Behavioral Type Inference for Concurrent Object-Oriented Languages

Cláudio Vasconcelos

Master thesis presentation

NOVA-LINCS and Dep. de Informática, FCT
Universidade NOVA de Lisboa

Advised by Professor António Ravara

November 28, 2016

Problem

Large-scale software systems rely on communication protocols

Mainstream programming languages do not cope with them:

- provide (some) data-type safety
- fail to give static support to stateful behaviour

Problem

Large-scale software systems rely on communication protocols

Mainstream programming languages do not cope with them:

- provide (some) data-type safety
- fail to give static support to stateful behaviour

Most (static) approaches to code correctness not well suited

- Usual safety guarantees not enough:
  An application that allows to open and close a file, but not read or write, is safe but not useful
- Liveness is hard to statically verify (sometimes not possible)

State-of-the-art

Several approaches to analyse code

- Deductive proof systems
- Model-checkers / Abstract Interpretation
- Type systems

We aim at an *automatic, decidable, tool, coping with safety and (weak) liveness properties* (like protocol completion)

Our language of choice

The Mool Language: `http://gloss.di.fc.ul.pt/mool`

- Small, rigorously defined, Java-like, and object-oriented
- Associates with each class a behavioural type
- Types express valid sequences of method calls
- Type system ensures statically safe usage of objects' protocols

Our language of choice

We will work with a new version of the language

- Aspects where the language was incorrect or too restricting were revised
  - Concurrency
  - Use of *null* as a value
  - Shared usages
  - ...
- Assertions were added
  - Boolean expressions on the state of fields and parameters

The problem we address

Observations

- Specifying objects intended behaviour as state machines is natural, but may be demanding and not easy to get right
- Stating, for each method, the required and ensured state of fields and parameters may be easier
- Assertions are part of Java since 2006

The problem we address

Observations

- Specifying objects intended behaviour as state machines is natural, but may be demanding and not easy to get right
- Stating, for each method, the required and ensured state of fields and parameters may be easier
- Assertions are part of Java since 2006

Question

Can we get behavioural types from code with assertions?

The envisaged contribution

Infer, from O.-O. code with assertions, behavioural (class) types ensuring safe interoperability

The envisaged contribution

Infer, from O.-O. code with assertions, behavioural (class) types ensuring safe interoperability

A type inference system: given a program

- either fails: the code is *not well-typed* (in the standard sense)

The envisaged contribution

Infer, from O.-O. code with assertions, behavioural (class) types ensuring safe interoperability

A type inference system: given a program

- either fails: the code is *not well-typed* (in the standard sense) or it may produce a run-time error due to calling methods in an *incorrect order*;

The envisaged contribution

Infer, from O.-O. code with assertions, behavioural (class) types ensuring safe interoperability

A type inference system: given a program

- either fails: the code is *not well-typed* (in the standard sense) or it may produce a run-time error due to calling methods in an *incorrect order*;
- or returns a new version of the code with the classes annotated with behavioural types, ensuring *object interoperability*.

Usage inference process

The process is composed by three stages

1. Typestate generation

Usage inference process

The process is composed by three stages

1. Typestate generation
2. Usage generation

Usage inference process

The process is composed by three stages

1. Typestate generation
2. Usage generation
3. Object usage state inference

Usage inference process

Input example - *File* class

```
class File {

    int linesInFile; int linesRead;
    boolean closed; boolean lineInBuffer; boolean isEof;

    //@invariant linesRead >= 0 && linesRead <= linesInFile;
    //@initial linesRead == 0 && linesInFile == 5
              && !closed && !lineInBuffer && !isEof;
    void File() { ... }

    ...
}
```

Usage inference process

Input example - *File* class

```
class File {

        ...

        //@requires linesRead < linesInFile && !closed && lineInBuffer
                    && !isEof;
        //@ensures linesRead + 1 <= linesInFile
                    && !closed && !lineInBuffer && !isEof;
        string read() { ... }

        ...
}
```

Usage inference process

Input example - *File* class

```
class File {

        ...

        //@requires linesRead <= linesInFile && !closed && !lineInBuffer
                    && !isEof;
        //@ensures (linesRead == linesInFile -> !lineInBuffer && isEof)
                    && !closed;
        boolean eof() { ... }

        ...
}
```

Usage inference process

Input example - *File* class

```
class File {

        ...

        //@requires linesRead == linesInFile && isEof && !closed;
        //@ensures linesRead == linesInFile && isEof && closed;
        void close() {
                closed = true;
        }
}
```

Stage 1 - Typestate generation

Based on the algorithm presented in

G. D. Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. "Enabledness-based Program Abstractions for Behavior Validation". In: ACM Trans. Softw. Eng. Methodol. 22.3 (July 2013), 25:1–25:46. ISSN: 1049-331X

- Enabledness-preserving automata extraction from source code equipped with assertions
- We modified the algorithm to apply it in code with preconditions and postconditions
- We then use a SMT solver to perform the validity checks

Stage 1 - Typestate generation

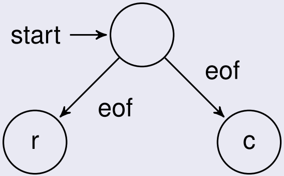
Choice-based transitions

- Mool allows transitions to have, at most, two target states, with these being based on choice

Stage 1 - Typestate generation

Choice-based transitions

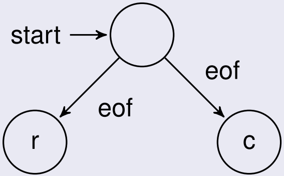
- Mool allows transitions to have, at most, two target states, with these being based on choice
- The algorithm allows nondeterministic transitions with multiple target states



Stage 1 - Typestate generation

Choice-based transitions

- Mool allows transitions to have, at most, two target states, with these being based on choice
- The algorithm allows nondeterministic transitions with multiple target states



Mool depends on the result of *eof* to know which state to transit to

Stage 1 - Typestate generation

The original algorithm produces the following transition relation

$$\delta(S_a, m) = S_b$$

Meaning that, when executing the method m in state S_a, the object transits to state S_b.

Stage 1 - Typestate generation

The original algorithm produces the following transition relation

$$\delta(S_a, m) = S_b$$

Meaning that, when executing the method m in state S_a, the object transits to state S_b.
In a nondeterministic context it is possible to have the following transition relation:

$$(S_a, m, S_{b_1}) \in \delta$$
$$(S_a, m, S_{b_2}) \in \delta$$

Where S_{b_1} and S_{b_2} are different states.

Stage 1 - Typestate generation

In our version the transition relation is *a function*, defined as follows

$$\delta(S_a, m, c) = S_b$$

Where c is the choice the transition corresponds to.

Stage 1 - Typestate generation

In our version the transition relation is *a function*, defined as follows

$$\delta(S_a, m, c) = S_b$$

Where c is the choice the transition corresponds to.
In the previous nondeterministic example, the transition relation could be:

$$\delta(S_a, m, \mathit{false}) = S_{b_1}$$

$$\delta(S_a, m, \mathit{true}) = S_{b_2}$$

Meaning that, if m returns *true* the object transits to state S_{b_2}, otherwise it transits to state S_{b_1}.

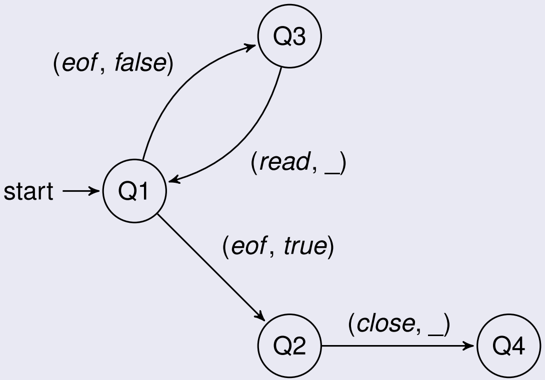
Stage 1 - Typestate generation

Transition relation extension

- To do this, the algorithm needs to know which state corresponds to both true and false branches
- The post-condition must specify the object state in both choices

Stage 1 - Typestate generation

Output example - Typestate of the *File* class



$Q1 = \{eof\}$

$Q3 = \{read\}$

$Q2 = \{close\}$

$Q4 = \{\}$

Stage 2 - Usage generation

Based on the algorithm presented in

P. Collingbourne and P. H. J. Kelly. “Inference of Session Types From Control Flow”. In: Electron. Notes Theor. Comput. Sci. 238.6 (June 2010), pp. 15–40. ISSN: 1571-0661.

- Session type inference for C
- We are only interested in stage three of the algorithm: Graph Simplification and Translation

Stage 2 - Usage generation

Choice-based transitions

- The translation function does not deal with choice-based transitions
- We extend it so that it translates these type of transitions into variant types

$$\begin{array}{l} \delta(S_a, m, \mathit{false}) = (S_{b_1}) \\ \delta(S_a, m, \mathit{true}) = (S_{b_2}) \end{array} \implies S_a = \{m; < S_{b2} \ + \ S_{b1} >\}$$

Stage 2 - Usage generation

Shared state of usage states

- An usage state can be defined as shared or non-shared
- We extend the translation function to infer the shared status of an usage state
- An usage state is considered shared if:
  - Its completely recursive

$$S_a = \{a; S_a + b; S_a\}$$

  - It only transits to equivalent usage states

$$S_a = \{a; S_a + b; S_b\} \qquad S_b = \{a; S_b + b; S_a\}$$

Stage 2 - Usage generation

Output example - Usage of the *File* class

```
usage lin{File; Q1} where
          Q1 = lin{eof; < Q2 + Q3 >}
          Q3 = lin{read; Q1}
          Q2 = lin{close; end};
```

Stage 3 - Object usage state inference

Specifying the initial usage state of fields

- Mool offers the possibility of indicating the state of the usage of a object in its declaration:

```
class FileReader {
        ...
        File[Q3] file;
        ...
}
```

Stage 3 - Object usage state inference

Specifying the usage state of parameters

- Programmers can also define the usage state of parameters:

```
void FileReader(File[Q3] f) {
        file = f;
        ...
}
```

Stage 3 - Object usage state inference

Specifying the usage state of a returned object

- It is also possible to define the usage state of an object returned by a method:

```
File[Q3] getFileToRead() {
        file;
}
```

Stage 3 - Object usage state inference

Usage state declaration in the context of our work

- We do not expect the programmer to know beforehand the states that will compose the generated usage
- But we can expect the programmer to know the state of an object when initialised

Stage 3 - Object usage state inference

Using preconditions to specify the state of an object

- It is possible to express the expected state of the instance received as a parameter in the precondition of the method:

```
class FileReader {
        File file; ...

        //@invariant counter >= 0;
        //@requires f != null && !f.eof();
        //@initial counter == 0 && new File() && !isEof;
        void FileReader(File f) { file = f; ... }
        ...
}
```

Stage 3 - Object usage state inference

Using preconditions to specify the state of an object

- It is possible to express the expected state of the instance received as a parameter in the precondition of the method:

```
class FileReader {
    File file; ...

    //@invariant counter >= 0;
    //@requires f != null && !f.eof();
    //@initial counter == 0 && new File() && !isEof;
    void FileReader(File[Q3] f) { file = f; ... }
}
```

Stage 3 - Object usage state inference

Using preconditions to specify the state of an object

- It is also possible to know the initial state of an object when it is initialized:

```
class FileReader {
        File[Q3] file; ...

        //@invariant counter >= 0;
        //@requires f != null && !f.eof();
        //@initial counter == 0 && new File() && !isEof;
        void FileReader(File[Q3] f) { file = f; ... }
}
```

Stage 3 - Object usage state inference

Using postconditions to specify the state of a returned object

- This usage state can be inferred using the method postcondition to express the expected state of the returned object:

```
//@requires !isEof;
//@ensures !isEof && !file.eof();
File getFileToRead() {
        file;
}
```

Stage 3 - Object usage state inference

Using postconditions to specify the state of a returned object

- This usage state can be inferred using the method postcondition to express the expected state of the returned object:

```
//@requires !isEof;
//@ensures !isEof && !file.eof();
File[Q3] getFileToRead() {
        file;
}
```

Stage 3 - Object usage state inference

Algorithm steps

1. Determines the usage state of every parameter using the preconditions
2. Analyses the code of the method and:
   - For every initialization, sets the usage state of the initialized variable with the usage state of the value
   - For every call, changes the current usage state of the object the method was called
3. Determines the usage state of the return type using the postconditions

Conclusion

Work summary

- In short, the algorithm does the following:
  - Generates typestates from the code equipped with assertions
  - Translates the typestates into usage types
  - Infers the correct usage state for each declared object

Conclusion

Work summary

- In short, the algorithm does the following:
  - Generates typestates from the code equipped with assertions
  - Translates the typestates into usage types
  - Infers the correct usage state for each declared object
- We implemented the algorithm:
  `http://usinfer.sourceforge.net/`

Conclusion

Assertions

- If the assertions are not correct, one of two things might happen:
  - The tool fails to infer the usage types
  - The tool produces usage types that may allow unwanted behaviour
    In that case, typechecking the code with such usage may fail, if it allows erroneous behaviour

Conclusion

Assertions

- If the assertions are not correct, one of two things might happen:
  - The tool fails to infer the usage types
  - The tool produces usage types that may allow unwanted behaviour
    In that case, typechecking the code with such usage may fail, if it allows erroneous behaviour
- Thus, the algorithm can also be used to verify the correctness of the assertions

Future work

Correctness

We want to:

- State the intended results
- Prove the algorithm sound

Future work

Correctness

We want to:

- State the intended results
- Prove the algorithm sound

Assertions

- Programming with assertions is also hard
- We want to infer them as automatically as possible

Thank you