Verification of Featherweight Java Programs via Transformation to Higher-order Functional Programs with Recursive Data Types

Hiroki Sakamoto Hiroshi Unno (University of Tsukuba)

1. Our ultimate goal

Precise and fully-automated verification of Java programs

This poster :
Proposes a precise and semi-automated verification method for assertion safety of programs in **Featherweight Java (FJ) [Igarashi et al. '01]** extended with **booleans, integers, conditional branches**, and **assertions**

2. Challenges

Assertion safety verification for ext. FJ often requires
- **Context-sensitive** analysis of dynamic dispatch
- **Path-sensitive** analysis of conditional branches

FJ program P_{list}

```
class List {
   Int length() { return assert false; }
}
class Nil extends List {
   Int length() { return 0; }
}
class Cons extends List {
   Int hd; List tl;
   Cons(Int hd, List tl) { this.hd=hd; this.tl=tl;  }
   Int length() {
      return 1 + this.tl.length();
   }
}
class Main {
   List mk_n_list(Int n) {
      return if n=0 then new Nil()
                    else new Cons(n, this.mk_n_list(n-1));
   }
}
main() {
   return assert (new Main().mk_n_list(3).length() = 3);
}
```

Behaves differently depending on **the context** of length

Behaves differently depending on the run time **value of n**

2. Approach

Extended FJ program

Our verifier

FJ to ML translator

ML program
with recursive data structures and
higher-order functions

Existing ML program verifier

Result: Valid / Invalid / Unknown

We use ***Refinement Caml*** (RCaml) [Unno et al. '09…] because it can perform path-sensitive analysis of conditional branches

3. Translation from FJ to ML

(This translation is inspired by [Kobayashi and Igarashi '13])

FJ program P_{list}

Simulate dynamic dispatch using higher-order functions and recursive data types

(Conjecture)
$FJ\ program \not\to^*_{FJ} assert\ false \Leftrightarrow$
$ML\ program \not\to^*_{ML} assert\ false$

ML program

```
type obj = Obj of (int -> obj-> obj) * (obj -> bool)
```

Encode objects as recursive data structures

```
let length_Main this = assert false
let mk_n_list_Main n this = ...
let send_mk_n_list (Obj (m1,m2)) = m1 (Obj(m1,m2))
...
let main () =
  assert (send_length
            send_mk_n_list_Main
              (Obj(mk_n_list_Main 3, length_Main)) = 3)
```

To **simulate dynamic dispatch**, extract and call a function from the recursive data structure that encodes an object

4. Limitation of RCaml and our solution

Limitation

RCaml **cannot** handle recursive data structures **context-sensitively** if they contain **functions** (otherwise, it can)

Solution

Extend the FJ-to-ML translation to insert context information to the **recursive data structures** that encode objects

```
type obj = Obj of cls * (int -> obj-> obj) * cls * (obj -> bool)
 and cls = Mk_n_list_Main of int | Length_Cons | ...
let send_mk_n_list (Obj (cm1,m1,cm2,m2)) = m2 (Obj(cm1,m1,cm2,m2))
...
let main () =
  assert (send_length
            send_mk_n_list_Main
              (Obj(Mk_n_list_Main(3),mk_n_list_Main 3,
                   Length_Main, length_Main)) = 3)
```

5. Experiments (with demonstrations)

Example 1

```
class List { Int contain() { return assert false; } }
class Nil extends List { Int contain() { return false; } }
class Cons extends List {
   Int hd; List tl;
   Cons(Int hd, List tl) { this.hd=hd; this.tl=tl;}
   Int contain(Int n) {
      return if key = hd then true else this.tl.contain(key);
   }
}
main() {
   return assert (new Cons(1, new Cons(2, new Nil()))).contain(2);
}
```

Result: Safe (+ refinement types as certificate)
➡ Achieved fully-automated context-sensitive verification

Example 2

FJ program P_{list} + Annotation for recursive data types

Result: Safe (+ refinement types as certificate)
➡ Achieved path- and context-sensitive verification with small annotation burden

6. Future work

- Improve RCaml (improve performance, etc.)
- Improve the translation (deal with other features such as multithreaded programs, assignment, exceptions, etc.)