Refactoring for Parameterizing Java Classes

Adam Kiezun (MIT), Michael D. Ernst (MIT), Frank Tip (IBM), Robert M. Fuhrer (IBM)
Parameterization

- Goal: migration of Java code to generics
- Generics (e.g., `List<String>`) enable creation of type-safe, more reusable classes
- Parameterization improves formality of specification in lightweight way
- Libraries and applications must be migrated
  - Hard to do by hand
class Wrapper {
    private Cell c;

    Object get() {
        return c.get();
    }

    void set(Object t) {
        c.set(t);
    }

    boolean in(Object o) {
        return o.equals(get());
    }
}

class Cell {
    private Object data;

    Object get() {
        return data;
    }

    void set(Object t) {
        data = t;
    }

    void copyFrom(Cell c) {
        data = c.get();
    }

    void addTo(Collection c) {
        c.add(data);
    }
}
Parameterization Example

class Wrapper<E1> {
    private Cell<E1> c;

    E1 get() {
        return c.get();
    }
    void set(E1 t) {
        c.set(t);
    }
    boolean in(Object o) {
        return o.equals(get());
    }
}

class Cell<E2> {
    private E2 data;

    E2 get() {
        return data;
    }
    void set(E2 t) {
        data = t;
    }
    void copyFrom(Cell<? extends E2> c) {
        data = c.get();
    }
    void addTo(Collection<? super E2> c) {
        c.add(data);
    }
}
Migration Problem: 2 parts

1. **Instantiation** – updating clients to use generic libraries, e.g.,

   \[
   \text{Graph } g; \quad \rightarrow \quad \text{Graph<}\text{Integer, String}> \, g;
   \]

   Efficient and accurate tools exist (e.g., Eclipse’s INFER TYPE ARGUMENTS, based on our work): OOPSLA’04, ECOOP’05

2. **Parameterization** – annotating classes with type parameters, e.g.,

   \[
   \text{class Graph} \quad \rightarrow \quad \text{class Graph}<\text{V,E}>
   \]

   No usable tools exist – generic libraries parameterized by hand. Parameterization subsumes instantiation.
Related Work

- Constraint-based type inference for OO:
  - Smalltalk: (Graver-Johnson’89), (Palsberg-Schwartzbach’93)
  - Java cast verification: (O’Callahan’99), (Wang-Smith’01)
- Refactoring using type constraints:
  - Decoupling classes (TipEtAl’03, SteimannEtAl’06)
  - Class library migration (BalabanEtAl’05)
  - Class customization (deSutterEtAl’04)
- Generic instantiation:
  - Context-sensitive analysis (DonovanEtAl’04)
  - Context-insensitive analysis (FuhrerEtAl’05)
- Generic parameterization:
  - Generalize C methods from operator overloading (RepsSiff’96)
  - Java methods, unification based (Pluemicke’06)
  - Start with over-generalizations, reduce imprecision heuristically (Duggan’97), (Donovan’03), (vonDincklageDiwan’04)
  - Only one implementation (vonDincklageDiwan’04) but incorrect results (changes program behavior)
Type Inference Approach to Parameterization

- Type inference using type constraints
- Type constraints
  - capture type relationships between program elements
  - additional constraints for behavior preservation (method overriding)
- Solution to constraint system is a correct typing of the program (and unchanged behavior)
Parameterization Algorithm

1. Generate type constraints for the program
   - Syntax-driven, from source
   - Close the constraint system using additional rules

2. Find types for constraint variables to satisfy all constraints
   - Iterative work-list algorithm
   - Many solutions possible: prefer eliminating more casts

3. Rewrite source code
Type Constraints

Notation:

- \( \alpha : \) constraint variable (type of a program element), e.g.:
  - \([e] : \) type of expression e
  - \([\text{Ret}(A.m)] : \) return type of method A.m
  - \([\text{Param}(2, A.m)] : \) type of the 2\(^{nd}\) parameter of A.m
  - String : type constant
  - ? extends \([a] : \) wildcard type upper-bounded by type of a

- \( \alpha \leq \alpha' : \) type constraint ("\( \alpha \) is equal to or a subtype of \( \alpha' \) ")

Examples of type constraints:

- Assignment: \( a = b; \)
  - constraint: \([b] \leq [a]\)

- Method overriding: SubClass.m overrides SuperClass.m:
  - \([\text{Ret}(\text{SubClass.m})] \leq [\text{Ret}(\text{SuperClass.m})] \) (return types)
  - \([\text{Param}(i, \text{SubClass.m})] = [\text{Param}(i, \text{SuperClass.m})] \) (parameters)
Context Variables

Given this declaration:

```java
class NumCell{
    void set(Number p){...}
}
```

consider this call: `c.set(arg)`

What constraint for `[arg]`?:

- `[arg] ≤ Number
  - no: type of p may change as result of parameterization
- `[arg] ≤ [p]
  - no: type of p may differ for receivers, if NumCell gets parameterized to NumCell<E>
    - If [c] is NumCell<Float>, then [p] is Float
- `[arg] ≤ I_c([p])
  - “type of p in the context of the type of the receiver, c”`
Context Variables: examples

Given declaration

class Cell{
    Object get(){...}
}

call c.get()

- constraint: \[ c.get() = I_c[\text{Ret(Cell.get)}] \]
  “type of the call is the return type of the method, in the context of the type of the receiver”

- Return type depends on the receiver (unlike non-generic type system)
Context Variables: examples

Method overriding revisited:
SubClass.m overrides SuperClass.m

- Types depend on subclass:
  - \([\text{Ret(SubClass.m)}] \leq I_{\text{SubClass}}[\text{Ret(SuperClass.m)}]\)
  - \([\text{Param}(i,\text{SubClass.m})] = I_{\text{SubClass}}[\text{Param}(i,\text{SuperClass.m})]\)

- Examples (two subclasses of class Cell\(<E>\)):

  ```java
class StringCell extends Cell<String>{
    String get(){...
    void set(String n){...
  }
class SubCell<T> extends Cell<T>{
    T get(){...
    void set(T n){...
  }
```
Type Constraints Closure

- Java’s type system enforces additional constraints
  - Invariance
    - e.g., \( \text{List}\langle A \rangle \leq \text{List}\langle B \rangle \) iff \( A = B \)
  - Subtyping of actual type parameters
    - e.g., given class \text{MyClass}<T1, T2 extends T1>, declaration \text{MyClass}<\text{String}, \text{Number}> is not allowed

- Algorithm adds constraints that enforce this (i.e., closes the constraint system)
Type Constraint Solving

- **Type estimate** (set of types) associated with each constraint variable
- Estimates initialized depending on element
- Estimates shrink during solving
  - Algorithm iteratively:
    - Selects a constraint
    - Satisfies it by shrinking estimates for both sides
- Finally, each estimate is a singleton
Solving: examples

**Example 1**
Constraint \( a \leq b \)
estimate(b) = \{Number, ? super Number, Date\}
estimate(a) = \{String, Number, ? super Integer\}

**Example 2**
Creating type parameters for inter-dependent classes:

\[
estimate(I_{[a][Ret(A.m)]}) = \{E \text{ extends Object}\} \quad (\text{type parameter})
\]

This implies that \([Ret(A.m)]\) must be a type parameter too

- If \([Ret(A.m)]\) is a non-parameter, so is \(I_{[a][Ret(A.m)]}\)
- E.g., if \([Ret(A.m)] = \text{String}\), then \(I_{[a][Ret(A.m)]} = \text{String}\)
  - because context is irrelevant for non-parametric types
Type Constraint Solving: pseudo-code

1 Initialize estimates
2 while (not every estimate is singleton):
3   repeat for each \( a \leq b \) until fix-point:
4     remove from estimate(\( a \)) all types that are not a subtype of a type in estimate(\( b \))
5     remove from estimate(\( b \)) all types that are not a supertype of a type in estimate(\( a \))
6   find variable \( v \) with non-singleton estimate
7   select a type for \( v \)
Heuristics for non-deterministic choice

...  

6  find variable v with non-singleton estimate  
7  select a type for v  

Step 7 uses heuristics:  
- preserves type erasure (to preserve behavior)  
- prefer wildcard types  
- prefer type parameters, if this propagates to return types  

Result: better solutions  
- eliminates more casts  
- more closely matches JDK style
Type Estimates

- Estimates are finite sets containing:
  - simple types: String, MyClass[]
  - type parameters: E extends Number
    - pre-existing or created during solving
  - wildcard types: ? super Date

- Estimate initialization:
  - Program elements from JDK have fixed types
  - User may restrict choices by selecting a set of references to parameterize – new type parameters
  - Other variables are initialized to set of all types
Optimization: Symbolic Representation of Estimates

- Symbolic representation, e.g.,
  - Sup(C)
    - set of all supertypes of type C
  - Sub(\(? extends Number\))
    - set of all subtypes of type \(? extends Number\)

- Efficient operations
  - Creation, e.g., Sup(Intersect(Sub(C), Sup(D)))
  - Simplifications, e.g.: Sub(Sub(D)) → Sub(D)

- Symbolic representation expanded only for explicit enumeration
Evaluation

- **Correctness:** program behavior is unchanged
  - We verified erasure preservation
- **Usability:** tool reduces work
  - We measured tool run-time and counted source edits
- **Accuracy:** result is close to what a human would do
  - We measured difference between manual and automatic parameterization
  - When manual parameterization was unavailable, we asked developers to examine results
Subject Programs

- Parameterized 16000+ LOC, largest class 1303 LOC

- Generic libraries (total more than 150kLOC)
  - Apache collections
  - jPaul
  - jUtil
  - java.util.concurrent
  - Amadeus
  - DSA

- Non-generic libraries
  - ANTLR
  - Eclipse
Correctness

- Correctness is a strict prerequisite for migration
- Preserving erasure guarantees correctness
  - Compiled bytecode remains the same
  - Generic type information unavailable on runtime
- Previous approaches (e.g., vonDincklage’04) did not achieve correctness
  - Bytecode modified
  - Method overriding relationships broken – affects method dispatch
- We verified erasure preservation
Usability

- **Performance:**
  - manual: “several weeks of work” (Apache developer)
  - automated: less than 3 seconds per class

- **Source modifications:**
  - manual: 1655 source edits (9% sub-optimal results)
  - automated: tool finds all edits (4% sub-optimal results)
Accuracy on Generic Libraries

- **Experiments:**
  - We removed generic types from source
  - Our tool reconstructed them
  - We compared manual parameterization with tool results

- **Results:**
  - In 87% of cases, computed results equal to manual
  - In 4% of cases, computed results are worse
    - too many type parameters (2 vs. 1) in two cases
    - reference left un-parameterized
  - In 9% of cases, computed results are better
    - wildcard inferred – improved flexibility of use
    - type parameter inferred in inner class – allows removing casts
    - confirmed by developers (Doug Lea, Alexandru Salcianu)
Accuracy on non-Generic Libraries

- We used the tool to infer generic types
- We asked developers to examine results
  - Developers found less than 1% of edits that they considered sub-optimal
  - “[results] look pretty good” (ANTLR developer)
  - “good and useful for code migration to Java 5.0” (Eclipse developer)
Future work: Data-independence for model checking

- Discover data-independent classes (manipulate data without examining it)
- Apply to software model-checking:
  - Environment generation
    - No need to exercise all inputs if values are ignored
  - State-matching abstraction
    - No need to store ignored portion of state
Conclusions

- Automatic parameterization of Java classes
- Correct: preserves behavior for clients
- Infers wildcards – increases flexibility of solution
- Evaluated on real library code:
  - 96% of results better or equal to manual parameterization
  - Fast – saves a lot of manual work
  - “Are there any doubts that such a refactoring would be useful? “ (Eclipse developer)