Advances in Programming Languages

APL20: Type-checking for SQLizeability

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 17 March 2010
Semester 2 Week 10

Topic: Domain-Specific vs. General-Purpose Languages

This is the fourth of four lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- LINQ: .NET Language Integrated Query
- Language integration in F#
- Type-checking for SQLizeability

Topic: Domain-Specific vs. General-Purpose Languages

This is the fourth of four lectures on integrating domain-specific languages with general-purpose programming languages. In particular, SQL for database queries.

- Using SQL from Java
- LINQ: .NET Language Integrated Query
- Language integration in F#
- Type-checking for SQLizeability

A Short Paper with a Long Title

This lecture presents results from the following research paper:

Ezra Cooper.
The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed.
In *Database Programming Languages: Proceedings of the 12th International Symposium DBPL 2009*, Lecture Notes in Computer Science 5708, pages 36–51. Springer-Verlag, 2009.

Ezra developed web applications for Amazon and Moveable Type; did a PhD here at Edinburgh; and now works in Boston on XQuery searching.
http://ezrakilty.net/.

Language Integrated Query

We have seen how LINQ in C# can lower the *impedance mismatch* between programming language and query language, making a host language more sensitive to the semantics of its guest.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

Language Integrated Query

There is more here than just extra SQL-like keywords. The Table<Person> has typed records, field selection u.Score can be checked at compile time, and each item has a correct static type.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = from u in users
              where low < u.Score && u.Score < high
              select new { u.Id, u.Name };

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

Language Integrated Query

The special SQL-like syntax is sugar that expands into a sequence of method invocations, using higher-order functions and anonymous closures.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = users.Where(u => (low < u.Score && u.Score < high))
                   .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

Language Integrated Query

This expansion into standard method calls opens up query handling to compiler optimisation: we are no longer just executing an SQL string, but building a structured query.

```
float findUsersInRange(SqlConnection con, float low, float high) {

  Table<Person> users = con.GetTable<Person>()

  var query = users.Where(u => (low < u.Score && u.Score < high))
                   .Select(u => new { u.Id, u.Name }) ;

  foreach(var item in query)
    { Console.WriteLine("{0}: {1}", item.Id, item.Name); }
}
```

LINQ limits

LINQ brings many good things:

- Creating SQL queries from C# syntax
- Static checking of syntax and database schema
- Parameterization and abstraction
- Compiler-led query amalgamation

But there are limitations:

- SQL conversion is best-effort — it may fail at runtime
- Abstraction not fully higher-order
- Exposes concrete *Expression* type with special properties

How it might be

Cooper sets out some examples, using the syntax of *Links*. It's a general-purpose functional language, with some syntax to make queries look natural. (All examples are from the paper.)

```
# Alice runs a local baseball league

fun overAgePlayers() {              # Alice wants a list of players over 12
  query { for (p <- players)
            where (p.age > 12)
              [(name = p.name)] }
}

# The "query" block indicates that this should be translated to SQL

# Here "for ... where" is a 'bag comprehension' that gathers together # a
multiset of records satisfying the guard
```

How it might be

Cooper sets out some examples, using the syntax of *Links*. It's a general-purpose functional language, with some syntax to make queries look natural. (All examples are from the paper.)

```
# We introduce the gratuitous complication of reversing player's names

fun overAgePlayersReversed() {
  query { for (p <- players)
          where (p.age > 12)
            [(name = reverse(p.name))] }      # ERROR !
}

# Because we specified a "query" block, the compiler raises
# an error: SQL has no string reverse operation
```

How it might be

Cooper sets out some examples, using the syntax of *Links*. It's a general-purpose functional language, with some syntax to make queries look natural. (All examples are from the paper.)

```
# Obtain team rosters as [(name:String, roster:[(playerName:String)])]
fun teamRosters() {
  for (t <- teams)
    [(name  = t.name,
      roster = for (p <- players)
               where (p.team == t.name) [(playerName=p.name)])];
}
fun usablePlayers() {                        # Identify players on full teams
  query { for (t <- teamRosters())           # For each team list
          where (length(t.roster) >= 9)      # If big enough
            t.roster }                       # Add members to mailing list
}
```

How it might be

Cooper sets out some examples, using the syntax of *Links*. It's a general-purpose functional language, with some syntax to make queries look natural. (All examples are from the paper.)

```
/*
  Although teamRosters returned a nested collection, which cannot be
  directly represented in SQL, we can still translate the overall query
  using nested SELECT queries.
*/

SELECT p.name AS playerName
  FROM players AS p, teams AS t
  WHERE
    (SELECT COUNT(*)
       FROM players AS p2 WHERE p2.team = t.name) < 9
```

How it might be

Cooper sets out some examples, using the syntax of *Links*. It's a general-purpose functional language, with some syntax to make queries look natural. (All examples are from the paper.)

```
fun playersBySelectedTeams(pred) { # Higher−order function taking
  query {                          # a predicate "pred" as argument
    for (t <− teamRosters())
    where (pred(t.roster))         # Can this be SQLized? It depends
      t.roster }
}

fun fullTeam(list) { length(list) >= 9 }

fun seniorPlayers(list) { for (x <− list) where (x.age >= 15) [x] }

playersBySelectedTeams(fun(x) { fullTeam(seniorPlayers(x)) } )
```

How it might be

Cooper sets out some examples, using the syntax of *Links*. It's a general-purpose functional language, with some syntax to make queries look natural. (All examples are from the paper.)

```
playersBySelectedTeams(fun(x) { fullTeam(seniorPlayers(x)) } )

-- In this case, we can translate into SQL

SELECT p.name AS playerName
  FROM players AS p, teams AS t
    WHERE ( SELECT COUNT(*) FROM players AS p2
        WHERE p2.team = t.name AND p2.age >= 15) >= 9

-- This requires expanding the predicate, and rearranging to
-- appropriate nest the SQL. Can we build a type system to check
-- all this in a modular way at compile time?
```

Can this be done?

Cooper does not, in fact, present an implementation in the paper (although an experimental one does exist).

Instead, he sets out three things:

- A method for statically checking whether conversion is possible
- A detailed explanation of a procedure to carry out the conversion
- *A proof that this always works*

This is a standard approach in programming language research: after all, from an algorithm and a proof you might build an implementation; but the reverse is much harder.

One step further is to including a machine-checked proof; this is rare as yet, but it's the future.

Types and effects

Static checking for SQLizability is done through a *type and effect system*.

Where a type system might have judgements like this:

$$x_1 : S_1, \dots, x_n : S_n \vdash M : T\,.$$

A type and effect system has judgements like this:

$$x_1 : S_1, \dots, x_n : S_n \vdash M : T \,!\, e\,.$$

Here e is the set of possible *effects* associated with the evaluation of M.

Deriving types and effects

A type and effect system comes with rules for deriving valid judgements. For example:

$$\frac{\Gamma \vdash M_1 : [\mathsf{T}] \mathrel{!} e_1 \quad \Gamma \vdash M_2 : [\mathsf{T}] \mathrel{!} e_2}{\Gamma \vdash M_1 \mathbin{++} M_2 : [\mathsf{T}] \mathrel{!} e_1 \cup e_2} \qquad \Gamma = x_1 : S_1 \dots x_n : S_n$$

These rules are chained together to make a complete derivation.

As with plain type systems, it is possible to automatically infer many effect annotations.

Effects within types

The types and effects may interact, as in function abstraction and application.

$$\frac{\Gamma, x : S \vdash M : T \mathrel{!} e}{\Gamma \vdash \lambda x.M : S \xrightarrow{e} T \mathrel{!} \emptyset} \qquad \frac{\Gamma \vdash F : S \xrightarrow{e} T \mathrel{!} e_1 \qquad \Gamma \vdash N : S \mathrel{!} e_2}{\Gamma \vdash FN : T \mathrel{!} (e_1 \cup e_2 \cup e)}$$

Here function type $S \xrightarrow{e} T$ includes a *latent* effect e, which emerges when the function is applied to an argument.

Effects for SQLizability

We need effects to track when code needs a feature *not* available in SQL.

We can do this with an effect *noqy*. For example:

$$(+) : \mathsf{int} \times \mathsf{int} \xrightarrow{\emptyset} \mathsf{int} \qquad \mathsf{length} : [\mathsf{T}] \xrightarrow{\emptyset} \mathsf{int} \qquad \mathsf{print} : \mathsf{string} \xrightarrow{\mathit{noqy}} ()$$

The rule for typing a "query" block checks this:

$$\frac{\Gamma \vdash M : \mathsf{T} \mathbin{!} \emptyset \qquad \mathsf{T} \text{ has the form } [(\overline{l : o})]}{\Gamma \vdash \mathsf{query}\{M\} : \mathsf{T} \mathbin{!} \emptyset}$$

Provided that the types check out, we can build arbitrary combinations of query blocks, abstraction, higher-order functions, application, comprehension, . . .

Results

The paper sets out a *rewrite system* $M \rightsquigarrow M'$ which flattens out and simplifies terms, with the following properties:

- Types are preserved: if $M : T \,!\, \emptyset$ and $M \rightsquigarrow M'$ then $M' : T \,!\, \emptyset$.
- Every term normalizes: $M \rightsquigarrow^* V$ for some $V \not\rightsquigarrow$.
- If $M : [(\overline{l : o})] \,!\, \emptyset$ then its normal form directly matches SQL constructions.

The result is that if a term does not have the *noqy* effect, then it can always be converted SQL. This might happen at compile time, run time, or both: but *it will always succeed*.

Summary

- LINQ offers language integration for queries, but only best-effort translation. Things can go wrong at runtime.
- An *effect* system refines types with information about side-effects that happen on execution.
- We can construe "not available in SQL" as a side-effect.
- Static inference and checking of types and effects is enough to know in advance which terms can be turned into SQL queries.

Specific properties of types and effects for SQLizability:

- Compositional, for modular checking
- Supports arbitrary higher-order types
- Complete integration of guest and host language terms
- Compile-time guarantees

The End

Please complete a course questionnaire, either on paper or online.
Paper copies can be left in the lecture theatre, or delivered to the ITO.

The Examination Timetable is available online:

http://www.registry.ed.ac.uk/Examinations/index.cfm

Good luck, and enjoy learning more programming languages.

I've worked with many languages, from BASIC to assembly code.
One of the last checkins I made when implementing generics for .NET,
C# and VB had a lot of x86 assembly code. My first job was in Prolog.
I think programmers should learn languages at all extremes.

Don Syme, F#