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)

Motivation: Rust

```
// tx is is the sending half (tx for transmission), and rx is the receiving
// half (rx for receiving).
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());
```

Reference Papers

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 A

- 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 B

- 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

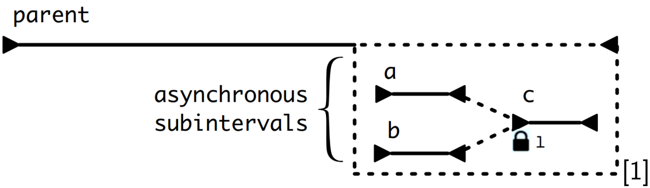
Examples
- Bounded-Buffer Producer-Consumer (BBPC)
- TStack

Conclusion

Intervals A

- 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

Asynchronous Intervals: Parallel Tasks A



```
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 B

- 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

B

```
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

A

```
// 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

Grammar for Inter Pt. 1 A

Example

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

Grammar for Inter Pt. 2 A

Example

```
stmt  := x.f = x
       | x = x.f
       | x = new c[fs eq paths] (xs)
       | x.m(xs)
       | assert x wcrel x
ty    := c[fs wcrels paths]
```

Grammar for Concurrent Java

Example

$$
\begin{array}{ll}
P & := defn{*}\ e \\
defn & := \texttt{class}\ cn\ \texttt{extends}\ c\ body \\
c & := cn \mid \texttt{Object} \\
body & := field{*}\ meth{*} \\
meth & := t\ mn(arg{*})\ e \\
field & := \texttt{[final]}\ t\ fd = e \\
arg & := \texttt{[final]}\ t\ x \\
t & := c \mid \texttt{int} \mid \texttt{boolean} \\
e & := \texttt{new}\ c \mid x \mid x = e \mid e.fd \mid e.fd = e \mid e.mn(e{*}) \\
 & \quad e;e \mid \texttt{let (}\ arg\ \texttt{=}\ e\ \texttt{) in}e \mid \texttt{if (}e\texttt{) then}\ e \mid \\
 & \quad \texttt{synchronized (}e\texttt{) in}\ e\ \texttt{|}\ \texttt{fork (}x{*}\texttt{)}\ e
\end{array}
$$

$$
\begin{array}{ll}
cn & \in \text{class names} \\
fd & \in \text{field names} \\
mn & \in \text{method names} \\
x & \in \text{variable names}
\end{array}
$$

Grammar extension for Race-Free Java

B

Example

$defn := \texttt{class}\ c{<}owner\ formal^*{>}\ \texttt{extends}\ c\ body$

$c := cn{<}owner{+}{>} \mid \texttt{Object}$

$c := cn{<}owner{+}{>} \mid \texttt{Object}$

$owner := formal \mid \texttt{self} \mid \texttt{thisThread} \mid efinal$

$meth := t\ mn(arg^*)\ \texttt{accesses}\ |(efinal^*)\ e$

$efinal := e$

$formal := f$

$f\ \epsilon$ owner names

Outline

Motivation

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

Formalization
- Grammars
- Type checker

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

Conclusion

Type checker: The Environment ε A

- 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

Type checker: Stable paths A

- 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

Type checker: Checking statements A

STMT-LOAD

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \qquad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (\mathit{ty}_{\mathtt{f}}; \mathit{path}_g) \qquad \Theta = [\mathtt{this} \to \mathtt{x}_o] \\ \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{stableBy}\ \mathtt{x}_l \qquad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{permitsRd}\ \mathtt{x}_l}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_d = \mathtt{x}_o.\mathtt{f}) \to \mathcal{E} + (\mathtt{x}_d : \Theta(\mathit{ty}_{\mathtt{f}}))}$$

STMT-LOAD-IMMUTABLE

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \qquad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (\mathit{ty}_{\mathtt{f}}; \mathit{path}_g) \qquad \Theta = [\mathtt{this} \to \mathtt{x}_o] \\ \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{stableBy}\ \mathtt{x}_l \qquad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{ensuresImm}\ \mathtt{x}_l}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_d = \mathtt{x}_o.\mathtt{f}) \to \mathcal{E} + (\mathtt{x}_d : \Theta(\mathit{ty}_{\mathtt{f}})) + (\mathtt{x}_d\ \mathtt{eq}\ \mathtt{x}_o.\mathtt{f})}$$

STMT-STORE

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \qquad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (\mathit{ty}_{\mathtt{f}}; \mathit{path}_g) \qquad \Theta = [\mathtt{this} \to \mathtt{x}_o] \\ \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{stableBy}\ \mathtt{x}_l \qquad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{permitsWr}\ \mathtt{x}_l \\ \mathcal{E} \vdash \mathtt{x}_v : \Theta(\mathit{ty}_{\mathtt{f}})}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_o.\mathtt{f} = \mathtt{x}_v) \to \mathcal{E}}$$

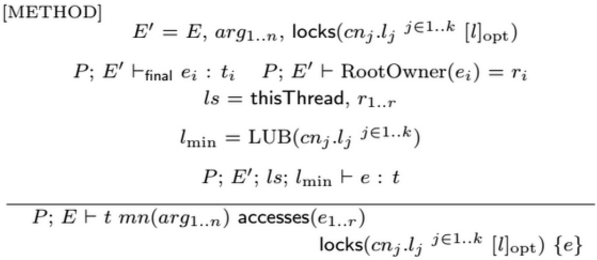
Type checker: The Environment ε B

- 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

Type checker: Ownership properties B

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.

Type checker: Checking statements B



[METHOD]

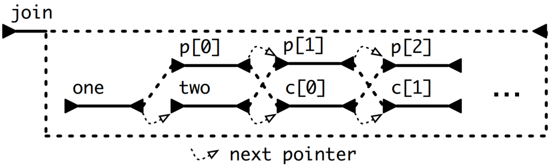
$$\frac{\begin{array}{c} E' = E, arg_{1..n}, \mathsf{locks}(cn_j.l_j^{\ j\in 1..k}\ [l]_{\mathrm{opt}}) \\ P; E' \vdash_{\mathsf{final}} e_i : t_i \qquad P; E' \vdash \mathrm{RootOwner}(e_i) = r_i \\ ls = \mathsf{thisThread}, r_{1..r} \\ l_{\mathrm{min}} = \mathrm{LUB}(cn_j.l_j^{\ j\in 1..k}) \\ P; E'; ls; l_{\mathrm{min}} \vdash e : t \end{array}}{\begin{array}{l} P; E \vdash t\ mn(arg_{1..n})\ \mathsf{accesses}(e_{1..r}) \\ \qquad\qquad \mathsf{locks}(cn_j.l_j^{\ j\in 1..k}\ [l]_{\mathrm{opt}})\ \{e\} \end{array}}$$

Outline

BBPC A

- 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

BBPC: Stream A



```
abstract class Stream {
    Stream next guardedBy inter;
    abstract interval inter;
}
```

BBPC: Consumer A

```
// 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);
    }
}
```

BBPC: Producer A

```
// 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);
    }
}
```

BBPC: Putting all together A

```
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` B

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

```
// 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`

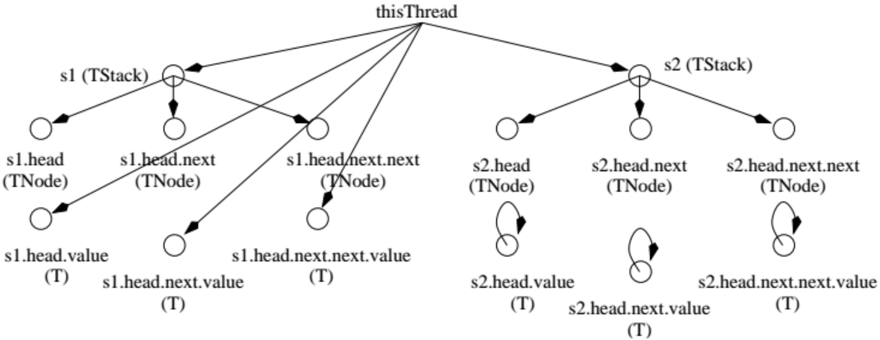
B

```
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 B

```
class T<thisOwner> { int x=0; }

TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>;
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 I

- 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
- Boyapati, C. and Lee, R. and Rinard, M. *Ownership Types for Safe Programming: Preventing Data Races and Deadlocks*. Laboratory of Computer Science, MIT, Cambridge, published in OOPSLA 2002
- 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