Java Library Specialization

The cJ approach

Yannis Smaragdakis
University of Massachusetts, Amherst

(research jointly with Shan Shan Huang, David Zook)

The Problem

- We want to safely specialize reusable code collections (e.g., class libraries)
  - ```
    class List<E> {
    <if it is an expandable list>
      boolean add(E e) {...}
    <end if>
    }
    ```
- This allows many good things
  - static checking of calling unsupported operations
  - possible optimization (removing code, special handling)
- Like meta-programming where the generator only has “if”, but no loops
    ```

Current Approaches

- Java: no static specialization, only runtime checking
  - ```
    interface List<E> {
      public E add(int index, E element)
      throws UnsupportedOperationException;
    }
    ```
- Alternative: manually maintaining an exponential number of related types
  - List, ModifiableList, ExpandableList, ShrinkableList, ModifiableShrinkableList, ModifiableShrinkableExpandableList, ...
  - alternative explicitly rejected in current Java libraries (notably, the Java Collections Framework)
    ```

Current Approaches

- C/C++: unsafe specialization, ***once per entire compilation unit!***
  - ```
    template <class E>
    class List {
    #ifdef Expandable
      bool add(const E &e) {...}
    #endif
    };
    class Client {
      void meth() {
        List<string> ls;
        ls.add("John");
      }
    };
    ```
- Low-level power:
  - can make any code fragment conditional
  - code in unsatisfied conditions not even parsed
    ```

Idea #1

- Let's allow arbitrary propositions a la C but try to ensure their safety
  - `#define Prop`
  - `#ifdef Prop { ... } #endif`
  - `#ifndef Prop { ... } #endif`

Safe Ifdefs

- Complexity builds up
  - ```
    #ifdef A { ... int i; ... } #endif
    #ifndef B { ... int i; ... } #endif
    ```
    - i is defined under the condition "A or not B"
  - ```
    #ifdef A { #ifdef B
      { ... int i; ... }
    #endif } #endif
    ```
    - i is defined under the condition "A and B"

Issues with Safe Ifdefs

- In general, can form arbitrary propositional clauses and may need to check their validity
  - NP-hard
  - type system needs integration with SAT-solver
  - need to consider exponential number of conditions
- This may be fine, but also language is artificial and not too expressive
  - programmer decides meaning of propositions

Idea #2: the cJ approach

- Define conditions using expressible type concepts
  - Java used as context for examples
  - #ifdef becomes <*cond*>?
  - Can define conditionally fields and entire methods
    - code fragments at the statement level also easy to support

cJ Example

- ```
  class C<X> {
    X xRef;
    ...
    <X extends DataSource>?
    void store() {... xRef.getConnection() ...}
  }
  ```
- immediate benefit: types maintain the appropriate conditions
  - we know **`xRef`** supports **`getConnection`** because of the type condition
  ```

cJ and Java Collections Framework

- Solves conciseness/safety issues of the Java Collections Framework

```
interface Collection<E, M> { ...
  <M extends VariableSize>?
  boolean add(E e);
}
interface List<E, M> extends Collection<E, M> {
  ...
  <M extends Modifiable>?
  E set(int index, E element);
}
```

Abstraction

- For this to really be general, need abstraction
- Two ways to abstract in OO languages:
  - be able to handle all objects that support same methods, even if they are from different classes
    - subtyping via interfaces
  - be able to handle all conditional instantiations of a class that support at least some functionality, without knowing exactly what
    - variance

Abstraction #1: Interfaces

- ```
  class C<X>
  <X extends DataSource>? implements Storable {
    X xRef;
    ...
    <X extends DataSource>?
    void store() {... xRef.getConnection() ...}
  }
  ```
- This should make you uneasy:
  - we conditionally change something that can affect other conditions
  - also, subtyping conditions can be recursive
  ```

Example

```
class C<X> extends D<C<C<X>>> {}
class D<Y> <Y extends E<C<Y>>>? extends E<Y> {}
class E<Z> {}
```

- consider checking
  - `C<A> extends E<C<C<A>>>`
  - this requires
    `C<C<A>> extends E<C<C<C<A>>>`
  - this requires
    `C<C<C<A>>> extends E<C<C<C<C<A>>>>`
  - …

Conditional Subtyping

- One more nudge and we can emulate a Turing machine in the type system!
  - which means our safety check is undecidable
  -
    ```
    class Add<X,Y>
    <X extends Succ<Z>>? extends Succ<Add<Z,Y>>
    {}
    ```
- We worked hard to make cJ decidable
  - same issue for any kind of specialization mechanism

Abstraction #2: Variance

- "I want my code to work with all list objects that have a set method (i.e., are modifiable), regardless of whether they are expandable, shrinkable, etc."
- In Java, this kind of abstraction is done with "variance" or "wildcards"
- cJ supports this, but it opens a new can of worms

```
List<Dog,? extends Modifiable> modList;
...
modList.set(1, new Dog("Sparky")); // OK
modList.add(new Dog("Spotty")); // NO!
```

Bottom Line

- Safe library specialization is very useful in practice
  - concise expression of many different combinations
- But not easy to really do and integrate in type system
  - issues of power of conditions, abstraction over them
    - too easy to fall off the deep end

Mission Statement

- WG 2.11 can play a key role in defining such mechanisms!
  - this is meta-programming at its finest
  - modest (only “if”, no “for”) yet still quite hard!
  - real need in practice

Many More Issues

- I concentrated on what is expressible and checkable
- Ignored several other issues
  - negative conditions, disjunctions
  - how to compile
    - keep all combinations, vs. remove unused code
  - conditions used for low-level solutions
    - e.g., platform specific code