Polymorphism and Type Inference in Database Programming

A paper by Peter Buneman and Atsushi Ohori

Presented by Mark Grechanik

A Problem

- Checki… SQL queries
  - SELEC… …ry > 100,000
- What … ?
  - Functi… WHERE
- A problem is …

How to type check database-dependent programs?

The Main Goal

- Develop a polymorphic type system that represents various database structures
- Propose an extension to ML
  - Express polymorphic nature of data types and operations
  - Create an appropriate basis for a general-purpose database programming languages
- Formalize proposed type system

Why ML?

- Standard ML is a safe, modular, strict, functional, polymorphic programming language with
  - compile-time type checking and type inference
  - garbage collection
  - exception handling
  - immutable data types and updatable references
  - abstract data types, and parametric modules
  - It has efficient implementations and a formal definition with a proof of soundness.
  - http://www.smlnj.org/sml.html

Basic ML Concepts

- Expressions
  - > 1 + 2 * 3
  - > val it = 7 : int
- …ions
  - fn: <domain type> → <range type>
  - > fun square(x : real) = x*x
  - > val square = fn : int → int

Value of expression

type

value

Previous expression

Basic ML Concepts

- Expressions
  - > 1 + 2 * 3
  - > val it = 7 : int
- …s
  - fn: <domain type> → <range type>
  - > fun square(x : real) = x*x
  - > val square = fn : int → int

keyword

identifier

expression

parameter

Basic Structures

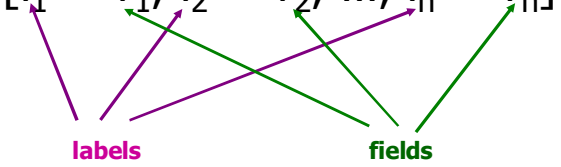
- Values and functions
- Labeled records and variants
- Sets
- Cyclic structures
- References

Values and Functions

- Borrowed from ML
- Use ***val*** to bind names to values
  - val seven = 5 + 2
- Functions
  - fun f(n) = if eq(n, 1) then 1 else n*f(n-1)
  - (fn x => x*x)(10)
- Let x = e_1 in e_2
  - evaluates e_2 in the environment in which x is bound to e_1
  - let x = 4+5 in x+x*x

Labeled Records

$[l_1 = v_1, l_2 = v_2, \ldots, l_n = v_n]$



Labeled Records

- Field selection
  - r.l, where r is a record, l is a label
- Field modification
  - modify(r, l, e)
  - create a new record identical to r
  - l field has value e
  - modify([Name="Mark", Age=6], Age, 10)
  - [Name="Mark", Age=10]
- Pairs: for construction of n-tuples
  - (e_1, e_2) means [first=e_1, second=e_2]

Variants

- Variants tag values in order to treat them uniformly
- Syntax
  - <l = v>
  - <Real = 9.0>
  - <Int = 3>
- Powerful use with case expressions

Variants

```
case <Person= [Name="Mark",
      CreditCard="1056354813238137",
      Phone="256-3765"]>
of
     <Person = x> => x.Phone,
     <Animal = y> => y.Kind
endcase
```

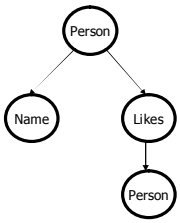
"256-3765"

Sets

- Description terms
- Four operations
  - {} : empty set
  - {x} : singleton set construct
  - Union(s_1, s_2) : set union
  - Homomorphic extensions
    - hom(f, op, z, s)
    - hom(f, op, z, {}) ≡ z
    - hom(f, op, z, {s}) ≡ f(s)
    - hom(f, op, z, union(s_1, s_2) ≡ op(hom(f, op, z, s_1), hom(f, op, z, s_2))

Cyclic Structures

- Pointer reassignment
  - val Person = (rec v.[Name="Mark", Likes=v])



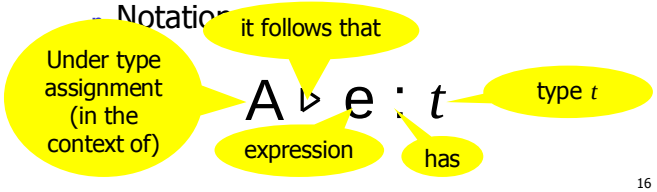
...

References

- Primitives
  - new(v) : reference creation
  - !r : dereferencing
  - r := v : assignment
- Uniqueness of references
  - Person = new ([Name="Mark", Age=11])

Type Systems

- A set of rules or axioms that are used to define legal programs in a language
- Typing rules determine types for all expressions
- Notation



Example of a Type Rule

if **eq(x*x, y)** then x - y

$$\frac{A \triangleright e_1 : t \qquad A \triangleright e_2 : t}{A \triangleright \mathbf{eq}(e_1, e_2) : \text{bool}}$$

Why Machiavelli?

- It is an ML-based language
- Developed at UPenn
- Serves as a framework to fulfill the main goal of this work
  - Develop a polymorphic type system that represents various database structures
- Example
  - **Function Wealthy(S)= SELECT x.Name FROM x ¬ S WHERE x. Salary > 100,000**

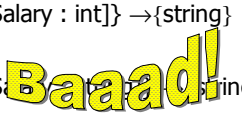
Kinded Types in Machiavelli

Wealthy: $\{\tau :: [\![\text{Name} : \delta, \text{Salary} : \text{int}]\!]\} \rightarrow \{\delta\}$

Polymorphic method **Class**

{[Name : string, Salary : int]} →{string}
{[Name : string, Age : int, Salary : int]} →{string}

{[Name : string} →{string}
{[Name : string, Age : int, S ring}



Relational Algebra

- Algebraic notation used to express queries by applying specialized operators to relations
  - Limited expressive power of operators
  - Finiteness of relations (complement operator)
  - Rich enough language to express things that make database systems useful

Operators of Relational Algebra

- Union: $R \cup S$
- Set difference: $R - S$
- Cartesian product: $R \times S$
- Projection: $\pi_{3,1}(R)$
- Selection: $\sigma_{\$2>\$3 \wedge \$1=\text{"Mark"}}$
- Join: $R \bowtie_{i\Theta j} S \equiv \sigma_{\$i\Theta\$j} R \times S$
- Semijoin: $R \ltimes S \equiv \pi_R(R \bowtie S)$

Operators of Relational Algebra

- Union: $R \cup S$
- Set difference: $R - S$
- Cartesian product: $R \times S$
- Projection: $\pi_{3,1}(R)$
- Selection: $\sigma_{\$2>\$3 \wedge \$1=\text{"Mark"}}$

A	B	C
1	2	3
4	5	6
7	8	9

D	E
3	1
6	2

A	B	C	D	E
1	2	3	3	1
1	2	3	6	2
4	5	6	6	2

$R \bowtie_{B<D} S$

- Join: $R \bowtie_{i\Theta j} S \equiv \sigma_{\$i\Theta\$j} R \times S$
- Semijoin: $R \ltimes S \equiv \pi_R(R \bowtie S)$

Operators of Relational Algebra

- Union: $R \cup S$
- Set difference: $R - S$
- Cartesian product: $R \times S$
- Projection: $\pi_{3,1}(R)$
- Selection: $\sigma_{\$2>\$3 \wedge \$1=\text{"Mark"}}$

A	B	C
a	b	c
d	b	c
b	b	f
c	a	d

B	C	D
b	c	d
b	c	e
a	d	b

A	B	C
a	b	c
d	b	c
c	a	d

- Semijoin: $R \ltimes S \equiv \pi_R(R \bowtie S)$

Why to Generalize Relational Algebra?

- Machiavelli does not have a number of required operators
  - Projection
  - Join
- Join operator requires to extend the notion of kinded types
  - Based on many papers

Generalization

- Add relational operators to Machiavelli
- Make them as polymorphic as possible
- Natural limitations: equality
- Operators
  - eq(e_1, e_2) : equality test
  - join(e_1, e_2) : database join
  - con(e_1, e_2) : consistency check
  - project (e, δ) : projection of e onto type δ

Join

- t_1 = [Name=[First="Joe"]]
- t_2 = [Name=[Last="Doe"]]
- join(t_1 , t_2) $\rightarrow$ t
- t = [Name=[First="Joe", Last="Doe"]]
- Ordering is induced by the inclusion of record fields
- t is the least upper bound of t_1 and t_2
  - join(t_1 , t_2) = $t_1 \sqcup t_2$

Equal Operator

- Tests the equality on the amount of information
- Characterized by the equivalence relation
- Information ordering
  - eq(d_1, d_2) = $d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_1$
  - Null values: null(b)

Projections

- Projection of a complex description onto some "substructure", i.e. it throws away some type information
  - Project([Name="Mark", Age=3, Salary=10], [Name="John", Age=7, Salary=27]}, [Name:string, Salary:int]}) = {[Name="Mark", Salary=10], [Name="John", Salary=27]}

Type Inference

- Principal conditional typing theorem
  - Theorem 4, page 32
  - Any term can either be typed or produces a failure
- Examples
  - Join3([Name="Mark"],[Age=7],[Office=523])
  - val it=[Name="Mark",Age=7,Office=523]: [Name:string,Age:int,Office=int]
  - project(it, [Name:string])
  - val it=[Name="Mark"]: [Name:string]

Heterogeneous Collections

- Contradictory use of inheritance
  - In OO languages it means code sharing
  - In databases it means entity inclusion
- Leads to losing information in type systems
- A number of solutions are proposed
- This solution
  - Do not allow an operation to be applied to a value of an inappropriate type
  - Dynamic type checking in a statically-typed framework

Conclusions

- Offered an extension to the type system of ML/Machiavelli
- Generalized relational algebra
- How does Machiavelli compare to SQL?
- Many updates
  - how garbage collector handles memory
- What about persistence?