specification does not label each transformation with a unique tag, so Table 3 cross-references them by listing a brief description for each rule as well as the section in [7] where it is covered.

In this section a notation is put forward to specify $LINQ \rightarrow SQO$ using LINQ itself (Sec. 4.1) and a precise formulation of rewriting order is given (Sec. 4.2). In order to be useful, a set of rewriting rules should be (a) *confluent* (i.e. rule application terminates in a finite number of steps), (b) *deterministic* (i.e. for any input AST just one output AST exists), and (c) free from multiple applicable rules during any intermediate step. Moreover, the rewrite rules should be (d) *semantics preserving*. In Sec. 4.3 we explain how the transformation rules fare with regard to these properties. Sec. 4.4 reviews the consequences (as for the semantics of Java with embedded LINQ) of our design choice to favor the query-shipping paradigm.

Table 3. Catalog of structural transformations in the $LINQ \rightarrow SQO$ translation

ID	Phase	Description	§ in C# spec.
T1	1	Inline query continuation	7.15.2.1
T2	2	<i>FromClause</i>	type annotation
T3		<i>JoinClause</i>	
T4		<i>JoinIntoClause</i>	
T5	3	Identity query	7.15.2.3
T6	4	<i>FromClause FromClause</i>	<i>SelectClause</i>
T7			otherwise
T8		<i>FromClause JoinClause</i>	<i>SelectClause</i>
T9			otherwise
T10		<i>FromClause JoinIntoClause</i>	<i>SelectClause</i>
T11			otherwise
T12			<i>OrderByClause</i>
T13			<i>WhereClause</i>
T14		<i>LetClause</i>	
T15		non-identity <i>SelectClause</i>	7.15.2.5
T16	<i>FromClause</i>	identity <i>SelectClause</i>	
T17		non-identity <i>GroupByClause</i>	7.15.2.6
T18		identity <i>GroupByClause</i>	

4.1 Notation

To clarify notation, Table 4 lists for transformation T1 (a) its informal formulation (from [7, §7.15.2]); (b) its applicability condition, in terms of the productions in Table 1; and (c) its functional definition, using LINQ itself. For completeness, Figure 1 depicts the parse tree of the resulting query.

In contrast to the cursory presentation in [7], the notation in Table 3 is precise (facilitating conformance across different implementations) and seems therefore well suited as an ingredient for a JSR standardizing LINQ for Java. As another advantage, this notation is closer to the logic formalism of model-checkers, which can be used to validate the translation algorithm (*i.e.*, to certify that the resulting ASTs are well-formed, for all valid input ASTs [15]) and to test the algorithm’s implementation against an *oracle* (*i.e.*, to corroborate whether an implementation produces for some given input the result expected by the declarative specification). Also based on that specification, a model-checker can generate input datasets for tests, achieving larger coverage than manual testing.

4.2 Phases

Transformations T1 - T18 are not applied all at once but in phases. The first phase comprises just T1 (“inline query continuations”), recursively rewriting subqueries to eliminate this syntax shorthand.

After the second phase (T2 - T4) all explicit type annotations have been reformulated in terms of *Application* and *Cast* productions (shown in Table 2).

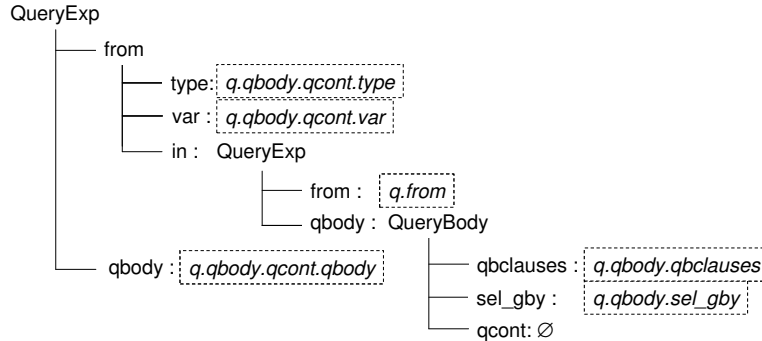


Fig. 1. Template of parse trees resulting from applying T1

The third phase rewrites identity queries (of the form `from x in srcSeq select x`) into `srcSeq.Select(x => x)`. In main-memory evaluation, returning items as in the first query would expose the object identities of the base data. In contrast, the second formulation allows an overridden `Select` to return clones of the items, clones that can later be mutated at will. In the query shipping scenario both formulations denote the same result.

The fourth phase iteratively applies the bulk of the transformations (T6 - T18). In each iteration, T6 to T18 are tried *in that order*, innermost queries first. After successful application of a rule, the next iteration tries again from T6 onwards.

4.3 Confluence and determinism

Regarding confluence and determinism, each of phase 1 and 3 involves applying only one rewriting whereby a construct is consumed (*i.e.* not copied to the output). Phase 2 involves three rules, whose applicability conditions are disjoint (each rule matching *one* of the constructs `FromClause`, `JoinClause`, and `Join-IntoClause`). While the same kind of clause will be copied to the output, it will not match again as the explicit cast has been removed. As can be seen, the first two phases are confluent and lead to a deterministic result for finite input.

Following a similar reasoning, Phase 3 can be shown to be well-behaved. And due to the fact that rewrites reduce syntax shorthands, there is no doubt that the output queries are semantically equivalent to the original queries.

The analysis of Phase 4 must consider more subcases. To recap, the shape of the input to this phase is: (a) because of T1, a legal query expression can now only end in either a `select` or a `groupby` clause, and (b) because of none of Phases 1 to 3 touches them, the initial mandatory `from` clause(s) are still there.

The translation sidesteps the issue of non-unique rule applicability by forcing the order T6-T18 on rewriting. A visual inspection of columns 3 and 4 in Table 3

Table 4. Definition of transformation T1

(a)	<p>T1: inline query continuation</p> <pre> from x1 in e1 ... into x2 ... → from x2 in (from x1 in e1 ...) ... </pre>
(b)	<pre> function T1 (q : QueryExp) : QueryExp when q.qbody.qcont ≠ ∅ </pre>
(c)	<pre> new QueryExp { from = new FromClause { type = q.qbody.qcont.type, var = q.qbody.qcont.var, in = new QueryExp { from = q.from qbody = new QueryBody { qbclauses = q.qbody.qbclauses, sel_gby = q.qbody.sel_gby } } } qbody = q.qbody.qcont.qbody } </pre>

reveals that the subcases partition the set of valid queries. To prove confluence, it suffices to show that each rule strictly diminishes a progress measure. This measure is the number of clauses in the query.

1. The rules that bring an outermost query into its final SQO form are T6, T8, T10, and T15 to T18. They have in common that the input query ends in a *GroupByClause* (T17 and T18) or a *SelectClause* (the rest).
2. In contrast, T7, T9, and T11 to T14 result in a non-SQO query, but diminish the number of clauses by one: T7 consumes a *FromClause*, T9 a *JoinClause* without *into*, and T11 a *JoinIntoClause*.
3. Finally, T12, T13, and T14 consume one *orderby*, *where*, and *let* clause resp.

The clauses that each of T7, T9, T11, and T14 consume happen to declare range variables. In order to avoid those usages becoming dangling in the output query, they are prepended with a prefix (a so called *transparent identifier*) to access hidden identifiers, as follows. The input to any of these rules starts with two clauses (the first of them a *FromClause*) that introduce two variables x_1 and x_2 . In all cases, those two clauses are reduced to a single *from prefix in srcSeq*, where *prefix* is a fresh name and *srcSeq* denotes a sequence of pairs (x_1, x_2) , *i.e.* the labels for the tuple components match the old variable identifiers. Pairs (x_1, x_2) result from instantiating an *anonymous type*, as done with the C# code `new { x1 = Ex1, x2 = Ex2 }`. With this name choice, usages of x_1 can be rewritten to *prefix.x₁* (similarly for x_2) thus making the output query well-formed.

The analysis of semantic equivalence for T6-T18 remains as future work, on the basis of the denotational semantics introduced in Sec. 3.2 for LINQ.

4.4 Language integration aspects

In the Microsoft implementation, the behavior of an SQO method chain is determined by the receiver object at its head. Depending on the runtime type of the receiver the following will happen:

- for an in-memory collection (an instance of `IEnumerable`) an iterator is configured to reel off results as they are found when traversing the object graph on the heap. The iterator idiom simulates lazy evaluation from functional languages, and the applicable optimizations are only those that the C# compiler and the virtual machine (VM) know about.
- for an `IQueryable`, an AST is built for shipping to a *LINQ provider*, which mediates the interaction with a DBMS engine. Any or both of provider and engine may optimize the AST.

In our prototype we focus only on the query shipping case, which implies:

- When evaluating queries on the heap (*LINQ to Objects*) C# does not restrict query constituents in any way: calls to `Thread.Sleep()` may appear, custom comparators may be given as arguments to sorting operators (in the spirit of `java.util.Comparator`); *i.e.* constructs may be used which in general cannot be translated into a DBMS native language. Such translation could be successfully performed *in some cases* with the help of bytecode inspection and rewriting techniques [5], but we find the supported LINQ textual syntax to be expressive enough for all practical purposes. In our implementation, constructs that cannot be translated are rejected.
- The LINQ grammar includes a production invoking *expression*, the most general syntactic category for expressions in C# [7, §B.2.4]. This defeats any hope of faithfully supporting 100% of the LINQ grammar (short of reimplementing the C# compiler). As done in the N.Linq open-source project⁶ (where LINQ capabilities are back-ported to previous versions of .NET) our grammar covers as large a subset of *expression* as practical.

Favoring the query shipping scenario results in two other behaviors of *LINQ to Objects* not being exhibited by our translation: (a) side-effects, and (b) variable capture. We argue in what follows that these behaviors are more a consequence of VM-semantics than desirable properties of a database query language.

Side-effects are possible in *LINQ to Objects*, *e.g.* `index++` in the query:

```
int index = 0;
List<Customer> top10 = (from c in customers
                      where index++ < 10 select c).ToList();
```

Besides rendering most optimizations useless, the stateful `index++` makes the query prone to race conditions, in case `index` can be accessed from other threads.

Variable capture occurs when a lambda expression refers to a variable in scope not hidden by a parameter. For example⁷, the following C# code prints 10 five times and not 0, 2, 4, 6, 8:

⁶ <http://www.codeplex.com/nlinq>

⁷ <http://lorgonblog.spaces.live.com/Blog/cns!701679AD17B6D310!689.entry>

```
List<Func<int>> actions = new List<Func<int>>();
for (int i = 0; i < 5; ++i) { actions.Add( () => i * 2 ); }
foreach (var act in actions) { Console.WriteLine( act() ); }
```

because all five instances of the `Func` objects created by the lambda capture a reference to the same mutable variable instance `i`. McNamara goes on to say,

[As in] every language that has both mutable variables and closures ... the lambda captures the mutable variable now, but gets evaluated later, after further mutations may have occurred. This is an instance of how “lazy evaluation” and “side effects” don’t always mix nicely.

In the query shipping scenario, no DBMS can callback the client VM to retrieve the then-current value of a captured variable. In our implementation, such variable references are re-formulated as parameter passing (and thus evaluated once, just before query shipping).

5 Rewriting of compiler-level trees

A Java compiler operates in phases, progressively decorating Abstract Syntax Trees (ASTs) and populating symbol tables with information needed for successive analyses. JSR-269 (Pluggable Annotation Processing) allows third-parties to provide *compiler plugins* to interact with any Java compiler during the early phases of compilation, for example to *rewrite* an AST. After rewriting an AST in phase N , a compiler plugin may (a) reconstruct the state that previous phases would have computed up to that point; or (b) pretty-print the updated AST and launch the compilation task anew. Depending on the amount of information required from surrounding nodes (the *program context*) transformations range from *desugaring* (performed based on the contents of a subtree alone) to *whole-program* (requiring knowledge of several compilation units).

The phases of the OpenJDK `javac` (Figure 2) are representative of those in other compilers. Rather than describe each phase in detail (as done in [11]) we review first the whole process, focusing afterwards on the contract between the *Annotation Processing* and the *Analyze and Generate* phases where our plugin gets activated. During the first phase (*Parse and Enter*), externally-visible information about each compilation unit is entered into symbol tables. Next, *Annotation processing* calls one or more annotation processors which may generate new source or class files, causing a compilation restart until no new files are created. The last phase, *Analyze and Generate* encapsulates several complex stages: (1) *Attribute* includes type checking and constant folding. Additionally, names, expressions and other elements in the AST are resolved to their corresponding type and symbol nodes. (2) *Flow* checks for definite assignment to variables and for unreachable statements, based on a class-level dataflow analysis. (3) *Generics erasure* is followed by (4) *Desugar* (e.g., simplification of nested and inner classes into normal ones, expansion of “foreach”); concluding with (5) *Generate*.

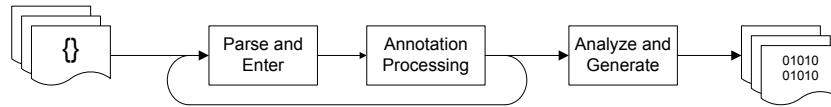


Fig. 2. Compilation workflow realized by `javax.tools.JavaCompiler`

5.1 Prototype

Annotation processors, standardized for Java 6 in JSR-269, access functionality from packages `javax.annotation.processing.*` and `javax.lang.model.*`. A custom processor (a subclass of `AbstractProcessor`) may declare interest in handling all compilation units, be they annotated or not⁸. With that, a method overriding `AbstractProcessor.process()` may inspect a set of `TypeElement`, for example to look up enclosed `ExecutableElement` elements, which stand for methods, constructors, or initializers of a class or interface. Still, no access is provided to the subtrees of their associated statement blocks, as necessary for detecting LINQ queries nested as strings. At least not through that interface, but in practice such navigation is possible for both the Sun and Eclipse compilers by downcasting an `elem` AST-node reference to its specific type in the compiler tree API, along the lines of:

```
if (elem.getKind() == ElementKind.CLASS) {
    String cName = elem.getClass().getName();
    if (cName.startsWith("org.eclipse.jdt.internal.compiler")) { ...
    } else if (cName.startsWith("com.sun.tools.javac.tree")) { ...
```

A factory provides at startup compiler-specific visitors (based on the runtime type of the obtained `javax.tools.JavaCompiler`). These visitors are later used to detect LINQ queries, expand them, and instantiate AST nodes. The first and last visitors admittedly require porting to each supported compiler, however the bulk of the plugin functionality (LINQ \rightarrow SQO) is compiler-independent. Moreover, the AST hierarchies between compilers exhibit variations mostly in naming conventions, while their organization always reflects the Java Language Specification (*e.g.*, `JCMethodDecl` vs. `MethodDeclaration`). Because of this, the porting effort is kept to a minimum.

Instead of letting compilation proceed on the rewritten AST, the compilation unit may be pretty-printed into a textual file (option `-printsource` in `javac`). This arrangement is convenient in conjunction with an IDE, where the project builder internal to the IDE picks up the output file as part of the build process. In this case, generated code can be inspected in read-only mode.

Our LINQ query expander gets activated immediately after Java ASTs have been built and performs a desugaring, without restarting the compilation task. Each statement block of the form `{JL.expand("<LINQ query>");}` is replaced with

⁸ `@SupportedAnnotationTypes("*")`

another containing invocations to SQO factory methods. In a typical arrangement, a method is declared for each LINQ query, whose expansion will constitute the method body, with method arguments corresponding to query parameters. As to the return type, the developer-provided dummy placeholder (*e.g.*, `Object`) is overwritten with the specific type computed for the query result.

5.2 Static semantics and schema awareness

Once expanded, the resulting queries are well-formed (*i.e.*, guaranteed not to cause syntax errors when shipped to a persistence engine) and immune to *injection attacks* (that is, for LINQ queries provided by the developer at compile-time). These guarantees result from (a) the checks performed by the compiler plugin, (b) the wrapping of query parameters as arguments to strongly-typed factory methods; and (c) the checks performed later during the *Attribute* phase of the compilation task.

Regarding the *runtime* assembly of LINQ queries immune to injection attacks, that capability can be achieved by directly building an AST using factory methods, rather than by concatenating strings⁹.

Our design decision does not prevent others from adapting our compiler plugin to target other query EDSLs or different DBMS drivers. Rather, it simplifies that customization by relieving them from performing the aforementioned well-formedness checks at the level of generated EDSL statements. Irrespective of the compile-time analyses performed, still there will be some queries that generate exceptions upon evaluation (*e.g.*, division by zero, or attempting to obtain the `single()` element of a non-singleton sequence). These queries would also have raised exceptions in the Microsoft implementation of LINQ.

Incidentally, it is not necessary for the generated code to serve simultaneously the purposes of (a) checking well-formedness, and (b) building the AST of its translation. As an extreme example, our compiler plugin might have generated code to null out at runtime the AST just built, only to send the original query string to a LINQ provider (*à la* JDBC, JPQL, etc.) Unlike its JDBC counterpart, such plugin would still have guaranteed well-formedness.

6 Related Work

Several grammar-centered approaches promote language extensibility, with case studies reporting typesafe nesting of SQL based on an *Attribute Grammar* formalism as implemented by the `Silver` system [33]. Similar extensions are reported for the `JastAdd` system [10]. These systems act as frontend processors, performing AST-to-AST transformations that are later fed to a standard Java compiler. As our prototype shows, non-trivial transformations can be achieved without the extra machinery offered by these systems. If needed, a compiler plugin can also perform elaborate rewriting, with the `MatchO` library [30].

⁹ http://blogs.msdn.com/swiss_dpe_team/archive/2008/06/05/composable-linq-to-sql-query-with-dynamic-orderby.aspx

The closest to a grammar-centered approach being addressed by Java compiler vendors is the *OpenJDK Compiler Grammar Project*¹⁰, an experimental version of the `javac` compiler based on a grammar written in ANTLR v3. Modifications to `javac` were ruled out by our design objectives. This decision will be revisited in case the Compiler Grammar Project graduates to release status.

Ideally, a standard API should be available to process ASTs (as intended by JSR-198), an idea that has not gained support among compiler and IDE vendors. AST manipulation is supported in Java by IDE-specific frameworks, *e.g.* Jackpot for NetBeans¹¹ and LTK [12] for Eclipse.

Embedded DSLs leverage the compiler to perform *impact analysis* upon changes to the object-oriented schema, as such code does not type-check when the classes involved in queries have been refactored. An early example of a tool reifying the schema is *Safe Query Objects* [5].

DSLs which do not assume an object model (*e.g.*, XQuery) may have well-formedness constraints that cannot be captured by the Java 5 type system. Such DSLs can still be embedded in a typesafe manner, given that the *Checker Framework*¹² of JSR-308 supports the implementation of custom static analyses in compiler plugins. Besides its current use cases (*e.g.* enforcing non-null references) JSR-308 can thus play a key role in improving language embedding.

7 Conclusions and Future Work

Rather than shoehorning a query language to stay within the confines of Java syntax (which raises new problems, *e.g.* extracting declarative queries from imperative code [31]) we have attempted instead to balance the desire for dedicated syntax with the realities of standard compiler infrastructure.

The work reported in this paper is part of a larger project on language engineering, aiming at efficiently supporting in one of the traditionally distinct execution environments (Virtual Machine, Database Manager) useful abstractions originating in the other. While functional queries were originally applied to main-memory object populations, their adoption in the DBMS setting is gaining momentum. In the other direction, ACID transactions are finding a new home in virtual machines with *Software Transactional Memory* [20].

Finally, we believe that while individual improvements (as addressed in this paper for nested query languages) contribute to advancing the case for Object Database Management Systems (ODBMSs), it is still necessary not to lose sight of a more encompassing research and engineering agenda if ODBMSs are to become a strategic technology for software development.

Acknowledgement. Kaichuan Wen proficiently contributed to the implementation of the prototype as part of his course project on query language translation.

¹⁰ <http://openjdk.java.net/projects/compiler-grammar/>

¹¹ A framework for Java source code reengineering, <http://jackpot.netbeans.org/>

¹² <http://groups.csail.mit.edu/pag/jsr308/>

References

1. 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 Intl Conf on Mgmt of Data*, pages 877–888, New York, NY, USA, 2007. ACM.
2. Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The object data standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
3. Gavin Bierman and Agathoniki Trigoni. Towards a formal type system for ODMG OQL. Technical Report 497, University of Cambridge Computer Laboratory, Sept. 2000. <http://research.microsoft.com/~gmb/papers/tr497.pdf>.
4. Nicolas Bruno and Pablo Castro. Towards declarative queries on adaptive data structures. In *ICDE*, pages 1249–1258. IEEE, 2008.
5. William R. Cook and Siddhartha Rai. Safe Query Objects: statically typed objects as remotely executable queries. In *ICSE '05: Proc. of the 27th Intl Conf. on Software Engineering*, pages 97–106, New York, NY, USA, 2005. ACM.
6. Ezra Cooper, Sam Lindley, Phil Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, pages 266–296. Springer, 2006.
7. Microsoft Corporation. C# version 3.0 language specification, 2007. <http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>.
8. Microsoft Corporation. Standard Query Operators Overview, 2009. <http://msdn.microsoft.com/en-us/library/bb397896.aspx>.
9. Yanlei Diao. Implementing memoization in a streaming XQuery processor. In *Proc. 2nd Intl XML Database and XML Technologies Symposium, XSym 2004*, volume 2004 of *LNCS*, pages 35–50, Toronto, Canada, 2004. Springer.
10. Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA '07: Proc. of the 22nd ACM SIGPLAN Conf. on Object Oriented Programming Systems and Applications*, pages 1–18, New York, NY, USA, 2007. ACM.
11. David Erni and Adrian Kuhn. The Hacker’s Guide to javac. Technical report, Software Composition Group (SCG), University of Bern, Switzerland, August 2008. <http://www.iam.unibe.ch/~scg/Archive/Projects/Erni08b.pdf>.
12. Leif Frenzel. LTK: an API for automated refactorings in Eclipse IDEs, 2006. Technical Article. <http://www.eclipse.org/articles/Article-LTK/ltk.html>.
13. Miguel Garcia. Formalizing the well-formedness rules of EJB3QL in UML + OCL. In T. Kühne, editor, *Reports and Revised Selected Papers, Workshops and Symposia at MoDELS 2006, Genoa, Italy*, LNCS 4364, pages 66–75. Springer-Verlag, 2006.
14. 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>.
15. Miguel Garcia. Formalization of QVT-Relations: OCL-based Static Semantics and Alloy-based Validation. In *Proc. of the 2nd Workshop on MDS Today*, pages 21–30. Shaker Verlag, October 2008. ISBN 978-3-8322-7627-0, <http://www.sts.tu-harburg.de/~mi.garcia/pubs/2008/qvtr/QVTRelationsFormalization.pdf>.
16. Miguel Garcia and Ralf Möller. Incremental Evaluation of OCL Invariants in the Essential MOF Object Model. In Thomas Kühne, Wolfgang Reisig, and Friedrich Steimann, editors, *Modellierung 2008*, volume 127 of *GI-Edition Lecture Notes in Informatics*, pages 11–26, 2008.
17. Peter M. D. Gray, Larry Kerschberg, Peter J.H. King, and Alexandra Poulouvasilis (Eds.). *The Functional Approach to Data Management: Modeling, Analyzing, and Integrating Heterogeneous Data*. SpringerVerlag, 2004.

18. Torsten Grust, Manuel Mayr, and Jan Rittinger. XQuery join graph isolation. In *Proc. of the 25th Intl. Conf. on Data Engineering (ICDE 2009), Shanghai, China, March/April 2009*. Extended version at <http://arxiv.org/abs/0810.4809>.
19. Torsten Grust and Marc H. Scholl. Translating OQL into Monoid Comprehensions—Stuck with Nested Loops? Technical Report 3a/1996, Database Research Group, Univ Konstanz, September 1996.
20. Simon Peyton Jones. *Beautiful Code: Leading Programmers Explain How They Think*, chapter 24 (Beautiful Concurrency). O'Reilly Media, Inc., 2007.
21. 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>.
22. Jevgeni Kabanov and Rein Raudj arv. Embedded typesafe domain specific languages for Java. In *PPPJ '08: Proc. of the 6th Intl. Symp. on Principles and Practice of Programming in Java*, pages 189–197, New York, NY, USA, 2008. ACM. <http://www.ekabanov.net/kabanov-raudjarv-pppj08.pdf>.
23. Dragos Manolescu, Brian Beckman, and Benjamin Livshits. Volta: Developing distributed applications by recompiling. *IEEE Softw.*, 25(5):53–59, 2008.
24. Krishna K. Mehra, Sriram K. Rajamani, A. Prasad Sistla, and Sumit K. Jha. Verification of object relational maps. In *SEFM '07: Proc. of the Fifth IEEE Intl. Conf. on Software Engineering and Formal Methods*, pages 283–292, Washington, DC, USA, 2007. IEEE Computer Society.
25. Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *SIGMOD '07: Proc. of the 2007 ACM SIGMOD Intl Conf on Mgmt of Data*, pages 461–472, New York, NY, USA, 2007. ACM.
26. Guido Moerkotte. Building Query Compilers. Draft; <http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf>, 2009.
27. Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007. ISBN 9780978739256.
28. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., NJ, USA, 1987. <http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/>.
29. Franklyn A. Turbak and David K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008.
30. Joost Visser. Matching Objects without Language Extension. *Journal of Object Technology*, 5(8):81–100, 2006.
31. Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. *SIGPLAN Not.*, 43(10):19–36, 2008.
32. Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the Java Query Language. In *OOPSLA '08: Proc. of the 23rd ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications*, pages 1–18, New York, NY, USA, 2008. ACM.
33. Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute grammar-based language extensions for Java. In Erik Ernst, editor, *ECOOP*, volume 4609 of *LNCS*, pages 575–599. Springer, 2007.