Behavioural types for memory safety in Mungo

António Ravara
With M. Bravetti (IT), A. Francalanza (MT), H. Hüttel (DK)

NOVA LINCS and Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa
Work partially supported by RISE and COST (EU), and EPSRC (UK)

December 19, 2019

Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233.

Aim of type systems: avoid programs to go wrong

Well-typed programs do not go wrong

R. Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 1978.

Typical wrong situations in OO prevented by type systems

- data-mismatch; method-not-understood
  (mostly) prevent by classical type systems
- flow-mismatch, protocol violations
  prevent by behavioural type systems

Behavioural types in Java-like languages

Associate with a class a description of an object's behaviour.

Class usages

- Declare the admissible sequences of method calls.
- Ensure at compile time that sequences of method calls follow the order declared by usages.
- Objects are linear to prevent interferences unexpectedly changing their state.

Some languages

- Campos and Vasconcelos MOOL – gloss.di.fc.ul.pt/mool
- Gay et al. Mungo – www.dcs.gla.ac.uk/research/mungo/
- Aldrich et al. Plaid – www.cs.cmu.edu/∼aldrich/plaid/
- Hu et al. SJ – www.doc.ic.ac.uk/∼rhu/sessionj.html

Class usage types, by example: a file

```
Class File {

// usage = {open;⟨rec X.
//    {neof;⟨true:{read;X},false:{close;end}⟩},end⟩}

  bool open(){...}

  bool neof(){...}

  String read(){...}

  void close(){...}

}
```

Class usages, by example: a “client” of the file

```
Class FileReader {
// usage = {init;{fetchData;{readData;end}}}
  File f;  String s;
  void init() {
   f = new File();
   s = "";
  }
  void fetchData () {
   if (f.open()) {
     while (f.neof())
      s ⊕= f.read();
     f.close();
  }
  String readData() { s; }
}
```

What can still go wrong?

Null de-referencing

Forget to initialise a field.

```
class FileReader {

// usage = {open;⟨rec X.
// {neof;⟨true:{read;X},false:{close;end}⟩},end⟩}

  File f;

  void init() {
    f.open();
  }

  ...
}
```

What can still go wrong?

Dangling pointers

- Re-initialise fields with linear objects.
- Pass a linear object as an argument of a method call and do not return it even if it did not complete its protocol.

```
class FileReader {
  ...
  void fetchData () {
   if (f.open()) {
     f = new File();
     while (f.neof())
      s ⊕= f.read();
     f.close();
  }
  ...
}
```

Aim of type systems: avoid programs to go wrong

- Forgetting to initialise a field before calling a method on it results in a null-pointer dereferencing, even though you follow the protocol (fidelity).
- If one looses reference to an object in the middle of a protocol, it may not be completed.
- OO type systems (even behavioural ones) DO NOT statically catch this situations.
- Languages (extensions) dealing with the issues require annotations.

What do we want to ensure?

Less wrong well-typed programs, by

- performing null-pointer analysis, and
- better linearity control to ensure protocol completion.

Aim of type systems: avoid programs to go wrong

- Forgetting to initialise a field before calling a method on it results in a null-pointer dereferencing, even though you follow the protocol (fidelity).
- If one looses reference to an object in the middle of a protocol, it may not be completed.
- OO type systems (even behavioural ones) DO NOT statically catch this situations.
- Languages (extensions) dealing with the issues require annotations.

What do we want to ensure?

Less wrong well-typed programs, by

- performing null-pointer analysis, and
- better linearity control to ensure protocol completion.

The Mungo language

$$
\begin{array}{rl}
\text{(usage types)} & u ::= \{m_i; w_i \mid i \in I\} \mid \mu X.u \\
\text{(choice usage types)} & w ::= u \mid \langle l_i; u_i \mid i \in I\rangle \mid X
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(basic types)} & b & ::= & \mathsf{void} \mid \mathsf{bool} \mid L \\
\text{(types)} & t & ::= & b \mid \bot \mid C[u]
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(enum and class declarations)} & D & ::= & \mathsf{enum}\ L\ \{\widetilde{l}\} \mid \mathsf{class}\ C\ \{u; \overrightarrow{F}; \overrightarrow{M}\} \\
\text{(field and method declarations)} & F & ::= & t\ f \qquad M\ ::=\ t\, m(t\, x)\{e\}
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(values)} & v & ::= & \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false} \mid l \mid \mathsf{null} \\
\text{(references)} & r & ::= & x \mid f \\
\text{(expressions)} & e & ::= & v \mid r \mid \mathsf{new}\ C \mid f = e \mid r.m(e) \mid e; e \\
& & & \mid \mathsf{if}\ (e)\ \{e\}\ \mathsf{else}\ \{e\} \mid \mathsf{switch}\ (r.m(e))\ \{\overrightarrow{l:e}\}_{l \in L} \\
& & & \mid \lambda : e \mid \mathsf{continue}\ \lambda
\end{array}
$$

Key features of our type-checking system

Null-pointer analysis

A dedicated type $\perp$ inhabit only by *null* that inhabits no other type.

BTW, the fact that *null* is a value of every type is the problem now with type-safety in Java and Scala!

Protocol completion ensures memory safety

We took a step further, not allowing objects to be dropped.

Efficient type-checking

Following the typestate may lead to check the same method as many times as it is mentioned.

Mungo's type-system should checks methods only once by inferring the usage of fields and parameters.

Key features of our type-checking system

Null-pointer analysis

A dedicated type $\perp$ inhabit only by *null* that inhabits no other type.

BTW, the fact that *null* is a value of every type is the problem now with type-safety in Java and Scala!

Protocol completion ensures memory safety

We took a step further, not allowing objects to be dropped.

Efficient type-checking

Following the typestate may lead to check the same method as many times as it is mentioned.

Mungo's type-system should checks methods only once by inferring the usage of fields and parameters.

Key features of our type-checking system

Null-pointer analysis

A dedicated type $\perp$ inhabit only by *null* that inhabits no other type.

BTW, the fact that *null* is a value of every type is the problem now with type-safety in Java and Scala!

Protocol completion ensures memory safety

We took a step further, not allowing objects to be dropped.

Efficient type-checking

Following the typestate may lead to check the same method as many times as it is mentioned.

Mungo's type-system should checks methods only once by inferring the usage of fields and parameters.

Guarantees (I): data-safety and no null de-referencing

Errors to avoid

$$\frac{s(x) = \text{null}}{\vdash_{\vec{D}} \langle h, (o, s) : env_S, x.m(v) \rangle \longrightarrow_{\text{err}}} \qquad \frac{h(o).f = \text{null}}{\vdash_{\vec{D}} \langle h, (o, s) : env_S, f.m(v) \rangle \longrightarrow_{\text{err}}}$$

How to avoid them

$$\frac{\text{lin}(t)}{\Lambda; \Delta : (o, [x \mapsto t]) \vdash_{\vec{D}} x : t \rhd \Lambda; \Delta : (o, [x \mapsto \bot])}$$

$$\frac{\text{lin}(t)}{\Lambda, o.f \mapsto t; \Delta : (o, S) \vdash_{\vec{D}} f : t \rhd \Lambda, o.f \mapsto \bot; \Delta : (o, S)}$$

$$\text{TCALL} \; \frac{\Lambda \vdash^{o}_{\boldsymbol{D}} e : t \rhd \Lambda', o.r : C[U] \qquad U \xrightarrow{m} W \qquad t' \; m(t\,x) \; \{e'\} \in C.\text{methods}}{\Lambda \vdash^{o}_{\boldsymbol{D}} r.m(e) : t' \rhd \Lambda', o.r : C[W]}$$

Guarantees (II): protocol completion

Protocol fidelity, *i.e.*, objects will *not* not follow the protocols.

Will objects actually follow the protocols?

Avoid loosing reference to objects with incomplete protocols.

$$\frac{\mathsf{lin}(v', h) \wedge v \neq v'}{\vdash_{\vec{D}} \langle h, (o, [x \mapsto v']) : env_S, \mathsf{return}\{v\}\rangle \longrightarrow_{\mathsf{err}}}$$

$$\frac{v \notin \{l_i \mid i \in I\} \qquad h(o).\mathsf{usage} = \langle l_i ; u_i \mid i \in I\rangle}{\vdash_{\vec{D}} \langle h, (o, s) : env_S, \mathsf{return}\{v\}\rangle \longrightarrow_{\mathsf{err}}} \qquad \frac{\mathsf{lin}(v, h)}{\vdash_{\vec{D}} \langle h, env_S, v; e\rangle \longrightarrow_{\mathsf{err}}}$$

$$\frac{\Lambda; \Delta \vdash_{\vec{D}} e : t \triangleright \Lambda'; \Delta' \qquad \Delta' = \Delta'' : (o', S') \qquad \mathsf{terminated}(S')}{\Lambda; \Delta : (o, S) \vdash_{\vec{D}} \mathsf{return}\{e\} : t \triangleright \Lambda'; \Delta' : (o, S)}$$

$$\frac{\Lambda; \Delta \vdash_{\vec{D}} e : t \triangleright \Lambda''; \Delta'' \qquad \neg\mathsf{lin}(t) \qquad \Lambda''; \Delta'' \vdash_{\vec{D}} e' : t' \triangleright \Lambda'; \Delta'}{\Lambda; \Delta \vdash_{\vec{D}} e; e' : t' \triangleright \Lambda'; \Delta'}$$

The real challenge: infer usages

- Specify object protocols when writing OO code is not common practice.
- Is it reasonable to ask programmers to define them?
  How would we deal with legacy code?

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 real challenge: infer usages

- Specify object protocols when writing OO code is not common practice.
- Is it reasonable to ask programmers to define them?
  How would we deal with legacy code?

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 real challenge: infer usages

- Specify object protocols when writing OO code is not common practice.
- Is it reasonable to ask programmers to define them?
  How would we deal with legacy code?

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 OO 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*.

Thanks!

The envisaged contribution

Infer, from OO 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*.

Thanks!

The envisaged contribution

Infer, from OO 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*.

Thanks!

The envisaged contribution

Infer, from OO 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*.

Thanks!

The envisaged contribution

Infer, from OO 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*.

Thanks!