Behaviour Inference for Deadlock Checking

Ka I Violet Pun, Martin Steffen, Volker Stolz

PMA Group, University of Oslo, Norway

BEAT, Madrid, Spain

23rd – 24th September, 2013
Goal

- Find *potential* deadlocks in programs *statically* by detecting cyclic wait
  - two or more processes form a circular chain, waiting for a shared resource that is held by the next process in the cycle.
  - shared resources here: *locks*
deadlock freedom: **global** (safety) property

two stage approach: local ⇔ global

1. **local** level:
   - behavioral effects for lock interactions
   - polymorphic

2. **global** level: exploration of the abstract behavior to detect deadlock

   - potentially $\infty$ state space
     - re-entrant lock counter
     - stack-structure for function calls
     - dynamic lock creation

$\Rightarrow$ abstractions needed
Techniques used

- type and effect system with behavioral effects (+ flow information)
- behavior effects as in e.g. [Amtoft et al., 1999]
- type/effect inference using constraint-based formulation as in loc.cit
- polymorphic analysis, for enhanced precision (let-polymorphism)
- for proving soundness
  - of the effect type system ("subject reduction")
  - of the abstractions

⇒ deadlock (and termination) sensitive simulation
1. Introduction

2. Syntax and semantics

3. Type and effect system

4. Abstract behaviour

5. Summary
Syntax

- small concurrent calculus
- dynamic lock/thread creation
- higher-order functions
- re-entrant, heap-allocated locks

\[
P ::= \emptyset \mid p\langle t \rangle \mid P \parallel P
\]

\[
t ::= v \mid \text{let } x:T = e \text{ in } t
\]

\[
e ::= t \mid v \cdot v \mid \text{if } v \text{ then } e \text{ else } e \mid \text{spawn } t \mid \text{new } L \mid v.\text{lock}
\]

\[
v ::= x \mid l' \mid \text{true} \mid \text{false} \mid \text{fn } x:T.t \mid \text{fun } f:T.x:T.t
\]
Syntax

- small concurrent calculus
- dynamic lock/thread creation
- higher-order functions
- re-entrant, heap-allocated locks

\[ P ::= \emptyset \mid p\langle t \rangle \mid P \parallel P \]
\[ t ::= v \mid \text{let } x : T = e \text{ in } t \]
\[ e ::= t \mid v \, v \mid \text{if } v \text{ then } e \text{ else } e \mid \text{spawn } t \mid \text{new } L \mid v.\text{lock} \]
\[ v ::= x \mid l' \mid \text{true} \mid \text{false} \mid \text{fn } x : T . t \mid \text{fun } f : T . x : T . t \]

Dining philosophers

```haskell
let l1 = new L; l2 = new L; l3 = new L /* create all locks */
phil = fn x:L,y:L . ( x.lock; y.lock; /* eat */
                   y.unlock; x.unlock; /* think */ )
in spawn(phil(l1,l2)); spawn(phil(l2,l3)); spawn(phil(l3,l1))
```
Operational semantics

\[
P ::= \emptyset \mid p\langle t \rangle \mid P \parallel P \quad \text{(Processes)}
\]

\[
\sigma \vdash P \rightarrow \sigma' \vdash P' \quad \text{with} \quad \sigma : L \mapsto \{\text{free, } p(n)\} \quad \text{(Configuration)}
\]

An example run:

\[
\emptyset \vdash p_0\langle t \rangle \rightarrow \ldots \rightarrow [l_1 \mapsto p_1(1), l_2 \mapsto p_0(1)] \vdash p_1\langle l_2. \text{lock} \rangle \parallel p_0\langle l_1. \text{lock} \rangle
\]
Circular wait

Definition (Waiting for a lock)
Given a configuration $\sigma \vdash P$, 

$$\text{wants}(\sigma \vdash P, p, l)$$

if it is not the case that $\sigma \vdash P \xrightarrow{p\langle l.\text{lock} \rangle} \sigma'$, and furthermore there exists a $\sigma'$ s.t. $\sigma' \vdash P \xrightarrow{p\langle l.\text{lock} \rangle} \sigma'' \vdash P'$.

Definition (Deadlock)
A configuration $\sigma \vdash P$ is deadlocked if $\sigma(l_i) = p_i(n_i)$ and furthermore $\text{wants}(\sigma \vdash P, p_i, l_{i+k})$ (where $k \geq 2$ and for all $0 \leq i \leq k - 1$).
Figure: Deadlock

Figure: Wait-for graph
- *behavioral* effects $\varphi$: interactions of a thread with locks:
- simple process “algebra”
- actions: locking/unlocking
- latent effects for function types: $T_1 \xrightarrow{\varphi} T_2$
- judgements

$$\Gamma \vdash t : T :: \varphi$$
## Types & effects for lock interaction

### Types

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( { \pi } \mid r \sqcup r )</td>
<td>lock/label sets</td>
</tr>
<tr>
<td>( \hat{T} )</td>
<td>( B \mid L^r \mid \hat{T} \xrightarrow{\varphi} \hat{T} )</td>
<td>types</td>
</tr>
</tbody>
</table>

### Effects

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Phi )</td>
<td>( 0 \mid p\langle\varphi\rangle \mid \Phi \parallel \Phi )</td>
<td>effects (global)</td>
</tr>
<tr>
<td>( \varphi )</td>
<td>( \epsilon \mid \varphi;\varphi \mid \varphi + \varphi \mid \alpha \mid X \mid \text{rec} ; X.\varphi )</td>
<td>effects (local)</td>
</tr>
<tr>
<td>( a )</td>
<td>( \text{spawn} ; \varphi \mid r.\text{lock} \mid r.\text{unlock} )</td>
<td>labels/basic effects</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>( a \mid \tau )</td>
<td>transition labels</td>
</tr>
</tbody>
</table>
type level variables
  for sets of \( \pi \)-locations ("regions")
  for behaviob

to enhance precision: type schemes

behavior and flow constraints
Type and effect system

Judgments

\[ \Gamma \vdash e : T :: \varphi \]

\[ Y ::= \varrho \mid X \quad \text{type-level variables} \]

\[ r ::= \varrho \mid \{ \pi \} \mid r \sqcap r \quad \text{lock/label sets} \]

\[ \hat{T} ::= B \mid Lr \mid \hat{T} \xrightarrow{\varphi} \hat{T} \quad \text{types} \]
Type and effect system

Judgments

\[ \Gamma \vdash e : T :: \varphi \]

\[ \begin{align*}
Y & ::= \varrho \mid X \quad \text{type-level variables} \\
r & ::= \varrho \mid \{\pi\} \mid r \sqcup r \quad \text{lock/label sets} \\
\hat{T} & ::= B \mid \mathbb{L}r \mid \hat{T} \xrightarrow{\varphi} \hat{T} \quad \text{types} \\
\hat{S} & ::= \forall \vec{Y}. \hat{T} \quad \text{type schemes}
\end{align*} \]
Judgments

\[ C; \Gamma \vdash e : T :: \varphi \]

\[ \begin{align*}
Y & ::= \varnothing \mid X & \text{type-level variables} \\
r & ::= \varnothing \mid \{\pi\} \mid r \sqcup r & \text{lock/label sets} \\
\hat{T} & ::= B \mid L^r \mid \hat{T} \to \hat{T} & \text{types} \\
\hat{S} & ::= \forall \vec{Y}: C. \hat{T} & \text{type schemes} \\
C & ::= \emptyset \mid \varnothing \sqsupseteq r, C \mid X \sqsupseteq \varphi, C & \text{constraints}
\end{align*} \]
For our example:

\[
\begin{align*}
\text{let } x : L^{\pi_1} &= \text{new}_{\pi_1} L \text{ in} \\
\text{let } y : L^{\pi_2} &= \text{new}_{\pi_2} L \text{ in} \\
&\quad \text{spawn } (y.\text{lock}; x.\text{lock}; \text{stop}); \ x.\text{lock}; y.\text{lock}; \text{stop}
\end{align*}
\]

Effect:

\[
\varphi = \nu L^{\pi_1}; \nu L^{\pi_2}; \text{spawn } (\pi_2.\text{lock}; \pi_1.\text{lock}); \pi_1.\text{lock}; \pi_2.\text{lock}
\]
Thread-local type and effect system

- instead of checking “subtyping” of “sub-effecting” \(\Rightarrow\) generating appropriate constraints on-the-fly, using fresh variables
- can be seen a generalization of “Algorithm W”
- type schemes for polymorphic analysis, constraints “qualifying” the bound variables
\[
\Gamma(x) = \forall \bar{Y}: C. \hat{T} \quad \bar{Y}' \text{ fresh} \quad \theta = [\bar{Y}'/\bar{Y}]
\]

\[
\Gamma \vdash x : \theta \hat{T} :: \epsilon; \theta C
\]

\[
\varrho \text{ fresh}
\]

\[
\Gamma \vdash \text{new}^\pi L : L^\varrho :: \epsilon; \varrho \sqsubseteq \{\pi\}
\]

\[
\hat{T}_1 = [T_1]_A \quad \Gamma, x: \hat{T}_1 \vdash e : \check{T}_2 :: \varphi; C \quad X \text{ fresh}
\]

\[
\Gamma \vdash \text{fn } x : T_1.e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \epsilon; C, X \sqsubseteq \varphi
\]

\[
\hat{T}_1 \xrightarrow{X} \hat{T}_2 = [T_1 \rightarrow T_2]_A \quad \Gamma, f: \hat{T}_1 \xrightarrow{X} \hat{T}_2, x: \hat{T}_1 \vdash e : \check{T}_2 :: \varphi; C_1 \quad \hat{T}_2' \leq \hat{T}_2 \vdash C_2
\]

\[
\Gamma \vdash \text{fun } f : T_1 \rightarrow T_2, x: T_1.e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \epsilon; C_1, C_2, X \sqsubseteq \varphi
\]

\[
\Gamma \vdash v_1 : \check{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \epsilon; C_1 \quad \Gamma \vdash v_2 : \check{T}_2' :: \epsilon; C_2 \quad \hat{T}_2' \leq \hat{T}_2 \vdash C_3 \quad X \text{ fresh}
\]

\[
\Gamma \vdash v_1 \; v_2 : \hat{T}_1 :: X; C_1, C_2, C_3, X \sqsubseteq \varphi
\]
Thread-local type and effect system

\[
[\hat{T}] = [\hat{T}_1] = [\hat{T}_2] \quad \hat{T}; C = \hat{T}_1 \lor \hat{T}_2 \quad X; C' = \varphi_1 \sqcup \varphi_2
\]

\[
\Gamma \vdash v : \text{Bool} :: \epsilon; C_0 \quad \Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1; C_1 \quad \Gamma \vdash e_2 : \hat{T}_2 :: \varphi_2; C_2
\]

\[
\Gamma \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 : \hat{T} :: X; C_0, C_1, C_2, C, C'
\]

\[
\hat{S}_1 = \text{close}(\Gamma, \varphi_1, C_1, \hat{T}_1) \quad \Gamma, x:\hat{S}_1 \vdash e_2 : \hat{T}_2 :: \varphi_2; C_2
\]

\[
\Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1; C_1 \quad [\hat{T}_1] = T_1
\]

\[
\hat{S}_1 = \text{close}(\Gamma, \varphi_1, C_1, \hat{T}_1) \quad \Gamma, x:\hat{S}_1 \vdash e_2 : \hat{T}_2 :: \varphi_2; C_2
\]

\[
\Gamma \vdash \text{let } x : T_1 = e_1 \text{ in } e_2 : \hat{T}_2 :: \varphi_1; \varphi_2; C_1, C_2
\]

\[
\Gamma \vdash t : \hat{T} :: \varphi; C \quad X \text{ fresh}
\]

\[
\Gamma \vdash \text{spawn } t : \text{Unit} :: X; C, X \sqsupset \text{spawn } \varphi
\]

\[
\Gamma \vdash v : L^\varrho :: \epsilon; C \quad X \text{ fresh}
\]

\[
\Gamma \vdash v. \text{lock} : L^\varrho :: X; C, X \sqsupset \varrho \text{ lock}
\]

\[
\Gamma \vdash v : L^\varrho :: \epsilon; C \quad X \text{ fresh}
\]

\[
\Gamma \vdash v. \text{lock} : L^\varrho :: X; C, X \sqsupset \varrho \text{ unlock}
\]
we need the operational “behavior” of the effects for

- **local** level: to relate the type system to the semantics (soundness, via subject reduction)
- **global** level: deadlock checking (see later)

defined using the constraints

**labelled (weak) transitions**

\[ C \vdash \varphi_1 \xrightarrow{a} \sqsubseteq \varphi_2 \] given by \[ C \vdash a; \varphi_2 \sqsubseteq \varphi_1. \]

subject reduction with effects: a form of simulation proof

however: beware of deadlocks
Subject reduction

\[
C_1; \hat{\sigma}_1 \vdash p\langle \varphi \rangle \quad \xrightarrow{p\langle a \rangle} \quad C_1; \hat{\sigma}_1' \vdash p\langle \varphi' \rangle
\]

\[
\sigma_2 \vdash p\langle t \rangle \quad \xrightarrow{p\langle a \rangle} \quad \sigma_2' \vdash p\langle t' \rangle
\]
for subject reduction: relating one thread with its effect

globally: compositionality wrt. $\parallel$

to relate effects at different levels of abstraction: relate effects

Definition (Deadlock sensitive simulation $\lessdot_{D}$)

Assume a heap-mapping $\theta$ and a corresponding wait-sensitive abstraction $\leq_{\theta}$. A binary relation $R$ between configurations is a deadlock sensitive simulation relation if the following holds.

Assume $C_1; \hat{\sigma}_1 \vdash \Phi_1 \ R \ C_2; \hat{\sigma}_2 \vdash \Phi_2$ with $\hat{\sigma}_1 \leq_{\theta} \hat{\sigma}_2$. Then:

1. If $C_1; \hat{\sigma}_1 \vdash p^{<a>} \not\rightarrow{\subseteq} C_1; \hat{\sigma}'_1 \vdash \Phi'_1$, then
   
   $\exists C_2; \hat{\sigma}_2 \vdash \Phi_2 \not\rightarrow{\subseteq} C_2; \hat{\sigma}'_2 \vdash \Phi'_2$ for some $C_2; \hat{\sigma}'_2 \vdash \Phi'_2$ with $\hat{\sigma}'_1 \leq_{\theta} \hat{\sigma}'_2$ and $C_1; \hat{\sigma}_1 \vdash \Phi'_1 \ R \ C_2; \hat{\sigma}'_2 \vdash \Phi'_2$.

2. If $w\text{aits}_{\subseteq}((C_1; \hat{\sigma}_1 \vdash \Phi_1), p, \varrho)$, then

   $w\text{aits}_{\subseteq}((C_2; \hat{\sigma}_2 \vdash \Phi_2), p, \theta(\varrho))$. 
Infinite state space

4 sources of infinity

1. dynamic lock creation
2. Unboundedness of *reentrant* lock counters
3. “control stack” of *non-tail recursive* behaviours
4. process creation
Lock abstraction

- summarizing locks by their point of creation
- non-injective abstraction
- improvement over [Pun et al., 2012]
- abstract heap:
  - abstract lock = location(s) of lock creation
  - abstract lock counter = \textit{sum} of all lock counters of concrete locks it represents
Problem in state space:
Unbounded lock counters counting up

Solution:
Fix upper bound; unlocking from upper bound becomes non-deterministic.

Lemma

Given a configuration $\sigma \vdash \Phi$, and let further denote $\sigma_1 \vdash^{n_1} \Phi$ and $\sigma_2 \vdash^{n_2} \Phi$ the corresponding configurations under the lock-counter abstraction. If $n_1 \geq n_2$, then $\sigma_1 \vdash^{n_1} \Phi \preceq^D \sigma_2 \vdash^{n_2} \Phi$. 
Getting rid of the stack: $\Omega$

- replacing context-free (stack) behavior by tail-recursive one
- beyond stack-depth-$k$: chaotic behavior $\Omega$
- again: beware of preserving deadlocks
- compositionality

Lemma ($\Omega$ is maximal wrt. $\preceq^{DT}$)

Assume $\varphi$ over a set of locations $r$, then $\sigma \vdash p\langle \varphi \rangle \preceq^{DT} \sigma \vdash p\langle \Omega \rangle$. 
Some results

Theorem (Finite abstractions)

The lock abstraction, the lock counter abstraction and behavior abstraction (when abstracting all locks and recursions) results in a finite state space.

Note: the “size” of the abstraction is adaptable

Theorem (Soundness of the abstraction)

Given $\Gamma \vdash P : \text{ok} :: \Phi$ and two heaps $\hat{\sigma}_1 \leq_\theta \hat{\sigma}_2$. Further, $\sigma'_2 \vdash \Phi'$ is obtained by the mentioned abstractions from $\sigma_2 \vdash \Phi$. Then if $\sigma'_2 \vdash \Phi'$ is deadlock free then so is $\sigma_1 \vdash P$. 
Summary

Conclusion:
- We have proven that our type systems is correct in the aspect of capturing behavior of a program.
- Abstract behavior correctly over-approximates the concrete one.
- Deadlocks in a program are correctly detected in the abstract run...
- Type system partially formalized with Ott and Coq (mono-case).

Future Work:
- Applying to communication analysis of asynchronous systems.
- Abstracting processes (probably hard).
- Implement our algorithm with model checker for real language.
- CEGAR - Counter-Example Guided Abstraction Refinement.

Type and Effect Systems: Behaviours for Concurrency.
Imperial College Press.

Deadlock checking by a behavioral effect system for lock handling.
A preliminary version was published as University of Oslo, Dept. of Computer Science Technical Report 404, March 2011.

Lock-polymorphic behaviour inference for deadlock checking.
Technical report 436, University of Oslo, Dept. of Informatics.