Formalization of a type and effect system for deadlock checking using Coq & Ott

Peter Brottveit Bock\textsuperscript{1,2} and Ka I Pun\textsuperscript{2}  Martin Steffen\textsuperscript{2}  Volker Stolz\textsuperscript{2}

IT University Copenhagen, Danemark
IfI UiO

Bad Honnef, 30. April 2014
Overview

Goal

Find *potential* deadlocks in programs by detecting *data races* through *static analysis*

- Data race:
  - Simultaneous access to shared data with $\geq 1$ write
  - Shared data: mutable, unprotected

- Deadlock:
  - Multiple processes wait for shared resources in a cycle
  - Here: *Locks*

General approach: Reduce deadlock to race checking

- instrument programs with *accesses* to additional global variables (*race variables*) at appropriate *program points*
- no data race $\implies$ deadlock free
• Concurrent language
• Higher-order
• Dynamic lock creation
• Reentrant locks
• Non-lexically scoped locks

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

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

\[
v ::= x \mid l \mid \text{fn } x:T.t \mid \text{fun } f:T.x:T.t
\]
• Concurrent language
• Higher-order
• Dynamic lock creation
• Reentrant locks
• Non-lexically scoped locks

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

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

\[
v ::= x \mid l \mid \text{fn } x : T.t \mid \text{fun } f : T.x : T.t
\]
Type and effect system

Types & effects

• Type: *results* of the computations
• Effect: *behaviour* during the computation(s)

here:

• Uses *program points* $\pi$, to characterize locks according to their origin
• Tracks *relative change* to the lock count
• Captures *static program points* where deadlocks may manifest themselves
• *constraints* to derive the *smallest* possible types and effects
• Context-sensitive analysis
  • Polymorphic types
  • Increase precision
Second lock point

- **Second lock point** (*slp*)
  - A *static* over-approximation of program points where deadlocks may manifest themselves
  - \( p \) holds \( \pi_1 \) and tries to take \( \pi_2 \)

- The type and effect system works thread-locally
- Derives *slp* per thread
Abstract cycle $\Theta$

- A cycle of locks
- A sequence of pairs $p_1 : \pi_1 ; \ldots p_n : \pi_n$
  - $p_i$: process identifiers
  - $\pi_i$: program points where locks are created
- Interpreted as: process $p_1$ has $\pi_1$ and wants $\pi_2$
Second lock point relative to $\Theta$

Given $\Delta_C$ and $\bullet \vdash p \ t_0 : \Delta$, $t$ is a *static second lock point* if:

1. $t = \text{let } x: \mathbb{L}\{...,\pi,...\} = v.\text{lock in } t'$.

2. $\Delta_1 \vdash p \ t :: \Delta_2$ within $\bullet \vdash t_0 :: \Delta$.

3. there exists $\pi'$ s.t.

   $\pi' \in \Delta_1$, $\Theta \vdash p$ has $\pi'$, and $\Theta \vdash p$ wants $\pi$
Type and effect system

**Judgments**

\[ \Gamma \vdash e : T \]

\[ T ::= B \mid \bot \mid T \rightarrow T \]

types

see: “Type and Effect Systems”, Amtoft/Nielson/Nielson, 1999
Type and effect system

Judgments

\[ \Gamma \vdash e : \hat{T} :: \varphi; C \]

\[ \hat{T} ::= B \mid □^r \mid \hat{T} \rightarrow \hat{T} \mid \forall \bar{Y} : C.\hat{T} \]

\[ r ::= \{ \pi \} \mid r \sqcup r \mid \varrho \]

\[ \varphi ::= \Delta \rightarrow \Delta \]

\[ \Delta ::= \bullet \mid \Delta, r : n \mid X \]

\[ C ::= \emptyset \mid \varrho \sqsubseteq r, C \mid X \geq \Delta, C \]

types
lock/label sets/data-flow information
effects/pre- and post specification
lock env./abstract state
constraints

see: “Type and Effect Systems”, Amtoft/Nielson/Nielson, 1999
Type and effect system

\[ \Gamma \vdash t : \hat{T} :: \bullet \rightarrow \Delta_2 ; C \]
\[ \Gamma \vdash \text{spawn } t : \text{Thread} :: \Delta_1 \rightarrow \Delta_1 ; C \]

T-Spawn

\[ \Gamma \vdash \text{new}^\pi L : L^{\{\pi\}} :: \Delta \rightarrow \Delta \]

T-NewL

\[ \rho \text{ fresh} \]
\[ \Gamma \vdash \text{new}^\pi L : L^\rho :: \Delta \rightarrow \Delta ; \rho \supseteq \{\pi\} \]

T-NewL

\[ \Gamma \vdash v : L^\rho :: \Delta \rightarrow \Delta \]
\[ \Gamma \vdash v. \text{lock} : L^\rho :: \Delta \rightarrow \Delta \oplus (\rho : 1) \]

T-Lock

\[ \Gamma \vdash v : L^\rho :: \Delta \rightarrow \Delta ; C_1 \]
\[ X \text{ fresh} \]
\[ C_2 = X \geq \Delta \oplus (\rho : 1) \]
\[ \Gamma \vdash v. \text{lock} : L^\rho :: \Delta \rightarrow X ; C_1 , C_2 \]

T-Lock

Inference system

about 80 inference rules.
An example of cycle length three

- $\Theta$ is given as
  
  $p_0 : \pi_0$
  $p_1 : \pi_1$
  $p_2 : \pi_2$

```
p 0
|   |
|---|--
|   |
|   |
```

```
p 1
|   |
|---|--
|   |
|   |
```

```
p 2
|   |
|---|--
|   |
|   |
```
An example of cycle length three

- $\Theta$ is given as
  
  $$
  p_0 : \pi_0 \\
  p_1 : \pi_1 \\
  p_2 : \pi_2
  $$

```
\begin{array}{lll}
\pi_0.\text{lock} & \pi_1.\text{lock} & \pi_2.\text{lock} \\
!z & !z & \\
\pi_1.\text{lock} & \pi_2.\text{lock} & \pi_0.\text{lock}
\end{array}
```
- Races are *binary*
- Deadlocks are *n-ary*
- To compensate, add *locks* appropriately
Gate locks

- Gate locks
  - *Short-lived*
    - No locking-step before a short-lived lock is released
    - does not lead to more deadlocks
  - Variable access between locking and unlocking steps
  - One variable is guarded by one gate lock
Gate locks

An example of cycle length three

- Θ is given as
  \[ p_0 : \pi_0 \]
  \[ p_1 : \pi_1 \]
  \[ p_2 : \pi_2 \]

- Add gate locks in \( n - 1 \) processes
- Analyze \( n \) combinations

\[ \pi_0.\text{lock} \]
\[ \pi_1.\text{lock} \]
\[ \pi_2.\text{lock} \]
Gate locks

An example of cycle length three

- $\Theta$ is given as:
  - $p_0 : \pi_0$
  - $p_1 : \pi_1$
  - $p_2 : \pi_2$

- Add gate locks in $n - 1$ processes
- Analyze $n$ combinations

\[
\begin{align*}
\pi_0 & . \text{lock} \\
\pi_0 & . \text{unlock} \\
\pi_1 & . \text{lock} \\
\pi_1 & . \text{unlock} \\
\pi_2 & . \text{lock} \\
\pi_2 & . \text{unlock} \\
\pi_0 & . \text{lock} \\
\end{align*}
\]
How reliable are the proofs

---

**Quote from POPLMark\(^a\)**

\(^a\)Mechanized Metatheory for the Masses: The POPLMARK Challenge, 2005, Aydemir et al.

How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?
Ott and Coq

Ott

- DSL for grammars and inference rules
- Simple checks for consistency
- Typeset in \texttt{\LaTeX}
- Translate to Coq, HOL, Isabelle, OCaml definitions
- Generate substitution and free-variables functions

Coq

- Powerful proof assistant: four-color, Feit-Thomson
- Dependent type theory; higher-order predicate logic
- Induction: data types and relations
- Recursive functions, inductive proofs
- Proofs: tactic language

Ott for definitions, Coq for theorems and proofs.
Ott and Coq

Ott

- DSL for grammars and inference rules
- Simple checks for consistency
- Typeset in \LaTeX
- Translate to Coq, HOL, Isabelle, OCaml definitions
- Generate substitution and free-variables functions

Coq

- Powerful proof assistant: four-color, Feit-Thomson
- Dependent type theory; higher-order predicate logic
- Induction: data types and relations
- Recursive functions, inductive proofs
- Proofs: tactic language

Ott for definitions, Coq for theorems and proofs.
metavar var, x ::= \{\text{coq nat}\} \{\text{ coq-equality } \}

grammar expr, e :: e_ ::= 
  | x :: :: \text{Var}
  | e e' :: :: \text{App}
  | \ x . e :: :: \text{Abs (+ bind x in e +)}
  | ( e ) :: S:: \text{Par} \{\text{icho [[e]] } \}
  | e [ x / e' ] :: M:: \text{Subst}

\{\text{coq (subst_expr [[e]] [[x]] [[e']])}\}

terminals :: '' ::= 
  | \ :: :: \text{Lam} \{\text{tex \lambda}\}
  | --> :: :: \text{Arr} \{\text{tex \longrightarrow}\}

substitutions
  single expr var :: subst

defns Step :: '' ::= 
  defn e --> e' :: :: \text{Step:: 's_'} by

-------------------- :: \text{Beta}
  (\ x.e) e' --> e[x/e']

  e --> e'

-------------------- :: \text{AppL}
  e e2 --> e' e2

  e --> e'

-------------------- :: \text{AppR}
  e1 e --> e1 e'
"expr, e ::=
  x
  e e'
  \lambda x. e \text{ bind } x \text{ in } e
  (e) \text{ S}
  e[x/e'] \text{ M}

\[ e \rightarrow e' \]

\[ (\lambda x. e) e' \rightarrow e[x/e'] \text{ S_BETA} \]

\[ e \rightarrow e' \]

\[ e e_2 \rightarrow e' e_2 \text{ S_APPL} \]

\[ e \rightarrow e' \]

\[ e_1 e \rightarrow e_1 e' \text{ S_APPR} \]
**Coq output: Syntax and substitution**

**Inductive expr : Set :=**
- e_Var (x:var)
- e_App (e:expr) (e’:expr)
- e_Abs (x:var) (e:expr).

/** substitutions */
**Fixpoint subst_expr (e5:expr) (x5:var) (e_6:expr) {struct e_6} : expr :=**
- match e_6 with
  - (e_Var x) ⇒ (if eq_var x x5 then e5 else (e_Var x))
  - (e_App e e’) ⇒ e_App (subst_expr e5 x5 e) (subst_expr e5 x5 e’)
  - (e_Abs x e) ⇒ e_Abs x (if list_mem eq_var x5 (cons x nil) then e else (subst_expr e5 x5 e))
- end.
CoQ output: Step relation

Inductive Step : expr → expr → Prop := (* defn Step *)
| s_Beta : forall (x:var) (e e’:expr),
  Step (e_App (e_Abs x e) e’) (subst_expr e x e’)
| s_AppL : forall (e e2 e’:expr),
  Step e e’ →
  Step (e_App e e2) (e_App e’ e2)
| s_AppR : forall (e1 e e’:expr),
  Step e e’ →
  Step (e_App e1 e) (e_App e1 e’).
Formalization in Ott

If code-generation is not used

- Transliteration of the grammars and inference rules into Ott: straightforward
- Only minor modifications needed, e.g. $C ::= \epsilon \mid \Delta \leq \Delta'$, $C$ does not allow $C_1, C_2$
- Rudimentary saneness-checking
  - Consistent usage of language constructs
  - Every string can be parsed unambiguously
- About 80 inference rules.
First challenge: Equivalences

- Program: $P ::= \emptyset \mid p\langle t \rangle \mid P \| P$
- $\|\$ is associative and commutative, with $\emptyset$ as the identity
- Solutions:
  - Quotient-types: not available in Ott/Coq
  - Coq supports setoids (set with equivalence relation) must prove compatibility
  - Modify type systems: add explicit structural rules
  - Use a data-type for sets, e.g. $\text{MSet}$

This is a recurring problem; also applies to constraints and $\alpha$-equivalence
Second challenge: Constraints

How to interpret $C \models C'$

- In the paper: not specified — not of interest
- Define decision algorithm
- Axiomatization — only soundness is needed
- Translate constraints to Coq-propositions
\(\alpha\)-equivalence and capture avoiding substitution

Several ways to represent variables

- Naïve
- Locally nameless
- (Parametric-/weak-) higher-order abstract syntax

Ott offers:

- Naïve and (experimental) locally nameless
- Locally nameless: max one Ott-file
- Generated substitution functions are *not* capture avoiding with naïve representation
- If user-defined translation of grammar is given, the substitution-functions will not work
Fresh Variables

- Claim of fresh variable is global
- Derivations not composable
- Solutions:
  - Create predicate on derivations
  - Rewrite the system so that it is local; threading a counter or stream
\[ \Gamma(x) = \forall \tilde{Y} C. \hat{T} \quad \tilde{Y}' \text{ fresh} \quad \theta = [\hat{T}' / \hat{T}] \]

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

TA_VAR
Formalized Rule

\[ \Gamma(x) = \forall \rho_1 \ldots \rho_j \ X_1 \ldots X_k : C. \hat{T} \]
\[ \theta = [X_1' \ldots X_m'/X_1 \ldots X_k][\rho_1' \ldots \rho_z'/\rho_1 \ldots \rho_j] \]

unique \( \rho_1' \ldots \rho_z' \)
unique \( X_1' \ldots X_m' \)
\( \rho_1' \ldots \rho_z' \) notIn \( FV(\Gamma) \)
\( \rho_1' \ldots \rho_z' \) notIn \( FV(C) \)
\( \rho_1' \ldots \rho_z' \) notIn \( FV(\Delta) \)
\( X_1' \ldots X_m' \) notIn \( FV(\Gamma) \)
\( X_1' \ldots X_m' \) notIn \( FV(C) \)
\( X_1' \ldots X_m' \) notIn \( FV(\Delta) \)
\( \rho_{\text{max}} < \min(\rho_1' \ldots \rho_z') \)
\( X_{\text{max}} < \min(X_1' \ldots X_m') \)

\((\rho_{\text{max}}, X_{\text{max}}) \Gamma \vdash x : \theta \ \hat{T} :: \Delta \rightarrow \Delta; \theta \ C(\max(\rho_1' \ldots \rho_z') + 1, \max(X_1' \ldots X_m') + 1) \)
Approach

- Top down
- Cut-off at “obviously true” lemmas
  - Trade-off: time / formal correctness
- Hint database

Totally ca. 40 lemmas

Proofs done

- numerous structural proofs
- soundness of algo wrt. specification
- one glitch spotted in the soundness-proof
Remarks

• Ott is suited for the first written version of grammars and relations
• Ott restricts choices – formalization even more clumsy
• Formalization highlighted several minor errors/typos and one larger error
• Problem areas:
  • Bound variables and substitution
  • Equivalence classes
  • Fresh variables
Summary

- formal type and effect system
- Transformation guarantees each slp is protected by the same variable
- soundness of the approach
  - Programs with deadlocks \( \implies \) data race in the transformed one
  - Race free in the transformed program \( \implies \) deadlock free in the original one
- Implementation of second lock point analysis in Goblint framework
- machine-checked proof of soundness
Formalization of a type and effect system using Coq and OTT.  
Master's thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, Nov. 2013.

Deadlock checking by data race detection.  
Submitted for journal publication, under review. The paper is an extended version of the conference contribution of FSEN’13, 2013.