Typestate Inference for Mungo: Algorithm and Implementation

Hans Hüttel – Department of Computer Science, Aalborg University, Denmark – with

Iaroslav Golovanov Mikkel Kettunen Mathias Jakobsen

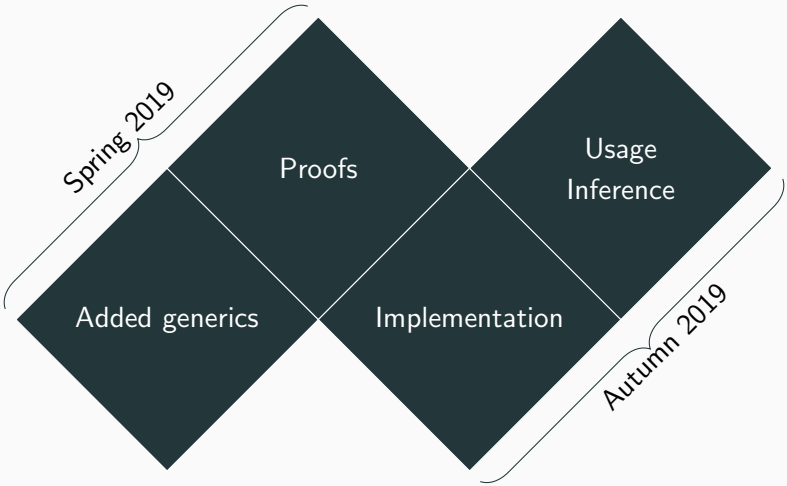
2 March 2020

Mungo

Mungo is a typed Java-like language in the style of Featherweight Java of Igarashi et al. that contains usual object-oriented and imperative constructs.

Mungo is also the associated programming tool developed at Glasgow University by Dardha, Gay et al.

Our contributions to Mungo



Introduction: Typestates

Typestate definitions model dynamic program properties by letting types have content that can give an overapproximation of reachable program states.

Typestates allow us to specify the allowed order of operations.

In our setting, we can statically verify that

- Null-dereferencing does not occur
- Protocols are not violated

Example: Protocol Error

```
1  public static void main(String[] args) {
2      List list = new LinkedList();
3      list.add("List");
4      list.add("Modification");
5      list.add("Example");
6
7      ListIterator lIterator = list.listIterator();
8      lIterator.next();
9      lIterator.remove();
10     lIterator.set("Updating");
11 }
```

Illegal state exception at line 10

Iterator should be invalidated after call to remove

Introduction: Usages

Commonly, pre- and postconditions are used to define typestates

- On variables in Strom & Yemini
- On methods in most OOP contexts

Usages are another approach to typestate definitions.

A usage defines a global typestate instead of annotating each method or variable.

Introduction: Usages cont.

Usages are similar to finite-state automata. We allow three behavioural constructs:

- Branching $\{m_i; w_i\}_{i \in I}^{\vec{E}}$
- Choice $\langle l_i : u_i \rangle_{l_i \in L}^{\vec{E}}$
- Recursion $X^{\vec{E} \uplus \{X = u\}}$

$$\text{Usage} = \{\,\text{init}\,; X\}^{\vec{E}}$$

$$\vec{E} = \{X = \{\,\text{test}\,; \left\langle \begin{array}{l} \text{T} : \{\text{doStuff}; X\} \\ \text{F} : X \end{array} \right\rangle \text{stop}; \text{end}\}\}$$

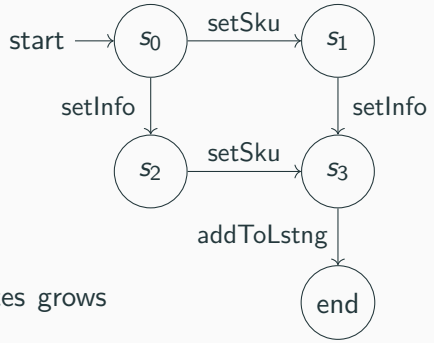
Each object is associated with a typestate consisting of a class and a usage

Introduction: Example

Let Listing be a class with method addProduct with two parameters

Consider the following Product class:

```
1   class Product {
2       String sku; PInfo info;
3
4       void setSku(String x) { sku = x;  }
5       void setInfo(PInfo x) { info = x; }
6
7       Listing addToLstng(Listing l) {
8           l.addProduct(info, sku);
9           return l;
10      }
11  }
```



Note that the number of reachable usage states grows exponentially with each additional linear field.

Introduction: Inference Motivation

Annotating each class with usages has some disadvantages:

- Usages can be large and trivial
- Costly to adopt
- Problems with maintainability
- Cannot check external modules

```
1  class Product{
2      {setSku; {setInfo; X}
3       setInfo; {setSku; X}}
4       [X = {addToLstng; end}]
5
6      ...
7  }
```

Introduction: Inference Motivation cont.

It would be good to be able infer usages when protocol specification is not important:

- Statically ensures no null-dereferencing errors
- Reduces overhead
- Better maintainability
- Open source external modules can be checked

Usage Inference: Problem

Problem

Infer usages for classes when not explicitly specified

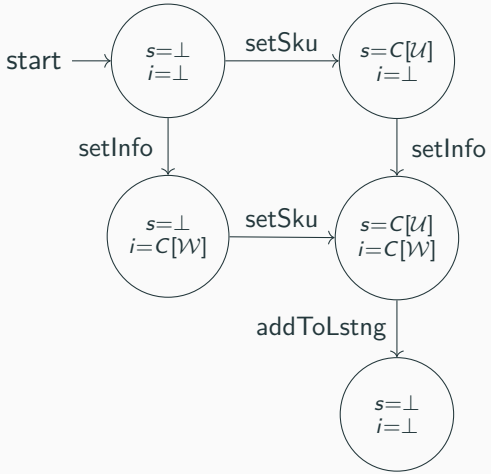
Principal Usage
A usage that can simulate every other usage that well-types the class

Usage subtyping $\mathcal{U} \sqsubseteq \mathcal{U}'$ or $(\mathcal{U}, \mathcal{U}') \in R$
R is a usage simulation iff for all $(\mathcal{U}_1, \mathcal{U}_2) \in R$ we have that:

1. If $\mathcal{U}_1 \xrightarrow{m} \mathcal{U}_1'$ then $\mathcal{U}_2 \xrightarrow{m} \mathcal{U}_2'$ such that $(\mathcal{U}_1', \mathcal{U}_2') \in R$
2. If $\mathcal{U}_1 \xrightarrow{l} \mathcal{U}_1'$ then $\mathcal{U}_2 \xrightarrow{l} \mathcal{U}_2'$ such that $(\mathcal{U}_1', \mathcal{U}_2') \in R$

Typesystem: Intuition

1. Start from the environment where all fields are null
2. Check that following the usage type-checks given the current environment
3. Finally, make sure that the environment is terminated when the usage is



Typesystem: Definition

Type judgements for classes are of the form

$$\Theta; envT_F \vdash_{\vec{D}} C[\mathcal{U}] \triangleright envT'_F$$

Typesystem: Definition

(TCBr) makes sure that when we have a branching usage, all possible branches are well typed w.r.t. the field typing environment

(TCCh) checks that all labels leads to well-typed behaviour and same resulting environment

$$(\mathrm{TCBr})\ \frac{I \neq \emptyset \qquad \forall i \in I\ .\ \exists envT''_F\ .\ \{\mathsf{this} \mapsto envT_F\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto t'_i]) \vdash_{\vec{D}} e_i : t_i \triangleright \{\mathsf{this} \mapsto envT''_F\}; \emptyset \cdot (\mathsf{this}, [x_i \mapsto t''_i] \quad \mathsf{terminated}(t''_i) \qquad t_i\ m_i(t'_i\ x_i)\{e_i\} \in C.\mathsf{methods}_{\vec{D}} \qquad \Theta; envT''_F \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT'_F}{\Theta; envT_F \vdash_{\vec{D}} C[\{m_i; u_i\}_{i \in I}^{\vec{E}}] \triangleright envT'_F}$$

$$(\mathrm{TCCh})\ \frac{\forall l_i \in L\ .\ \Theta; envT_F \vdash_{\vec{D}} C[u_i^{\vec{E}}] \triangleright envT'_F}{\Theta; envT_F \vdash_{\vec{D}} C[\langle l_i : u_i \rangle_{l_i \in L}^{\vec{E}}] \triangleright envT'_F}$$

Typesystem: Definition

(TCallF) type checks a method call of a field

It does so by making sure that a method call is available at the current type

$$(\text{TCallF})\frac{\begin{array}{c}\Lambda; \Delta \cdot (o, S) \vdash e : t \triangleright \Lambda'\{o.f \mapsto C[\mathcal{U}]\}; \Delta' \cdot (o, S') \\ t'\ m(t\ x)\{e'\} \in C.\text{methods}_{\vec{D}} \qquad\qquad \mathcal{U} \xrightarrow{m} \mathcal{W}\end{array}}{\Lambda; \Delta \cdot (o, S) \vdash f.m(e) : t' \triangleright \Lambda'\{o.f \mapsto C[\mathcal{W}]\}; \Delta' \cdot (o, S')}$$

Inference Algorithm: Intuition

We need to do what $(\mathrm{TCB}_\mathrm{R})$ and $(\mathrm{TCC}_\mathrm{H})$ typing does in reverse

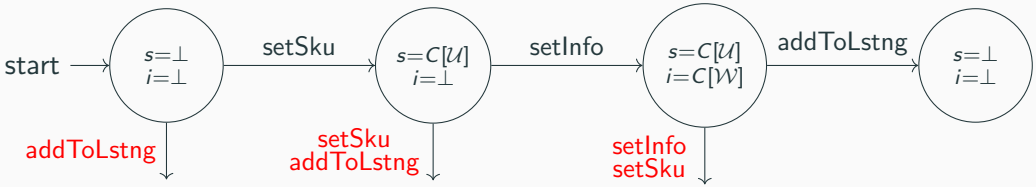
1. Start from the initial field typing environment
2. Typecheck all method bodies with the field typing environment and filter those from that leads to errors
3. Now continue to typecheck all new field typing environment
4. Build the usage based on the graph generated

Inference Algorithm: Intuition

Creating a method availability graph for the previous example

Graph shows a valid sequence of method calls

The red transitions are method calls that lead to errors



Inference Algorithm: Rules

We define a transition relation between field typing environments.

A transition only exists if the method call is well typed.

$$(\textsc{Class})\frac{}{envT_F \xrightarrow{m} envT'_F}$$

Inference Algorithm: Rules

We define a transition relation between field typing environments.

A transition only exists if the method call is well typed.

$$\{\text{this} \mapsto envT_F\}; \emptyset \cdot (\text{this}, [x \mapsto t]) \vdash^{\emptyset} e : t' \triangleright \{\text{this} \mapsto envT'_F\}; \emptyset \cdot (\text{this}, [x \mapsto t''])$$

$$(\textsc{Class}) \frac{}{envT_F \xrightarrow{m} envT'_F}$$

Inference Algorithm: Rules

We define a transition relation between field typing environments.

A transition only exists if the method call is well typed.

$$(\textsc{Class})\ \frac{\{\text{this} \mapsto envT_F\}; \emptyset \cdot (\text{this}, [x \mapsto t]) \vdash^{\emptyset} e : t' \triangleright \{\text{this} \mapsto envT'_F\}; \emptyset \cdot (\text{this}, [x \mapsto t'']) \qquad t'\ m(t\ x)\ \{e\} \in C.\text{methods} \qquad \neg\text{lin}(t'')}{envT_F \xrightarrow{m} envT'_F}$$

Inference Algorithm: Rules

S is a set of the environments which can lead to a terminated environment.

$$S = \{envT_F \mid envT_{F_\perp} \rightarrow^* envT_F \wedge envT_F \rightarrow^* envT_{F_\perp}\} \cup \{\text{end}\}$$

The usage graph allows us to reach the terminated usage end.

$$(\textsc{Trans})\frac{envT_F \xrightarrow{m} envT'_F}{envT_F \overset{m}{\Rightarrow} envT'_F} \qquad (\textsc{End})\frac{envT_F \xrightarrow{m} envT_{F_\perp}}{envT_F \overset{m}{\Rightarrow} \text{end}}$$

Inference Algorithm: Rules

The usage is created by creating a usage variable and usage for each field typing environment

---

1: **function** $\textsc{Infer}(S, A, \Rightarrow, envT_{F_\perp})$
2: **function** $\textsc{Reach}(envT_F)$
3: **return** $\{envT'_F \mid \exists m \in A. envT_F \overset{m}{\Rightarrow} envT'_F\}$
4: **if** $\textsc{Reach}(envT_{F_\perp}) = \emptyset$ **then**
5: **return** $\text{end}^{\emptyset}$
6: $E \leftarrow \emptyset$
7: **for all** $envT_F \in S \setminus \{\text{end}\}$ **do**
8: $E \leftarrow E \cup \{X_{envT_F} = \textsc{CreateState}(envT_F)\}$
9: **return** $X^E_{envT_{F_\perp}}$

Inference Algorithm: Rules

1: **function** CreateState($envT_F$)
2: $u \leftarrow \emptyset$
3: **for all** $m \in A$ **do**
4: **for all** $s \in S$ **do**
5: **if** $envT_F \stackrel{m}{\Rightarrow} s$ **then**
6: **if** $s = \text{end}$ **then**
7: $u \leftarrow u \cup \{m; \text{end}\}$
8: **if** $L\ m(\_\ x)\{\_\} \in C.\text{methods}$ **then**
9: $u \leftarrow u \cup \{m; \langle l_i : \text{end}\rangle_{l_i \in L}\}$
10: **else**
11: $u \leftarrow u \cup \{m; X_s\}$
12: **if** $L\ m(\_\ x)\{\_\} \in C.\text{methods}$ **then**
13: $u \leftarrow u \cup \{m; \langle l_i : X_s\rangle_{l_i \in L}\}$
14: **return** u

1. Find all transitions from $envT_F$
2. Create a branch for a transition
3. If the method returns a label, add a choice usage

Implementation

Haskell implementation of type system and inference module in `mungoi`

Implemented for only the formalised subset of Java

Motivation: Verify complexity in real-life situations

Implementation: Example

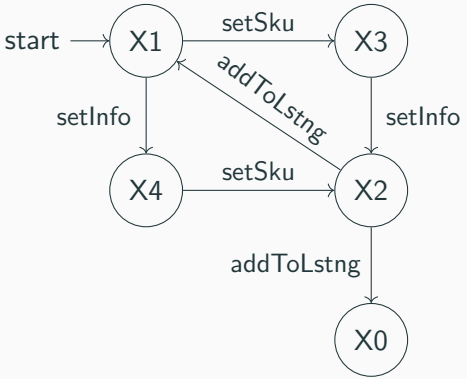
We introduce the special usage variable `infer` to indicate inference

```
1  class Product {
2      infer[]
3
4      String sku
5      PInfo info
6
7      void setSku(String[initialised] x) { sku = x  }
8      void setInfo(PInfo[initialised] x) { info = x }
9      Listing[intialised] addToLstng(Listing[uninitialised] l) {
10         l.addPinfo(info); l.addSku(sku);
11         l
12     }
13 }
```

Implementation

The example can be inferred, and is also well-typed

```
1  $ stack run Product.mg
2  X1[ X0 = end
3      X1 = { setSku; X3
4             setInfo; X4 }
5      X2 = { addToLstng; X0
6             addToLstng; X1 }
7      X3 = { setInfo; X2 }
8      X4 = { setSku; X2 } ]
9  new Right ()
```



Algorithm: Properties

Theorem (Principal Usage Inference)
Let C be a class and $\mathcal{U}_I$ be an inferred usage, then $\mathcal{U}_I$ is a principal usage for C.

Proof.
By showing that the inferred usage is the largest and that it makes the class well typed □

Complexity

Size of usages are bounded by

$$O((|e| + |f| \cdot |\mathcal{U}|)^2 + 2^{2 \cdot |f| \cdot |\mathcal{U}|} \cdot |m|) \approx O(2^{2 \cdot |f| \cdot |\mathcal{U}|} \cdot |m|)$$

Worst case assumption

Every usage state can transition to every other usage state, with every method call

While technically possible, it is a huge overestimation for actual programs

Practical Complexity

Large usages are plausible for usual programs

Classes with n unrelated linear fields will have usages of size $O(2^n)$

Non-determinism does not seem to be useful often

Practical Complexity

```
1  class SmartHomeController {
2      DoorController dc;
3      TemperatureController tc;
4      ElectronicsController ec;
5      [....]
6      void initTempController(TemperatureController c) {
7          this.tc = c;
8          this.tc.initialise();
9      }
10     [....]
11 }
```

Practical Complexity

Large usages are plausible for usual programs

Classes with n unrelated linear fields will have usages of size $O(2^n)$

Non-determinism does not seem to be useful often

Discussion & Conclusion

Inference makes usages more practical

We infer the largest (*principal*) usage

- Infer once, use everywhere

Is the extra effort and complexity worth it?