Lock Analysis for an Asynchronous Object Calculus

Elena Giachino Tudor A. Lascu

Focus INRIA - Università di Bologna

19 September 2012

http://www.hats-project.eu

programme

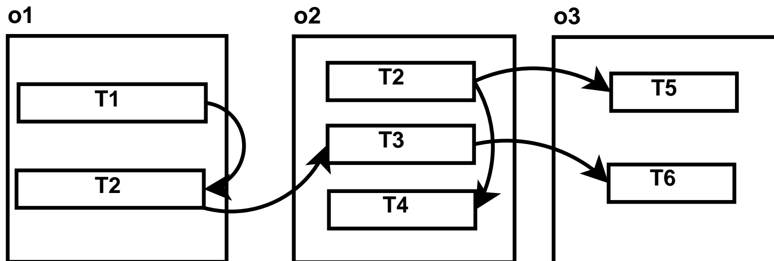
- define a calculus for objects interacting asynchronously

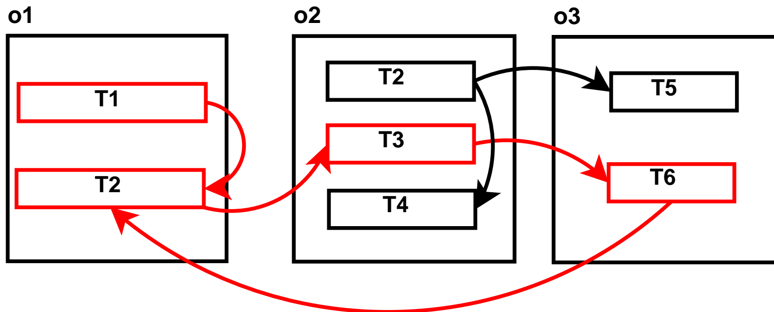
  Featherweight Java with futures (FJf)

- introduce contracts (behavioural types) $\Rightarrow$ lock analysis for FJf
- natural extensions to FJf $\longrightarrow$ FJf$^+$
- design framework evolution for lock analysis of FJf$^+$

the world as we see it

- objects execute concurrently
- an object's activity $\leftrightarrow$ set of tasks
- one lock per object – one active task per object
- asynchronous interaction


o1
T1
T2
o2
T2
T3
T4
o3
T5
T6



circularity = dangerous configuration

language FJf

- based on Featherweight Java (FJ)

$$e \;::=\; x \mid e.f \mid \mathtt{new}\ C(\bar{e}) \mid e;e \mid e!m(\bar{e}) \mid e.\mathtt{get} \mid e.\mathtt{await}$$

$$T \;::=\; C \mid \mathtt{Fut}(T)$$

- features

e!m(ē)	method invocation is asynchronous
e.get	retrieve result, keeping lock
e.await	wait for result, releasing lock
Fut(T)	explicit future type

type future

a variable `X` with type `Fut(T)` is:

- a placeholder for a value of type `T` that will be available (when its computation ends)
  ```
  X = e!m();
  ```
- used as a (future) reference to retrieve the actual value when ready
  ```
  return X.get;
  ```
- typically used in calculi with asynchronous method calls

analysis' idea

- collect object name dependencies in a program
- by means of contracts (behavioural types)
- (automagically) through type-inference
- look for circularities

method contracts

methods $\xrightarrow{\text{type inference}}$ contracts

```
class C {
    C m() { return new C ;}
    C r(C x) { return x!m().get ;}
}
```

methods $\xrightarrow{\text{type inference}}$ contracts

```
class C {

  a[] ( ) { 0 } b[]
    C m() { return new C ;}

  a[] (b[]) { C.m b[] ( ).(a, b) } d[]
    C r(C x) { return x!m().get ;}
}
```

method contracts - structure

$$\mathtt{a}[\,]\ (\mathtt{b}[\,])\ \{\ \mathtt{C.m}\ \mathtt{b}[\,]\ (\,).(\mathtt{a},\mathtt{b})\ \}\ \mathtt{d}[\,]$$

$$\mathbb{r}(\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$$

$$\underbrace{\mathbb{r}}_{\text{object record}}\ (\ \underbrace{\bar{\mathbb{s}}}_{\text{parameter record}}\ )\{\ \underbrace{\mathbb{c}}_{\text{contract body}}\ \}\ \underbrace{\mathbb{r}'}_{\text{returned record}}$$

future records

- (made of) object names
- used to (statically) track objects in a program
  a fresh name for every `new`
- grammar

$$
\begin{array}{rcll}
\mathbb{r} & ::= & \mathtt{X} & \text{variable} \\
 & & \mathtt{a}[\bar{\mathtt{f}} : \bar{\mathbb{r}}] & \text{object's structure} \\
 & & \mathtt{a} \rightsquigarrow \mathbb{r} & \text{access to } \mathbb{r} \text{ granted through } \mathtt{a}
\end{array}
$$

more method contracts

```
class A {
     Object f ;

     A m() { return new A(this.f) ; }

     Fut(A) n(A x) { return x!m() ; }

     A r(A x) { return x!m().await.get ; }
}
```

more method contracts

```
class A {
    Object f ;
```

$a[f : b]\ (\,)\ \{0\}\ \ a'[f : b]$

```
    A m() { return new A(this.f) ; }
```

$a[f : X]\ (a'[f : Y])\ \{A.m\ a'[f : Y](\,)\}\ \ a'[f : Y] \rightsquigarrow a''[f : Y]$

```
    Fut(A) n(A x) { return x!m() ; }
```

$a[f : X]\ (a'[f : Y])\ \{A.m\ a'[f : Y](\,)\bullet(a, a')^{a}\}\ \ a''[f : Y]$

```
    A r(A x) { return x!m().await.get ; }
}
```

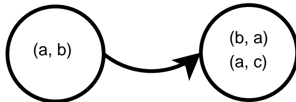
- core of a method contract $\rightarrow$ contract body
- contains contract associated to expression in method's body
- keeps info on object dependencies

$\mathbb{c} ::= 0$	no info
$\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}})$	method $\mathtt{C.m}$ invoked on $\mathbb{r}$ with parameters $\bar{\mathbb{s}}$
$\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}).(\mathtt{a},\mathtt{b})$	" plus *get* operation on result
$\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}).(\mathtt{a},\mathtt{b})^{\mathtt{a}}$	" plus *await* operation on result
$(\mathtt{a},\mathtt{b})$	*get* operation on a field
$(\mathtt{a},\mathtt{b})^{\mathtt{a}}$	*await* operation on a field
$\mathbb{c} \fatsemi \mathbb{c}$	sequential composition

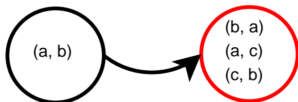
```
class C {

  a[] ( ) { 0 } b[]
    C m() { return new C ;}

  a[] (b[]) { C.m b[] ( ).(a, b) } d[]
    C r(C x) { return x!m().get ;}
}
```

lock analysis for FJf

- extracted contracts $\xrightarrow{\text{compose}}$ automaton
- states are sets of dependencies
- transitions model how are dependencies activated and discarded
- circularity $\Rightarrow$ possible dead (or live-)lock



lock-free



potential misbehaviour

- FJf is a functional language (just as FJ)
  (fields only initialized, by constructor)
- let's introduce notion of state by field update
- add `this.f = e` to the syntax of expressions

can we adjust previous framework?

extensions - intuition

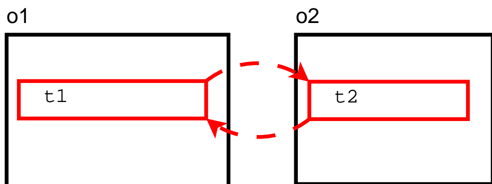
- consider `x!m(); x!n()` where
  - `m()` does `this.f = e1`
  - `n()` does `this.f = e2`
- no way to know order in which updates take place
- static analysis ⇒ keep track of all different possibilities

⇒ future records must contain sets of names

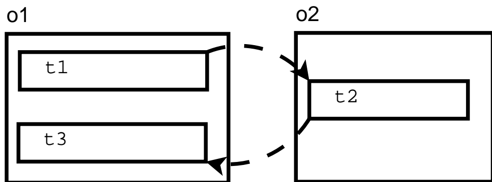
$$\mathbb{r} ::= \mathtt{X} \mid \mathsf{a}[\bar{\mathtt{f}} : \bar{\mathbb{r}}] \mid \mathsf{a} \rightsquigarrow \mathbb{r} \qquad \text{old syntax}$$

$$\mathbb{r} ::= \mathtt{X} \mid \mathcal{A}[\bar{\mathtt{f}} : \bar{\mathbb{r}}] \mid \mathcal{A} \rightsquigarrow \mathbb{r} \qquad \text{new syntax}$$

now: possible to write pure livelocks (`await`-only circularities)



before: `await`-only circularities were non-problematic (by construction)



extensions - side effects 2

- essential to distinguish good from bad circularities
- need to introduce also names for tasks
  $\hookrightarrow$ dependencies btw. tasks
- create a new task name for every method invocation
- future record of invocation tagged by associated task: $\mathcal{A} \rightsquigarrow_{\mathtt{t}} \mathbb{S}$
- states of the automaton contain also dependencies btw. tasks $(\mathtt{t}, \mathtt{t}')$

the type system must also be tuned ...

old style: $\Gamma \vdash_{a} e : (T, \mathbb{r}), \mathbb{c}$

where:

- $\Gamma \rightarrow$ environment
- $a \rightarrow$ name of the object `this`
- $e \rightarrow$ FJf expression
- $T \rightarrow$ expression e's (class or future) type
- $\mathbb{r} \rightarrow$ expression e's future record
- $\mathbb{c} \rightarrow$ expression e's contract

updated version: $\Gamma \vdash^{t}_{a} e : (T, \mathbb{r}), \mathbb{c}$

old rule:

$$\frac{\Gamma \vdash_{\mathtt{a}} \mathtt{e} \;:\; (\mathtt{Fut}(\mathtt{T}), \mathtt{a}' \leadsto \mathbb{s})\,,\ \mathbb{c}}{\Gamma \vdash_{\mathtt{a}} \mathtt{e.get} \;:\; (\mathtt{T}, \mathbb{s})\,,\ \mathbb{c} \between (\mathtt{a}, \mathtt{a}')}$$

new rule:

$$\frac{\Gamma \vdash^{\mathtt{t}}_{\mathtt{a}} \mathtt{e} \;:\; (\mathtt{Fut}(\mathtt{T}), \mathcal{A} \leadsto_{\mathtt{t}'} \mathbb{s})\,,\ \mathbb{c}}{\Gamma \vdash^{\mathtt{t}}_{\mathtt{a}} \mathtt{e.get} \;:\; (\mathtt{T}, \mathbb{s})\,,\ \mathbb{c} \between (\mathtt{t}, \mathtt{t}') \between (\mathtt{a}, \mathtt{a}_i)\, \forall \mathtt{a}_i \in \mathcal{A}}$$

conclusions

- start from ABS language
  (programming lang. developed @HATS project (7th FP) - Highly Adaptable and Trustworthy Software using Formal Models)
- extract core (functional) fragment $\rightarrow$ `FJf`
  (ref. Deadlock and Livelock Analysis in Concurrent Objects with Futures (MSCS'11) - Giachino, Laneve & Lascu)
- develop contracts for lock analysis of `FJf`
  (ref. same as above)
- extend language $\rightarrow$ $\texttt{FJf}^{+}$
- ideas to adapt framework to $\texttt{FJf}^{+}$

future work

- improve analysis' precision
  (infinite names dealt with finite approximants)
- proof of concept prototype (under construction)
- fix analysis' details with addition of tasks
- introduce `null` value ⇒
  circular lists + recursive records ??
- extend analysis to (full) ABS language

thanks!