Domain-Specific Embedded Compilers and SchemeQL

Presenter: Walter Chang

Overview

- Embedding languages
- Why embed languages?
- Embedded SQL vs language embeddings
- SchemeQL
- Embedding SQL into Haskell

Embedding Languages

- Database queries can be embedded into applications in different ways
- Instead of embedding SQL strings, we can embed higher-level constructs with more linguistical meaning
- Instead of

```
db_query("SELECT id FROM emp...")
```

we can have

```
dbQuery q = new dbQuery(id, emp, ...);
results = q.executeQuery(db);
```


Why embedded query languages?

Embedding queries directly into the host language has several advantages over embedding SQL as text:

- Embedded SQL statements are not checked until run time
- Queries as embedded SQL statements are not first-class in the language
- Embedding queries in the host language allows composition, abstraction, and type-checking of queries
- Embedded queries in the host language are syntactically correct by design
- Embedding queries reduces the number of languages implicitly present in a source program

Embedding Queries, the old way

Example (from Leijen & Meijer)

```
Q =     "SELECT *"
Q = Q & "FROM Rogerson AS r"
Q = Q & "WHERE r.President = FALSE"

Set RS = CreateObject("ADODB.Recordset")
RS.Open Q "Rogerson"

Do While Not RS.EOF
  Print RS("Object")
  ...
  RS.MoveNext
Loop
```


Embedding Queries, the old way

Example (from Leijen & Meijer)

The Real Work

```
Q =     "SELECT *"
Q = Q & "FROM Rogerson AS r"
Q = Q & "WHERE r.President = FALSE"

Set RS = CreateObject("ADODB.Recordset")
RS.Open Q "Rogerson"

Do While Not RS.EOF
  Print RS("Object")
  ...
  RS.MoveNext
Loop
```

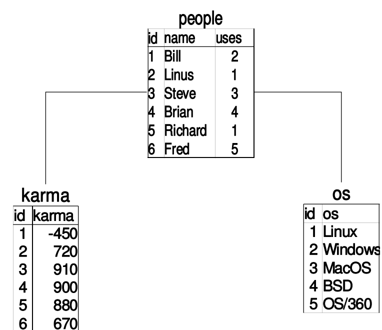
Extra infrastructure

SchemeQL

- Embeds a fragment of SQL in Scheme
- Allows queries to be treated as first-class entities
- Similar to SQL: offers a "gentle slope from existing SQL... to higher level abstractions"
- Creates SQL queries by composing simple operations
- Is very simple to use!

Let's see some examples...

SchemeQL Example database



people

id	name	uses
1	Bill	2
2	Linus	1
3	Steve	3
4	Brian	4
5	Richard	1
6	Fred	5

karma

id	karma
1	-450
2	720
3	910
4	900
5	880
6	670

os

id	os
1	Linux
2	Windows
3	MacOS
4	BSD
5	OS/360

SchemeQL Example Query 1

Get the ids of the Linux users:

In SQL:

```
SELECT people.id
  FROM people, os
  WHERE people.uses = os.id
  AND os.os = 'Linux'
```

In SchemeQL:

```
(query ((people id)
        (people os)
        ((= (people uses) (os id))
         (= (os os) “'Linux'”)))
```


SchemeQL Example Query 2

How about the names of high-karma people?

In SQL:

```
SELECT people.name
  FROM people, karma
  WHERE people.id = karma.id
  AND karma.karma > 800
```

In SchemeQL:

```
(query ((people name)
        (people karma)
        ((= (people id) (karma id))
         (> (karma karma) 800)))
```


SchemeQL Example 2

We can parameterize over them:

```
(define (find-by-os field osname)
        (query ((people field))
               (people os)
               ((= (people uses) (os id))
                (= (os os) osname))))

(define (find-by-karma field pred num)
        (query ((people field))
               (people karma)
               ((= (people id) (karma id))
               (pred (karma karma) num))))

(define (good-karma threshold)
        (find-by-karma 'id '> threshold))
```


SchemeQL Example

How about all Linux users with good (> 800) karma?

In SQL:

```
SELECT id
  FROM people, karma
  WHERE people.uses = os.id
  AND os.os = 'Linux'
  INTERSECT ( SELECT id
              FROM karma
              WHERE karma > '800')
```

In SchemeQL (with our previous definitions)

```
(let ((linux-users (find-by-os 'id “Linux”))
      (saints (good-karma 800)))
  (intersect linux-users saints))
```


SchemeQL Example

How about our VB example?

```
Q =     “SELECT *”
Q = Q & “FROM Rogerson AS r”
Q = Q & “WHERE r.President = FALSE”

Set RS = CreateObject(“ADODB.Recordset”)
RS.Open Q “Rogerson”

Do While Not RS.EOF
  Print RS(“Object”)
  ...
  RS.MoveNext
Loop
```

```
(cursor-map
  stuff
  (schemeql-execute
    (query ((as all r))
           (Rogerson)
           (= (r President) FALSE))))
```


Embedding SQL in Haskell

- Define an abstract syntax for the terms of the language to be embedded (here, SQL)
- Define operators (for syntactic sugar)
- Define necessary type rules (via introducing an artificial polymorphic type for the domain)
- Define transformation from abstract syntax to concrete syntax (a mini-compiler)
- Overload the evaluator

Quick and Dirty Intro to Haskell

Haskell is functional:
A function, line:
`line = \a -> \b -> \x -> a*x + b`
The type:
`line :: Int -> Int -> Int -> Int`

In Scheme

```
(define line
  (lambda (a)
    (lambda (b)
      (lambda (x) (+ (* a x) b)))))
```

(or just `(define (line a b x) (+ (*a x) b))`)

In SML:
`val line = fn a => fn b => fn x => a*x + b`

More Quick and Dirty Haskell

Case expressions can define recursive functions:

```
fac = \n ->
  case n of
    { 0 -> 1
    ; n -> n * fac (n-1)
    }
```

You can do{} things for side-effects:

```
do{ c <- getChar
  ; putChar c
  }
```

Comprehensions (Haskell/DB)

Haskell/DB supports comprehensions. How do we do a query?

```
SELECT *
FROM Rogerson AS r
WHERE r.President = FALSE
```

```
do{ r <- table rogerson
  ; restrict
    (r!president .==. constant False)
  ; return r
  }
```


Embedding Haskell (AST)

First we must define the abstract syntax for relation algebra:

```
type TableName = String
type Attribute = String
…
data PrimQuery
  = BaseTable TableName Scheme
  | Project Assoc PrimQuery
  | Restrict PrimExpr PrimQuery
  | Binary RelOp PrimQuery PrimQuery
  | Empty

data RelOp
  = Times | Union | Intersect
  | Divide | Difference
```

... and so on

Embedding Haskell (Types)

Trick: introduce a polymorphic type `Expr a` such that `expr :: Expr a` means `expr` is an expression of type `a`

```
data Expr a = Expr PrimExpr
```

Now define functions that operate on expressions to return values of type `Expr a`

```
(.==.) :: Eq a => Expr a -> Expr a -> Expr
Bool

(Expr x) .==. (Expr y)
    = Expr (BinExpr OpEq x y)
```

Now the AST for queries can be “type-checked”

Towards Comprehensions

In relational algebra, attributes are specified only by name. To do something like take a cartesian product with itself, we need a rename mechanism.

```
SELECT X.Name, X.Mark
FROM Students AS X, Students AS Y
WHERE X.Mark = Y.Mark
AND X.Name <> Y.Name
```

```
do{ x <- table students
  ; y <- table students
  ; restrict (x!mark .==. y!mark)
  ; restrict (x!name .<>. y!name)
  ; project (name = x!name, grade = x!grade)
  }
```

This doesn't address type-checking, though.

Towards Comprehensions, cont.

Untyped Comprehensions

- Define a Query monad
- operation returnQ
  returnQ :: a -> Query a
- operation bindQ
  bindQ :: Query a ->
    (a -> Query b) ->
    Query b
- Details are “rather subtle and a through discussion is outside the scope of this paper”

Typed Comprehensions

- Parameterize the selection operator (!) to return an expression of the same type
- Parameterize Rel by its “scheme” to prevent selecting nonexistent members
- (!) :: Rel r -> Attr r a -> Expr a

Recap

- Embedding SQL allows more flexible and natural manipulation than plain strings
- Embedding SQL allows static validity and type checking of queries
- Embedding SQL reduces or hides the details of the language/database interface

Discussion

- Is language embedding worthwhile?
- How about imperative examples?
- Does embedding generalize beyond databases to other domains?