Behavioural Type Inference for Object-Oriented Languages

Ongoing work

António Ravara
Work with Adrian Francalanza, Hans Hüttel and Mario Bravetti

NOVA LINCS and DI-FCT, Univ NOVA de Lisboa

April 17, 2015

The BETTY vision

Goal

- Behavioural types: basis for communication-intensive distributed systems.
- Aim: certified software for global services (by automatically checking behavioural properties of communicating systems).
- To encourage the industrial adoption of advanced programming languages and tools.

Present situation

- Widely used programming languages still give poor support to ensure protocol compatibility.
- Component-based software development deals with legacy code, assembling new applications from code-bases.
- Some languages (BICA, MOOL, Plaid, SJ) already do behavioural type-checking.

The BETTY vision

Goal

- Behavioural types: basis for communication-intensive distributed systems.
- Aim: certified software for global services (by automatically checking behavioural properties of communicating systems).
- To encourage the industrial adoption of advanced programming languages and tools.

Present situation

- Widely used programming languages still give poor support to ensure protocol compatibility.
- Component-based software development deals with legacy code, assembling new applications from code-bases.
- Some languages (BICA, MOOL, Plaid, SJ) already do behavioural type-checking.

The BETTY vision

Goal

- Behavioural types: basis for communication-intensive distributed systems.
- Aim: certified software for global services (by automatically checking behavioural properties of communicating systems).
- To encourage the industrial adoption of advanced programming languages and tools.

Present situation

- Widely used programming languages still give poor support to ensure protocol compatibility.
- Component-based software development deals with legacy code, assembling new applications from code-bases.
- Some languages (BICA, MOOL, Plaid, SJ) already do behavioural type-checking.

The BETTY vision

Goal

- Behavioural types: basis for communication-intensive distributed systems.
- Aim: certified software for global services (by automatically checking behavioural properties of communicating systems).
- To encourage the industrial adoption of advanced programming languages and tools.

Present situation

- Widely used programming languages still give poor support to ensure protocol compatibility.
- Component-based software development deals with legacy code, assembling new applications from code-bases.
- Some languages (BICA, MOOL, Plaid, SJ) already do behavioural type-checking.

The BETTY vision

Goal

- Behavioural types: basis for communication-intensive distributed systems.
- Aim: certified software for global services (by automatically checking behavioural properties of communicating systems).
- To encourage the industrial adoption of advanced programming languages and tools.

Present situation

- Widely used programming languages still give poor support to ensure protocol compatibility.
- Component-based software development deals with legacy code, assembling new applications from code-bases.
- Some languages (BICA, MOOL, Plaid, SJ) already do behavioural type-checking.

The BETTY vision

Goal

- Behavioural types: basis for communication-intensive distributed systems.
- Aim: certified software for global services (by automatically checking behavioural properties of communicating systems).
- To encourage the industrial adoption of advanced programming languages and tools.

Present situation

- Widely used programming languages still give poor support to ensure protocol compatibility.
- Component-based software development deals with legacy code, assembling new applications from code-bases.
- Some languages (BICA, MOOL, Plaid, SJ) already do behavioural type-checking.

The BETTY vision

Goal

- Behavioural types: basis for communication-intensive distributed systems.
- Aim: certified software for global services (by automatically checking behavioural properties of communicating systems).
- To encourage the industrial adoption of advanced programming languages and tools.

Present situation

- Widely used programming languages still give poor support to ensure protocol compatibility.
- Component-based software development deals with legacy code, assembling new applications from code-bases.
- Some languages (BICA, MOOL, Plaid, SJ) already do behavioural type-checking.

Classical motivational example

- A file system – no *read* before *open* – or an iterator – no *next* before *hasNext*.
- Valid method usage sequences require a protocol, even in a sequential setting.
- One needs to delimit the set of admissible call sequences: *protocols as (class) types*.

safe use of a file: the behavioural type

Classical motivational example

- A file system – no *read* before *open* – or an iterator – no *next* before *hasNext*.
- Valid method usage sequences require a protocol, even in a sequential setting.
- One needs to delimit the set of admissible call sequences: *protocols as (class) types*.

safe use of a file: the behavioural type

Classical motivational example

- A file system – no *read* before *open* – or an iterator – no *next* before *hasNext*.
- Valid method usage sequences require a protocol, even in a sequential setting.
- One needs to delimit the set of admissible call sequences: *protocols as (class) types*.

safe use of a file: the behavioural type

Classical motivational example

- A file system – no *read* before *open* – or an iterator – no *next* before *hasNext*.
- Valid method usage sequences require a protocol, even in a sequential setting.
- One needs to delimit the set of admissible call sequences: *protocols as (class) types.*

safe use of a file: the behavioural type

Classical motivational example

- A file system – no *read* before *open* – or an iterator – no *next* before *hasNext*.
- Valid method usage sequences require a protocol, even in a sequential setting.
- One needs to delimit the set of admissible call sequences: *protocols as (class) types*.

safe use of a file: the behavioural type

Classical motivational example

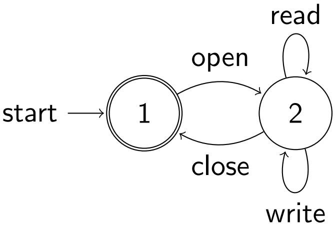
- A file system – no *read* before *open* – or an iterator – no *next* before *hasNext*.
- Valid method usage sequences require a protocol, even in a sequential setting.
- One needs to delimit the set of admissible call sequences: *protocols as (class) types*.

safe use of a file: the behavioural type

Classical motivational example

- A file system – no *read* before *open* – or an iterator – no *next* before *hasNext*.
- Valid method usage sequences require a protocol, even in a sequential setting.
- One needs to delimit the set of admissible call sequences: *protocols as (class) types*.

safe use of a file: the behavioural type



The envisaged contribution

Our Goal

To *infer*, from "standard" (concurrent) O.-O. code, behavioural (class) types ensuring safe interoperability.

A type inference system

A tool that takes a program written in a subset of Java and

- 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*.

The envisaged contribution

Our Goal

To *infer*, from "standard" (concurrent) O.-O. code, behavioural (class) types ensuring safe interoperability.

A type inference system

A tool that takes a program written in a subset of Java and

- 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*.

The envisaged contribution

Our Goal

To *infer*, from "standard" (concurrent) O.-O. code, behavioural (class) types ensuring safe interoperability.

A type inference system

A tool that takes a program written in a subset of Java and

- 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*.

The envisaged contribution

Our Goal

To *infer*, from "standard" (concurrent) O.-O. code, behavioural (class) types ensuring safe interoperability.

A type inference system

A tool that takes a program written in a subset of Java and

- 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*.

The envisaged contribution

Our Goal

To *infer*, from "standard" (concurrent) O.-O. code, behavioural (class) types ensuring safe interoperability.

A type inference system

A tool that takes a program written in a subset of Java and

- 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*.

The envisaged contribution

Our Goal

To *infer*, from "standard" (concurrent) O.-O. code, behavioural (class) types ensuring safe interoperability.

A type inference system

A tool that takes a program written in a subset of Java and

- 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*.

Type-safety

What can go wrong?

- Why would a sequence of method calls produce an error?
  One may get a null pointer exception – calling *read* before *open* amounts to read a un-instantiated variable.

  ```
  Class F{
    String s;
  // @req null(s) @ens !null(s)
    void open(){s = "";}
  // @req !null(s) @ens !null(s)
    String read(){return s;}
  }
  ```
- Why would a program get stuck?
  An object may not complete its protocol – a caller may wait forever for the result of a callee that does not return.
  String read(){*if null*(*s*) *return s*; }

Type-safety

What can go wrong?

- Why would a sequence of method calls produce an error?
One may get a null pointer exception – calling *read* before *open* amounts to read a un-instantiated variable.

```
Class F{
  String s;
// @req null(s) @ens !null(s)
  void open(){s = "";}
// @req !null(s) @ens !null(s)
  String read(){return s;}
}
```

- Why would a program get stuck?
An object may not complete its protocol – a caller may wait forever for the result of a callee that does not return.
String read(){if null(s) return s;}

Type-safety

What can go wrong?

- Why would a sequence of method calls produce an error?
  One may get a null pointer exception – calling *read* before *open* amounts to read a un-instantiated variable.
  Class F{
  String s;
  // @*req null*(*s*) @*ens* !*null*(*s*)
  void open(){*s* = ""; }
  // @*req* !*null*(*s*) @*ens* !*null*(*s*)
  String read(){*return s*; }
  }
- Why would a program get stuck?
  An object may not complete its protocol – a caller may wait forever for the result of a callee that does not return.
  String read(){*if null*(*s*) *return s*; }

Type-safety

What can go wrong?

- Why would a sequence of method calls produce an error?
  One may get a null pointer exception – calling *read* before *open* amounts to read a un-instantiated variable.

  ```
  Class F{
    String s;
  // @req null(s) @ens !null(s)
    void open(){s = "";}
  // @req !null(s) @ens !null(s)
    String read(){return s;}
  }
  ```

- Why would a program get stuck?
  An object may not complete its protocol – a caller may wait forever for the result of a callee that does not return.
  String read(){if null(s) return s;}

Type-safety

What can go wrong?

- Why would a sequence of method calls produce an error?
  One may get a null pointer exception – calling *read* before *open* amounts to read a un-instantiated variable.

  ```
  Class F{
    String s;
  // @req null(s) @ens !null(s)
    void open(){s = "";}
  // @req !null(s) @ens !null(s)
    String read(){return s;}
  }
  ```

- Why would a program get stuck?
  An object may not complete its protocol – a caller may wait forever for the result of a callee that does not return.
  String read(){if null(s) return s;}

The work plan

Type-safety

- Absence of (null pointer) exceptions.
- The protocol of critical resources is fully executed.

Procedure

1. Infer pre- and post-conditions for each method.
2. Generate a finite-state representation of all possible safe sequences of methods calls, one for each class – its usage.
3. Since the availability of some method may depend on the return value of the previous method, the usage language should support both external and internal choice.
4. Type-check the main class to verify correct class usage.

The work plan

Type-safety

- Absence of (null pointer) exceptions.
- The protocol of critical resources is fully executed.

Procedure

1. Infer pre- and post-conditions for each method.
2. Generate a finite-state representation of all possible safe sequences of methods calls, one for each class – its usage.
3. Since the availability of some method may depend on the return value of the previous method, the usage language should support both external and internal choice.
4. Type-check the main class to verify correct class usage.

The work plan

Type-safety

- Absence of (null pointer) exceptions.
- The protocol of critical resources is fully executed.

Procedure

1. Infer pre- and post-conditions for each method.
2. Generate a finite-state representation of all possible safe sequences of methods calls, one for each class – its usage.
3. Since the availability of some method may depend on the return value of the previous method, the usage language should support both external and internal choice.
4. Type-check the main class to verify correct class usage.

The work plan

Type-safety

- Absence of (null pointer) exceptions.
- The protocol of critical resources is fully executed.

Procedure

1. Infer pre- and post-conditions for each method.
2. Generate a finite-state representation of all possible safe sequences of methods calls, one for each class – its usage.
3. Since the availability of some method may depend on the return value of the previous method, the usage language should support both external and internal choice.
4. Type-check the main class to verify correct class usage.

The work plan

Type-safety

- Absence of (null pointer) exceptions.
- The protocol of critical resources is fully executed.

Procedure

1. Infer pre- and post-conditions for each method.
2. Generate a finite-state representation of all possible safe sequences of methods calls, one for each class – its usage.
3. Since the availability of some method may depend on the return value of the previous method, the usage language should support both external and internal choice.
4. Type-check the main class to verify correct class usage.

The work plan

Type-safety

- Absence of (null pointer) exceptions.
- The protocol of critical resources is fully executed.

Procedure

1. Infer pre- and post-conditions for each method.
2. Generate a finite-state representation of all possible safe sequences of methods calls, one for each class – its usage.
3. Since the availability of some method may depend on the return value of the previous method, the usage language should support both external and internal choice.
4. Type-check the main class to verify correct class usage.

State-of-the-art

Three closely related works, addressing the problem of inferring behavioural interfaces for (sequential) object-oriented code.

- Whaley et al. consider that interfaces are just finite state machines and do not ensure safety.
- Alur et al. start from Whaley's paper and derive history-dependent behavioural interfaces, but do not deal with (inter-object) references.
- Nanda et al. improve on Alur's work deriving the abstract typestate graph representing the transitions of the abstract heap state.

State-of-the-art

Three closely related works, addressing the problem of inferring behavioural interfaces for (sequential) object-oriented code.

- Whaley et al. consider that interfaces are just finite state machines and do not ensure safety.
- Alur et al. start from Whaley's paper and derive history-dependent behavioural interfaces, but do not deal with (inter-object) references.
- Nanda et al. improve on Alur's work deriving the abstract typestate graph representing the transitions of the abstract heap state.

State-of-the-art

Three closely related works, addressing the problem of inferring behavioural interfaces for (sequential) object-oriented code.

- Whaley et al. consider that interfaces are just finite state machines and do not ensure safety.
- Alur et al. start from Whaley's paper and derive history-dependent behavioural interfaces, but do not deal with (inter-object) references.
- Nanda et al. improve on Alur's work deriving the abstract typestate graph representing the transitions of the abstract heap state.

State-of-the-art

Three closely related works, addressing the problem of inferring behavioural interfaces for (sequential) object-oriented code.

- Whaley et al. consider that interfaces are just finite state machines and do not ensure safety.
- Alur et al. start from Whaley's paper and derive history-dependent behavioural interfaces, but do not deal with (inter-object) references.
- Nanda et al. improve on Alur's work deriving the abstract typestate graph representing the transitions of the abstract heap state.

References

Papers

- R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In POPL'05, ACM.
- M. Nanda, C. Grothoff, and S. Chandra. Deriving object type-states in the presence of inter-object references. SIGPLAN Not., 40(10):7796, ACM, 2005.
- J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In ISSTA'02, ACM.

Tools

- BICA – http://gloss.di.fc.ul.pt/bica
- MOOL – http://gloss.di.fc.ul.pt/mool
- Plaid – http://www.cs.cmu.edu/∼aldrich/plaid/
- SJ – http://www.doc.ic.ac.uk/∼rhu/sessionj.html