A Design for Type-Directed Programming in Java

Stephanie Weirich
University of Pennsylvania
joint work with Liang Huang

Type-directed programming?

- Defining operations that can be used for many types of data
- Behavior of operation depends on the type of the data
- Third form of polymorphism
  - Subtype polymorphism (Java)
  - Parametric polymorphism (GJ, ML)
  - Ad-hoc polymorphism (TDP)
- Poster child: Serialization

Serialization

```
public String serialize(Object obj) {
    if (obj == null) return "null";
    if (obj instanceof Integer) {
        return Integer.toString(
            ((Integer)obj).intValue());
    } else if (obj instanceof Boolean) {
        if ((Boolean)obj) { return "true"; }
        else {  return "false"; }
    } else if (obj instanceof Float) { …
    } else { …
```

Serialization (continued)

```
try {
        Class objClass = obj.getClass();
        String result = “[” + objClass + “ ”;
        Field[] f = objClass.getDeclaredFields();
        for (int = 0; i<f.length; i++) {
                f[i].setAccessible(true);
                result += f[i].getName() + “=”;
                result += serialize( f[i].get( obj ));
                if (i<(f.length –1)) result += “,”;
        }
        return result += “]”;
  } catch (IllegalAccessException e)  { return “Impossible”; }
}
```

TDP is not OOP

- Instructors for OO-langs often tell students to replace:

```
if (x instanceof C1) {  dosomething1(); }
else if (x instanceof C2) { dosomething2(); }
else if (x instanceof C3) { dosomething3(); }
```

with

```
    x.dosomething();
```

and put the functionality in C1,C2,C3.

Why not in this case?

- Serialization is used by *many* classes
- *Each* class needs a method called serialize. Implementation dispersed throughout the program.
  - Annoying because there is a general way to define that method.
  - Difficult to change. What if extra state is necessary?
  - May not have access to all classes.

More examples of TDP

- Operations on data structures:
  - Structural equality, cloning, iterators, visitor pattern
- Proxies/Adaptors
  - Add new functionality to an interface
  - Examples: logging, tracing, profiling
- Dynamic objects
  - Checking interface of dynamically loaded code/data
- JavaBeans
  - Presenting components to users
- Runtime debugging tools

Java provides TDP

- Analyze names of types
  - instanceof, cast
- Analyze structure of types
  - Reflection API
  - discover and access the fields and methods of classes
- Both are important

Problems with instanceof and Java Reflection

- Weak guarantees of correctness
  - Almost always requires run-time type casting
- Doesn’t integrate well with generics
  - Type parameters are erased in GJ
- Breaks abstraction
  - Can find out “real” type of an object
  - Can access public and private fields of methods

Our proposal

- Analyze first-class type parameters

```
<T>void m (T x) {
  // To learn about the type of x
  // analyze the type parameter T
}
```

- New operators for discovering the name and structure of run-time type information.

Run-time type information

- Type information provided at run-time to parameterized classes and methods
- NextGen, PolyJ but not GJ
- More expressive: new T(), (T)
- Downside: Not as compatible with existing code

Nominal analysis of type params

```
<T>void m (T x) {
  typematch T {
    case Integer:    … x.intValue() …
    case Boolean:    … if (x) then …
    case C:  …  x.m() …

    default:    … can't do anything special with x
  }
}
```

No casting needed! Change the type of x in the branch

Matches if T is a subtype of C

Comparison with instanceof

- How does typematch compare to instanceof in terms of
  - Eliminating type casts
  - Generics
  - Abstraction

Eliminating casts

- Could we change instanceof to add refinement?

```
Object x;
if (x instanceof Integer) {
   …  x.intValue() + 1 …
}
```

Not a sound change

```
class C {
    Object f = new Integer(3);
}

C x;
if (x.f instanceof Integer) {
    g();
    …  x.f.intValue() …
}
```

```
void g() {
    x.f = new Boolean(false);
}
```

With typematch

```
class C<T> {
   T f = null;   // Initialize in constructor
}
…
C<T> x;
typematch T  {
  case Integer:
        g();
        …  x.f.intValue() …
}
```

g() cannot assign to x.f unless it determines the identity of T

Typematch eliminates many casts

- Can refine the type of many objects

```
T[] arr;
typematch T {
   case Integer:
      // Know all elements of arr are Integers
}
```

- With instanceof, must cast each element individually.

Generics with typematch

- Pattern matching for parameterized classes.

```
typematch T {
   case List<Integer>:
         // only matches lists of Integers
   case List<U>:
         // matches any list
}
```

Generics with instanceof

```
Object x;
if (x instanceof ???) {
    …
}
```

- GJ: only match general lists.
  - List<Integer> is the “same” type as List at run time.
  - Can’t distinguish List<Integer> from List<Boolean>
- NextGen: only match specific instances.
  - List<Object> is not a supertype of List<Integer>.

No abstraction with instanceof

- Subtyping is not an abstraction mechanism.

```
class D extends C { … }
public void m(C x) {
   if (x instanceof D) {
     …
   }
}
```

- m's caller cannot hide the fact that obj is actually a D

```
D obj = …;
m(obj);
```

Abstraction w/ typematch

- Even with parameter analysis, still some information hiding.

```
class D extends C { … }
public <T extends C>void m(T x) {
  typematch T {
    case D: …
  }
```

- m's caller can hide the type by changing the type parameter.

```
D obj = …
m<C>(obj);
```

Structural Analysis (Summary)

- Replacement for Java Reflection
- Use pattern matching to iterate over fields and methods of objects.

```
T obj;
forfield (U f in T) {
        U field =  obj.f;
        ….
}
```

- Same issues arise as with typematch

Conclusion

- Analyzing type parameters is a more principled approach to type-directed programming than instanceof or Java Reflection.

In the paper

- Formalization of typematch and field/method iteration in a core calculus
  - Typing rules and operational semantics for small, FGJ-like language.
- More detailed description of related work
- Companion technical report contains proof of type soundness.

Future Work

- Currently working on an implementation
  - Using Polyglot to extend PolyJ implementation from Cornell University
- Will help us weigh trade off between abstraction and expressiveness
  - How to deal with public/private/protected for fields and methods?
  - Allow access to the run-time type of an object as a first-class type parameter?

Structural analysis

- Additional operations to determine type information
  - getName<T>
  - getNumFields<T>
  - getNumMethods<T>
  - getFieldName<T,f>
  - getMethodName<T,m>
- Easy but still important.

Type-Directed Serialization

```
public <T>String pickle ( T obj ) {
        if (obj == null) return “null”;
        typematch T {
           case Integer:
              return Integer.toString(obj.intValue());
           case Boolean:
              return Boolean.toStrong(obj.boolValue());
           default:
              String result = “[“ + getName<T>” + “ “;
              forfield (U f in T) {
                    result += getFieldName<T,f> + “=“ + pickle<U>( obj.f );
                }
                return result + “]”;
        }
```

Pattern matching is natural

- Can iterate over all integer-valued fields.

```
for field(Integer f in T) {
   sum += obj.f.intValue();
}
```

- Can iterate over all void methods.

```
formethod (void m() in T) {
    if (getMethodName<T,m> == "test") {
        obj.m();
    }
}
```

Limitation to method iteration

- Can't write a single method pattern to match any method
  - formethod (U0 m() in T) { … }
  - formethod (U0 m(U1) in T) { … }
  - formethod (U0 m(U1,U2) in T) { … }
- Also must specify type parameters
  - formethod (X <X>m(X, U) in T) { …}

Formal Language

- We have formalized these ideas in a small language (called TDJ).
  - Explicitly states how type checking works.
  - Necessary to show type soundness.

- TDJ is based on FGJ but has a type-passing semantics.

TDJ Syntax

- New expression forms, compatible with functional core OO language.
- e ::= …

  | typematch T with $\bar{U}$:$\bar{e}$ default :e'

  | fieldfold$_i$ (x = e; T f_x in U) e'

  | methfold$_i$ (x = e; MT m_x in T) e'

  | e.f_x | e.m_x

New assumptions in context

- Typing context contains new forms of assumptions:

  $\Delta ::= \text{empty} \mid X <: T$

  $\quad \mid \Delta,\ T <<: U$

  $\quad \mid \Delta,\ T <<: \{\ U\ f_x\ \}$

  $\quad \mid \Delta,\ T <<: \{\ MT\ m_x\ \}$

- Used when determining subtyping, checking field/method access.

$\text{matches}(N,T) = \Sigma \quad \text{when} \quad \vdash N <: \Sigma(T)$

Execution of typematch

$$\frac{\text{matches}(N, T) = \Sigma}{\text{typematch } N \text{ with } T{:}e \ \bar{U}{:}\bar{e} \text{ default}{:}e' \mapsto \Sigma(e)}$$

$$\frac{\text{matches}(N,T) \text{ not defined}}{\text{typematch } N \text{ with } T{:}e \ \bar{U}{:}\bar{e} \text{ default}{:}e' \mapsto \text{typematch } \bar{U}{:}\bar{e} \text{ default}{:}e'}$$

$$\frac{}{\text{typematch } N \text{ with default}{:}e' \mapsto e'}$$

Type checking typematch

- Add assumptions to context when checking branches

$$\frac{\Delta \vdash T, U \text{ ok} \qquad \Delta_i \vdash U_i \text{ minok} \qquad \Delta, \Delta_i, T <\!\!<: U_i ; \Gamma \vdash e_i \in V_i <: U}{\Delta; \Gamma \vdash \text{typematch } T \text{ with } \bar{U}{:}\bar{e} \text{ default: } e' \in U}$$

Can add unsatisfiable assumptions to context

- Occurs when checking dead code.

```
typematch Integer {
   case Boolean:
      //  who cares whether this branch typechecks?
}
```

- Smart compiler could omit checking branch.

Typechecking fieldfold

- Similar to typematch

$i>0 \qquad \Delta; \Gamma \vdash e: U'' <: U$

$\Delta \vdash T' \text{ ok} \qquad \Delta' \vdash T \text{ minok}$

$\Delta, \Delta', T' <<: \{T\ f_x\}; \Gamma, x{:}U \vdash e' \in U' <: U$

$\Delta; \Gamma \vdash \text{fieldfold}_i\ (x{=}e;\ T\ f_x \text{ in } T')\ e' \in U$

Related work

- Lots of related work on run-time type analysis.
- Closest to intensional type analysis
- Adding assumptions to context like GADTs

Comparison with ITA

- Both systems: type system must propagate discovered type information
  - typecase: type equalities mean that substitution is sufficient.
  - Here: constraints required to propagate information about subtyping.
- Discovering type equalities is more expressive

Typecase/typematch

- Basing typematch on subtyping limits expressiveness
- With typecase the result type of a branch can depend on the pattern

```
 (typecase a of
     int =>  0
     bool => false)   :  a
```

- Unsound for typematch

```
 typematch T {
    case Integer:   0     // assume T <: Integer, not =
    case C:  new C();
 }  : T
```

Future work

- Cases for typematch that require the exact type:

```
typematch D {
    case = C:
        //  even if D <: C this would not fire
    case sub C:
      // instead this branch is taken
}
```