Comparison Of Two Approaches For Data-Race Prevention

Erik Kundt
Freie Universität Berlin

Seminar on Software Engineering 2015
Outline

Motivation

Introduction
  - Key contributions
  - Ideas and elements
  - The type system

Formalization
  - Grammars
  - Type checker

Examples
  - Bounded-Buffer Producer-Consumer (BBPC)
  - TStack

Conclusion
Motivation: Data-Races / Race conditions

- When an application depends on the sequence or timing of processes or threads for it to operate properly
- A data-race happens when there are two memory accesses in a program where both:
  - Target the same location
  - Are performed concurrently by two threads
  - Are not reads
  - Are not synchronization operations
Motivation: Improve design quality

- Attributes of software quality:
  - efficiency, modifiability, readability, correctness, robustness, usability
- Decrease defect rate (here: synchronization)
  - Counted during inspections and testings (expensive)
  - Less likely to produce synchronization defects because of reduced complexity
  - Static formal verification
- Difficult to find empirical analyses of the root cause of defects in the literature
Motivation: Trends in programming languages

- Rust[3]: Rust tasks
  - Like threads
  - Developer can choose the low-level details of how they operate
- Go[4]: goroutine
  - independent concurrent thread of control
  - same address space
  - Go has runtime (dynamic checks)
- Paradigm shift: Functional programming
  - High degree of parallelization (Scala)
let (tx, rx) = channel();

// Spawn off an expensive computation
spawn(proc() {
    tx.send(expensive_computation());
});

// Let's see what that answer was
println!("{}", rx.recv());
A

A Time-Aware Type System For Data-Race Protection and Guaranteed Initialization[1]

Nicholas D. Matsakis and Thomas R. Gross
ETH Zurich, Switzerland
{nmatsaki,trg}@inf.ethz.ch

B

Ownership Types for Safe Programming: Preventing Data Races and Deadlocks[2]

Chandrasekhr Boyapati, Robert Lee and Martin Rinard
Laboratory of Computer Science, MIT, Cambridge
{chandra,rhlee,rinard}@lcs.mit.edu
Outline

Motivation

Introduction
   Key contributions
   Ideas and elements
   The type system

Formalization
   Grammars
   Type checker

Examples
   Bounded-Buffer Producer-Consumer (BBPC)
   TStack

Conclusion
About the papers: Key contributions

- Intervals
- Type system that is aware of intervals and the happens before relation
- Flexible data-race protection:
  - lock-free and locked parallel patterns
  - field-by-field, object-by-object
- Controlled object initialization:
  - access control system for fields and methods
- Language Inter and a suitable type checker
About the papers: Key contributions

- Ownership types
- Type system that is aware of ownerships in Java programs
- Simple data-race and deadlock prevention:
  - lock levels and partial order among them
  - lock level polymorphism
- Language extensions for Race-Free Java
- Formal rules for type checking
- Runtime ordering of locks
Outline

Motivation

Introduction
   Key contributions
   Ideas and elements
   The type system

Formalization
   Grammars
   Type checker

Examples
   Bounded-Buffer Producer-Consumer (BBPC)
   TStack

Conclusion
Intervals

- Time as first-class objects
- Represents the span of program time in which a certain piece of code executes
- Statement, method call or asynchronous task
- Prevent data-races: Partially ordered through a happens-before relation
- Can be nested inside another interval
class Subintervals(Interval parent, Lock l) {
    interval a(this.parent) { /* code for a */ }
    interval b(this.parent) { /* code for b */ }
    interval c(this.parent) { /* code for c */ }
    this.a hb this.c;
    this.b hb this.c;
    this.c locks this.l;
}
Ownership types

- Every object has an owner
- Locking discipline specified using type declarations
- Statically enforceable way of specifying object encapsulation
- A class definition in Race-Free Java is parameterized by a list of owners
Ownership types: Tree example

```java
class BalancedTree {
    LockLevel l = new;
    Node<self:l> root = new Node;
}

class Node<self:k> {
    tree Node<self:k> left;
    tree Node<self:k> right;

    synchronized void rotateRight() locks(this) {
        final Node x = this.right; if (x == null) return;
        synchronized (x) {
            final Node v = x.left; if (v == null) return;
            synchronized (v) {
                final Node w = v.right;
                v.right = null;
                x.left = w;
                this.right = v;
                v.right = x;
            }
        }
    }
}
```
Outline

Motivation

Introduction
  Key contributions
  Ideas and elements
  The type system

Formalization
  Grammars
  Type checker

Examples
  Bounded-Buffer Producer-Consumer (BBPC)
  TStack

Conclusion
The Guard interface

// Objects which will serve as guards must implement the interface Guard
Interface Guard {

    // checks whether the guard g permits i to write to the fields guarded by g
    boolean permitsWr(Interval current);

    // same as permitsWr but for reads
    boolean permitsRd(Interval current);

    // checks whether the fields protected by g are immutable for the interval i
    boolean ensuresImm(Interval current);
}

- Intervals and locks implement this interface
Outline

Motivation

Introduction
  Key contributions
  Ideas and elements
  The type system

Formalization
  Grammars
  Type checker

Examples
  Bounded-Buffer Producer-Consumer (BBPC)
  TStack

Conclusion
## Example

### CDeclaration Syntax

```plaintext
cdecl ::= class c(tys fs) extends c(paths) { members }
member ::= gdecl | fdecl | mdecl | idecl | ldecl | hbdecl

gdecl ::= ghost f
fdecl ::= ty f guardedBy path
mdecl ::= void m(tys xs) reqs { lstmts }
req ::= path rel path
idecl ::= interval f(path) { lstmts }
ldecl ::= path locks path
hbdecl ::= path hb path
path ::= x.fs
rel ::= trel | wcrel | locks
trel ::= subOf | inlineSubOf | hb
wcrel ::= permitsWr | permitsRd | ensuresImm | eq
lstmt ::= x : stmt
```
Example

\[
stmt ::= x.f = x \\
| x = x.f \\
| x = \text{new } c[fs \text{ eq } paths] (xs) \\
| x.m(xs) \\
| \text{assert } x \ wcrel \ x \\
\]

\[
ty ::= c[fs \ wcrels \ paths]
\]
Example

\[ P ::= \text{defn}^* \ e \]

\[ \text{defn} ::= \text{class} \ cn \ \text{extends} \ c \ \text{body} \]

\[ c ::= cn \mid \text{Object} \]

\[ \text{body} ::= \text{field}^* \ \text{meth}^* \]

\[ \text{meth} ::= t \ mn(\text{arg}^*) \ e \]

\[ \text{field} ::= [\text{final}] \ t \ fd = e \]

\[ \text{arg} ::= [\text{final}] \ t \ x \]

\[ t ::= c \mid \text{int} \mid \text{boolean} \]

\[ e ::= \text{new} \ c \mid x \mid x = e \mid e.fd \mid e.fd = e \mid e.mn(e^*) \]
\[ e;e \mid \text{let} (\ \text{arg} = e) \ \text{ine} \mid \text{if} (e) \ \text{then} \ e \mid \]
\[ \text{synchronized} (e) \ \text{in} \ e \mid \text{fork} (x^*) \ e \]

\[ cn \in \text{class names} \]

\[ fd \in \text{field names} \]

\[ mn \in \text{method names} \]

\[ x \in \text{variable names} \]
**Example**

\[
\begin{align*}
\text{defn} & := \text{class } c<\text{owner formal}^* > \text{ extends } c \text{ body} \\
\text{c} & := cn<\text{owner}^+ > \mid \text{Object} \\
\text{c} & := cn<\text{owner}^+ > \mid \text{Object} \\
\text{owner} & := \text{formal} \mid \text{self} \mid \text{thisThread} \mid \text{efinal} \\
\text{meth} & := t \text{ mn(arg}^* ) \text{ accesses } | (\text{efinal}^* ) e \\
\text{efinal} & := e \\
\text{formal} & := f \\

f & \in \text{ owner names}
\end{align*}
\]
Outline

Motivation

Introduction
   Key contributions
   Ideas and elements
   The type system

Formalization
   Grammars
   Type checker

Examples
   Bounded-Buffer Producer-Consumer (BBPC)
   TStack

Conclusion
Heart of the type checker
- Records facts deduced by compiler
- As compiler proceeds, new facts are added; nothing is ever removed
- Facts in the environment take the form of tuples:
  - `path rel path` relate two paths
  - `x : ty` record the type for a local variable
A path is just a local variable name[2]

Facts about paths have to be always true as the program executes

Will not become invalidated as fields are updated

Ensuring that all paths referenced from the environment are *stable*

Always stable:

- x or this, variable reassignment is forbidden
- this.parent, parent is constructor argument and immutable

Only stable at certain times:

- aResult: When the class is first created, interval a has not yet executed, and so aResult is not stable
**STMT-LOAD**

\[ \mathcal{E} \leftarrow x_o : c \quad \text{reified}(c, f) = (ty_f; path_g) \quad \Theta = [\text{this} \rightarrow x_o] \]

\[ \mathcal{E} \leftarrow \Theta(path_g) \text{stableBy } x_l \quad \mathcal{E} \leftarrow \Theta(path_g) \text{permitsRd } x_l \]

\[ \mathcal{E} \leftarrow x_l : (x_d = x_o.f) \rightarrow \mathcal{E} + (x_d : \Theta(ty_f)) \]

**STMT-LOAD-IMMUTABLE**

\[ \mathcal{E} \leftarrow x_o : c \quad \text{reified}(c, f) = (ty_f; path_g) \quad \Theta = [\text{this} \rightarrow x_o] \]

\[ \mathcal{E} \leftarrow \Theta(path_g) \text{stableBy } x_l \quad \mathcal{E} \leftarrow \Theta(path_g) \text{ensuresImm } x_l \]

\[ \mathcal{E} \leftarrow x_l : (x_d = x_o.f) \rightarrow \mathcal{E} + (x_d : \Theta(ty_f)) + (x_d \text{ eq } x_o.f) \]

**STMT-STORE**

\[ \mathcal{E} \leftarrow x_o : c \quad \text{reified}(c, f) = (ty_f; path_g) \quad \Theta = [\text{this} \rightarrow x_o] \]

\[ \mathcal{E} \leftarrow \Theta(path_g) \text{stableBy } x_l \quad \mathcal{E} \leftarrow \Theta(path_g) \text{permitsWr } x_l \]

\[ \mathcal{E} \leftarrow x_v : \Theta(ty_f) \]

\[ \mathcal{E} \leftarrow x_l : (x_o.f = x_v) \rightarrow \mathcal{E} \]
A Type checker: The Environment

- Set of rules for reasoning about the typing judgement
- Holds information about class definitions
- Judgements based on state of environment
- Ensures that the program is well typed
1. The owner of an object does not change over time
2. The ownership relation forms a forest of rooted trees, where the roots can have self loops.
3. The necessary and sufficient condition for a thread to access to an object is that the thread must hold the lock on the root of the ownership tree that the object belongs to.
4. Every thread implicitly holds the lock on the corresponding thisThread owner. A thread can therefore access any object owned by its corresponding thisThread owner without any synchronization.
[METHOD]

\[ E' = E, \text{arg}_1..n, \text{locks}(cn_j.l_j j \in 1..k \ [l]_{\text{opt}}) \]

\[ P; E' \vdash_{\text{final}} e_i : t_i \quad P; E' \vdash \text{RootOwner}(e_i) = r_i \]

\[ ls = \text{thisThread}, r_{1..r} \]

\[ l_{\text{min}} = \text{LUB}(cn_j.l_j j \in 1..k) \]

\[ P; E'; ls; l_{\text{min}} \vdash e : t \]

\[ P; E \vdash t \text{mn(\text{arg}_1..n) accesses(e}_{1..r}) \]

\[ \text{locks}(cn_j.l_j j \in 1..k \ [l]_{\text{opt}}) \{e\} \]
Outline

Motivation

Introduction
   Key contributions
   Ideas and elements
   The type system

Formalization
   Grammars
   Type checker

Examples
   Bounded-Buffer Producer-Consumer (BBPC)
   TStack

Conclusion
Two independent parallel tasks
Producer task writes data into a buffer, consumer reads it out
If producer is too fast, buffer becomes full, producer has to wait for consumer to catch up
Reference: synchronous programming
abstract class Stream {
    Stream next guardedBy inter;
    abstract interval inter;
}
// this == c[i], prod == p[i]
class Consumer(Interval parent, Producer prod) extends Stream {
    prod.inter hb inter; // p[i] -> c[i]
    interval inter(parent) {
        /* consume prod.result */
        next = new Consumer(parent, (Producer)prod.next);
    }
}
// this == p[i], cons == c[i-2]
class Producer(Interval parent, Stream cons) extends Stream {
    Object result guardedBy inter;
    cons.inter hb inter; // c[i-2] ⟷ p[i]
    interval inter(parent) {
        result = /* produce item */;
        next = new Producer(parent, cons.next);
    }
}
class DummyConsumer(Interval parent) extends Stream {
    Stream link guardedBy parent;
    interval inter(parent) {
        next = link;
    }
}

class Start {
    void main() {
        join: {
            DummyConsumer one = new DummyConsumer(join);
            DummyConsumer two = new DummyConsumer(join);
            Producer prod = new Producer(join, one);
            one.link = two;
            two.link = new Consumer(join, prod);
        }
    }
}
Outline

Motivation

Introduction
  Key contributions
  Ideas and elements
  The type system

Formalization
  Grammars
  Type checker

Examples
  Bounded-Buffer Producer-Consumer (BBPC)
  TStack

Conclusion
The TStack program: TStack

- Stack of T objects
- TStack is implemented using a linked list

```java
// thisOwner owns the TStack object
// TOwner owns the T objects in the stack.
class TStack<thisOwner, TOwner> {
    TNode<this, TOwner> head = null;

    T<TOwner> pop() accesses (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
    ...
}
```
The TStack program: TNode

```cpp
class TNode<thisOwner, TOwner> {
    T<TOwner> value;
    TNode<thisOwner, TOwner> next;

    T<TOwner> value() accesses (this) {
        return value;
    }

    TNode<thisOwner, TOwner> next() accesses (this) {
        return next;
    }

    ...
}
```
The TStack program

```java
1  class T<thisOwner> { int x=0; }
2
3  TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>;
4  TStack<thisThread, self> s2 = new TStack<thisThread, self>;
```
The TStack program: Ownership relation
Summary

- Both give a practical approach for giving users the ability to implement lock discipline
- Race-Free Java more practical in comparison to Inter
- Proven that the type systems protects against data races
- Usage of similar technique in Rust proves that approach is suitable
For Further Reading

Matsakis, N. and Gross, T. *A Time-Aware Type System For Data-Race Protection and Guaranteed Initialization*. ETH Zurich, Switzerland, published in OOPSLA/SPLASH 2010


Rust programming language *Rust*. http://www.rust-lang.org

Go programming language *Go*. http://www.golang.org

Matsakis, N. *E-Mail conversation*. 03.11.2014