Concurrent Programming with Futures

Presented by

Michael Hicks

Oregon Summer School 2006
on Concurrent and Distributed Software

1

This Lecture

- Introducing Futures
  - Programming model
  - Implementation in Multilisp
    (Halstead 1985, Mohr et al 1991, Flanagan and Felleisen 1995)
- Futures in Java
  - Java.util.concurrent
  - Transparency with static typing
    (Pratikakis et al, 2004)
  - Safety
    (Welc et al, 2005)

2

Thanks

- Adam Welc at Purdue for most of the Safe Futures slides

3

Scheme Merge Sort

```
(define (split x) …)
(define (merge x y) … (car x) …)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (mergesort y) (mergesort z))))
```

How to make parallel?

4

Explicit Approach

- Threads, Message Passing
- Problems
  - Message passing requires partitioning the data among different address spaces
  - Must write code to exploit resources of underlying platform
  - Significant code changes

5

Implicit Approach

- Rely on the compiler to figure out opportunities for parallelism
- Problems
  - Really hard!
  - Instruction-level and loop-level parallelism can be inferred, but
  - Inferring larger "subroutine"-level parallelism has had less success.

6

Middle Ground: Futures

- Use **future** annotation [Halstead 85]
  - **(future e)** indicates **e** may run concurrently with parent
- Benefits
  - Notationally lightweight
    - Sequential algorithm still manifest
  - Implement to let concurrency be determined by the run-time system, based on system resources
  - Coordination between concurrent computations is transparent

7

Where to annotate?

```
(define (split x) …)
(define (merge x y) … (car x) …)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (mergesort y) (mergesort z))))
```

No - result is used immediately in following call

8

Where to annotate?

```
(define (split x) …)
(define (merge x y) … (car x) …)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (mergesort y) (mergesort z))))
```

Yes - recursive calls can operate in parallel

9

Multilisp Merge Sort

```
(define (split x) …)
(define (merge x y) … (car x) …)
(define (mergesort x)
  (let ((y,z) (split x))
    (merge (future (mergesort y)
                (future (mergesort z)))))
```

10

Basic Implementation Approach

- **(future e)**
  - fork a new thread **T** to evaluate **e**
  - return a proxy **p** to the parent
    - called a *future* or *promise*
- **T** stores result of **e** into **p**
- Run-time system extracts result from **p** when accessed by the parent
  - Called a *touch* or *claim*

11

Implementing Touches

```
(define (merge x y) … (car x) …)
```

Could be a future…

- Futurized implementation of **(car x)**

  (if (pair? (**touch** x))
    (*get first elem of x*)
    (*error*))

- Where **(touch x)** is

  (if (future? x) (get x) x)

Blocks until result has been computed

12

Optimization I

- Forking a thread per future could be expensive and without advantage
  - Particularly if not many CPUs
- Idea: only use as many threads as there are processors [Mohr et al 91]
  - At a **future** call, use idle thread, if any
  - Otherwise, continue using current thread
    - Save continuation on a separate queue
  - When a thread would block, save the current continuation and grab one from the queue

13

Optimization II

- Once a **future** computation completes, its result is immutable
  - Proxy and further touches redundant
- Thus
  - Use garbage collector to throw away the proxy and replace with the result [Halstead 85]
  - Avoid touching at all if static analysis can prove it's unnecessary [Flanagan & Felleisen 95]

14

What about side effects?

```
(let ((x 1)
      (_ (set! x 2))
  x)
(let ((x 1)
      (_ (future (set! x 2))
  x)
```

- Sequential version: 2
- Parallel version: either 1 or 2

15

Safety and Concurrency

- Most Multilisp code is functional
  - No worry about inconsistencies
- Non-functional code
  - Encapsulate abstractions that are mutable
  - Synchronize all accesses
    - Like "fully synchronized" Vector class in Java
- What if the programmer makes a mistake?
  - Will look at this later in the talk

16

Futures in Java

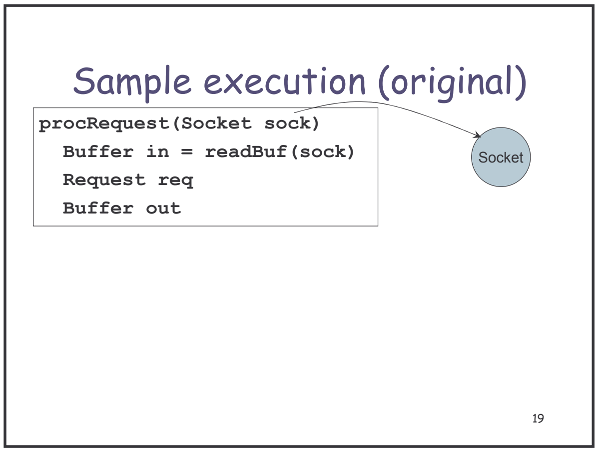
- Java is not Lisp/Scheme
  - Static typing
  - Side-effects are far more prevalent
- Approach
  - Static analysis and transformation [Pratikakis et al 2004]
  - Detect safety problems at run-time [Welc et al 2005]

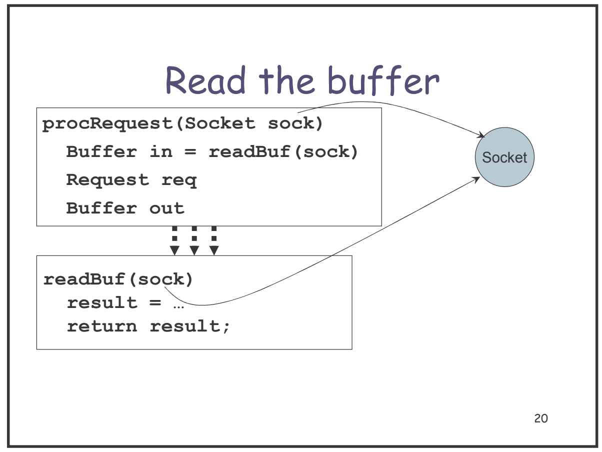
17

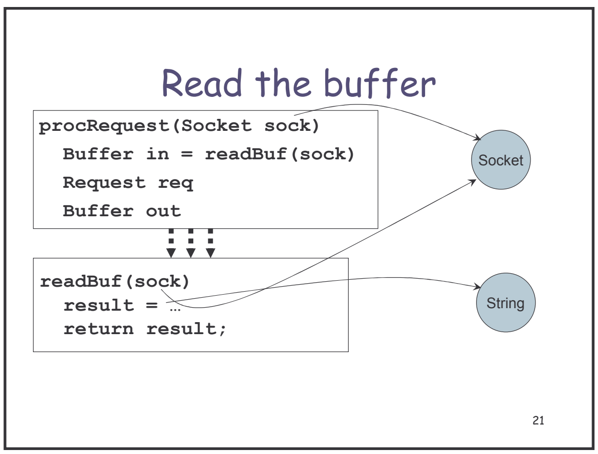
Example: HTTP handler

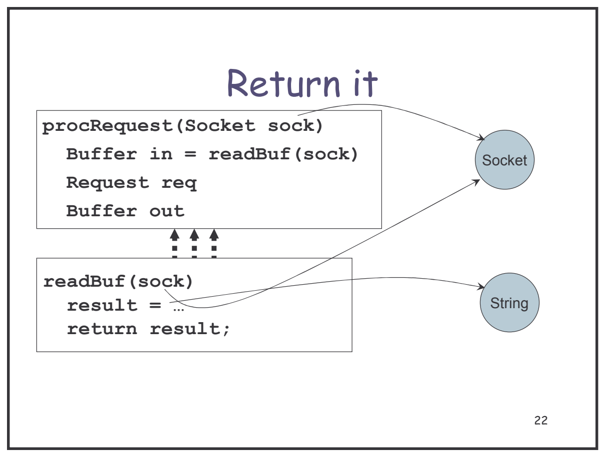
```
procRequest(Socket sock) {
  Buffer in = readBuf(sock);
  Request req = translate(in);
  Buffer out = process(req);
  writeBuf(sock,out);
}
Request translate(Buffer in) {
  Request result;
  … in.foo() …
  return result;
}
…
```

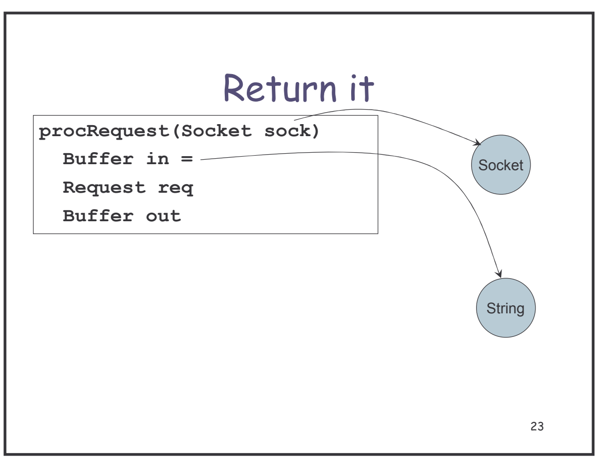
18

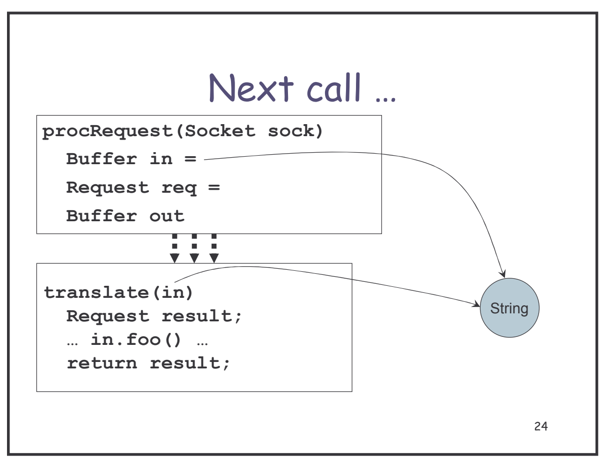

Sample execution (original)
procRequest(Socket sock)
Buffer in = readBuf(sock)
Request req
Buffer out
Socket
19


Read the buffer
procRequest(Socket sock)
Buffer in = readBuf(sock)
Request req
Buffer out
readBuf(sock)
result = …
return result;
Socket
20


Read the buffer
procRequest(Socket sock)
Buffer in = readBuf(sock)
Request req
Buffer out
readBuf(sock)
result = …
return result;
Socket
String
21


Return it
procRequest(Socket sock)
Buffer in = readBuf(sock)
Request req
Buffer out
readBuf(sock)
result = …
return result;
Socket
String
22


Return it
procRequest(Socket sock)
Buffer in =
Request req
Buffer out
Socket
String
23


Next call …
procRequest(Socket sock)
Buffer in =
Request req =
Buffer out
translate(in)
Request result;
… in.foo() …
return result;
String
24

Suppose we had **future**

```
procRequest(Socket sock) {
  Buffer in = future readBuf(sock);
  Request req = future translate(in);
  Buffer out = future process(req);
  writeBuf(sock,out);
}
```

25

Sample execution (async)

```
procRequest(Socket sock)
  Buffer in = future readBuf(sock);
  Request req =
  Buffer out
```

Socket

26

Read the buffer in new thread

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Socket

spawn thread

```
readBuf(sock)
  result = …
  return result;
```

27

Placeholder to caller

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Socket

Future

```
readBuf(sock)
  result = …
  return result;
```

28

Calculate result in child

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Socket

Future

```
readBuf(sock)
  result = …
  return result;
```

String

29

Store in placeholder

```
procRequest(Socket sock)
  Buffer in =
  Request req =
  Buffer out
```

Socket

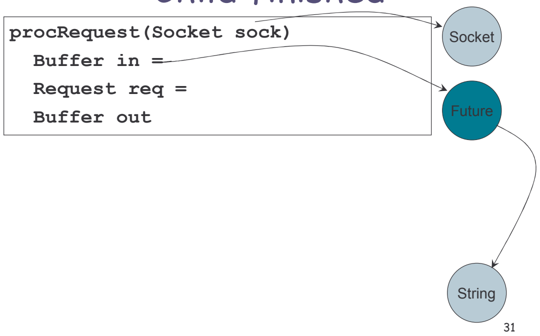
Future

```
readBuf(sock)
  result = …
  return result;
```

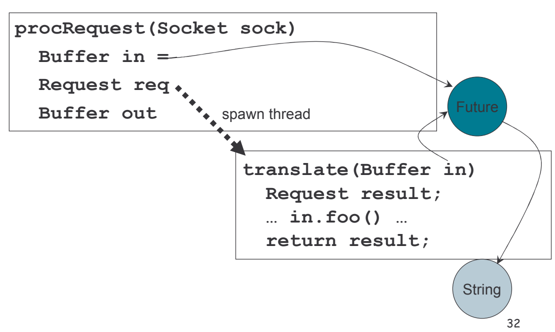
String

30

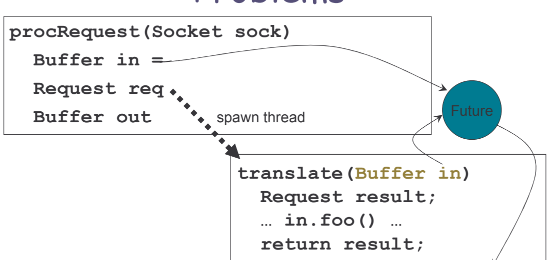
Child finished



Next call

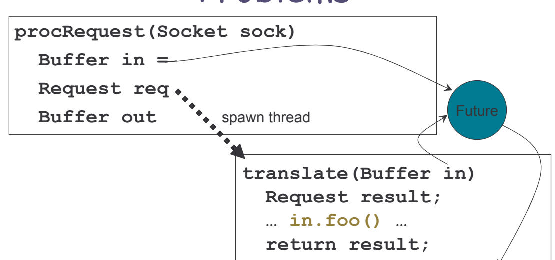


Problems



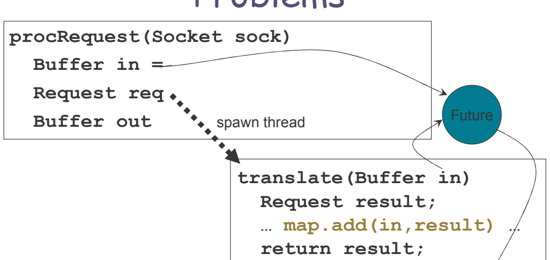
Callee and caller type not correct
(should be Future or Object)

Problems



Callee operations on argument
assume a Buffer, not a Future

Problems



Callee operations might violate
transparency

java.util.concurrent

- Concurrency library in Java 1.5

```
public interface Future<T> {
  T get();
  …
}
```

```
public class FutureTask<T>
  implements Future<T> { … }
```

java.util.concurrent

- Could convert our HTTP program by hand to use this library, but
  - Would take a lot of code rewriting
    - Adjust the types, insert code to spawn the thread, to extract the underlying object from the future when needed, catch any exceptions that could be thrown …
  - Makes it hard to change policies later
    - What if I later want only one of the methods to be async?
  - Might result in inadvertent transparency violation

37

Proxy Design Pattern

- The proxy and object share an interface
- Addresses typing and code problems, but
  - Still might have to change the program to introduce an interface type, rather than the concrete type
  - Interfaces only name methods
    - Thus field accesses disallowed
  - Does not solve the transparency problems
    - Still can use ==, instanceof, etc. to distinguish between the object and its proxy

38

Solution: Proxy Programming Framework

[Pratikakis et al 2004]

- User indicates
  - where proxies are introduced, e.g. by **future** annotations on method calls.
  - what to do when a proxy's underlying object is required, e.g. when calling a method or extracting a field from a proxy
- An automatic program transformation inserts necessary code
  - For proxy introduction and coercion, avoiding transparency violations

39

Benefits

- No code changes needed by hand
- Policies can be changed easily
- Prevents violations of transparency
- Has applications beyond futures
  - Tracking of security-sensitive data
  - Not-null types
  - Stack allocation of objects

40

Summary of Approach

- Formalization of analysis and transformation
  - Formally proven correct
- Prototype implementation
  - Built on the SOOT Java bytecode analysis toolkit
- Experimental evaluation, considering
  - Analysis running time
  - Quality of generated code

41

Three-Stage Transformation

- Inference
  - Generate constraint graph describing how proxies could flow through the program.
- Constraint solving
  - Solve the constraints, identifying where coercions are needed.
- Transformation
  - Rewrite any classes requiring coercions, type changes, etc.

42

Inference

- Each type has *qualifier* **proxy** or **nonproxy**
  - Like **final**, but never appears in source programs
  - **proxy** indicates the value *may* be a proxy
  - **nonproxy** indicates it is *definitely not* a proxy
  - **nonproxy** < **proxy**
- *Qualifier inference* is used to assign qualifiers to types in the program, based on
  - Where proxies are introduced
  - Where non-proxies are required
  - How values flow between these locations

Inference

- Whenever a **nonproxy** is required, e.g. to call a method, the analysis notes that the value may need to be coerced
  - E.g., get the underlying object from a future
- Coercions are flow-sensitive
  - Once we check at runtime that a value is a non-proxy, we can assume it is from thereon
  - Like touch optimization in Multilisp
    - Can discard placeholder and avoid later touches

Constraint Solving

- Standard
  - Based on graph reachability
- If a possible **proxy** indeed flows to a location requiring a **nonproxy**, there will be a path between the two in the graph.
  - Requires a coercion as **proxy** $\not\leq$ **nonproxy**

Transformation

- For each class that
  - Requires a coercion
  - Introduces a proxy
- … rewrite the class as necessary to insert code to implement them
  - Code provided by the user
- Must avoid transparency violations
  - Forward calls to .equals(), .hashcode(), etc.

Before Analysis: procRequest

```
procRequest(Socket sock) {
  Buffer in = future readBuf(sock);
  Request req = future translate(in);
  Buffer out = future process(req);
  writeBuf(sock,out);
}
```


Inference constraints

```
procRequest(Socket sock) {
  Buffer in = proxy readBuf(sock);

  Request req = proxy translate(in);

  Buffer out = proxy process(req);

  writeBuf(sock,out);
}
```

To method body for translate

After transcformation

```
procRequest(Socket sock) {
  Object in = new Proxy {
                 private Object result;
                 public void run() {
                   result = readBuf(sock); }
                 public synchronized Object get() {
                   … return result; }
                 public bool equals(Object o) {
                   return get().equals(o); }
               }();
               TPE.run((Runnable)in);
  Object req = new Proxy { …translate(in)…
  Object out = new Proxy { …process(req)…
  writeBuf(sock,out);
}
```

49

Before Analysis: translate

```
Request translate(Buffer in) {
  Request result;
  … in.foo() …
  return result;
}
```

50

Inference Constraints

From call in procRequest

```
Request translate(Buffer in) {
  Request result;

  … nonproxy in.foo() …

  return result;
}
```

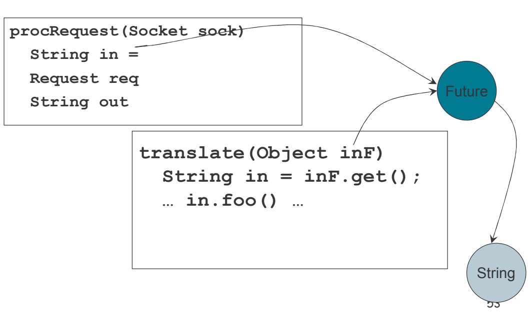
51

After transformation

```
Request translate(Object inF) {
  Request result;
  String in =
    (String)(inF instanceof Proxy ?
             inF.get() :
             inF)
  … in.foo() …
  return result;
}
```

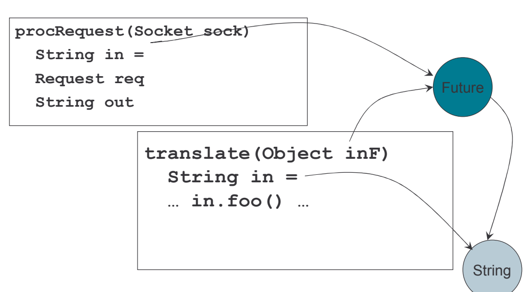
52

Example



53

After executing coercion



54

User control of Analysis

- Analysis determines where coercions are needed, then rewrites classes.
- What code to insert depends on the proxy being used; provided by the user
  - Can support *lazy computation* using the same code for coercions, with proxy.get() to run the invocation.

55

Analysis Characterization

- Analysis is context-insensitive, path-insensitive, and partly flow-sensitive (only with regard to coercions).
- Operates on whole program
  - User can control whether standard class libraries should also be rewritten

56

Other Applications

- Checking for transparency violations
  - Follow the flow of design-pattern proxies (which use an interface)
  - Require identity-revealing operations to be only on non-proxies
    - Argument to ==
    - Argument to instanceof
    - Argument to downcast
  - If any coercions are needed, reveals potential transparency violation

57

Other Applications

- Not-null types
  - Two qualifiers *null* and *notnull*
    - *notnull* < *null*
  - Coercion implemented as null-check
- Stack-allocated objects
  - Two qualifiers *stack* and *nonstack*
    - *stack* < *nonstack*
  - Coercions introduced when
    - assigning *nonstack* to a field or return value
    - Performing an identity-revealing operation (e.g. hashcode)

58

Implementation

- Modified the SOOT bytecode analysis framework
  - Three-address code, SSA-like intermediate representation called Jimple
  - Extended Jimple with opcode to indicate proxy introduction
- User-provided classes dictate what expression forms may require coercions and how they are implemented

59

Experiments

- Overhead of inserted dynamic checks
- Cost of running the analysis
- Benefits to target applications

60

Dynamic Check Overhead

```
Object p, o = …;
for (int i = 0; i<N; i++) {
  p = o; p.m();
}
```

test	tot (s)	per-check (ns)	% ovr
no claim	2.154	*n/a*	*n/a*
spurious claim	2.401	35	10%
necessary claim	3.567	141	65%

61

Sample Application: Async RMI

```
Service findService(LocalPeer self,
                    String name) {
  Service s = self.getService(name);
  if (s != null) return s;
  self.forward(…);
  return getRemoteService(self,name);
}
```

62

Sample Application: Async RMI

```
Service findService(LocalPeer self,
                    String name) {
  Service s = self.getService(name);
  if (s != null) return s;
  Async.invoke(self.forward(…));
  return getRemoteService(self,name);
}
```

63

Sample Application: Async RMI

```
Service findService(LocalPeer self,
                    String name) {
  Service s = self.getService(name);
  if (s != null) return s;
  Async.invoke(…,self.forward(…));
  return Lazy.invoke(
    getRemoteService(self,name));
}
```

64

Sample Application: Async RMI

Version	Services requested and used									
	1	2	3	4	5	6	7	8	9	10
Orig	11	22	30	41	54	60	85	78	96	104
Async	11	24	32	43	53	61	76	81	90	101
Orig + delay	100	192	282	370	462	562	647	738	828	914
Async + delay	100	107	110	120	124	137	138	143	151	156

- Adding asychrony provides a performance benefit for higher-latency networks when messages can be retrieved in parallel
- Otherwise, network latency dominates, so asychrony not helpful

65

Async RMI Analysis Time

Analysis	Time	classes			claims
		analyzed	w/ fut.	re-written	
FI	139	1319	17	3	3
FS	218	1319	9	2	1
spark	126	1320	*n/a*	*n/a*	*n/a*

- Flow-insensitive version adds little cost to points-to analysis
- Flow-sensitive version adds greater cost
  - Currently over-eagerly introduces flow-sensitive nodes; can be more on-demand

66

Other Applications

- Checking for transparency violations of design-pattern proxies
  - In SOAP/RMI library (2087 classes analyzed)
  - In SOOT framework (2510 classes analyzed)
- Chose various locations to introduce a design-pattern proxy
  - Found that doing so would have introduced as many as 7 transparency violations.

67

Summary

- Proxy programming framework provides a way to introduce futures to Java *transparently*
  - Write the annotation as in Multilisp
  - Compiler inserts code to touch possible futures, with some optimizations
  - Ensures placeholder not mistaken for original object
- Next up: worrying about side effects ...

68

Futures - Safety

If sequential program *P* is annotated with futures to yield concurrent program P_F, then the observable behavior of *P* is equivalent to P_F

- Logical serial order trivially satisfied when no side-effects
- Problems arise with mutation of shared data

69

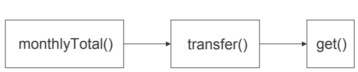
Running Example

```
Account s;   // savings
Account c;   // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

70

Terminology

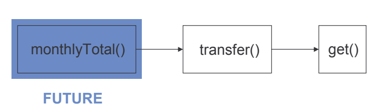
```
Account s;   // savings
Account c;   // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```



71

Terminology

```
Account s;   // savings
Account c;   // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```



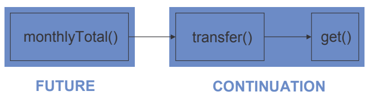
72

Terminology

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```



73

Logical Serial Order

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s = 100 c = 100 global = 0

T_F
future
continuation
T_M

74

Logical Serial Order

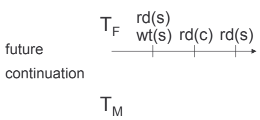
Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

210

s = 110 c = 100 global = 0



75

Logical Serial Order

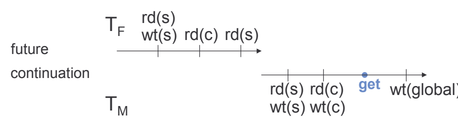
Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

210

s = 60 c = 150 global = 210



76

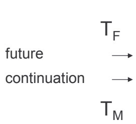
Arbitrary Interleaving

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s = 100 c = 100 global = 0



77

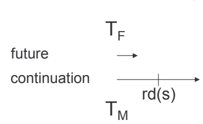
Arbitrary Interleaving

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }

void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s = 100 c = 100 global = 0

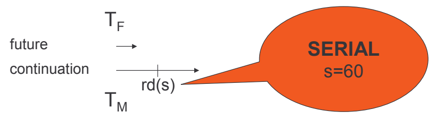


78

Arbitrary Interleaving

```
Account s;   // savings
Account c;   // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount)  {
   s.withdraw(amount);
   c.deposit(amount);  }
```

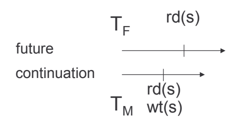
s = 100   c = 100   global = 0



Arbitrary Interleaving

```
Account s;   // savings
Account c;   // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount)  {
   s.withdraw(amount);
   c.deposit(amount);  }
```

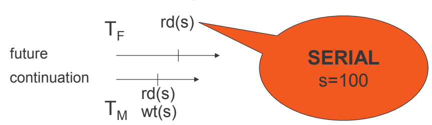
s = 50   c = 100   global = 0



Arbitrary Interleaving

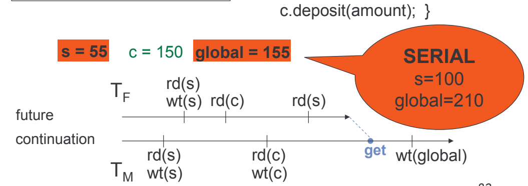
```
Account s;   // savings
Account c;   // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount)  {
   s.withdraw(amount);
   c.deposit(amount);  }
```

s = 50   c = 100   global = 0



Arbitrary Interleaving

```
Account s;   // savings
Account c;   // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {                       155
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount)  {
   s.withdraw(amount);
   c.deposit(amount);  }
```



What Happened?

- Concurrency of shared updates led to unexpected behavior
- Updates from continuation leaked into future
  - monthlyTotal() *should not* see results of transfer()
- Results computed by future were not available for continuation
  - transfer() *supposed to* see results of monthlyTotal()

Two Kinds of Violations

- *Forward Dependency* Violation
  - Continuation *does not* observe an effect of the future computation when it should have serially (or observes the wrong one)
- *Backward Dependency* Violation
  - Future *does* observe an effect of the continuation when it would not have serially

Avoiding Safety Violations
[Welc et al 2005]

- Formal framework for reasoning about safe futures
  - Proof that schedules that do not exhibit forward or backward dependency violations are equivalent to serial
- Implementation that ensures safe schedules
  - Uses optimistic techniques

85

Implementation Overview

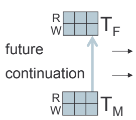
- Data accesses hashed into read and write maps. Maps used by continuation to *detect* conflicts for accesses from its future
  - Detects forward dependency violations
- Versions used by future to *prevent* seeing updates by its continuation
  - Prevents backward dependency violations
- Automatic roll-back when conflict detected

86

Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

s = 100   c = 100   global = 0

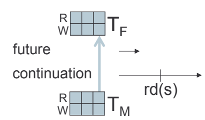


87

Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

s = 100   c = 100   global = 0

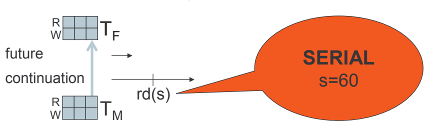


88

Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

s = 100   c = 100   global = 0

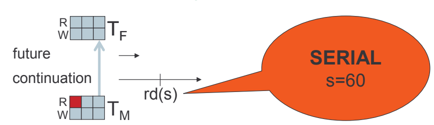


89

Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

s = 100   c = 100   global = 0

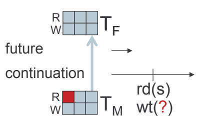


90

Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

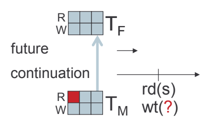
s = 100 c = 100 global = 0



Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

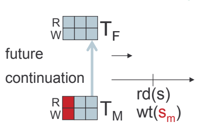
s_M
s = 100 c = 100 global = 0



Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

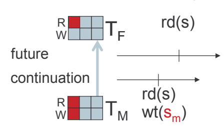
s_M= 50
s = 100 c = 100 global = 0



Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

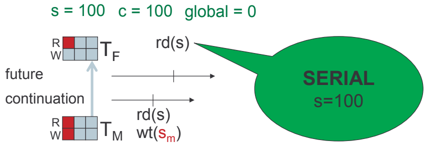
s_M= 50
s = 100 c = 100 global = 0



Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

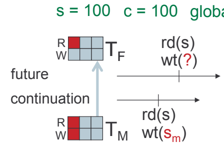
s_M= 50
s = 100 c = 100 global = 0

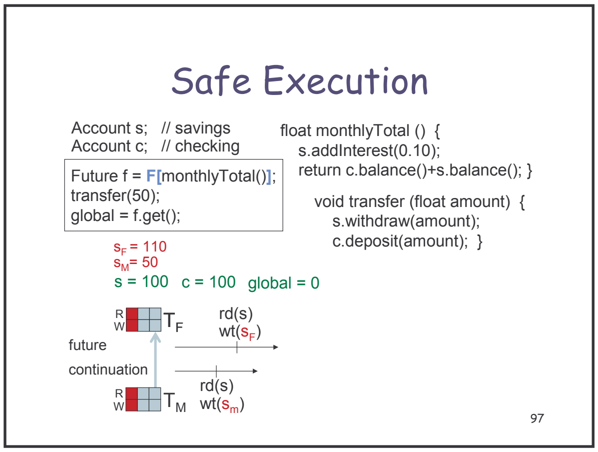


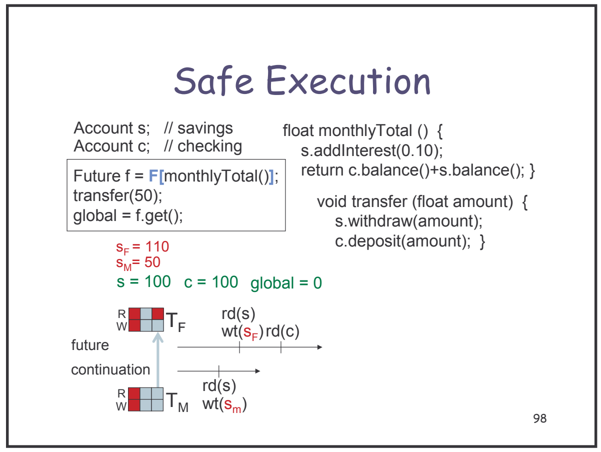
Safe Execution

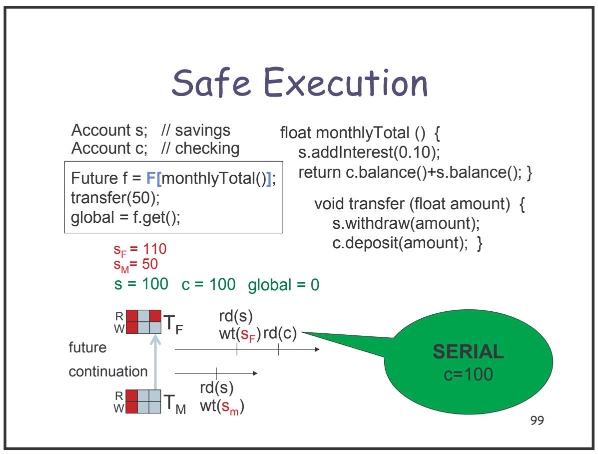
```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

   void transfer (float amount) {
      s.withdraw(amount);
      c.deposit(amount); }
```

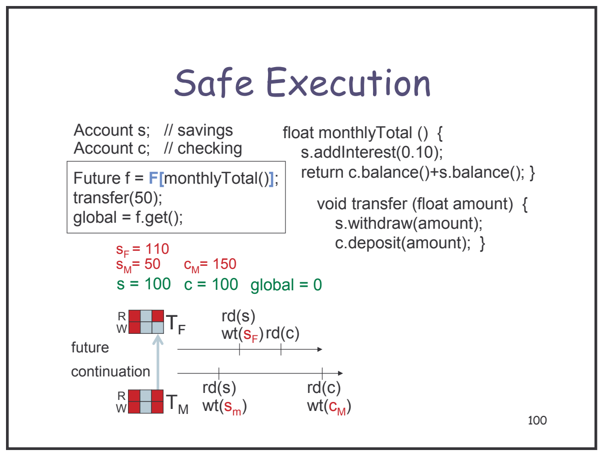
s_M= 50
s = 100 c = 100 global = 0

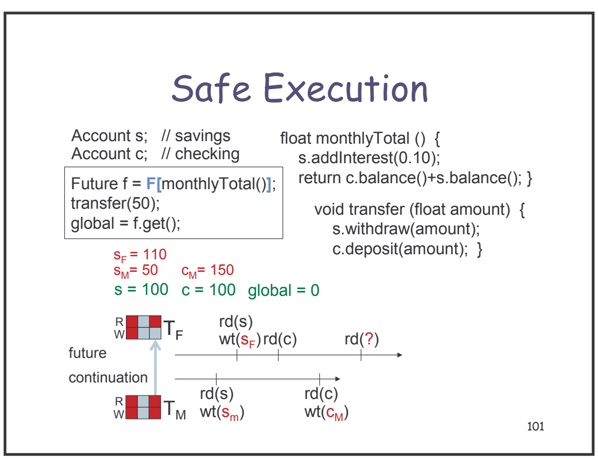


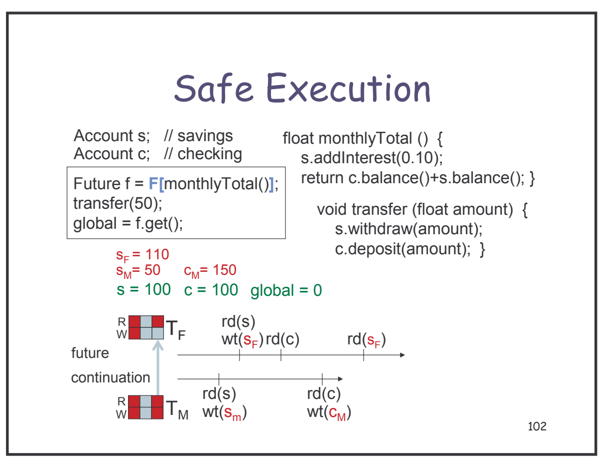

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
sF = 110
sM= 50
s = 100 c = 100 global = 0
R
W
TF
rd(s)
wt(sF)
future
continuation
R
W
TM
rd(s)
wt(sm)
97

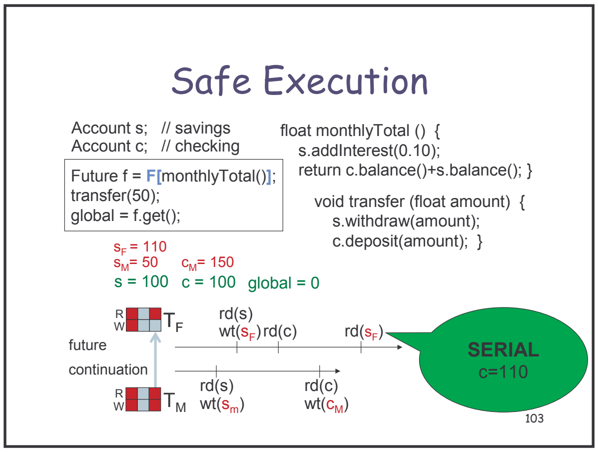

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
sF = 110
sM= 50
s = 100 c = 100 global = 0
R
W
TF
rd(s)
wt(sF) rd(c)
future
continuation
R
W
TM
rd(s)
wt(sm)
98

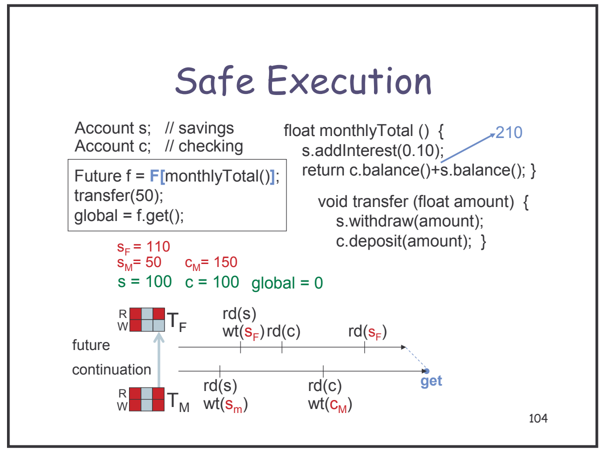

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
sF = 110
sM= 50
s = 100 c = 100 global = 0
R
W
TF
rd(s)
wt(sF) rd(c)
SERIAL
c=100
future
continuation
R
W
TM
rd(s)
wt(sm)
99

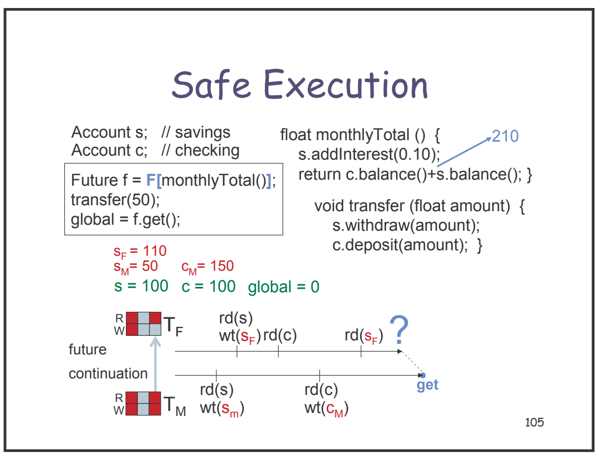

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
sF = 110
sM= 50 cM= 150
s = 100 c = 100 global = 0
R
W
TF
rd(s)
wt(sF) rd(c)
future
continuation
R
W
TM
rd(s)
wt(sm)
rd(c)
wt(cM)
100

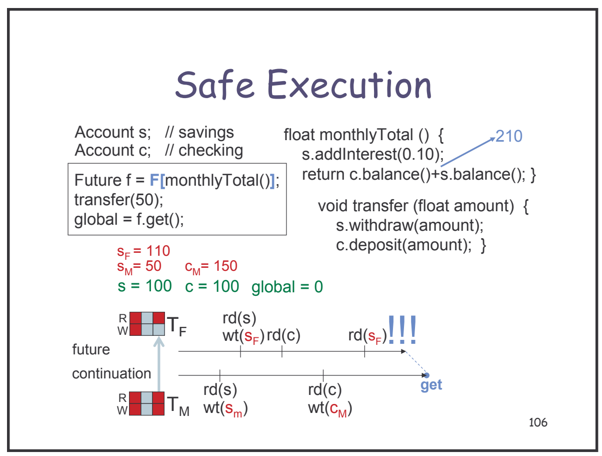

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
sF = 110
sM= 50 cM= 150
s = 100 c = 100 global = 0
R
W
TF
rd(s)
wt(sF) rd(c)
rd(?)
future
continuation
R
W
TM
rd(s)
wt(sm)
rd(c)
wt(cM)
101

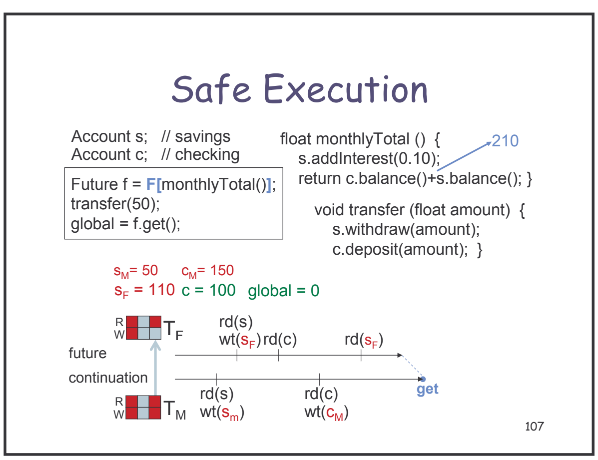

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
sF = 110
sM= 50 cM= 150
s = 100 c = 100 global = 0
R
W
TF
rd(s)
wt(sF) rd(c)
rd(sF)
future
continuation
R
W
TM
rd(s)
wt(sm)
rd(c)
wt(cM)
102

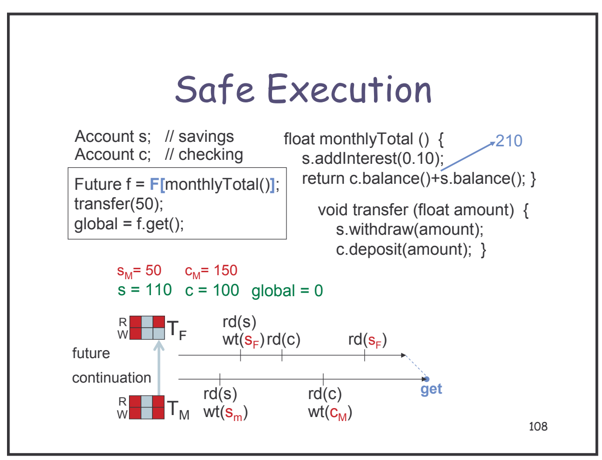

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_F = 110
s_M= 50 c_M= 150
s = 100 c = 100 global = 0
R W T_F
rd(s)
wt(s_F) rd(c) rd(s_F)
future
continuation
R W T_M
rd(s) wt(s_m)
rd(c) wt(c_M)
SERIAL
c=110
103

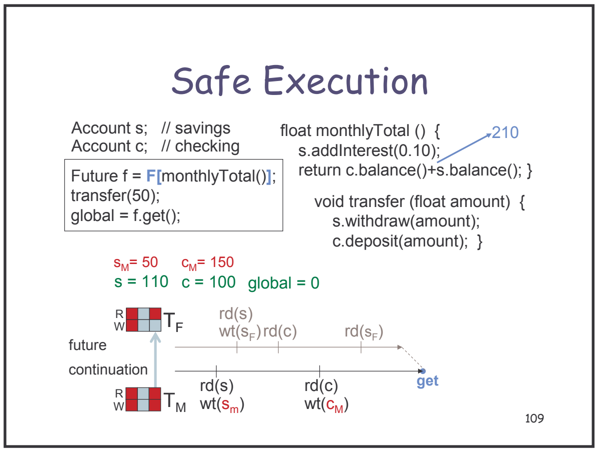

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_F = 110
s_M= 50 c_M= 150
s = 100 c = 100 global = 0
R W T_F
rd(s)
wt(s_F) rd(c) rd(s_F)
future
continuation
get
R W T_M
rd(s) wt(s_m)
rd(c) wt(c_M)
104

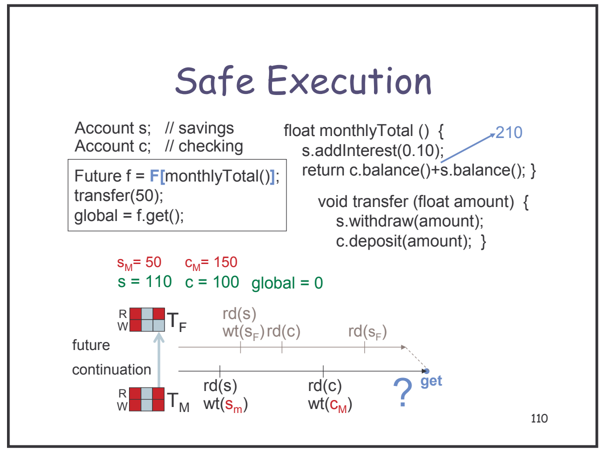

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_F = 110
s_M= 50 c_M= 150
s = 100 c = 100 global = 0
R W T_F
rd(s)
wt(s_F) rd(c) rd(s_F) ?
future
continuation
get
R W T_M
rd(s) wt(s_m)
rd(c) wt(c_M)
105

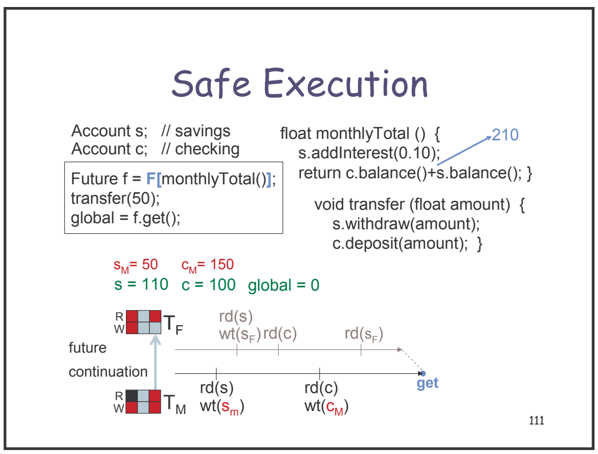

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_F = 110
s_M= 50 c_M= 150
s = 100 c = 100 global = 0
R W T_F
rd(s)
wt(s_F) rd(c) rd(s_F) !!!
future
continuation
get
R W T_M
rd(s) wt(s_m)
rd(c) wt(c_M)
106

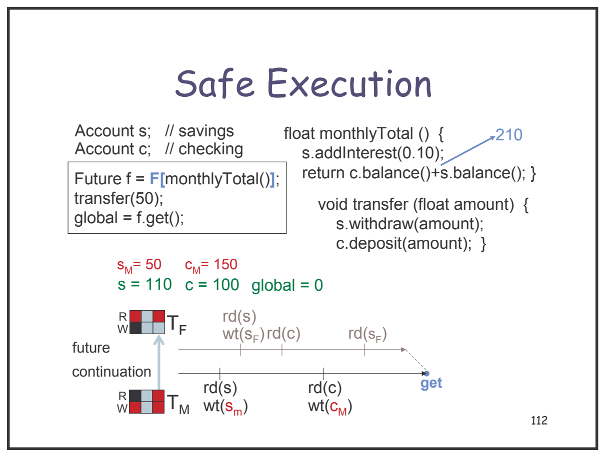

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s_F = 110 c = 100 global = 0
R W T_F
rd(s)
wt(s_F) rd(c) rd(s_F)
future
continuation
get
R W T_M
rd(s) wt(s_m)
rd(c) wt(c_M)
107

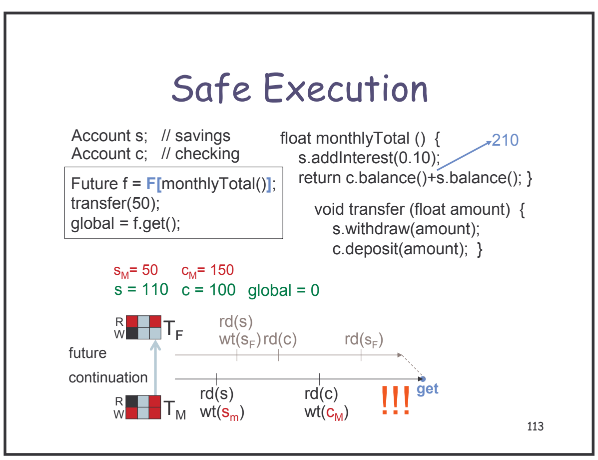

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s = 110 c = 100 global = 0
R W T_F
rd(s)
wt(s_F) rd(c) rd(s_F)
future
continuation
get
R W T_M
rd(s) wt(s_m)
rd(c) wt(c_M)
108

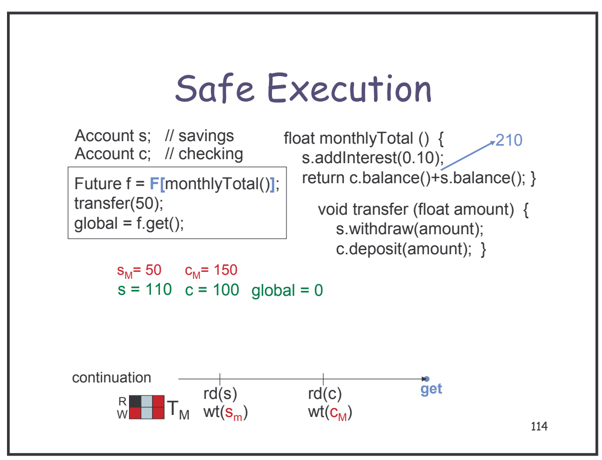

Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s = 110 c = 100 global = 0
R W T_F
rd(s) wt(s_F) rd(c) rd(s_F)
future
continuation
get
R W T_M
rd(s) wt(s_m) rd(c) wt(c_M)
109


Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s = 110 c = 100 global = 0
R W T_F
rd(s) wt(s_F) rd(c) rd(s_F)
future
continuation
? get
R W T_M
rd(s) wt(s_m) rd(c) wt(c_M)
110


Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s = 110 c = 100 global = 0
R W T_F
rd(s) wt(s_F) rd(c) rd(s_F)
future
continuation
get
R W T_M
rd(s) wt(s_m) rd(c) wt(c_M)
111


Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s = 110 c = 100 global = 0
R W T_F
rd(s) wt(s_F) rd(c) rd(s_F)
future
continuation
get
R W T_M
rd(s) wt(s_m) rd(c) wt(c_M)
112


Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s = 110 c = 100 global = 0
R W T_F
rd(s) wt(s_F) rd(c) rd(s_F)
future
continuation
!!! get
R W T_M
rd(s) wt(s_m) rd(c) wt(c_M)
113


Safe Execution
Account s; // savings
Account c; // checking
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
float monthlyTotal () {
s.addInterest(0.10);
return c.balance()+s.balance(); }
210
void transfer (float amount) {
s.withdraw(amount);
c.deposit(amount); }
s_M= 50 c_M= 150
s = 110 c = 100 global = 0
continuation
get
R W T_M
rd(s) wt(s_m) rd(c) wt(c_M)
114

Safe Execution

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }
```
210

```
void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s = 110 c = 100 global = 0

continuation — rd(s) wt(s_m) — rd(c) wt(c_M) — get

R W T_M

115

Safe Execution

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }
```
210

```
void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s = 110 c = 100 global = 0

continuation — rd(s) wt(s_m) — rd(c) wt(c_M) — get

R W T_M

116

Safe Execution

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }
```
210

```
void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s = 110 c = 100 global = 0

continuation →

R W T_M

117

Safe Execution

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }
```
210

```
void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s_M= 60 c_M= 150
s = 110 c = 100 global = 0

continuation — rd(s) wt(s_M) — rd(c) wt(c_M) — get

R W T_M

118

Safe Execution

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }
```
210

```
void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s_M= 60 c_M= 150
s = 110 c = 100 global = 0

continuation — rd(s) wt(s_m) — rd(c) wt(c_M) — ? get

R W T_M

119

Safe Execution

Account s; // savings
Account c; // checking

```
Future f = F[monthlyTotal()];
transfer(50);
global = f.get();
```

```
float monthlyTotal () {
  s.addInterest(0.10);
  return c.balance()+s.balance(); }
```
210

```
void transfer (float amount) {
  s.withdraw(amount);
  c.deposit(amount); }
```

s_M= 60 c_M= 150
s = 110 c = 100 global = 0

continuation — rd(s) wt(s_m) — rd(c) wt(c_M) — !!! get

R W T_M

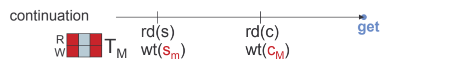
120

Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount) {
   s.withdraw(amount);
   c.deposit(amount); }
```

210

s_M = 60 c_M = 150 global = 0

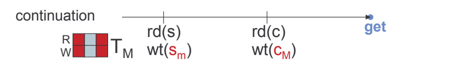


Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount) {
   s.withdraw(amount);
   c.deposit(amount); }
```

210

s = 60 c = 150 global = 0

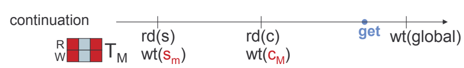


Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount) {
   s.withdraw(amount);
   c.deposit(amount); }
```

210

s = 60 c = 150 global = 210



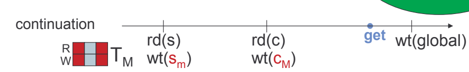
Safe Execution

```
Account s;  // savings
Account c;  // checking

Future f = F[monthlyTotal()];
transfer(50);
global = f.get();

float monthlyTotal () {
   s.addInterest(0.10);
   return c.balance()+s.balance(); }

void transfer (float amount) {
   s.withdraw(amount);
   c.deposit(amount); }
```

210

s = 60 c = 150 global = 210

SERIAL
s=60 c=150
global=210



Prototype Implementation

- Based on IBM's Jikes RVM
- Compiler-injected read and write barriers to intercept shared data accesses
- Bytecode rewriting plus run-time support for automatic roll-back
- Modification of object headers
  - Version access via forwarding pointers

Barrier Optimizations

- **Goal:** omit barriers on loads of primitive values
- **Problem:** accesses through stale on-stack references
- **Solution:** update references on stack using modified GC stack scanning procedure
  - At version creation
  - At pre-specified "synchronization" points

Automatic Rollback

- Discard versions
- Futures:
  - evaluated within separate thread so just re-run
- Continuations:
  - Rewrite bytecodes to save state at start
  - On rollback throw **revoke** exception
  - Modify run-time to unwind **revoke** exceptions without running user handlers
  - Handler restores state and restarts continuation

127

Challenges

- Continuations escaping method scope
  - Perform **get** early
- Serial order for multiple futures
  - Different threads for separate futures
  - The same thread for all continuations
  - Nested futures
- Interaction with existing mechanisms
  - Java threads, native methods may foil safety

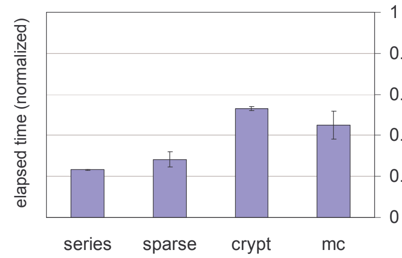
128

Benchmarks

- Selected Java Grande benchmarks
- Modified Multi-User OO7 benchmark
  - Standard OO7 design database
    - Multi-level hierarchy of composite parts
    - Shared and private modules
  - Mixed-mode read/write traversals
- Configuration
  - 700MHz Pentium 3 (4 CPUs)
  - Average of 5 "hot" runs (no compilation)

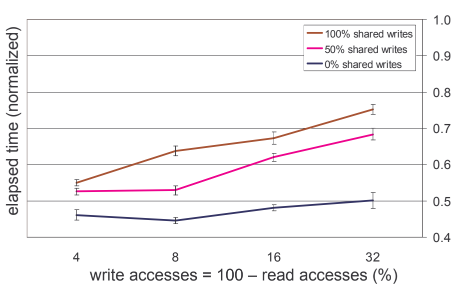
129

Java Grande – 4 Futures



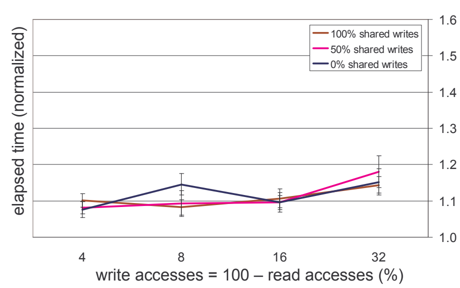
130

OO7 – 4 Futures
All reads to shared module



131

OO7 – 1 Future
All reads to shared module



132

Conclusions

- Futures are a lighter-weight alternative to programming for parallelism
- Multilisp pioneered the idea
- Applying to Java requires more work
  - Proxy inference
  - Safety checking

133

Future Work

- Better run-time support
  - Lazy task creation a la Multilisp
- Safety checking for non-serial futures
  - HTTP example rejected by safety checking scheme
- Incremental analysis for better software development

134

Further Reading

- Static Analysis
  - Points-to analysis (many)
  - Qualifier inference (Foster et al.)
  - Value flow analysis (Heintze and Tardieu)

135

Further Reading

- Parallelization (Rinard *et. al.*)
- Transactional memory (Herlihy *et. al.*, Shavit-Touitou)
- Atomicity (Flanagan *et. al.*, Harris *et. al.*)
- Traditional lock optimizations (Bacon *et. al.*)
- Lock-free data structures (Rajwar-Goodman, Jensen *et. al.*)

136

It's break time!

137