

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



- 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 $\Leftrightarrow$ global
 1. **local** level:
 - behavioral effects for lock interactions
 - polymorphic
 2. **global** level: exploration of the abstract behavior to detect deadlock
- potentially ∞ state space
 - re-entrant lock counter
 - stack-structure for function calls
 - dynamic lock creation

$\Rightarrow$ abstractions needed

- 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

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

$$\begin{aligned} P &::= \emptyset \mid p\langle t \rangle \mid P \parallel P \\ t &::= v \mid \text{let } x:T = e \text{ in } t \\ e &::= t \mid v \ v \mid \text{if } v \text{ then } e \text{ else } e \mid \text{spawn } t \mid \text{new } L \mid v. \text{lock} \\ v &::= x \mid l^r \mid \text{true} \mid \text{false} \mid \text{fn } x:T. t \mid \text{fun } f:T. x:T. t \end{aligned}$$

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

$$P ::= \emptyset \mid p\langle t \rangle \mid P \parallel P$$
$$t ::= v \mid \text{let } x:T = e \text{ in } t$$
$$e ::= t \mid v \ v \mid \text{if } v \text{ then } e \text{ else } e \mid \text{spawn } t \mid \text{new } L \mid v. \text{lock}$$
$$v ::= x \mid l^r \mid \text{true} \mid \text{false} \mid \text{fn } x:T. t \mid \text{fun } f:T.x:T. t$$

Dining philosophers

```
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))
```

$$\begin{array}{l} P ::= \emptyset \mid p\langle t \rangle \mid P \parallel P \quad (\text{Processes}) \\ \sigma \vdash P \rightarrow \sigma' \vdash P' \text{ with } \sigma : L \mapsto \{\text{free}, p(n)\} \quad (\text{Configuration}) \end{array}$$

An example run:

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

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 $\sigma \vdash P \xrightarrow{p\langle l.\text{lock} \rangle}$, and furthermore there exists a σ' 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 $\text{waits}(\sigma \vdash P, p_i, l_{i+k1})$ (where $k \geq 2$ and for all $0 \leq i \leq k-1$).

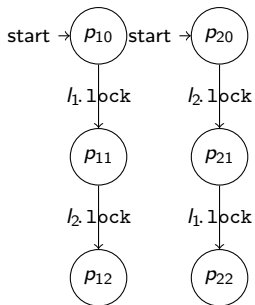


Figure : Deadlock

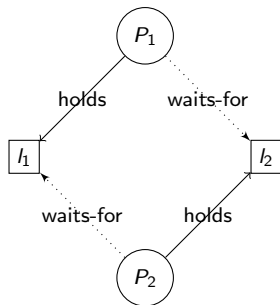


Figure : Wait-for graph

- *behavioral* effects φ : 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

$$\begin{array}{ll} r ::= \{\pi\} \mid r \sqcup r & \text{lock/label sets} \\ \hat{T} ::= B \mid L^r \mid \hat{T} \xrightarrow{\varphi} \hat{T} & \text{types} \end{array}$$

effects

$$\begin{array}{ll} \Phi ::= \mathbf{0} \mid p\langle\varphi\rangle \mid \Phi \parallel \Phi & \text{effects (global)} \\ \varphi ::= \epsilon \mid \varphi; \varphi \mid \varphi + \varphi \mid \alpha \mid X \mid \text{rec } X.\varphi & \text{effects (local)} \\ a ::= \text{spawn } \varphi \mid r.\text{lock} \mid r.\text{unlock} & \text{labels/basic effects} \\ \alpha ::= a \mid \tau & \text{transition labels} \end{array}$$

- type level variables
 - for sets of π -locations (“regions”)
 - for behavior
- to enhance precision: type **schemes**
- behavior and flow **constraints**

Judgments

$$\Gamma \vdash e : T :: \varphi$$

Y	$::=$	$\varrho \mid X$	type-level variables
r	$::=$	$\varrho \mid \{\pi\} \mid r \sqcup r$	lock/label sets
$\hat{T}$	$::=$	$B \mid L^r \mid \hat{T} \xrightarrow{\varphi} \hat{T}$	types

Judgments

$$\Gamma \vdash e : T :: \varphi$$

$\mathcal{Y}$	$::= \varrho \mid X$	type-level variables
r	$::= \varrho \mid \{\pi\} \mid r \sqcup r$	lock/label sets
$\hat{T}$	$::= B \mid \mathsf{L}^r \mid \hat{T} \xrightarrow{\varphi} \hat{T}$	types
$\hat{S}$	$::= \forall \vec{\mathcal{Y}}. \hat{T}$	type schemes

Judgments

$$C; \Gamma \vdash e : T :: \varphi$$

Y	$::= \varrho \mid X$	type-level variables
r	$::= \varrho \mid \{\pi\} \mid r \sqcup r$	lock/label sets
$\hat{T}$	$::= B \mid \mathbb{L}^r \mid \hat{T} \xrightarrow{\varphi} \hat{T}$	types
$\hat{S}$	$::= \forall \vec{Y} : C. \hat{T}$	type schemes
C	$::= \emptyset \mid \varrho \sqsupseteq r, C \mid X \sqsupseteq \varphi, C$	constraints

For our example:

```
let x : Lπ1 = newπ1 L in  
let y : Lπ2 = newπ2 L in  
  spawn (y.lock ; x.lock ; stop) ; x.lock ; y.lock ; stop
```

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

Thread-local type and effect system

$$\frac{\Gamma(x) = \forall \vec{Y} : C. \hat{T} \quad \vec{Y}' \text{ fresh} \quad \theta = [\vec{Y}' / \vec{Y}]}{\Gamma \vdash x : \theta \hat{T} :: \epsilon; \theta C} \text{TA-VAR}$$

$$\frac{\varrho \text{ fresh}}{\Gamma \vdash \text{new}^\pi L : L^\varrho :: \epsilon; \varrho \sqsupseteq \{\pi\}} \text{TA-NEWL}$$

$$\frac{\hat{T}_1 = [T_1]_A \quad \Gamma, x : \hat{T}_1 \vdash e : \hat{T}_2 :: \varphi; C \quad X \text{ fresh}}{\Gamma \vdash \text{fn } x : T_1. e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \epsilon; C, X \sqsupseteq \varphi} \text{TA-ABS}_1$$

$$\frac{\hat{T}_1 \xrightarrow{X} \hat{T}_2 = [T_1 \rightarrow T_2]_A \quad \Gamma, f : \hat{T}_1 \xrightarrow{X} \hat{T}_2, x : \hat{T}_1 \vdash e : \hat{T}_2' :: \varphi; C_1 \quad \hat{T}_2' \leq \hat{T}_2 \vdash C_2}{\Gamma \vdash \text{fun } f : T_1 \rightarrow T_2, x : T_1. e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \epsilon; C_1, C_2, X \sqsupseteq \varphi} \text{TA-AB}$$

$$\frac{\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \epsilon; C_1 \quad \Gamma \vdash v_2 : \hat{T}_2' :: \epsilon; C_2 \quad \hat{T}_2' \leq \hat{T}_2 \vdash C_3 \quad X \text{ fresh}}{\Gamma \vdash v_1 \ v_2 : \hat{T}_1 :: X; C_1, C_2, C_3, X \sqsupseteq \varphi} \text{TA-APP}$$

Thread-local type and effect system

$$\frac{\begin{array}{l} \llbracket \hat{T} \rrbracket = \llbracket \hat{T}_1 \rrbracket = \llbracket \hat{T}_2 \rrbracket \quad \hat{T}; C = \hat{T}_1 \vee \hat{T}_2 \quad X; C' = \varphi_1 \sqcup \varphi_2 \\ \Gamma \vdash v : \text{Bool} :: \epsilon; C_0 \quad \Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1; C_1 \quad \Gamma \vdash e_2 : \hat{T}_2 :: \varphi_2; C_2 \end{array}}{\Gamma \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 : \hat{T} :: X; C_0, C_1, C_2, C, C'} \text{TA-COND}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1; C_1 \quad \llbracket \hat{T}_1 \rrbracket = T_1 \\ \hat{S}_1 = \text{close}(\Gamma, \varphi_1, C_1, \hat{T}_1) \quad \Gamma, x:\hat{S}_1 \vdash e_2 : \hat{T}_2 :: \varphi_2; C_2 \end{array}}{\Gamma \vdash \text{let } x:T_1 = e_1 \text{ in } e_2 : \hat{T}_2 :: \varphi_1; \varphi_2; C_1, C_2} \text{TA-LET}$$

$$\frac{\Gamma \vdash t : \hat{T} :: \varphi; C \quad X \text{ fresh}}{\Gamma \vdash \text{spawn } t : \text{Unit} :: X; C, X \sqsubseteq \text{spawn } \varphi} \text{TA-SPAWN}$$

$$\frac{\Gamma \vdash v : L^\varrho :: \epsilon; C \quad X \text{ fresh}}{\Gamma \vdash v. \text{lock} : L^\varrho :: X; C, X \sqsubseteq \varrho \text{lock}} \text{TA-LOCK}$$

$$\frac{\Gamma \vdash v : L^\varrho :: \epsilon; C \quad X \text{ fresh}}{\Gamma \vdash v. \text{lock} : L^\varrho :: X; C, X \sqsubseteq \varrho \text{unlock}} \text{TA-UNLOCK}$$

Soundness of the abstraction (thread local)

- 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

$$\begin{array}{ccc} C_1; \hat{\sigma}_1 \vdash p\langle\varphi\rangle & \xRightarrow[p\langle a \rangle]{\equiv} & C_1; \hat{\sigma}'_1 \vdash p\langle\varphi'\rangle \\ | & & | \\ R & & R \\ | & & | \\ \sigma_2 \vdash p\langle t \rangle & \xRightarrow[p\langle a \rangle]{} & \sigma'_2 \vdash p\langle t' \rangle \end{array}$$

Deadlock sensitive simulation

- 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 $\lesssim_{\sqsubseteq}^D$)

Assume a heap-mapping θ 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 $C_1; \hat{\sigma}_1 \vdash \Phi_1 \xrightarrow{p\langle a \rangle}_{\sqsubseteq} C_1; \hat{\sigma}'_1 \vdash \Phi'_1$, then
 $C_2; \hat{\sigma}_2 \vdash \Phi_2 \xrightarrow{p\langle a \rangle}_{\sqsubseteq} C_2; \hat{\sigma}'_2 \vdash \Phi'_2$ 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 $\text{waits}_{\sqsubseteq}((C_1; \hat{\sigma}_1 \vdash \Phi_1), p, \varrho)$, then
 $\text{waits}_{\sqsubseteq}((C_2; \hat{\sigma}_2 \vdash \Phi_2), p, \theta(\varrho))$.

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

- 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 = *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

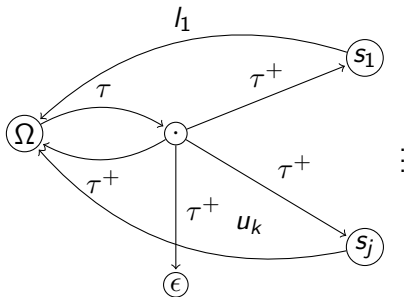
Given a configuration $\sigma \vdash \Phi$, and let further denote $\sigma_1 \vdash^{n_1} \Phi$ and $\sigma_2 \vdash^{n_2} \Phi$ the corresponding configurations under the lock-counter abstraction. If $n_1 \geq n_2$, then $\sigma_1 \vdash^{n_1} \Phi \lesssim^D \sigma_2 \vdash^{n_2} \Phi$.

Getting rid of the stack: Ω

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

Lemma (Ω is maximal wrt. $\lesssim^{DT}$)

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



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 $\Gamma \vdash P : ok :: \Phi$ 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 $\sigma_1 \vdash P$.

- 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

[Amtoft et al., 1999] Amtoft, T., Nielson, H. R., and Nielson, F. (1999).

Type and Effect Systems: Behaviours for Concurrency.

Imperial College Press.

[Pun et al., 2012] Pun, K. I., Steffen, M., and Stolz, V. (2012).

Deadlock checking by a behavioral effect system for lock handling.

Journal of Logic and Algebraic Programming, 81(3):331–354.

A preliminary version was published as University of Oslo, Dept. of Computer Science Technical Report 404, March 2011.

[Pun et al., 2013] Pun, K. I., Steffen, M., and Stolz, V. (2013).

Lock-polymorphic behaviour inference for deadlock checking.

Technical report 436, University of Oslo, Dept. of Informatics.

available electronically at <http://www.ifi.uio.no/~msteffen/download/13/lockpolymorphic-rep.pdf>.