Polymorphic Behavioural Lock Effects for Deadlock Checking

Ka I Violet Pun, Martin Steffen, Volker Stolz

PMA Group, University of Oslo, Norway

The 23rd Nordic Workshop for Programming Theory - NWPT ’11
Västerås, Sweden

26th ~ 28th October, 2011
Find potential deadlocks in programs statically by detecting cyclic wait

- Each of two or more processes, which form a circular chain, wait for a shared resource that is held by the next process in the chain.
- Shared resources here: locks
Overview

• Capture **abstract behaviour** as effects with a type and effect system

• Use *program points* \( \pi \), to characterize locks according to their origin

• **Execute** the abstract behaviour to detect deadlock

• Limit potential infinite state space by:
  - Put an upper bound for reentrant lock counter
  - Transform effects into coarser, tail-recursive effect
  - Don’t allow recursive thread/lock creation

• Prove deadlock preservation by defining a *Deadlock and Termination Sensitive Simulation*
Syntax

\[
\begin{align*}
t & ::= \text{stop} \mid v \mid \text{let } x:T = e \text{ in } t \\
e & ::= t \mid v \mid \text{if } e \text{ then } e \text{ else } e \mid \text{spawn } t \\
& \quad \mid \text{new } L \mid v.\text{lock} \mid v.\text{unlock} \\
v & ::= x \mid l \mid \text{fn } x:T.t \mid \text{fun } f:T.x:T.t
\end{align*}
\]

Sequential composition \(e_1;e_2\) is represented by let-construct

\[
\text{let } x:T = e_1 \text{ in } e_2 , \quad x \notin \text{fv}(e_2)
\]
Syntax

\[ t ::= \text{stop} | \nu | \text{let } x: T = e \text{ in } t \]
\[ e ::= t | \nu \nu | \text{if } e \text{ then } e \text{ else } e | \text{spawn } t \]
\[ | \text{new } L | \nu. \text{lock} | \nu. \text{unlock} \]
\[ \nu ::= x | l | \text{fn } x: T. t | \text{fun } f: T. x: T. t \]

Sequential composition \( e_1; e_2 \) is represented by let-construct

\[ \text{let } x: T = e_1 \text{ in } e_2, \quad x \notin \text{fv}(e_2) \]

Dining Philosophers

\[
\begin{align*}
\text{let } l_1 &= \text{new}_{\pi_1} L, l_2 = \text{new}_{\pi_2} L, l_3 = \text{new}_{\pi_3} L, \\
l_4 &= \text{new}_{\pi_4} L, l_5 = \text{new}_{\pi_5} L \text{ in} \\
\text{let } \text{grab} &= \text{fn}:L \times L \rightarrow L. (l, r). l. \text{lock}; r. \text{lock} \text{ in} \\
\text{let } \text{release} &= \text{fn}:L \times L \rightarrow L. (l, r). l. \text{unlock}; r. \text{unlock} \text{ in} \\
\text{let } \text{phil} &= \text{fun } \text{PHIL}:L \times L \rightarrow L. (l, r). \text{think}; \text{grab}(l, r); \\
&\quad \text{eat}; \text{release}(l, r); \text{PHIL}(l, r) \text{ in} \\
&\quad \text{spawn}(\text{phil}(l_1, l_2)); \ldots; \text{spawn}(\text{phil}(l_5, l_1))
\end{align*}
\]
Operational semantics

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

(Processes)

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

(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 $\sigma \vdash P$,

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

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 deadlock if $\sigma(l_i) = p_i(n_i)$ and furthermore $\text{waits}(\sigma \vdash P, p_i, l_{i+k1})$ (where $k \geq 2$ and for all $0 \leq i \leq k - 1$).
Figure: Deadlock

Figure: Wait-for graph
The judgment of our type and effect system is given by:

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

Types and effects are described by:

- **Basic types**:
  
  \[
  U ::= \text{Bool} \mid \text{Int} \mid L' \mid \text{Thread}
  \]

- **Types**:
  
  \[
  T ::= U \mid \vec{U} \varphi \rightarrow U \mid \forall \rho . T
  \]

- **Location annotations**:
  
  \[
  r ::= \pi \mid \rho
  \]
The judgment of our type and effect system is given by:

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

Types and effects are described by:

- \[ U ::= \text{Bool} | \text{Int} | L' | \text{Thread} \]  
  \( U \) basic types

- \[ T ::= U | \overrightarrow{U} \xrightarrow{\varphi} U | \forall \varrho. T \]  
  \( T \) types

- \[ r ::= \pi | \varrho \]  
  location annotations

- \[ \Phi ::= 0 | p\langle \varphi \rangle | \Phi \parallel \Phi \]  
  effects (global)

- \[ a ::= \text{spawn } \varphi | \nu L' | L'.lock | L'.unlock \]  
  labels-basic effects

- \[ \alpha ::= a | \tau \]  
  transition labels
Type and Effect System

The judgment of our type and effect system is given by:

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

Types and effects are described by:

\[
\begin{align*}
U & ::= \text{Bool} \mid \text{Int} \mid \text{L}' \mid \text{Thread} & \text{basic types} \\
T & ::= U \mid \vec{U} \varphi \rightarrow U \mid \forall \varphi. T & \text{types} \\
r & ::= \pi \mid \varrho & \text{location annotations} \\
\Phi & ::= 0 \mid p\langle \varphi \rangle \mid \Phi \parallel \Phi & \text{effects (global)} \\
\varphi & ::= \epsilon \mid X \mid \varphi;\varphi \mid \varphi + \varphi \mid \text{rec } X.\varphi \mid \alpha & \text{effects (local)} \\
a & ::= \text{spawn } \varphi \mid \nu L' \mid L'.lock \mid L'.unlock & \text{labels/basic effects} \\
\alpha & ::= a \mid \tau & \text{transition labels}
\end{align*}
\]
Deadlock Checking

To detect a deadlock in a program, we execute the abstract behaviour of the program. In our example:

```ml
let l1 = new π1 L, l2 = new π2 L, l3 = new π3 L,
      l4 = new π4 L, l5 = new π5 L in
let grab = fn:L×L→L. (l, r). l.lock; r.lock in
let release = fn:L×L→L. (l, r). l.unlock; r.unlock in
let phil = fun PHIL:L×L→L. (l, r). think; grab(l, r);
         eat; release(l, r); PHIL (l, r) in
spawn (phil(l1,l2));...;spawn (phil(l5,l1))
```

We have the effect:

\[ \nu L^{\pi_1};...;\nu L^{\pi_5}; \text{spawn} \left( \varphi_p(\pi_1,\pi_2) \right);...;\text{spawn} \left( \varphi_p(\pi_5,\pi_1) \right) \]

\[ \varphi_p(\varrho_1,\varrho_2) = \text{rec } X. \text{ think; } L^{\varrho_1}.\text{lock; } L^{\varrho_2}.\text{lock; } \text{eat; } L^{\varrho_1}.\text{unlock; } L^{\varrho_2}.\text{unlock; } X \]
\begin{align*}
\emptyset \vdash p(\nu \pi_1 \ldots ; \nu \pi_5 ; \text{spawn } (\varphi_p(\pi_1, \pi_2)) \ldots ; \text{spawn } (\varphi_p(\pi_5, \pi_1))) \\
[\pi_1 \mapsto \text{free}] \ldots [\pi_5 \mapsto \text{free}] \vdash p(\text{spawn } (\varphi_p(\pi_1, \pi_2)) \ldots ; \text{spawn } (\varphi_p(\pi_5, \pi_1))) \\
\vdots \\
\vdots \\
[\pi_1 \mapsto \text{free}] \ldots [\pi_5 \mapsto \text{free}] \vdash p_1(\pi_1.\text{lock};\pi_2.\text{lock};\pi_1.\text{unlock};\pi_2.\text{unlock}; \\
\text{rec } X.\pi_1.\text{lock}; \ldots) \parallel \ldots \parallel \\
p_5(\pi_5.\text{lock};\pi_1.\text{lock};\pi_5.\text{unlock};\pi_1.\text{unlock}; \\
\text{rec } X.\pi_5.\text{lock}; \ldots) \\
\vdots \\
[\pi_1 \mapsto p_1(1)][\pi_2 \mapsto p_2(2)] \ldots [\pi_5 \mapsto p_5(1)] \vdash p_1(\pi_2.\text{lock};\pi_1.\text{unlock};\pi_2.\text{unlock}; \\
\text{rec } X.\pi_1.\text{lock}; \ldots) \parallel \ldots \parallel \\
p_5(\pi_1.\text{lock};\pi_5.\text{unlock};\pi_1.\text{unlock}; \\
\text{rec } X.\pi_5.\text{lock}; \ldots) \\
p_1(\pi_1.\text{lock}) \quad p_5(\pi_1.\text{lock};\pi_5.\text{unlock};\pi_1.\text{unlock}; \\
\text{rec } X.\pi_5.\text{lock}; \ldots) \
\end{align*}
Two sources of infinity

- Unboundedness of *reentrant* lock counters
- Unboundedness of the “control stack” of *non-tail recursive* behaviour descriptions
Problem in state space:
Unbounded lock counters counting uuuuuuuupppppp (with recursion)...

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 $\sigma_1 \vdash_{n_1} \Phi \sim_D^D \sigma_2 \vdash_{n_2} \Phi$. 
Lemma (Ω is maximal wrt. $\preceq^{DT}$)

Assume $\varphi$ over a set of locations $r$, then $\sigma \vdash p\langle \varphi \rangle \preceq^{DT} \sigma \vdash p\langle \Omega \rangle$. 
Theorem (Finite abstractions)

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

Theorem (Soundness of the abstraction)

Given $\Gamma \vdash P : ok :: \Phi$ and two heaps $\sigma_1 \equiv \sigma_2$. Further, $\sigma'_2 \vdash \Phi'$ is obtained by lock-counter resp. behavior abstraction of $\sigma_2 \vdash \Phi$. Then if $\sigma'_2 \vdash \Phi'$ is deadlock free then so is $\sigma_1 \vdash P$. 
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...
- Inference algorithm is partially formalized with Ott and Coq

Future Work:
- Applying to communication analysis of asynchronous systems
- Relaxing the condition (e.g. lock creation in loop)
- Abstracting processes
- Implement our algorithm with model checker for real language
- CEGAR - Counter-Example Guided Abstraction Refinement