On Type Systems for Object-Oriented Database Programming Languages

Authors : Yuri Leontiev
M. Tamer Ozsu
Duane Szafron
Presenter: Jun Yuan

Purpose of This Paper

- This paper addresses two problems:
  - What are the requirements for a type system of an OODBPL
  - Are there any type systems satisfying these requirements?

Outline

- Type systems features
- OOPL requirements
- DBPL requirements
- OODB requirements
- Requirements summary
- Test programs
- Type system review
- Conclusion

Type System Features

- A type system should be
  - Verifiable
    - Exist a type checking algorithm
  - Transparent
    - Report sufficient error message to a programmer
  - Enforceable
    - Prevent execution of type incorrect programs
  - Extensible
    - Can be extended for new applications

OOPL Requirements

- Inheritance
  - interface
  - implementation
- Method types
  - Methods are objects
- Reflexivity
  - Types are objects
- Method uniformity
  - No public instance variables
- Multi-methods
  - Use types of all arguments during dispatch
- Substitutability

DBPL Requirements

- Persistence independence
- Type constructors
- Encapsulation
- Parametric types
- Partial type specification
- Verifiable && sound type system
- Capability of typing SQL-like queries
- Incremental type checking

OODB Requirements

- Complex objects
- Extensibility
  - User can add new primitive types
  - Treat system and user defined types uniformly
- View mechanism
- Dynamic schema evolution

Requirement Summary

- Theoretical requirements
  - Verifiability
- Inheritance requirements
  - Interface and implementation
  - Substitutability
- Expressibility requirements
  - Method types
  - Parametric types
  - Type constructor
  - Encapsulation
  - Mutable objects
  - Multi-methods
  - SQL-like queries

Requirement Summary (Cont)

- Uniformity requirements
  - Extensibility
  - Persistence independency
  - Method uniformity
- Reflexive requirements
- Dynamic requirements
  - Partial type specification
  - Schema evolution
- Others
  - Transparency
  - Incremental type checking
  - View mechanism

Test Program - PERSON

- Subtype can change super-type

```
type T_Integer;
type T_SmallInteger subtype of T_Integer;
type T_Person {
    getAge(): T_Integer;
};
type T_Child subtype of T_Person {
    getAge(): T_SmallInteger;
};
```


Test Program – POINT

- Multiple dispatch for binary methods

```
type T_Point {
   equal(T_Point p):T_Bool
          implementation ...  // equal1
};
type T_ColorPoint subtype of T_Point {
    equal(T_ColorPoint p):T_Bool
          implementation ... ;// equal2
};
T_Point p1 := new T_Point (...);
T_ColorPoint p2 := new T_ColorPoint (...);

p1.equal(p2);
// should call equal1
p2.equal(p1);
// should call equal1
```


Test Program - STREAM

- Parametric && inclusion polymorphism

```
type T_InputStream(covar X) {
    get():X;};
type T_OutputStream(contravar X) {
    put(X arg);
};
type T_IOStream(novar X) subtype of T_InputStream(X), T_OutputStream(X);
T_OutputStream(T_Point) osp;
T_IOStream(T_Point) iosp;
T_OutputStream(T_ColorPoint) oscp;
T_IOStream(T_ColorPoint) ioscp;

oscp = iosp;  Is this correct?
Correct!
osp = ioscp; Is this correct?
No!
```


Test Program - SORT

- Ability to deal with bounded quantification

```
…
sort(T_List(X) list): T_List(X)
    where (X implements I_Comparable)
…
```


Test Program - SET

- Ability to type SQL-like queries

```
type T_Set(X) {
    union(T_Set(Y) summand): T_Set(lub(X,Y));
    Intersection(T_Set(Y) summand): T_Set(glb(X,Y));
};
```


Test Program - BROWSER

- Dynamic type checking

```
printNumber(T_Number num)
                implementation ... ;
type T_Person {
    getAge():T_Integer;
};
T_Object root;      T_Database db;
db.open();
root := db.getRoot();
Type case root.typeOf() {
    subtype of T_Number: {
          printNumber(root); ... };
    subtype of T_Person: {
          printNumber(root.getAge());
... };}
```


Test Programs - Other

- COMPARABLE
  - Substitutablity
- GENSORT
  - Parametric types
  - Method types
- LIST
  - Support code reuse beyond subtyping
- APPLY
  - Method types

Type System Review

- Over 60 languages are evaluated using test programms
- We review a couple of them
  - C++
  - Java && extension
  - ODMG object model
  - ML
  - Machiavelli

C++

- Good aspects
  - Parametric types
  - Type constructors
  - Method types
- Limitations
  - No separation of interface and implementation
  - No intersection and union types
  - Not Uniform
  - Not verifiable
  - Limited substitutability
  - Lack of multiple dispatch
- Test passed
  - GENSORT(1 out of 10)

Java

- Shares many features with that of C++
- Difference
  - Separation of interface and implementation
  - Lack of method types
  - Lack of parametric types
- Test passed
  - SORT (1 out of 10)

GJ

- GJ extends Java by adding parametric types
- Parameter can only be of reference types
- Tests passed (2.5 out of 10)
  - SORT
  - COMPARABLE
  - Union part of SET

Pizza

- Pizza extends Java by adding
  - Parametric types
  - Method types
- Tests passed (4 out of 10)
  - GENSORT
  - APPLY
  - COMPARABLE
  - SORT

ODMG Model

- Good aspects
  - Separation of interface and implementation
  - Partial type specification
- Limitations
  - Lack of multiple dispatch
  - Parametric types only for property specification
  - Set operation only performs on types with a least upper bound. No notion of greatest lower bound
- Tests passed (1.5 out of 10)
  - BROWSER
  - Union part of SET

ML

- Good aspects
  - Decidable && sound type checking
  - Parametric types
  - Uniform
- Limitations
  - No subtyping
  - No inheritance
- Tests passed (4 out of 10)
  - PERSON
  - SORT
  - GENSORT
  - APPLY

Machiavelli

- Good aspects
  - Verifiable && sound type system
  - Record polymorphism
  - View mechanism
- Limitations
  - Not uniform
  - Other limitation of ML
- Tests passed (5 out of 10)
  - PERSON
  - SORT
  - GENSORT
  - APPLY
  - SET

Conclusion

- Every test is passed by at least one system
- None of the provably sound type system passes majority of the tests