Programming Languages

Concurrency: Memory Consistency

Dr. Michael Petter
Winter term 2018

Strict Consistency

Motivated by sequential computing, we intuitively implicitly transfer our idea of semantics of memory accesses to concurrent computation. This leads to our idealistic model strict consistency.

Definition (Strict consistency)

Independently of which process reads or writes, the value from the most recent write to a location is observable by reads from the respective location immediately after the write occurs.

Although idealistically desired, practically not existing

- absolute global time problematic
- physically not possible

⇝ strict consistency is too strong to be realistic

Abandoning absolute time

Thread A

```c
void foo(void) {
    a = 1;
    b = 1;
}
```

Thread B

```c
void bar(void) {
    while (b == 0){};
    assert (a==1);
}
```

Intuition: the assertion will never fail

Real execution: given enough tries, the assertion may eventually fail

⇝ in need of defining a Memory Model

Memory Models

Memory interactions behave differently in presence of

- multiple concurrent threads
- data replication in hierarchical and/or distributed memory systems
- deferred communication of updates

Memory Models are a product of negotiating

- restrictions of freedom of implementation to guarantee race related properties
- establishment of freedom of implementation to enable program and machine model optimizations

⇝ Modern Languages include the memory model in their language definition

Happend-Before Relation and Diagram

Still, any of the following may happen:

- foo: a and b happen before a sees b
- bar: a sees b before it writes b
- the assert statement should always hold
  - here correctness means: writing a : to a happens before reading a : in b

⇝ idea: state correctness in terms of what event may happen before another one
Events in a Distributed System

A process as a series of events [Lam78]: Given a distributed system of processes $P, Q, \ldots$, each process $P$ consists of events $p_1, p_2, \ldots$. An event $p_i$ in process $P$ happened before an event $q_j$ in $Q$ that receives this message and $p_i$ happened before $q_j$.

The Happened-Before Relation

**Definition**

If an event $p$ happened before an event $q$ then $p \rightarrow q$.

Observe:
- $\rightarrow$ is partial (neither $p \rightarrow q$ or $q \rightarrow p$ may hold)
- $\rightarrow$ is irreflexive ($p \rightarrow p$ never holds)
- $\rightarrow$ is transitive ($p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$)
- $\rightarrow$ is asymmetric (if $p \rightarrow q$ then $\neg(q \rightarrow p)$)

$\rightarrow$ the $\rightarrow$ relation is a strict partial order.

Summing up Happened-Before Relations

We can model concurrency using processes and events: there is a happened-before relation between the events of each process, there is a happened-before relation between communicating events, happened-before is a strict partial order, a clock is a total strict order that embeds the happened-before partial order.

Ordering

Let $C$ be a logical clock i.e. $C$ assigns a globally unique time-stamp $C(p)$ to each event $p$.

**Definition (Clock Condition)**

Function $C$ satisfies the clock condition if for any events $p, q$:

$p \rightarrow q \implies C(p) < C(q)$

For a distributed system the clock condition holds if:
- $p$, $q$, $r$, and $p_i$ are events of $P$ and $p_i \rightarrow p$ then $C(p_i) < C(p)$
- $p$ is the sending of a message by process $P$ and $q$ is the reception of this message by process $Q$ then $C(p) < C(q)$

$\rightarrow$ a logical clock $C$ that satisfies the clock condition describes a total order $a < b$ (with $C(a) < C(b)$) that embeds the strict partial order $\rightarrow$

The set defined by all $C$ that satisfy the clock condition is exactly the set of executions possible in the system.

- Use the process model and $\rightarrow$ to define better consistency model

Defining $C$ Satisfying the Clock Condition

Given:

$C(e) = C(p_1) < C(p_2) < \ldots$

Summing up Happened-Before Relations

We can model concurrency using processes and events:
- there is a happened-before relation between the events of each process
- there is a happened-before relation between communicating events
- happened-before is a strict partial order
- a clock is a total strict order that embeds the happened-before partial order
Memory Consistency Models based on the Happened-Before Relation

Happened-Before Based Memory Models

Idea: use happened-before diagrams to model more relaxed memory models.

Given a path through each of the threads of a program:
- consider the actions of each thread as events of a process
- use more processes to model memory
  - for here: one process per variable in memory
  - \( \cdots \) concisely represent some interleavings

We establish a model for Sequential Consistency.

Sequential Consistency

Definition (Sequential Consistency Condition [Lam78]):

The result of any execution is the same as if the memory operations
- of each individual processor appear in the order specified by its program
- of all processors joined were executed in some sequential order

Sequential Consistency applied to Multiprocessor Programs:

Given a program with \( n \) threads,
- for fixed event sequences \( p_0^0, p_1^1, \ldots \) and \( p_0^n, p_1^n, \ldots \) keeping the program order,
- executions obeying the clock condition on the \( p_j \),
- all executions have the same result

Yet, in other words:
- \( \Rightarrow \) defines the execution path of each thread
- each execution mentioned in \( \cdots \) is one interleaving of processes
- \( \Rightarrow \) declares that the result of running the threads with these interleavings is always the same.

Weakening the Model

Observation: more concurrency possible, if we model each memory location separately, i.e. as a different process

Sequential consistency still obeyed:
- the accesses of \( \text{foo} \) to \( a \) occurs before \( b \)
- the first two read accesses to \( b \) are in parallel to \( a=1 \)

Conclusion: There is no observable change if accesses to different memory locations can happen in parallel.

Benefits of Sequential Consistency

- concisely represent all interleavings that are due to variations in timing
- synchronization using time is uncommon for software
- \( \Rightarrow \) a good model for correct behaviors of concurrent programs
- \( \Rightarrow \) program results besides SC results are undesirable (they contain races)

Realistic model for simple hardware architectures:
- sequential consistency model suitable for concurrent processors that acquire exclusive access to memory
- processors can speed up computation by using caches and still made to maintain sequential consistency

Not realistic for elaborate hardware with out-of-order stores:
- \( \Rightarrow \) what other processors see is determined by complex optimizations to cacheline management
- \( \Rightarrow \) internal workings of caches
Introducing Caches: The MESI Protocol

The MESI Protocol: States [PP84]
Processors use caches to avoid a costly round-trip to RAM for every memory access.
- Programs often access the same memory area repeatedly (e.g., stack)
- Keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

The global state of cache lines is kept consistent by sending messages

MESI Example
Consider how the following code might execute:
Thread A
a = 1; // A.1
b = 1; // A.2
Thread B
while (b == 0) {}; // B.1
assert(a == 1); // ... CPU B

The MESI Protocol: Messages
Moving data between caches is coordinated by sending messages [McK10]:
- Read: sent if CPU needs to read from an address
- Read Response: when in state E or S, response to a Read message, carries the data for the requested address
- Invalidate: asks others to evict a cache line
- Invalidate Acknowledge: reply indicating that a cache line has been evicted
- Read Invalidate: like Read + Invalidate (also called "read with intend to modify")
- Writeback: Read Response when in state M, as a side effect noticing main memory about modifications to the cacheline, changing sender's state to S

We mostly consider messages between processors. Upon Read Invalidate, a processor replies with Read Response/Writeback before the Invalidate Acknowledge is sent.
**MESI Example: Happened Before Model**

Idea: each cache line one process, A caches b=0 as E, B caches a=0 as E

Observations:
- each memory access must complete before executing next instruction
- second execution of test b==0 stays within cache \(\Rightarrow\) no traffic

**Summary: MESI cache coherence protocol**

Sequential consistency:
- a characterization of well-behaved programs
- a model for differing speed of execution
- for fixed paths through the threads and a total order between accesses to the same variable: executions can be illustrated by happened-before diagram with one process per variable
- MESI cache coherence protocol ensures SC for processors with caches

**Introducing Store Buffers: Out-Of-Order Stores**

Abstract Machine Model: defines semantics of memory accesses

Out-of-Order Execution:
- performance problem: writes always stall

Thread A
- \(a = 1\); // A.1
- while (b == 0) {} // B.1
- assert (a == 1); // B.2

CPU A should continue executing after \(a=1\)

Store Buffers
- Abstract Machine Model: defines semantics of memory accesses
- put each store into a store buffer and continue execution
- Store buffers apply stores in various orders: FIFO (Sparc/x86-TSO), unordered (Spec PSO)
- Some program order still needs to be observed locally
  - store buffer snoops read channel and
  - on matching address, returns the youngest value in buffer

**TSO Model: Formal Spec [SI92]**

Definition (Total Store Order)

- The store order wrt. memory \(\preceq\) is total
- \(\forall a,b \in \text{addr} \quad a \preceq b \lor b \preceq a\)
- Stores in program order \(\preceq\) are embedded into the memory order \(\preceq\)
- \(a \preceq [a] \Rightarrow [b] \preceq [b]\)
- Loads preceding an operation w.r.t. program order \(\preceq\) are embedded into the memory order \(\preceq\)
- A load's value is determined by the latest write as observed by the local CPU

\[\forall a \in \text{addr} \Rightarrow \forall l \in \text{load} \quad l(a) = \max \{l_i(a) \mid l_i \in \text{load} \land l_i \preceq [a]\}\]

Particularly, one ordering property is not guaranteed:
\[a \preceq [a] \Rightarrow b \preceq [b]\]

Local stores may be observed earlier by local loads then from somewhere else!
Happened-Before Model for TSO

Thread A

\[ a = 1; \]
\[ printf("%d",b); \]

Thread B

\[ b = 1; \]
\[ printf("%d",a); \]

Assume cache A contains: a: S0, b: S0, cache B contains: a: S0, b: S0

TSO in the Wild: x86

The x86 CPUs, powering desktops and servers around the world is a common representative of a TSO Memory Model based CPU.

- FIFO store buffers keep quite strong consistency properties
- The major obstacle to Sequential Consistency is modern x86 CPUs provide the mfence instruction
- mfence orders all memory instructions:
  \[ Op_i \preceq mfence() \preceq Op'_i \]
- a fence between write and loads gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)
- use fences only when necessary

Explicit Synchronization: Write Barrier

Overtaking of messages may be desirable and does not need to be prohibited in general:

- generalized store buffers render programs incorrect that assume sequential consistency between different CPUs
- whenever a store in front of another operation in one CPU must be observable in this order by a different CPU, an explicit write barrier has to be inserted
  - a write barrier marks all current store operations in the store buffer
  - the next store operation is only executed when all marked stores in the buffer have completed

Happened-Before Model for PSO

Thread A

\[ a = 1; \]
\[ b = 1; \]
\[ printf("%d",a); \]

Thread B

while (b == 0) {};
assert(a == 1);

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I

PSO Model: Formal Spec [SI92]

Definition (Partial Store Order)

1 The store order wrt. memory (\( \preceq \)) is total
\[ \forall a,b \in \text{memory} : a \preceq b \lor b \preceq a \]

2 Fenced stores in program order (\( \preceq \)) are embedded into the memory order (\( \preceq \))
\[ a \preceq b \] and does not need to be prohibited in general

3 Stores to the same address in program order (\( \preceq \)) are embedded into the memory order (\( \preceq \))
\[ a \preceq b \]

4 Loads preceding an other operation (wrt. program order (\( \preceq \)) are embedded into the memory order (\( \preceq \))
\[ a \preceq b \]

5 A load’s value is determined by the latest write as observed by the local CPU
\[ \text{val}(Ld_i) = \text{val}(St_i) \] on \( i \) \[ \text{addr}_i \]

Now also stores are not guaranteed to be in order any more:
\[ a \preceq b \]

What about sequential consistency for the whole system?
Further weakening the model: O-o-O Reads

Relaxed Memory Order
Communication of cache updates is still costly:
- a cache-intense computation can fill up store buffers in CPUs
- waiting for invalidation acknowledgements may still happen
- invalidation acknowledgements are delayed on busy caches

- immediately acknowledge an invalidation and apply it later
- put each invalidate message into an invalidate queue
- if a MESI message needs to be sent regarding a cache line in the invalidate queue then wait until the line is invalidated

local loads and stores do not consult the invalidate queue
- What about sequential consistency?

Happened-Before Model for Invalidate Queues
Thread A
a = 1;
sfence();
b = 1;
Thread B
while (b == 0) {};
assert(a == 1);

Assume cache A contains: a: S0, b: E0, cache B contains: a: S0, b: I

Explicit Synchronization: Read Barriers
Read accesses do not consult the invalidate queue.
- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit read barrier before the read access
  - a read barrier marks all entries in the invalidate queue
  - the next read operation is only executed once all marked invalidations have completed
- a read barrier before each read gives sequentially consistent read behavior (and is as slow as a system without invalidate queue)

match each write barrier in one process with a read barrier in another process

Happened-Before Model for Read Barriers
Thread A
a = 1;
sfence();
b = 1;
Thread B
while (b == 0) {};
sfence();
assert(a == 1);
Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```c
//flag[] is boolean array; and turn is an integer
flag[0] = false;
flag[1] = false;
turn = 0; // or 1

P0:
flag[0] = true;
while (flag[1] == true) {
    if (turn != 0) {
        flag[0] = false;
        while (turn != 0) {
            // busy wait
        }
    }
    flag[0] = true;
}
// critical section
turn = 1;
flag[0] = false;
flag[1] = true;
```

```c
P1:
flag[1] = true;
while (flag[1] == true) {
    if (turn != 1) {
        flag[1] = false;
        while (turn != 1) {
            // busy wait
        }
    }
    flag[1] = true;
}
// critical section
turn = 0;
flag[0] = false;
flag[1] = false;
```

The Idea Behind Dekker

Communication via three variables:
- flag[1]==true: process P1 wants to enter its critical section
- turn==i: process P1 has priority when both want to enter

- In process P0:
  - if P1 does not want to enter, proceed immediately to the critical section
  - if P1 also wants to enter, wait for turn to be set to 1
  - while waiting for turn, reset flag[0] to enable P1 to progress

Dekker's Algorithm and RMO

Problem: Dekker’s algorithm requires sequential consistency.

Idea: insert memory barriers between all variables common to both threads.

```c
flag[0] = true;
while (flag[1] == true) {
    if (turn != 0) {
        flag[0] = false;
        interrupt();
    }
    flag[0] = true;
}
// critical section
turn = 1;
flag[0] = false;
```

Discussion

Memory barriers reside at the lowest level of synchronization primitives.

Where are they useful?
- when blocking should not de-schedule threads
- when several processes implement automata and coordinate their transitions via common synchronized variables
- protocol implementations
- OS provides synchronization facilities based on memory barriers

Why might they not be appropriate?
- difficult to get right, best suited for specific well-understood algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

Summary: Relaxed Memory Models

Highly optimized CPUs may use a relaxed memory model:
- reads and writes are not synchronized unless requested by the user
- many kinds of memory barriers exist with subtle differences
  - ARM, PowerPC, Alpha, ia-64, even x86 (→ SSE Write Combining)

⇝ memory barriers are the “lowest-level” of synchronization
An Introduction to the Intel QuickPath Interconnect

Figure 6. Intel Interconnect Overview

QuickPath Interconnect comprises loop-invariant code motion and dead store elimination, e.g.

- Compiler also depends on consistency guarantees
- Demand for Memory Models on language level

Future Many-Core Systems: NUMA

In that case: Intel's MESIF (Forward) to reduce communication overhead.

- But in general, Symmetric multi-processing (SMP) has its limits:
  - a memory-intensive computation may cause contention on the bus
  - the speed of the bus is limited since the electrical signal has to travel to all participants
  - point-to-point connections are faster than a bus, but do not provide possibility of forming consensus
  - use a bus locally, use point-to-point links globally: NUMA
  - non-uniform memory access partitions the memory amongst CPUs
  - a directory states which CPU holds a memory region
  - Interprocess communication between Cache-Controlers (ccNUMA): onchip on Opteron or in chipset on Itanium

Overhead of NUMA Systems

Communication overhead in a NUMA system.

- Processors in a NUMA system may be fully or partially connected.
- The directory of who stores an address is partitioned amongst processors.
- A cache miss that cannot be satisfied by the local memory at \( A \):
  - \( A \) sends a request to processor \( B \) owning the directory
  - \( B \) tells the processor \( C \) who holds the content
  - \( C \) sends data (or status) to \( A \) and sends acknowledge to \( B \)
  - \( B \) completes transmission by an acknowledge to \( A \)

Memory Consistency Wrapping Up

Keeping semantics I

```c
int x = 0;
for (int i=0;i<100;i++){
    printf("%d",x);
}
```

Keeping semantics II

```c
volatile int x = 0;
for (int i=0;i<100;i++){
    x = 1;
    printf("%d",x);
}
```

- Compilers may also reorder store instructions
- Write barriers keep the compiler from reordering across
- The specification of `volatile` keeps the C-Compiler from reordering memory accesses to this address
- Java Compilers even generate barriers around accesses to `volatile` variables

Learning Outcomes

- Strict Consistency
- Happened-before Relation
- Sequential Consistency
- The MESI Cache Model
- TSO: FIFO store buffers
- PSO: store buffers
- RMO: invalidate queues
- Reestablishing Sequential Consistency with memory barriers
- Dekker’s Algorithm for Mutual Exclusion

References

**Why Memory Barriers are not Enough**

Often, multiple memory locations may only be modified exclusively by one thread during a computation.

- Use barriers to implement automata that ensure mutual exclusion
- Generalize the re-occurring concept of enforcing mutual exclusion

Needed: interaction with multiple memory locations within a single step:

\[
\begin{array}{c}
A \\
a=1, b=1 \\
b
\end{array}
\]

**Definition (Atomic Execution)**

A computation forms an atomic execution if its effect can only be observed as a single transformation on the memory.
Overview

We will address the established ways of managing synchronization. The presented techniques
are available on most platforms
likely to be found in most existing (concurrent) software
provide solutions to common concurrency tasks
are the source of common concurrency problems

The techniques are applicable to C, C++ (pthread), Java, C# and other imperative languages.

Wait-Free Atomic Executions

Atomic Executions, Locks and Monitors

Wait-Free Algorithms based on Atomic Operations

Wait-Free Updates

Which operations on a CPU are atomic? (j, k and tmp are registers)

Program 1
i++;

Program 2
j = i;

Program 3
i = i+k;

Answer:
none by default (even without store and invalidate buffers, why?)

The load and store (even i++) may be interleaved with a store from another processor.
Wait-Free Updates

Which operations on a CPU are atomic? (\(j, x\) and \(tmp\) are registers)

Program 1
\[ i++; \]
Program 2
\[ j = i; \]
Program 3
\[ int\; tmp = i; \]
\[ i = j; \]
\[ j = \text{tmp}; \]

Answer:
- none by default (even without store and invalidate buffers, why?)

The load and store (even \(i++\)) may be interleaved with a store from another processor.

All of the programs can be made atomic executions (e.g. on x86):
- \(i\) must be in memory
- Idea: lock the cache bus for an address for the duration of an instruction

Marking Statements as Atomic

Rather than writing assembler: use made-up keyword \texttt{atomic}:

Program 1
\[
\begin{array}{l}
\text{atomic}\
\{\
i++;
\}
\end{array}
\]

Program 2
\[
\begin{array}{l}
\text{atomic}\
\{\
j = i;\
\}
\end{array}
\]

Program 3
\[
\begin{array}{l}
\text{atomic}\
\{\
int\; tmp = i;\
i = j;\
j = \text{tmp};\
\}
\end{array}
\]

The statements in an \texttt{atomic} block execute as \texttt{atomic execution}:

Bumper Pointer Allocation

Garbage collectors often use a bumper pointer to allocated memory:

```c
char heap[2^20];
char* firstFree = &heap[0];
char* alloc(int size) {
    char* start = firstFree;
    firstFree = firstFree + size;
    if (start+size>sizeof(heap)) garbage_collect();
    return start;
}
```

- firstFree points to the first unused byte
- each allocation reserves the next size bytes in heap

Thread-safe implementation:
- alloc's core functionality matches Program 2
  - inline assembler (GCC/AT&T syntax in the example)
Marking Statements as Atomic

Rather than writing assembler: use made-up keyword atomic:

```
Program 1
atomic {
    i++;  
}
```

The statements in an atomic block execute as atomic execution:

- atomic only translatable when a corresponding atomic CPU instruction exist
- the notion of requesting atomic execution is a general concept

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:
- no control flow possible, no behavioral change depending on data
- often, there are instructions that execute an operation conditionally

```
Program 4
atomic {
    r = b;  
    b = 0;  
}
```

Operations update a memory cell and return the previous value.
- the first two operations can be seen as setting a flag \( b \) to \( v \in \{0, 1\} \) and returning its previous state.
  - this operation is called set-and-test
- the third case generalizes this to setting a variable \( i \) to the value of \( j \), if \( i \)'s old value is equal to \( i \)'
  - this operation is called compare-and-swap
- compute a new value

```
Program 5
atomic {
    r = (k==i);  
    if (r) i = j;  
}
```

Operations update a memory cell and return the previous value.
- the first two operations can be seen as setting a flag \( b \) to \( v \in \{0, 1\} \) and returning its previous state.
  - this operation is called set-and-test
- the third case generalizes this to setting a variable \( i \) to the value of \( j \), if \( i \)'s old value is equal to \( i \)'
  - this operation is called compare-and-swap

- use as building blocks for algorithms that can fail

Lock-Free Algorithms

If a wait-free implementation is not possible, a lock-free implementation might still be viable.

```
Program 6
atomic {
    r = (k==i);  
    if (r) i = j;  
}
```

Operations update a memory cell and return the previous value.
- the first two operations can be seen as setting a flag \( b \) to \( v \in \{0, 1\} \) and returning its previous state.
  - this operation is called set-and-test
- the third case generalizes this to setting a variable \( i \) to the value of \( j \), if \( i \)'s old value is equal to \( i \)'
  - this operation is called compare-and-swap

- use as building blocks for algorithms that can fail

```
## Lock-Free Algorithms

If a wait-free implementation is not possible, a lock-free implementation might still be viable. Common usage pattern for compare and swap:

1. read the initial value in `i` into `k` (using memory barriers)
2. compute a new value `j = f(k)`
3. update `i` to `j` if `i < k` still holds
4. go to first step if `j ≠ k` meanwhile

⚠️ note: `i = k` must imply that no thread has updated `i`

### General recipe for lock-free algorithms
- given a compare-and-swap operation for `n` bytes
- try to group variables for which an invariant must hold into `n` bytes
- read these bytes atomically
- compute a new value
- perform a compare-and-swap operation on these `n` bytes

---

## Limitations of Wait- and Lock-Free Algorithms

Wait-/Lock-Free algorithms are severely limited in terms of their computation:
- restricted to the semantics of a single atomic operation
- set of atomic operations is architecture specific, but often includes
  - exchange of a memory cell with a register
  - compare-and-swap of a register with a memory cell
  - fetch-and-add on integers in memory
  - modify-and-test on bits in memory
- provided instructions usually allow only one memory operand

---

## Locked Atomic Executions

```java
// Locked Atomic Executions
```

## Linearizability

```
// Linearizability
```
**Locks**

**Definition (Lock)**

A lock is a data structure that
- can be acquired and released
- ensures mutual exclusion: only one thread may hold the lock at a time
- blocks other threads attempting to acquire while held by a different thread
- protects a critical section: a piece of code that may produce incorrect results when entered concurrently from several threads

⚠️ may deadlock the program

---

**Semaphores and Mutexes**

A (counting) semaphore is an integer \( s \) with the following operations:

```c
void signal(int *s) {
    atomic {
        *s = *s + 1;
        if (*s > 0) {
            *s = 0;
        }
    } while (!avail);
}
```

A counting semaphore can track how many resources are still available.
- a thread acquiring a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()` to release

---

**Semaphores and Mutexes**

A (counting) semaphore is an integer \( s \) with the following operations:

```c
void signal(int *s) {
    atomic {
        *s = *s + 1;
        if (*s > 0) {
            *s = 0;
        }
    } while (!avail);
}
```

A counting semaphore can track how many resources are still available.
- a thread acquiring a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()` to release

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`

---

**Implementation of Semaphores**

A semaphore does not have to wait busily:

```c
void wait(int *s) {
    bool avail;
    do {
        atomic {
            *s = *s + 1;
            if (*s > 0) {
                *s = 0;
            }
        } while (!avail);
    }
}
```

Busy waiting is avoided:
- a thread failing to decrease \( *s \) executes `de_schedule()`
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when \( *s \) becomes non-zero, usually by `monitoring writes` to \( *s \) (syscall to futex `WAIT`)
- once a thread calls `wake()`, the first thread waiting on \( s \) is extracted
- the operating system lets \( t \) return from its call to `de_schedule()`
Practical Implementation of Semaphores

Certain optimisations are possible:

```c
void signal(int *s) {
atomic { *s = *s + 1; ...
/* critical section */
(*s)++; } ...
```

Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by

- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Mutexes

One common use of semaphores is to guarantee mutual exclusion.

- in this case, a binary semaphore is also called a mutex
- e.g. add a lock to the double-ended queue data structure

Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by

- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:

1. a mechanism to address these problems:
   - if a thread is busy waiting and produces contention on the lock
   - if that lock is already taken, proceed if it is taken by the current thread
   - a procedure associated with a monitor acquires a lock on entry and releases it on exit
   - if that lock is already taken, proceed if it is taken by the current thread

Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by

- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:

- a mechanism to address these problems:
  - if a thread is busy waiting and produces contention on the lock
  - if that lock is already taken, proceed if it is taken by the current thread
Monitors: An Automatic, Re-entrant Mutex

Often, a data structure can be made thread-safe by
- acquiring a lock upon entering a function of the data structure
- releasing the lock upon exit from this function

Locking each procedure body that accesses a data structure:
- a re-occurring pattern, should be generalized
- becomes problematic in recursive calls: it blocks
- if a thread t waits for a data structure to be filled:
  - t will call e.g. pop() and obtain -t
  - t then has to call again, until an element is available
- Monitor: a mechanism to address these problems:
  - a procedure associated with a monitor acquires a lock on entry and releases it on exit
- if that lock is already taken, proceed if it is taken by the current thread
  - need a way to release the lock after the return of the last recursive call

Implementation of a Basic Monitor

A monitor contains a semaphore count and the id tid of the occupying thread:

```c
typedef struct monitor mon_t;
struct monitor {
  int tid; int count;
};
void monitor_init(mon_t *m) {
  m->count = 0;
}
```

Define `monitor_enter` and `monitor_leave`:
- ensure mutual exclusion of accesses to mon_t
- track how many times we called a monitored procedure recursively

```c
void monitor_enter(mon_t *m) {
  atomic {
    m->tid = thread_id();
  }
  atomic {
    m->tid = 0;
  }
  m->count++;
  m->tid = -m->tid;
  if (m->tid == 0) {
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = 0;
    }
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = thread_id();
    }
    atomic {
      m->tid = thread_id();
    }
  }
}
```

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.
Still: Efficiency problem when using resource to synchronize:
- if a thread t waits for a data structure to be filled:
  - t will call e.g. pop() and obtain -t
  - t then has to call again, until an element is available
  - t is busy waiting and produces contention on the lock
- if a thread t calls e.g. pop() and obtains -t
  - t then has to call again, until an element is available
  - t is busy waiting and produces contention on the lock
- if a thread t calls e.g. pop() and obtains -t
  - t then has to call again, until an element is available
  - t is busy waiting and produces contention on the lock

Condition Variable Semantics

Requires one queue for each condition cand a suspended queue s:
- the signal-and-urgent-wait
- a call to wait on condition c
- a call to signal for c adds thread to the queue c
- a call to signal for c removes thread from the queue c
- one thread form the queue c is awaken
- signal on a queue c is a no-op if the queue c is empty
- if a thread leaves, it wakes up one thread waiting on c
- if a thread leaves, it wakes up one thread waiting on c
- if a thread leaves, it wakes up one thread waiting on c
- if a thread leaves, it wakes up one thread waiting on c

Signal-And-Urgent-Wait Semantics

Requires one queue for each condition c and a suspended queue s:
- a thread that tries to enter a monitor is added to queue c if the monitor is occupied
- a call to wait on condition c
- a call to signal for c adds thread to the queue c
- one thread form the queue c is awaken
- signal on a queue c is a no-op if the queue c is empty
- if a thread leaves, it wakes up one thread waiting on c
- if a thread leaves, it wakes up one thread waiting on c
- if a thread leaves, it wakes up one thread waiting on c
- if a thread leaves, it wakes up one thread waiting on c
Signal-And-Urgent-Wait Semantics
Requires one queue for each condition \( c \) and a suspended queue \( s \):

- a thread who tries to enter a monitor is added to queue \( c \) if the monitor is occupied
- a call to \( \text{wait} \) on condition \( c \) adds thread to the queue \( c \)
- a call to \( \text{signal} \) for \( a \) adds thread to queue \( s \) (suspended)
- one thread form the \( c \) queue is woken up
- \( \text{signal} \) on \( a \) is a no-op if \( a,q \) is empty
- if a thread leaves, it wakes up one thread waiting on \( a \)
- if \( s \) is empty, it wakes up one thread from \( c \)

\( \Rightarrow \) queue \( s \) has priority over \( c \)

Atomic Executions, Locks and Monitors Locked

Signal-And-Continue Semantics
Here, the \( \text{signal} \) function is usually called \( \text{notify} \).

- a call to \( \text{wait} \) on condition \( c \) adds thread to the queue \( a,q \)
- a call to \( \text{notify} \) for \( a \) adds one thread from \( a,q \) to \( c \) (unless \( a,q \) is empty)
- if a thread leaves, it wakes up one thread waiting on \( c \)
- \( \Rightarrow \) queue \( s \) has priority over \( c \)

Atomic Executions, Locks and Monitors Locked

A Note on \( \text{Notify} \)
With signal-and-continue semantics, two \( \text{notify} \) functions exist:

1. \( \text{notify} \): wakes up exactly one thread waiting on condition variable
2. \( \text{notifyAll} \): wakes up all threads waiting on a condition variable

With \( \text{signal-and-continue} \) semantics, \( \text{notify} \) is a no-op if \( \text{signal} \) on \( a \) is empty

Atomic Executions, Locks and Monitors Locked

A Note on \( \text{Notify} \)
With \( \text{signal-and-continue} \) semantics, two \( \text{notify} \) functions exist:

1. \( \text{notify} \): wakes up exactly one thread waiting on condition variable
2. \( \text{notifyAll} \): wakes up all threads waiting on a condition variable

\( \Rightarrow \) an implementation often becomes easier if \( \text{notify} \) means \( \text{notify some} \)

\( \Rightarrow \) programmer should assume that thread is not the only one woken up

Atomic Executions, Locks and Monitors Locked
Monitors with a Single Condition Variable

Monitors with a single condition variable are built into Java and C#:

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter(this);
        // body of f
        monitor_exit(this);
    }
}
```

with `Object` containing:
- private int mon_var;
- private int mon_count;
- private int cond_var;
- protected void monitor_enter();
- protected void monitor_exit();

Deadlocks

**Definition (Deadlock)**

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

**Sequence leading to a deadlock:**

- `A` blocks on the monitor of `B`
- `B` blocks on the monitor of `A`
- `A` happens to execute `A.bar()` and `B` executes `B.bar()`
- `B` happens to execute `B.bar()` and `A` executes `A.bar()`

**Consider this Java class:**

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        b.bar() acquires the monitor of b
        a happens to execute a.bar()
        b happens to execute b.bar()
        a blocks on the monitor of b
        b blocks on the monitor of a
        // in parallel
        a.bar() || b.bar();
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter(this);
        // body of f
        monitor_exit(this);
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
    public synchronized void g() {
        // body of g
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
}
```

is the only safe approach on standard operating systems but what about algorithms that require locking?

**Deadlocks with Monitors**

**Definition (Deadlock)**

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

**Consider this Java class:**

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        b.bar() acquires the monitor of b
        a happens to execute a.bar()
        b happens to execute b.bar()
        a blocks on the monitor of b
        b blocks on the monitor of a
        // in parallel
        a.bar() || b.bar();
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter(this);
        // body of f
        monitor_exit(this);
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
    public synchronized void g() {
        // body of g
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
}
```

is the only safe approach on standard operating systems but what about algorithms that require locking?

**Deadlocks with Monitors**

**Definition (Deadlock)**

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

**Consider this Java class:**

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        b.bar() acquires the monitor of b
        a happens to execute a.bar()
        b happens to execute b.bar()
        a blocks on the monitor of b
        b blocks on the monitor of a
        // in parallel
        a.bar() || b.bar();
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter(this);
        // body of f
        monitor_exit(this);
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
    public synchronized void g() {
        // body of g
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
}
```

is the only safe approach on standard operating systems but what about algorithms that require locking?

**Deadlocks with Monitors**

**Definition (Deadlock)**

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

**Consider this Java class:**

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        b.bar() acquires the monitor of b
        a happens to execute a.bar()
        b happens to execute b.bar()
        a blocks on the monitor of b
        b blocks on the monitor of a
        // in parallel
        a.bar() || b.bar();
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter(this);
        // body of f
        monitor_exit(this);
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
    public synchronized void g() {
        // body of g
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
}
```

is the only safe approach on standard operating systems but what about algorithms that require locking?

**Deadlocks with Monitors**

**Definition (Deadlock)**

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

**Consider this Java class:**

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        b.bar() acquires the monitor of b
        a happens to execute a.bar()
        b happens to execute b.bar()
        a blocks on the monitor of b
        b blocks on the monitor of a
        // in parallel
        a.bar() || b.bar();
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter(this);
        // body of f
        monitor_exit(this);
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
    public synchronized void g() {
        // body of g
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
}
```

is the only safe approach on standard operating systems but what about algorithms that require locking?

**Deadlocks with Monitors**

**Definition (Deadlock)**

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

**Consider this Java class:**

```java
class Foo {
    public Foo other = null;
    public synchronized void bar() {
        a.bar() acquires the monitor of a
        b.bar() acquires the monitor of b
        a happens to execute a.bar()
        b happens to execute b.bar()
        a blocks on the monitor of b
        b blocks on the monitor of a
        // in parallel
        a.bar() || b.bar();
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
}
```

is equivalent to

```java
class C {
    public void f() {
        monitor_enter(this);
        // body of f
        monitor_exit(this);
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
    public synchronized void other() {
        // body of other
    }
}
```

**Consider this Java class:**

```java
class C {
    public synchronized void f() {
        // body of f
    }
    public synchronized void g() {
        // body of g
    }
}
```

**Consider this Java class:**

```java
class Foo {
    public synchronized void bar() {
        // body of bar
    }
}
```

is the only safe approach on standard operating systems but what about algorithms that require locking?
Treatment of Deadlocks

Observation: Deadlocks occur if the following four conditions hold:
- **mutual exclusion**: processes require exclusive access
- **no preemption**: resources cannot be taken away form processes
- **circular wait**: waiting processes form a cycle

The occurrence of deadlocks can be:
- **detected**: check within OS for a cycle, requires ability to preempt
- **prevented**: design programs to be deadlock-free
- **avoided**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks are partially ordered.

Definition (lock sets)

Let \( L \) denote the set of locks. We call \( \lambda(p) \subseteq L \) the lock set at \( p \), i.e. the set of locks that may be in the "acquired" state at program point \( p \).

We require the transitive closure \( \sigma^+ \) of a relation \( \sigma \):

Definition (transitive closure)

Let \( \sigma \subseteq X \times X \) be a relation. Its transitive closure is \( \sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i \) where:

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \{(x_1, x_2) \mid \exists x_2 \in X. (x_1, x_2) \in \sigma^i \land (x_2, x_1) \in \sigma^i \} \cup \sigma^i
\]

Each time a lock is acquired, we track the lock set at \( p \):

Definition (lock order)

Let \( \sigma \subseteq X \times X \) be a relation. Its transitive closure is \( \sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i \) where:

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \{(x_1, x_3) \mid \exists x_2 \in X. (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma^i \} \cup \sigma^i
\]

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists \( \sigma \subseteq X \times X \) such that \( l \prec^f l' \) if and only if \( \lambda(l) \subseteq L \), then the program is free of deadlocks.
Freedom of Deadlock
The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)
If there exists no \( a \in L \) with \( a \prec a \) then the program is free of deadlocks.

Suppose a program blocks on semaphores (mutexes) \( L_S \) and on monitors \( L_M \) such that \( L = L_S \cup L_M \).

Freedom of Deadlock
The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)
If there exists no \( a \in L \) with \( a \prec a \) then the program is free of deadlocks.

Suppose a program blocks on semaphores (mutexes) \( L_S \) and on monitors \( L_M \) such that \( L = L_S \cup L_M \).

Avoiding Deadlocks in Practice
How can we verify that a program contains no deadlocks?
1. identify mutex locks \( L_S \) and summarized monitor locks \( L_M \subseteq L_M \)
2. identify non-summary monitor locks \( L_M^S \subseteq L_M \setminus L_M \)
3. sort locks into ascending order according to lock sets
4. check that no cycles exist except for self-cycles of non-summary monitors

Avoiding Deadlocks in Practice
How can we verify that a program contains no deadlocks?
1. identify mutex locks \( L_S \) and summarized monitor locks \( L_M \subseteq L_M \)
2. identify non-summary monitor locks \( L_M^S \subseteq L_M \setminus L_M \)
3. sort locks into ascending order according to lock sets
4. check that no cycles exist except for self-cycles of non-summary monitors

Locks Roundup
An example for the latter is the \( \text{Foo} \) class: two instances of the same class call each other
Atomic Execution and Locks
Consider replacing the specific locks with atomic annotations:

```c
void pop() {
    ... wait(&q->t);
    ... if (*q) { signal(&q->t); return; }
    ... if (c) wait(&q->s);
    ... if (c) signal(&q->s);
    signal(&q->t);
}
```

- nested atomic blocks still describe one atomic execution
- locks convey additional information over atomic
- locks cannot easily be recovered from atomic declarations

Outlook
Writing atomic annotations around sequences of statements is a convenient way of programming.

Idea of mutexes: Implement atomic sections with locks:
- a single lock could be used to protect all atomic blocks
- more concurrency is possible by using several locks
- some statements might modify variables that are never read by other threads ← no lock required
- statements in one atomic block might access variables in a different order to another atomic block ← deadlock possible with locks implementation
- creating too many locks can decrease the performance, especially when required to release locks in \( \lambda(l) \) when acquiring!

Concurrency across Languages
In most systems programming languages (C,C++) we have
- the ability to use atomic operations
- → we can implement wait-free algorithms

In Java, C# and other higher-level languages
- Atomic annotations around sequences of statements is a convenient way of programming.
Concurrency across Languages

In most systems programming languages (C, C++) we have

- the ability to use atomic operations
- we can implement wait-free algorithms
- provide monitors and possibly other concepts
- often simplify the programming but incur the same problems

<table>
<thead>
<tr>
<th>language</th>
<th>barriers</th>
<th>wait-lock-free</th>
<th>semaphore</th>
<th>mutex</th>
<th>monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Java, C#</td>
<td>-</td>
<td>(b)</td>
<td>(c)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(a) some pthread implementations allow a reentrant attribute
(b) newer API extensions (java.util.concurrent.atomic.* and System.Threading.Interlocked resp.)
(c) simulate semaphores using an object with two synchronized methods

Abstraction and Concurrency

Two fundamental concepts to build larger software are:

- abstraction: an object storing certain data and providing certain functionality may be used without reference to its internals
- composition: several objects can be combined to a new object without interference

Both, abstraction and composition are closely related, since the ability to compose depends on the ability to abstract from details. Consider an example:

- a linked list data structure exposes a fixed set of operations to modify the list structure, such as push() and forAll()
- a set object may internally use the list object and expose a set of operations, including push()

The insert() operation uses the forAll() operation to check if the element already exists and uses push() if not.

Wrapping the linked list in a mutex does not help to make the set thread-safe.

- wrap the two calls in insert() in a mutex
- but other list operations can still be called -- use the same mutex
- unlike sequential algorithms, thread-safe algorithms cannot always be composed to give new thread-safe algorithms

Programming Languages

Concurrency: Transactions

Dr. Michael Petter
Winter term 2018

Summary

Classification of concurrency algorithms:

- wait-free, lock-free, locked
- next on the agenda: transactional

Wait-free algorithms:

- never block, always succeed, never deadlock, no starvation
- very limited in expressivity

Lock-free algorithms:

- never block, may fail, never deadlock, may starve
- invariant may only span a few bytes (8 on Intel)

Locking algorithms:

- can guard arbitrary code
- can use several locks to enable more fine grained concurrency
- may deadlock
- semaphores are not re-entrant, monitors are
- use algorithm that is best fit

References

E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ISSN 0360-0300.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
**Transactional Memory [2]**

Idea: automatically convert atomic blocks into code that ensures atomic execution of the statements.

```c
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
}
```

Execute code as transaction:
- execute the code of an atomic block
- nested atomic blocks act like a single atomic block
- check that it runs without conflicts due to accesses from another thread
- if another thread interferes through conflicting updates:
  - undo the computation done so far
  - re-start the transaction
- provide a retry keyword similar to the wait of monitors

---

**Weak- and Strong Isolation**

If guarantees are only given about memory accessed inside atomic, a TM implementation provides weak isolation. Can we mix transactions with code accessing memory non-transactionally?

- no conflict detection for non-transactional accesses
- standard race problems as in unlocked shared accesses
- give programs with races the same semantics as if using a single global lock for all atomic blocks
- strong isolation retains order between accesses to TM and non-TM

**Definition (SLA)**

The single-lock atomicity is a model in which the program executes as if all transactions acquire a single, program-wide mutual exclusion lock.

---

**Semantics of Transactions**

The goal is to use transactions to specify atomic executions. Transactions are rooted in databases where they have the ACID properties:

- atomicity: a transaction completes or seems to have run
- consistency: each transaction transforms a consistent state to another consistent state
- isolation: transactions do not interfere with each other
- durability: the effects are permanent

Transactions themselves must be serializable:
- the result of running concurrent transactions must be identical to one execution of them in sequence
- serializability for transactions is insufficient to perform synchronization between threads

---

**Consistency During Transactions**

**Concurrent Monitors**

ACID states how committed transactions behave but not what may happen until a transaction commits.

- a transaction that is run on an inconsistent state may generate an inconsistent state — zombie transaction
- in the best case, the zombie transaction will be aborted eventually
- may race and generate erratic behavior
- transactions may cause havoc when run on inconsistent states

```c
atomic {
    // preserved invariant: x==y
    int tmp1 = x;
    atomic {
        int tmp2 = y;
        assert(tmp1-tmp2==0);
        y = 10;
    }
}
```

---

**Disadvantages of the SLA model**

- SLA has a weaker progress guarantee than a transaction should have
- SLA correctness is too strong in practice

**Definition (opacity)**

A TM system provides opacity if failing transactions are serializable w.r.t. committing transactions.

---

**Transactional Sequential Consistency**

How about a more permissive view of transaction semantics?

- TM should not have the blocking behaviour of locks
- the programmer cannot rely on synchronization
- actual implementations use TSC with some race-free re-orderings

**Definition (TSC)**

The transactional sequential consistency is a model in which the accesses within each transaction are sequentially consistent.
Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:
- convert every read access \( x \) from a shared variable to `ReadTx(x)`
- convert every write access \( y \) to a shared variable to `WriteTx(x,y)`

**Translation of atomic-Blocks**

A TM system must track which shared memory locations are accessed:
- convert every read access \( x \) to a shared variable to `ReadTx(x)`
- convert every write access \( y \) to a shared variable to `WriteTx(x,y)`

**Principles of TL2**

The idea: obtain a version from the global counter on starting the transaction, the `read-version`, and watch out for accesses to newer versions throughout the transaction.
- A read `ReadTx` from a field at offset of object `obj` aborts,
  - when the objects version is younger than the transaction
  - otherwise it is locked at the moment of access
- `WriteTx` is successively
  - picks up locks for each written object
  - increments the global version
  - checks the read objects for being up to date
- `CommitTx` successively
  - atomically replace a pointer to the old object with a pointer to the new object
  - marks the object as not accessed
  - atomically replaces the pointer to the old object with the pointer to the new object
  - marks the object as accessed

**General Challenges when using STM**

Executing atomic blocks by repeatedly trying to execute them non-atomically creates new problems:
- a transaction might unnecessarily be aborted
  - the granularity of what is locked might be too large
- a TM implementation might impose restrictions:
  - `atomic { // clock=12
    // code
  }`
  - `atomic { // clock=13
    // code
  }

**A Software TM Implementation**

A software TM implementation allocates a transaction descriptor to store data specific to each atomic block, for instance:
- `atom`
  - `undo-log` of writes if writes have to be undone if a commit fails
  - `redo-log` of writes if writes are postponed until a commit
- `read- and write-set` locations accessed so far
- `read- and write-version`: time stamp when value was accessed

**Example:**

Consider the TL2 STM (software transactional memory) implementation [1]:
- provides opacity: zombie transactions do not see inconsistent state
- uses lazy versioning: writes are stored in a redo-log and done on commit
- validating conflict detection: accessing a modified address aborts

```c
atomic { // clock=12
    // code
} while (!CommitTx());
```

```c
atomic { // clock=13
    // code
}
```

```c
int r = ReadTx(kz,0); // if kz.kv12 + clock
WriteTx(kz,0) = 42; // clock=13
```
Integrating Non-TM Resources

Allowing access to other resources than memory inside an atomic block poses problems:
- storage management, condition variables, volatile variables, input/output
- semantics should be as if atomic implements SLA or TSC semantics

Example choice is one of the following:
- Prohibit It. Certain constructs do not make sense. Use compiler to reject these programs.
- Execute It. I/O operations may only happen in some runs (e.g. file writes usually go to a buffer). Abort if I/O happens.
- Irrevocably Execute It. Universal way to deal with operations that cannot be undone: enforce that this transaction terminates (possibly before starting) by making all other transactions conflict.
- Integrate It. Re-write code to be transactional: error logging, writing data to a file, etc.
  — currently best to use TM only for memory; check if TM supports irrevocable transactions

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:
- additional hardware to track read- and write-sets
- conflict detection is easier using the cache:
  - additional hardware makes it cheap to perform conflict detection
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts
- limited by fixed hardware resources, a software backup must be provided

Two principal implementation of HTM:
- Explicit Transactional Memory: each access is marked as transactional
  - similar to StartTx, ReadTx, WriteTx, and CommitTx
  - requires separate transaction instructions
- Implicit Transactional Memory: only the beginning and ending of a transaction are marked
  - same instructions can be used, hardware interprets them as transactional
  - only instructions affecting memory that can be cached can be executed transactionally
  - hardware access, OS calls, page table changes, etc. all abort a transaction
  - provides strong isolation

Example for HTM

AMD Advanced Synchronization Facilities (ASF):
- defines a logical speculative region
- LOCK MOV instructions provide explicit data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel’s TSX in Broadwell/Skylake microarchitecture (since Aug 2014):
- implicitly transactional, can use normal instructions within transactions
- tracks read/write set using a single transaction bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...) on a simple counter to support nested transactions
- may abort at any time due to lack of resources
- aborting in an inner transaction means aborting all of them

Intel provides two software interfaces to TM:
- Restricted Transactional Memory (RTM)
- Hardware Lock Elision (HLE)

Implementing RTM using the Cache (Intel)

Supporting Transactional operations:
- augment each cache line with an extra bit $T$
- introduce a nesting counter $C$ and a backup register set

Restricted Transactional Memory

- additional transaction logic:
  - $\text{begin}$ increments $C$ and, if $C = 0$, backs up registers and flushes buffer
  - subsequent read or write access to a cache line sets $T$ if $C > 0$
  - applying an $\text{invalidate}$ message to a cache line with 7 flag issues $\text{abort}$
  - observing a read for a modified cache line with 7 flag issues $\text{abort}$
  - $\text{abort}$ clears all 7 flags and the store buffer, invalidates the former TM lines, sets $C = 0$, and restores CPU registers
  - $\text{read}$ decrements $C$ and, if $C = 0$, clears all 7 flags, flushes store buffer
Restricted Transactional Memory

Provides new instructions `xbegin`, `xend`, `xabort`, and `xtest`:
- `xbegin` on transaction start skips to the next instruction or on abort
  - implicitly stores an error code in `eax`
- `xend` commits the transaction started by the most recent `xbegin`
- `xabort` aborts the whole transaction with an error code
- `xtest` checks if the processor is executing transactionally

The instruction `xbegin` is made accessible via library function `xbegin()`:

```
xbegin()
```

Protecting the Fall-Back Path

Use a lock to prevent the transaction from interrupting the fall-back path:

```
int ...
```

Concurrency: Transactions Hardware Transactional Memory

Happened Before Diagram for Transactions

Augment MESI states with extra bit `T`. CPU A: d:E5 t:E0, CPU B: d:I

```
_txnL1: 
_txnL1: 
```

Common Code Pattern for Mutexes

Using HTM in order to implement mutex:

```
void update(int idx, int val) {
    if(!_xbegin()==_XBEGIN_STARTED) {
        data[idx] += val;
        _xend();
    } else {
        data[idx] += val;
    }
}
```

Protection for the Fall-Back Path

Consider executing the following code concurrently with itself:

```
int data[100]; // shared
void update(int idx, int value) {
    _xbegin();
    data[idx] += value;
    _xend();
}
```

Several problems:
- the fall-back code may execute racing itself
- the fall-back code is not isolated from the transaction

First idea: ensure that the fall-back path is executed atomically

```
if (!mutex>0) _xabort();
```

Hardware Lock Elision

```
void update(int idx, int value) {
    if(!_xbegin()==_XBEGIN_STARTED) {
        data[idx] += value;
        _xend();
    } else {
        data[idx] += value;
        signal(mutex);
    }
}
```

- the fall-back code does not execute racing itself
- the fall-back code is now isolated from the transaction

void unlock(int* mutex) {
    if (*mutex>0) signal(mutex);
    data[idx] += value;
}
```
Hardware Lock Elision

Observation: Using RTM to implement lock elision is a common pattern to provide special handling in hardware: HLE

- Hardware Lock Elision
  - By default defer actual acquisition of the lock
  - Instead rely on HTM to sort out conflicting concurrent accesses
  - Fail back to actual locking only in case of conflicts
  - Support legacy lock code by locally acting as if semaphore value is actually modified

- Requires annotations for lock instructions:
  - Instruction that increments the semaphore must be prefixed with acquire
  - Instruction setting the semaphore to 0 must be prefixed with xrelease
  - These prefixes are ignored on older platforms
- For a successful elision, all signal/wait operations of a lock must be annotated

Implementing Lock Elision

Transactional operation:
- Re-uses infrastructure for Restricted Transactional Memory
- Add a buffer for elided locks, similar to store buffer

- xacquire of lock ensures
  - shared/exclusive cache line state with T, issues xbegin and keeps the modified lock value in elided lock buffer
  - rw access to other cache lines sets T
  - Applying an invalidate message to a T
  - Cache line issues xabort, analogous to read message to a TM cache line
- A local CPU load from the address of the elided lock accesses the buffer
- On xrelease on the same lock, decrement C and, if C = 0, clear T flags and elided locks buffer flush the store buffer

Transactional Memory: Summary

Transactional memory aims to provide atomic blocks for general code:
- Frees the user from deciding how to lock data structures
- Compositional way of communicating concurrently
- Can be implemented using software (locks, atomic updates) or hardware
- It is hard to get the details right:
  - Semantics of explicit HTM and STM transactions quite subtle when mixing with non-TM (weak vs. strong isolation)
  - Single-lock atomicity vs. transactional sequential consistency semantics
  - STM not the right tool to synchronize threads without shared variables
  - TM providing opacity (serializability) requires eager conflict detection or lazy version management

Pitfalls in implicit HTM:
- RTM requires a fall-back path
- No progress guarantee
- HLE can be implemented in software using RTM

TM in Practice

Availability of TM Implementations:
- GCC can translate accesses in _transaction atomic regions into libtm library calls
- The library libitm provides different TM implementations:
  - On systems with TSX, it maps atomic blocks to HTM instructions
  - On systems without TSX and for the fallback path, it resorts to STM
- C++20 standardizes synchronized atomic 2.3.3 blocks
- RTM support slowly introduced to OpenJDK Hotspot monitors

Use of hardware lock elision is limited:
- Allows to easily convert existing locks
- Pthread locks in glibc use RTM https://lwn.net/Articles/534768/
  - Allows implementation of back-off mechanisms
- HLE only special case of general lock
- Implementing monitors is challenging
  - Lock count and thread id may lead to conflicting accesses
  - In pthreads error conditions often not checked anymore

Outlook

Several other principles exist for concurrent programming:
- Non-blocking message passing (the actor model)
  - A program consists of actors that send messages
  - Each actor has a queue of incoming messages
  - Messages can be processed and new messages can be sent
  - Special filtering of incoming messages
  - Example: Erlang, many add-ons to existing languages
- Blocking message passing (CSF, π-calculus, join-calculus)
  - A process sends a message over a channel and blocks until the recipient accepts it
  - Channels can be send over channels (π-calculus)
  - Examples: Occam, Occam-π, Go
- (Immediate) priority ceiling
  - Declare processes with priority and resources that each process may acquire
  - Each resource has the maximum (ceiling) priority of all processes that may acquire it
  - A process’ priority at run-time increases to the maximum of the priorities of held resources
  - The process with the maximum (run-time) priority executes

References

- D. Dice, O. Shalev, and N. Shavit.
  Transactional Locking II.
- T. Harris, J. Larus, and R. Rajwar.
  Transactional memory, 2nd edition.

Online resources on Intel HLT and GCC’s STM:
  fun-with-intel-transactional-synchronization-extensions
Dispatching - Outline

- Motivation
- Formal Model
- Quiz
- Dispatching from the Inside

Solutions in Single-Dispatching
- Type introspection
- Generic interface

Multi-Dispatching
- Formal Model
- Multi-Java
- Multi-dispatching in Perl6
- Multi-dispatching in Clojure

Section 1
Direct Function Calls

Function Dispatching (ANSI C89)

```c
#include <stdio.h>
void fun(int i) { }
void bar(int i, double j) { }

int main()
{
    fun(1);
    bar(1,1.2);
    void (*foo)(int);
    foo = fun;
    return 0;
}
```

Functions with same names but different parameters not legal
Generic Selection (C11)

generic-selection ↦→ Generic(exp, generic-assoclist)
generic-assoclist ↦→ ... println(x) printf(printf_dec_format(x), x), printf("
");
int main(){
println(1.2);
println(1);
return 0;
}

Static Methods are Statically Dispatched
Signature
f'(e1,...,en)
 t0  f(t1 p1,...,tn pn)
dispatches to
handles
determines is applicable to
t0',..., tn'

Overloading with Scopes (C++)
#include<iostream>
using namespace std;
class B { public:
int f(int i) { cout << "f(int): "; return i+1; }
};
class D extends B { public:
public double f(double d) { cout << "f(double): "; return d+1.3; }
public int f(int i) { cout << "f(int): "; return i; }
public static void p(Object o) { System.out.print(o); }
};

Overloading Hassles

class D {
public static void p(Object o) { System.out.print(o); }
public int f(int i) { p("f(int): "); return i+1; }
public double f(double d) { p("f(double): "); return d+1.3; }
};

class D extends B {
public double f(double d) { p("f(double): "); return d+1.3; }
public int f(int i) { p("f(int): "); return i; }
};

Overloading with Inheritance (Java)

class B {
public int f(int i) { System.out.print("f(int): "); return i+1; }
}
class D extends B {
public double f(double d) { System.out.print("f(double): "); return d+1.3; }
public int f(int i) { System.out.print("f(int): "); return i; }
};

Overloading with Inheritance (Java)

class B {
public static void p(Object o) { System.out.print(o); }
public int f(int i) { p("f(int): "); return i+1; }
}
class D extends B {
public double f(double d) { p("f(double): "); return d+1.3; }
};
Inside the Javac – Predicates

Concept of methods being applicable for arguments:

// true if the given method is applicable to the given arguments
boolean isApplicable(MemberDefinition m, Type args[]) {
    // sanity check
    Type mType = m.getType();
    if (mType == null) return false;
    Type moreType = more.getClassDeclaration().getType();
    if (mType == null) return false;
    for (int i = args.length ; --i >= 0 ;)
        if (!isApplicable(mArgs[i], mType.getArgumentTypes())) return false;
    return true;
}

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor,
                                Identifier methodName, Type[] argumentTypes) throws ... {
    // A tentative maximally specific method.
    MemberDefinition tentative = null;
    // A list of other methods which may be maximally specific too.
    List candidateList = null;
    // Get all the methods inherited by this class which have the same 'methodName'
    for (MemberDefinition method : allMethods.lookupName(methodName)) {
        // see if this method is applicable
        if (isApplicable(method, argumentTypes)) continue;
        // see if this method is accessible
        if (env.isMoreSpecific(tentative, method)) continue;
        // method becomes our tentative maximally specific method.
        tentative = method;
        if (candidateList == null) candidateList = new ArrayList();
    }
    if (candidateList == null) candidateList = new ArrayList();
    if (env.isMoreSpecific(tentative, method)) throw new AmbiguousMember(tentative, method);
    return tentative;
}

Section 3

Overriding Methods

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor,
                                Identifier methodName, Type[] argumentTypes) throws ... {
    // A tentative maximally specific method.
    MemberDefinition tentative = null;
    // A list of other methods which may be maximally specific too.
    List candidateList = null;
    // Get all the methods inherited by this class which have the same 'methodName'
    for (MemberDefinition method : allMethods.lookupName(methodName)) {
        // see if this method is applicable
        if (isApplicable(method, argumentTypes)) continue;
        // see if this method is accessible
        if (env.isMoreSpecific(tentative, method)) continue;
        // method becomes our tentative maximally specific method.
        tentative = method;
        if (candidateList == null) candidateList = new ArrayList();
    }
    if (candidateList == null) candidateList = new ArrayList();
    if (env.isMoreSpecific(tentative, method)) throw new AmbiguousMember(tentative, method);
    return tentative;
}

Object Orientation

In Object Orientation, we see objects associating strongly with particular procedures, a.k.a. Methods.

Subtyping in Object Orientation

An Object Oriented Subtype is supposed to take responsibility for calls to Methods that are associated with the type, that it specializes.

In OO, at runtime subtypes can inherit statically more general typed variables

N implicit call the specialized method!
How can we implement that?

Let's look at what Java does!

The Java platform as example for state of the art OO systems:
- Static Java-based compiler
- Dynamic Hotspot JIT-Compiler/Interpreter

Let's watch the following code on its way to the CPU:

```java
public static void main(String[] args){
    Integral i = new Natural(1);
    i.incBy(42);
}
```

Bytecode

```java
⇝ matchMethod returns the statically most specific signature
⇝ Codegeneration hardcodes invokevirtual with this signature

Code:
0: new 4 // class Natural
1: 1
2: dup2
3: getfield #3 // Field number:I
4: iload_1
5: invokevirtual #2 // Method ... // Field number:I
6: return
```

Inside the Hotspot VM

The method lookup recursively traverses the super class chain:

```java
MethodDesc* klass::find_method(ObjArrayDesc* methods, Symbol* name, Symbol* signature) {
    int len = methods->length();
    if (len <= 0) return NULL;
    int mid = (l + h) >> 1;
    for (int i = l; i <= h; i = mid) {
        MethodDesc* m = (MethodDesc*)methods->obj_at(i);
        if (m->name() != name) break;
        if (m->signature() == signature) return m;
    }
    return NULL;
}
```

Inside the Hotspot VM

```java
void LinkResolver::resolve_method(methodHandle& resolved_method, KlassHandle resolved_klass,Symbol* name, Symbol* signature, KlassHandle current_klass) {
    // 6. access checks, etc.
}
```

Single-Dispatching: Summary

Compile Time

```java
MethodDesc::find_method(ObjArrayDesc* methods, Symbol* name, Symbol* signature) {
    int len = methods->length();
    // methods are sorted, do binary search
    int i, l = 0 , h = len - 1;
    while (l <= h) {
        int mid = (l + h) >> 1;
        MethodDesc* m = (MethodDesc*)methods->obj_at(mid);
        if (m->name() == name) return m;
        if (m->signature() == signature) return m;
        if (m->name() != name) break;
        if (m->signature() == signature) return m;
        if (m->name() != name) break;
    }
    return NULL;
}
```

Runtime

```java
void LinkResolver::resolve_method(methodHandle& resolved_method, KlassHandle resolved_klass,Symbol* name, Symbol* signature) {
    // 6. access checks, etc.
}
```

What is the semantics of invokevirtual?

- Check the runtime interpreter: Hotspot VM calls `resolve_method`!

Insider the Hotspot VM

```java
Inside the Hotspot VM

ObjectHandle resolve_method(ObjectHandle resolved_method, ObjectHandle resolved_klass,Symbol* name, Symbol* signature) {
    // 6. access checks, etc.
}
```

What is the semantics of invokevirtual?

- Check the runtime interpreter: Hotspot VM calls `resolve_method`!

Insider the Hotspot VM

```java
Inside the Hotspot VM

ObjectHandle resolve_method(ObjectHandle resolved_method, ObjectHandle resolved_klass,Symbol* name, Symbol* signature) {
    // 6. access checks, etc.
}
```

What is the semantics of invokevirtual?

- Check the runtime interpreter: Hotspot VM calls `resolve_method`!

Insider the Hotspot VM

```java
Inside the Hotspot VM

ObjectHandle resolve_method(ObjectHandle resolved_method, ObjectHandle resolved_klass,Symbol* name, Symbol* signature) {
    // 6. access checks, etc.
}
```

What is the semantics of invokevirtual?

- Check the runtime interpreter: Hotspot VM calls `resolve_method`!
Example: Sets of Natural Numbers

```java
class Natural {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Natural n){
        return n.number == number;
    }
}
```

```java
Set<Natural> set = new HashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
```

```bash
$ java Natural
[0]
```

Why? Is HashSet buggy?

⇝ Keep attention to exact signature!

Mini-Quiz: Java Method Dispatching

```java
class A {
    public static void p (Object o) { System.out.println(o); }
    public void m1 (A a) { p("m1(A) in A"); }
    public void m1 () { m1(new B()); }
    public void m2 (A a) { p("m2(A) in A"); }
    public void m2 () { m2(this); }
}
class B extends A {
    public void m1 (B b) { p("m1(B) in B"); }
    public void m2 (A a) { p("m2(A) in B"); }
    public void m3 () { super.m1(this); }
}
```

```java
B b = new B(); A a = b; a.m1(b);
B b = new B(); B a = b; b.m1(a);
B b = new B(); b.m2();
B b = new B(); b.m1();
B b = new B(); b.m3();
```

Section 4

Multi-Dispatching

Can we expect more than Single-Dispatching?

Mainstream languages support specialization of first parameter:
C++, Java, C#, Smalltalk, Lisp

So how do we solve the `equals()` problem?

1 introspection?
2 generic programming?
3 double dispatching?

Introspection

```java
class Natural {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Object n){
        if (!(n instanceof Natural)) return false;
        return ((Natural)n).number == number;
    }
}
```

```bash
$ java Natural
[D,0]
```

✓ Works but burdens programmer with type safety
and is only available for languages with type introspection

Generic Programming

```java
interface Equalizable<T>{
    boolean equals(T other);
}
class Natural implements Equalizable<Natural> {
    Natural(int n){ number=Math.abs(n); }
    int number;
    public boolean equals(Natural n){
        return n.number == number;
    }
}
```

```java
EqualizableAwareSet<Natural> set = new MyHashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
```

Why? Is HashSet buggy?

⇝ Keep attention to exact signature!

So how do we solve the `equals()` problem?

1 introspection?
2 generic programming?
3 double dispatching?

Introspection
Double Dispatching

abstract class EqualsDispatcher{
  boolean dispatch(Natural) { return false };
  boolean dispatch(Object) { return false };
}
class Natural {
  int number;
  public boolean equals(Object n){
    return n.equals(this);
  }
  public boolean doubleDispatch(EqualsDispatcher ed){
    return ed.dispatch(this);
  }
}
✓ Works but needs Dispatcher to know complete class hierarchies

Formal Model of Multi-Dispatching [7]

Signature
f'(e1,...,en)
 t0  f(t1 p1,...,tn pn)
dispatches to
handles
determines
Specializer
specialized by
is applicable to
t0',..., tn'

How it works
- Specializers as subtype annotations to parameter types
- Dispatcher selects Most Specific Concrete Method

Implications of the implementation

Type-Checking
- Typechecking families of concrete methods introduces checking the existence of unique most specific methods for all valid visible type tuples.
- Multiple-Inheritance or interfaces as specializers introduce ambiguities, and thus induce runtime ambiguity exceptions

Code-Generation
- Specialized methods generated separately
- Dispatcher method calls specialized methods
- Order of the dispatch tests determines the most specialized method

Performance penalty
The runtime-penalty for multi-dispatching is related to the number of parameters of a multi-method many instanceof tests.

Natural Numbers Behind the Scenes

$ javap -c Natural
public boolean equals(java.lang.Object);
Code:
0: aload_1
1: ... #31; //Method equals$body3$1:(LObject;)Z
21: ireturn
⇝ Redirection to methods equals$body3$1 and equals$body3$0

Section 5
Natively multidispatching Languages

Natural Numbers in Multi-Java [3]
class Natural {
  public Natural(int n){ number=Math.abs(n); }
  private int number;
  public boolean equals(Object n){
    return n.equals(this);
  }
};
Set<Natural> set = new HashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
>$ java Natural
[0]
✓ Clean Code!

Natural Numbers in Multi-Java [3]
class Natural {
  public Natural(int n){ number=Math.abs(n); }
  private int number;
  public boolean equals(Object@Natural n){
    return n.number == number;
  }
};
...
Set<Natural> set = new HashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set);
>$ java Natural
[0]
✓ Clean Code!

Natural Numbers Behind the Scenes

$ javap -c Natural
public boolean equals(java.lang.Object);
my Cool $foo;
my Cool $bar;
multi fun(Cool $one, Cool $two){
    say "Dispatch base"
}
multi fun(Int $one, Str ... "Dispatch 2"
}
$foo=1;
$bar="blabla";
fun($foo,$bar);
$foo="bla";
fun($foo,$bar)
Dispatch 1
Dispatch base

Clojure
... is a top dialect for the JVM with:

- Prefix notation
- [] – Brackets for lists
- :: – Userdefined keyword constructor :keyword
- [] – Vector constructor
- fn – Creates a lambda expression
  \( \text{fn} \left( x, y \right) \rightarrow x + y \)\n
- derive – Generates hierarchical relationships
- derive ::parent
- defmulti – Creates new generic method
- defmethod – Creates new concrete method

(defn salary [amount]
  (cond (< amount 600) ::poor
        (>= amount 5000) ::rich
        :else ::average))
(defrecord UniPerson [name wage])
(defmulti print (fn [person] (salary (-> person ::wage))))
(defmethod print ::poor [person] (str "HiWi " (-> person ::name)))
(defmethod print ::rich [person] (str "Prof. " (-> person ::name)))
(pr (print (UniPerson. "Stefan" 2000)))
(pr (print (UniPerson. "Seidl" 16000)))
Dr. Simon
HiWi Stefan
Prof. Seidl

Lessons Learned
Lessons Learned
1. Dynamically dispatched methods are complex interaction of static and dynamic techniques
2. Single Dispatching as in major OO-Languages
3. Making use of Open Source Compilers
4. Multi Dispatching generalizes single dispatching
5. Multi Dispatching Perl6
6. Multi Dispatching Clojure

Con
- Counters privileged 1st parameter
- Runtime overhead
- New exceptions when used with multi-inheritance
- Most Specific Method ambiguous

Clojure
... is a lisp dialect for the JVM with:

- Prefix notation
- [] – Brackets for lists
- :: – Userdefined keyword constructor ::keyword
- [] – Vector constructor
- fn – Creates a lambda expression
  \( \text{fn} \left( x, y \right) \rightarrow x + y \)\n
- derive – Generates hierarchical relationships
- derive ::parent
- defmulti – Creates new generic method
- defmethod – Creates new concrete method

(defmulti name dispatch-fn)
defmethod – Creates new concrete method
(defmethod name dispatch-val &fn-tail)

Principle of Multidispatching in Clojure
(derive ::child ::parent)
(defmulti fun (fn [a b] [a b]))
(defmethod fun [::child ::child] [a b] "child equals")
(defmethod fun [::parent ::parent] [a b] "parent equals")
(pr (fun ::child ::child))

Lessons Learned
Lessons Learned
1. Dynamically dispatched methods are complex interaction of static and dynamic techniques
2. Single Dispatching as in major OO-Languages
3. Making use of Open Source Compilers
4. Multi Dispatching generalizes single dispatching
5. Multi Dispatching Perl6
6. Multi Dispatching Clojure

dispatch methods

More Creative dispatching in Clojure
(defn salary [amount]
  (cond (< amount 600) ::poor
        (>= amount 5000) ::rich
        :else ::average))
(defrecord UniPerson [name wage])
(defmulti print (fn [person] (salary (-> person ::wage))))
(defmethod print ::poor [person] (str "HiWi " (-> person ::name)))
(defmethod print ::rich [person] (str "Prof. " (-> person ::name)))
(pr (print (UniPerson. "Stefan" 2000)))
(pr (print (UniPerson. "Seidl" 16000)))
Dr. Simon
HiWi Stefan
Prof. Seidl

Multidispatching

Pro
- Generalization of an established technique
- Directly solves problem
- Eliminates boilerplate code
- Compatible with modular compilation/type checking

Con
- Counters privileged 1st parameter
- Runtime overhead
- New exceptions when used with multi-inheritance
- Most Specific Method ambiguous

Other Solutions (extract)
- Dylan
- Scala
Section 6
Further materials

Further reading...

OpenJDK 7 Hotspot JIT VM.
http://hg.openjdk.java.net/jdk7/jdk7.

OpenJDK 7 Javac.
http://hg.openjdk.java.net/jdk7/jdk7.

Multijava: Design rationale, compiler implementation, and applications.
ACM Transactions on Programming Languages and Systems (TOPLAS), May 2006.


Programming Clojure.


Multiple dispatch in practice.
23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA), September 2008.

Outline

Inheritance Principles
- Interface Inheritance
- Implementation Inheritance
- Dispatching implementation choices

C++ Object Heap Layout
- Basics
- Single-Inheritance
- Virtual Methods

C++ Multiple Parents Heap Layout
- Multiple-Inheritance
- Virtual Methods
- Common Parents

Excursion: Linearization
- Ambiguous common parents
- Principles of Linearization
- Linearization algorithms

Interface vs. Implementation inheritance

The classic motivation for inheritance is implementation inheritance
- Code reuse
- Child specializes parents, replacing particular methods with custom ones
- Parent acts as library of common behaviours
- Implemented in languages like C++ or Lisp

Code sharing in interface inheritance inverts this relation
- Behaviour contract
- Child provides methods, with signatures predetermined by the parent
- Parent acts as generic code frame with room for customization
- Implemented in languages like Java or C#
Interface Inheritance

Queue
enqueue(x)
List
...
Stack
push(x)
CircularGraph
insertNodeAt(x,i)
dequeue() pop()
removeNodeAt(x,i)

<<interface>> <<interface>>

Implementation Inheritance

Ship
Aircraft Carrier
toot()
strikeAt(x,y)
Airport
shelter(Plane)moveTo(x,y)

“So how do we lay out objects in memory anyway?”

Excursion: Brief introduction to LLVM IR

LLVM intermediate representation as reference semantics:

; (recursive) struct definitions
%struct.A = type { i32, %struct.B, i32(i32)* }
%struct.B = type { i64, [10 x [20 x i32]], i8 }

; (stack-) allocation of objects
%a = alloca %struct.A

; address computation for selection in structure (pointers):
%1 = getelementptr %struct.A* %a, i64 0, i64 2
%2 = load i32* %1
%3 = add i32 %2, %p
ret i32 %3

Retrieve the memory layout of a compilation unit with:
clang -cc1 -x c++ -v -fdump-record-layouts -emit-llvm source.cpp
Retrieve the IR Code of a compilation unit with:
clang -O1 -S -emit-llvm source.cpp -o IR.llvm

Object layout

class A {
int a; int f(int);
};
class B : public A {
int b; int g(int);
};
class C : public B {
int c; int h(int);
};
...
C c;
c.g(42);

Translation of a method body

class A {
int a; int f(int);
};
class B : public A {
int b; int g(int);
};
class C : public B {
int c; int h(int);
};
B::g(int p) {
return p+b;
}
define i32 @_g(%class.B* %this, i32 %p) {
%1 = getelementptr %class.B* %this, i64 0, i32 1
%2 = load i32* %1
%3 = add i32 %2, %p
ret i32 %3
}
“Now what about polymorphic calls?”

Single-Dispatching implementation choices

1. Manual search run through the super-chain (Java Interpreter -> last talk)
   ```
   call i32 @__dispatch(%class.C* %c, 42, i32* "f(int, void)"
   ```

2. Caching the dispatch result (→ Hotspot/JIT)
   ```
   assert (%c type %class.D) ; verify objects class presumption
   call i32 @_f_from_D(%class.C* %c, 42) ; directly call f
   ```

3. Precomputing the dispatching result in tables
   - Full 2-dim matrix
   - 1-dim Row Displacement Dispatch Tables
   - Virtual Tables
     ```
     %class.C = type { %class.B, i32, [4 x i8] }
     %class.B = type { [12 x i8], i32 }
     %class.A = type { i32 (**), i32 }
     ```

Object layout – virtual methods

```
C
int a
int b
int c
A::f
B::g
C::h
```
Keeping Calling Conventions

```cpp
class A {
  int a; int f(int);
};
class B {
  int b; int g(int);
};
class C : public A, public B {
  int c;
};
```

```cpp
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

MRO via Refined Postorder DFS

Refined RPRDFS

Monotonicity is not guaranteed!

Extension Principle: Monotonicity

If \( C(A, B) \) \( \Rightarrow C \rightarrow A \wedge C \rightarrow B \)

In General:

1. Inheritance is a uniform mechanism, and its searches (\( \rightarrow \) total order) apply identically for all object fields or methods.
2. In the literature, we also find the set of constraints to create a linearization as Method Resolution Order.
3. Linearization is a best-effort approach at best.

MRO via DFS

Leftmost Preorder Depth-First Search

**Principle 1:** Inheritance Relation

 Defined by parent-child. Example: \( C(A, B) \) \( \Rightarrow C \rightarrow A \wedge C \rightarrow B \)

Pyton: classical python objects (\( \leq 2.1 \)) use LPDFS!

**Principle 2:** Multiplicity Relation

 Defined by the succession of multiple parents. Example: \( C(A, B) \) \( \Rightarrow A \rightarrow B \)

In General:

1. Linearization is a best-effort approach at best.

Ambiguities

**Solution I:** Explicit qualification

```cpp
pc->A::f(42);
```

**Solution II:** Automagical resolution

Idea: The Compiler introduces a

linear order on the nodes of the inheritance graph

```cpp
pc->B::f(42);
```

Linearization

**Principle 1:** Inheritance Relation

Defined by parent-child. Example: \( C(A, B) \) \( \Rightarrow C \rightarrow A \wedge C \rightarrow B \)

**Principle 2:** Multiplicity Relation

Defined by the succession of multiple parents. Example: \( C(A, B) \) \( \Rightarrow A \rightarrow B \)

In General:

1. Inheritance is a uniform mechanism, and its searches (\( \rightarrow \) total order) apply identically for all object fields or methods.
2. In the literature, we also find the set of constraints to create a linearization as Method Resolution Order.
3. Linearization is a best-effort approach at best.

MRO via Refined Postorder DFS

**Refined RPDFS**

Monotonicity is not guaranteed!

Extension Principle: Monotonicity

If \( C_1 \rightarrow C_2 \) in \( C \)'s linearization, then \( C_1 \rightarrow C_2 \)

for every linearization of \( C \)'s children.

```cpp
L[A] = A B C D E F G \( \Rightarrow F \rightarrow G \)
L[C] = C D G E F \( \Rightarrow G \rightarrow F \)
```
MRO via C3 Linearization

A linearization $L$ is an attribute $L[C]$ of a class $C$. Classes $B_1, \ldots, B_n$ are superclasses to child class $C$, defined in the local precedence order $C(B_1 \ldots B_n)$. Then

$$L[C] = C \cup L[B_1] \cup \ldots \cup L[B_n] \cup \{ \} \mid C(B_1 \ldots B_n)$$

$L[Object] = Object$ with

$$\bigcup \ni (Li) = \begin{cases} 
   c \cdot (\bigcup \ni (Li) \setminus c) & \text{if } \exists \min k \forall j \ c = \text{head}(Lk) \notin \text{tail}(Lj) \\
   \triangle \text{fail} & \text{else}
\end{cases}$$

C3 detects and reports a violation of monotonicity with the addition of $A(B,C)$ to the class set.

C3 linearization [1]: is used in Python 3, Perl 6, and Solidity

Virtual Tables for Multiple Inheritance

```
class A {
    int a; virtual int f(int);
};
class B {
    int b; virtual int f(int);
    virtual int g(int);
};
class C : public A, public B {
    int c; int f(int);
};
```

```
C::f
```

```
virtual int f(int){
    \text{switch to A or B}
};
```

```
A(B, C) B(F, G) C(D, E)
D(G) E(F)
```

“And what about dynamic dispatching in Multiple Inheritance?”

Languages with automatic linearization exist

- CLOS Common Lisp Object System
- Solidity, Python 3 and Perl 6 with C3
- Prerequisite for \texttt{Mixins}

Linearization vs. explicit qualification

- No switch/duplexer code necessary
- No explicit naming of qualifiers
- Unique super reference
- Reduces number of multi-dispatching conflicts

Solidity, Python 3, and Perl 6 with C3
Basic Virtual Tables (⇝C++-ABI)

A Basic Virtual Table consists of different parts:
- **offset to top** of an enclosing object's memory representation
- **vptr** to an RTTI object (not relevant for us)
- **virtual function pointers** for resolving virtual methods

- Virtual tables are composed when multiple inheritance is used
- The vptr fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- clang + -fdump-vtable-layouts -emit-llvm code.cpp yields the vtables of a compilation unit

Casting Issues

class A { int a; };
class B { virtual int f(int);};
class C : public A , public B {
int c; int ... b = new C();
b->f(42);
! this-Pointer for C::f is expected to point to C
C::f
C::Bf
C::f
RTTI
B
0
RTTI
C::Bf

Thunks

Solution: thunks
... are trampoline methods, delegating the virtual method to its original implementation with an adapted this-reference

define i32 @_f(class B* this, i32 %i) {
  %1 = bitcast %class.B* this to i8*
  %2 = getelementptr i8* %1, i64 -16 ; sizeof(A)=16
  %3 = bitcast i8* %2 to %class.C*
  %4 = call i32 @f(%class.C* %3, i32 %i)
  ret i32 %4
}

- B-in-vtable entry for f(int) is the thunk _f(int)
- _f(int) adds a compiletime constant ∆B to this before calling f(int)
- f(int) addresses its locals relative to what it assumes to be a C pointer

Common Bases – Duplicated Bases

Standard C++ multiple inheritance conceptually duplicates representations for common ancestors:

```
L {
  int l; virtual void f(int);
};
class A : public L {
  int a; void f(int);
};
class B : public L {
  int b; void f(int);
};
class C : public A , public B {
  int c;
};
```

```
C* pc = (C*)pl;
L* pl = (L*)pc;
pl->f(42); // where to dispatch?
L+ lpl = (L*)pl;
C* pc = (C*)lpl;
```

Ambiguity!

```
L {
  int l; virtual void f(int);
};
class A : public L {
  int a; void f(int);
};
class B : public L {
  int b; void f(int);
};
class C : public A , public B {
  int c;
};
```
Common Bases – Shared Base Class

Optionally, C++ multiple inheritance enables a shared representation for common ancestors, creating the *diamond pattern*:

```
A
int f(int)
B
int g(int)
int b
int a
int f(int)
int w
W
int h(int)
int c
C
int g(int)
int h(int)
```

### Shared Base Class

```
class W {
int w; virtual void f(int);
virtual void g(int);
virtual void h(int);
};
class A : virtual W {...};
class B : virtual W {...};
```

### Ambiguities

- e.g. overriding `f` in `A` and `B`

### Offsets to virtual base

### Again: Casting Issues

```
class W { virtual int f(int); };
class A : virtual W { int a; };
class B : virtual W { int b; };
```

### Dynamic Type Casts

```
class A : public virtual W {...};
class B : public virtual W {...};
class C : public A , public B {...};
```

### Dynamic casting makes use of offset-to-top

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```

### Virtual Tables for Virtual Bases (⇝C++-ABI)

A Virtual Table for a Virtual Subclass gets a virtual base pointer

```
A
B
C
W
```

### Virtual Thunks

```
A
B
C
W
```

#### Virtual Thunks

```
A
B
C
W
```
Compiler and Runtime Collaboration

Compiler generates:
1. . . one code block for each method
2. . . one virtual table for each class-composition, with
   ▶ references to the most recent implementations of methods of a unique common signature (→ single dispatching)
   ▶ sub-tables for the composed subclasses
   ▶ static top-of-object and virtual bases offsets per sub-table
   ▶ (virtual) thunks as this-adapter per method and subclass if needed

Runtime:
1. At program startup virtual tables are globally created
2. Allocation of memory space for each object followed by constructor calls
3. Constructor stores pointers to virtual table (or fragments) in the objects
4. Method calls transparently call methods statically or from virtual tables, unaware of real class identity
5. Dynamic casts may use offset-to-top field in objects

Sidenote for MS VC++

- the presented approach is implemented in GNU C++ and LLVM
- Microsoft’s MS VC++ approaches multiple inheritance differently
  ▶ splits the virtual table into several smaller tables
  ▶ keeps a vbptr (virtual base pointer) in the object representation, pointing to the virtual base of a subclass.

Lessons Learned

1. Different purposes of inheritance
2. Heap Layouts of hierarchically constructed objects in C++
3. Virtual Table layout
4. LLVM IR representation of object access code
5. Linearization as alternative to explicit disambiguation
6. Pitfalls of Multiple Inheritance

Further reading...

A monotonic superclass linearization for Dylan.
In OOPSLA ’96, pages 17–34, 1996.

B. Stroustrup.
Multiple inheritance for C++.

Mixins and Traits

Programming Languages

Dr. Michael Petter
Winter 2018/19
What modularization techniques are there besides multiple implementation inheritance?

**Outline**

**Design Problems**
- Inheritance vs Aggregation
- (De-)Composition Problems

**Inheritance in Detail**
- A Model for single inheritance
- Inheritance Calculus with Inheritance Expressions
- Modeling Mixins

**Mixins in Languages**
- Simulating Mixins
- Native Mixins

**Traits in Languages**
- (Virtual) Extension Methods
- Squeak

**Cons of Implementation Inheritance**
- Lack of fine grained Control
- Inappropriate Hierarchies

**A Focus on Traits**
- Separation of Composition and Modeling
- Trait Calculus

**Mixins in Languages**

**Traits in Languages**

**Reusability ≡ Inheritance?**

- Codesharing in Object Oriented Systems is often inheritance-centric
- Inheritance itself comes in different flavours:
  - single inheritance
  - multiple inheritance
- All flavours of inheritance tackle problems of decomposition and composition

**The Adventure Game**

- Door
- LockedDoor
- ShortDoor
- ShortLockedDoor

- canOpen(Person p)
- canPass(Person p)
- canOpen(Person p)
- canPass(Person p)

**The Wrapper**

- SocketStream
- FileStream
- SynchRW

- write()
- read()
- write()
- read()
- acquireLock()
- releaseLock()

- Unclear relations
  - Cannot inherit from both in turn with Multiple Inheritance
  - (Many-to-One instead of One-to-Many Relation)
(De-)Composition Problems

All the problems of
  ● Relation
  ● Duplication
  ● Hierarchy
are centered around the question

"How do I distribute functionality over a hierarchy"

→ functional (de-)composition

Aggregation

- Undoes specialization
- Needs common ancestor

Duplication

With multiple inheritance, read/write Code is essentially identical but duplicated for each particular wrapper

Inappropriate Hierarchies

Implemented methods (acquireLock/releaseLock) to high

Classes and Methods

The building blocks for classes are
  ● a countable set of method names \( N \)
  ● a countable set of method bodies \( B \)

Classes map names to elements from the flat lattice \( B \) (called bindings), consisting of:
  ● method bodies \( b \) or classes \( c \) ∈ \( B \)
  ● \( ⊥ \) abstract
  ● \( ⊤ \) in conflict
and the partial order \( ⊏ \) for each \( b, c \) ∈ \( B \)

Definition (Abstract Class \( c \))

A general function \( c : N \to B \) is called a class.

Definition (Interface and Class)

A class \( c \) is called
  ● interface if \( c(\text{pre}(n)) = ⊥ \),
  ● abstract class if \( c(\text{pre}(n)) = ⊥ \) ∈ \( B \),
  ● concrete class if \( c(\text{pre}(n)) ∈ B \) for each \( n \).

Computing with Classes and Methods

Definition (Family of classes \( C \))

We call the set of all maps from names to bindings the family of classes \( C : N \to B \).

Several possibilities for composing maps \( C \sqcup C' \):
  ● the symmetric join \( \sqcup \) defined componentwise:

\[
(\text{pre}(n)) \begin{cases}
  b_2 & \text{if } b_1 = ⊥ \text{ or } n \notin \text{pre}(c_2) \\
  b_1 & \text{if } b_2 = ⊥ \text{ or } n \notin \text{pre}(c_2) \\
  b_2 & \text{if } b_1 = b_2 \\
  ⊤ & \text{otherwise}
\end{cases}
\]

where \( b_n = c_n(\text{pre}(n)) \)

- in contrast, the asymmetric join \( \sqcap \) defined componentwise:

\[
(\text{pre}(n)) \begin{cases}
  c_1(n) & \text{if } n \in \text{pre}(c_1) \\
  c_2(n) & \text{otherwise}
\end{cases}
\]
Smalltalk-Inheritance

- children's methods dominate parents' methods
- is the archetype for inheritance in mainstream languages like Java or C#
- inheriting smalltalk-style establishes a reference to the parent

**Definition (Smalltalk inheritance)**

Smalltalk inheritance is the binary operator $\triangleright$, defined by

$$c_1 \triangleright c_2 = \{\text{super} \mapsto \rightarrow \}
\sqcup \leftrightarrow (c_1 \sqcup \leftrightarrow c_2)$$

**Example:**

Door = \{canPass \mapsto \rightarrow \⊥, canOpen \mapsto \rightarrow \⊥\}
LockedDoor = \{canOpen \mapsto 0x4204711\} $\triangleright$ Door
= \{super $\mapsto$ Door\} $\sqcup$ (\{canOpen $\mapsto 0x4204711\} \leftrightarrow Door)
= \{super $\mapsto$ Door, canOpen $\mapsto 0x4204711$, canPass $\mapsto \rightarrow \⊥\}$

Beta-Inheritance

- the design goal is to provide security wrt. replacement of a method by a different method.
- methods in parents dominate methods in subclass
- the keyword `inner` explicitly delegates control to the subclass

**Definition (Beta inheritance)**

Beta inheritance is the binary operator $\triangleleft$, defined by

$$c_1 \triangleleft c_2 = \{\text{inner} \mapsto \rightarrow \}
\sqcup \leftrightarrow (c_2 \sqcup \leftrightarrow c_1)$$

**Example (equivalent syntax):**

```
class Person {
    String name = "Axel Simon";
    public String toString(){ return name+inner.toString();};
}
```
```
class Graduate extends Person {
    public extension String toString(){ return ", Ph.D."; }
}
```

**So what do we really want?**

**Adventure Game with Code Duplication**

- Door
- LockedDoor
- ShortDoor
- ShortLockedDoor

**Adventure Game with Mixins**

```java
class Door {
    boolean canOpen(Person p) { return true; };
    boolean canPass(Person p) { return p.size() < 210; };
}
mixin Locked {
    boolean canOpen(Person p){
        if (!p.hasItem(key)) return false; else return super.canOpen(p);
    }
}
mixin Short {
    boolean canPass(Person p){
        if (p.height()>1) return false; else return super.canPass(p);
    }
}
class ShortDoor = Short(Door);
class LockedDoor = Locked(Door);
mixin ShortLocked = Short o Locked;
class ShortLockedDoor = Short(Locked(Door));
class ShortLockedDoor2 = ShortLocked(Door);
```
Abstract model for Mixins

A Mixin is a unary second order type expression. In principle it is a curried version of the Smalltalk-style inheritance operator. In certain languages, programmers can create such mixin operators:

Definition (Mixin)

The mixin constructor \( \text{mixin} : C \mapsto (C \mapsto C) \) is a unary class function, creating a unary class operator, defined by:

\[
\text{mixin}(c) = \lambda x. c \circ x
\]

Note: Mixins can also be composed \( \circ \):

Example: Doors

\[
\begin{align*}
\text{Locked} &= \{\text{canOpen} \mapsto \{\text{super} \mapsto \text{Locked} \circ x\}\} \\
\text{Short} &= \{\text{canPass} \mapsto \{\text{super} \mapsto \text{Locked} \circ x\}\}
\end{align*}
\]

\[
\text{Composed} = \text{mixin(Short)} \circ \text{mixin(Locked)} = \lambda x. \text{Short} \circ (\text{Locked} \circ x)
\]

= \lambda x. \{\text{super} \mapsto (\text{Locked} \circ x)\} \cup \{\text{canOpen} \mapsto 0x1234, \text{canPass} \mapsto 0x4711\} \circ x
\]

Simulating Mixins in C++

```cpp
template <class Super>
class SyncRW : public Super {
public:
    virtual int read() {
        acquireLock();
        int result = Super::read();
        releaseLock();
        return result;
    };
    virtual void write(int n) {
        acquireLock();
        Super::write(n);
        releaseLock();
    };
};
```

Surely multiple inheritance is powerful enough to simulate mixins?
Simulating Mixins in C++

```cpp
template <class Super>
class LogOpenClose : public Super {
public: virtual void open()
Super::open();
log("opened");
};
virtual void close()
Super::close();
log("closed");
protected: virtual void log(char*s) { ... };
};
class MyDocument : public SyncRW<LogOpenClose<Document>> {};
```

True Mixins vs. C++ Mixins

**True Mixins**
- super natively supported
- Mixins can be seen as coding pattern

**C++ Mixins**
- Mixins reduced to templated superclasses
- Can be seen as coding pattern

Common properties of Mixins
- Linearization is necessary
- Exact sequence of Mixins is relevant

Ruby

```ruby
class Person
  attr_accessor :size
  def initialize
    @size = 160
  end
  def hasKey
    true
  end
end
class Door
  def canOpen (p)
    true
  end
  def canPass(person)
    person.size < 210
  end
end
module Short
  def canPass(p)
    p.size < 160 and super(p)
  end
end
module Locked
  def canOpen(p)
    p.hasKey() and super(p)
  end
end
class ShortLockedDoor < Door
  include Short
  include Locked
end
p = Person.new
d = ShortLockedDoor.new
puts d.canPass(p)
```

Is Inheritance the Ultimate Principle in Reusability?
Lack of Control

- Common base classes are shared or duplicated at class level
- super as ancestor reference vs. qualified specification
- No fine-grained specification of duplication or sharing

Inappropriate Hierarchies

- High up specified methods turn obsolete, but there is no statically safe way to remove them
- Liskov Substitution Principle!

Excerpt from the Java 8 API documentation for class Properties:

"Because Properties inherits from Hashtable, the put and putAll methods can be applied to a properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail..."

Misuse of Implementation Inheritance

Implementation Inheritance itself as a pattern for code reusage is often misused!
- All that is not explicitly prohibited will eventually be done!

The Idea Behind Traits

- A lot of the problems originate from the coupling of implementation and modelling
- Interfaces seem to be hierarchical
- Functionality seems to be modular

Central idea

Separate object creation from modelling hierarchies and composing functionality.

- Use interfaces to design hierarchical signature propagation
- Use traits as modules for assembling functionality
- Use classes as frames for entities, which can create objects

Theorem 1

Definition (Trait ∈ T)

A class t is without attributes is called trait.

The trait sum + : T × T ↦→ T is the componentwise least upper bound:

\[
(c_1 + c_2)(n) = b_1 \sqcup b_2 = \begin{cases} 
  b_1 & \text{if } b_1 = b_2 \sqcup n \notin \text{pre}(c_1) \\
  b_2 & \text{if } b_2 = b_1 \sqcup n \notin \text{pre}(c_2) \\
  \top & \text{if } b_1 = b_2 \\
  \bot & \text{otherwise}
\end{cases}
\]

with \( b_1 = c_1(n) \) and \( b_2 = c_2(n) \) for \( n \neq \bot \) and \( n \neq \top \).

Trait Expressions also comprise:

- exclusion \( - : T \times N \rightarrow T \):

\[
(t - a)(n) = \begin{cases} 
  \text{undef} & \text{if } a = n \\
  t(n) & \text{otherwise}
\end{cases}
\]

- aliasing \( \llbracket \cdot \rrbracket : T \times N \times N \rightarrow T \):

\[
t(a \rightarrow b)(n) = \begin{cases} 
  t(n) & \text{if } n \neq a \\
  t(b) & \text{if } n = a
\end{cases}
\]

trait \( t \) can be connected to classes \( c \) by the asymmetric join:

\[
\langle c | t \rangle(n) = \begin{cases} 
  c(n) & \text{if } n \in \text{pre}(c) \\
  t(n) & \text{otherwise}
\end{cases}
\]

Usually, this connection is reserved for the last composition level.
**Traits – Concepts**

**Trait composition principles**

- **Flat ordering** All traits have the same precedence under + → explicit disambiguation with aliasing and exclusion
- **Precedence** Under asymmetric join \(\sqcup\) class methods take precedence over trait methods
- **Flattening** After asymmetric join \(\sqcup\): Non-overridden trait methods have the same semantics as class methods

**Can we augment classical languages by traits?**

```csharp
public class Person{
    public int size = 160;
    public bool hasKey() { return true; }
}
public interface Short {}
public ... static void Main() {
    ShortLockedDoor d = new ShortLockedDoor();
    Console.WriteLine(d.canOpen(new Person()));
}
```

**Extension Methods (C#)**

**Central Idea:**
Uncouple method definitions from class bodies.

**Purpose:**
- retrospectively add methods to complex types
- especially provide definitions of interface methods
  -- poor man’s multiple inheritance!

**Syntax:**
- Declare a static class with definitions of static methods
- Explicitly declare first parameter as receiver with modifier this
- Import the carrier class into scope (if needed)
- Call extension method in infix form with emphasis on the receiver

**Extension Methods as Traits**

**Extension Methods**
- transparently extend arbitrary types externally
- provide quick relief for plagued programmers

**...but not traits**
- Interface declarations empty, thus kind of purposeless
- Flattening not implemented
- Static scope only

Static scope of extension methods causes unexpected errors:

```csharp
public interface Locked {
    default boolean canOpen(Person p) { return p.hasKey(); }
}
public interface Short {
    default boolean canPass(Person p) { return p.size<160; }
}
public class ShortLockedDoor implements Short, Locked, Door {
    public boolean canPass(Person p) {
        return p.size<160;
    }
    public boolean canOpen(Person p) {
        return p.hasKey();
    }
}
```

**Virtual Extension Methods (Java 8)**

Java 8 advances one step further:

```java
interface Door {
    boolean canOpen(Person ... when composed
}
```

**Implementation**
... consists in adding an interface phase to invokevirtual's name resolution

**Precedence**
Still, default methods do not override methods from abstract classes when composed
Traits as General Composition Mechanism

Central Idea
Separate class generation from hierarchy specification and functional modeling
- model hierarchical relations with interfaces
- compose functionality with traits
- adapt functionality to interfaces and add state via glue code in classes

Simplified multiple Inheritance without adverse effects

So let's do the language with real traits?!

Squeak
Smalltalk
Squeak is a smalltalk implementation, extended with a system for traits.

Syntax:
name: param1 and: param2
declares method name with param1 and param2
| ident1 ident2 |
declares Variables ident1 and ident2
ident := expr
assignment
object name: content
sends message name with content to object (= call:
object name(content))
.
line terminator
^ expr
return statement

Traits in Squeak
Trait named: #TRStream uses: TPositionableStream
on: aCollection
self collection: aCollection.
self setToStart.
next
| self atEnd |
ifTrue: [nil]
ifFalse: [self collection at: self nextPosition].
Trait named: #TSynch uses: {}
acquireLock
self semaphore wait.
releaseLock
self semaphore signal.
Trait named: #TSyncRStream uses: TSynch+(TRStream@(#readNext -> #next))
next
| read |
self acquireLock.
read := self readNext.
self releaseLock.
" read.

Disambiguation

Traits vs. Mixins vs. Class-Inheritance
All different kinds of type expressions:
- Definition of curried second order type operators + Linearization
- Finegrained flat-ordered composition of modules
- Definition of (local) partial order on precedence of types wrt. MRO
- Combination of principles

Explicitly: Traits differ from Mixins
- Traits are applied to a class in parallel, Mixins sequentially
- Trait composition is unordered, avoiding linearization effects
- Traits do not contain attributes, avoiding state conflicts
- With traits, glue code is concentrated in single classes

Lessons learned
Mixins
- Mixins as low-effort alternative to multiple inheritance
- Mixins lift type expressions to second order type expressions

Traits
- Implementation Inheritance based approaches leave room for improvement in modularity in real world situations
- Traits offer fine-grained control of composition of functionality
- Native trait languages offer separation of composition of functionality from specification of interfaces
“Why bother with modelling types for my quick hack?”

“Let’s go back to basic concepts – *Lua*”

Motivation – Polemic

Bothersome features

- Specifying types for singletons
- Getting generic types right inspite of co- and contra-variance
- Subjugate language-imposed inheritance to (mostly) avoid redundancy

Prototype based programming

- Start by creating examples
- Only very basic concepts
- Introduce complexity only by need
- Shape language features yourself!

Outline

Prototype based programming

- Basic language features
- Structured data
- Code reusage
- Imitating Object Orientation

Programming Languages

Prototypes

Dr. Michael Petter
Winter 2018/19

Further reading...

- Gilad Bracha and William Cook.  
  Mixin-based inheritance. 
  European conference on object-oriented programming on Object-oriented programming systems, languages, and applications (OOPSLA/ECOOP), 1990.

- James R. Rohn.  
  Ruby 2.1.5 core reference, December 2014.  

- Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schäli, Roel Wuyts, and Andrew P. Black.  
  Traits: A mechanism for fine-grained reuse. 
  ACM Transactions on Programming Languages and Systems (TOPLAS), 2006.

- Matthew Flatt, Oregon Kimmell, and Nathanael Schäli.  
  Classes and mixins.  
  Principles of Programming Languages (POPL), 1998.

- Brian Goetz.  
  Interface evolution via virtual extension methods.  
  JSR 335: Lambda Expressions for the Java Programming Language, 2011.

- Anders Hejlsberg, Scott Wiltamuth, and Peter Golde.  
  C# Language Specification.  

- Nathanael Schäli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.  
  Traits: Composable units of behaviour.  
Basic Language Features

- Chunks being sequences of statements.
- Global variables implicitly defined

```plaintext
s = 0;
i = 1 -- Single line comment
p = i+s p=42 -- Multiline comment
s = l
```

Basic Types and Values

- Dynamical types – no type definitions
- Each value carries its type
- `type()` returns a string representation of a value's type

```plaintext
a = true
 type(a) -- boolean
 type("42") -- number
 type("Petter "..1) -- string
 type(type) -- function
 type(nil) -- nil
 type(["html"><body>pretty long string</body>
 <html>"]) -- string
```

Functions for Code

- First class citizens

```plaintext
function prettyprint(title, name, age)
 return title.. " ..name..", born in "..(2018-age)
end
a = prettyprint
a("Dr.","Petter",42)
```

Introducing Structure

- only one complex data type
- indexing via arbitrary values except `nil` (→ Runtime Error)
- arbitrary large and dynamically growing/shrinking

```plaintext
a = {} -- create empty table
k = 42
a[k] = 3.14159 -- entry 3.14159 at key 42
a["k"] = k -- entry 42 at key "k"
a[k] = nil -- deleted entry at key 42
print(a.k) -- syntactic sugar for a["k"]
```

Table Lifecycle

- created from scratch
- modification is persistent
- assignment with reference-semantics
- garbage collection

```plaintext
a = {} -- create empty table
a.k = 42
b = a -- b refers to same as a
b["n"] = "n" -- entry "n" at key "n"
print(a.k) -- yields "k"
a = nil
print(b.k) -- still "k"
b = nil
print(b.k) -- nil now
```

“So far nothing special – let’s compose types”
**Table Behaviour**

**Metatables**

- are ordinary tables, used as collections of special functions
- Naming conventions for special functions
- Connect to a table via `setmetatable`, retrieve via `getmetatable`

*Changes behaviour of tables*

- Overload operators like `add`, `mul`, `sub`, `div`, `pow`, `concat`, `unm`
- Overload comparators like `eq`, `lt`, `le`, `mul`, `div`

**Delegation 2**

⇝ Conveniently, `_index` does not need to be a function

```lua
meta = {} -- create as plain empty table
function meta.__tostring(person)
    return person.prefix .. " " .. person.name
end
function meta.__index(tbl, key)
    return tbl.prototype[key]
end
print(person) -- print "Dr. Petter"
```

**Delegation 3**

- `_newindex` handles unresolved updates
- frequently used to implement protection of objects

```lua
meta = {} -- create as plain empty table
function meta.__tostring(person)
return person.prefix .. " " .. person.name
end
function meta.__index(tbl, key)
return tbl.prototype[key]
end
person.title = "Dr." -- try to give him Dr.
```

**Delegation**

- `setmetatable` function is __connect__ to a table via `setmetatable`, retrieve via `getmetatable`

```lua
person = { name="Petter" } -- create Michael
setmetatable(person,meta) -- install metatable
print(person) -- print "Dr. Petter"
```

**Delegation**

- Connect to a table via `setmetatable`, retrieve via `getmetatable`

```lua
person = { prefix="Dr.", name="Petter" } -- create Michael
setmetatable(person,meta) -- install metatable
print(person) -- print "Dr. Petter"
```

**Delegation**

- Connect to a table via `setmetatable`, retrieve via `getmetatable`

```lua
meta = {} -- create as plain empty table
function meta.__newindex(tbl, key, val)
    return tbl.prototype[key]
end
function meta.__tostring(tbl)
    return tbl.name
end
meta.__index = job -- delegate to job
meta = {} -- create as plain empty table
function meta.__tostring(person)
    return person.prefix .. " " .. person.name
end
function meta.__newindex(tbl, key, val)
    return tbl.prototype[key]
end
function meta.__index(tbl, key)
    return tbl.prototype[key]
end
person = { prefix="Dr.", name="Petter" } -- create Michael
setmetatable(person,meta) -- install metatable
print(person) -- print "Dr. Petter"
```
Object Oriented Programming

- so far no concept for multiple objects

```lua
Account = { balance=0 }
function Account.withdraw (val)
    acc.balance=acc.balance-val
end
function Account.tostring(acc)
    return "Balance is ".acc.balance
end
setmetatable(Account,Account)
Account.withdraw(10)
print(Account)
```

Introducing Identity

- Concept of an object's own identity via parameter
- Programming aware of multiple instances
- Share code between instances

```lua
function Account.withdraw (acc, val)
    acc.balance=acc.balance-val
end
function Account.tostring(acc)
    return "Balance is ".acc.balance
end
```

Introducing Identity

- Delegation leads to chain-like inheritance

```lua
function createClass (parent1,parent2)
local c = {} ...
return o or {}
setmetatable(o, c) -- c is also metatable
return o
end
return c -- finally return c
end
```

Introducing Identity

- Introducing "Classes"
- Particular tables used like classes
- self table for accessing object-relative attributes
- connection via creator function new (like a constructor)

```lua
function Account:new(template)
    self.balance=self.balance-val
end
function Account:tostring()
    return "Balance is ".self.balance
end
function Account:new(template)
    template = template or (template=0) -- initialize
setmetatable(template, {__index=self})-- delegate to Account
getmetatable(template).tostring = Account.tostring
return template
end
gio = Account:new({balance=10}) -- create instance
print(gio)
```

Introducing Identity

- Inheriting Functionality
- Differential description possible in child class style
- Easily creating particular singletons

```lua
LimitedAccount = { }
setmetatable(LimitedAccount, {__index=Account})
function LimitedAccount:new()
    instance = { balance=0,limit=100 }
    setmetatable(instance,{__index=self})
end
function LimitedAccount:withdraw(val)
    if (self.balance+self.limit < val) then
        error("Limit exceeded")
    end
    self.balance=self.balance-val
end
setmetatable(LimitedAccount:withdraw,Account)
print(specialgiro)
```

Multiple Inheritance

- Delegation leads to chain-like inheritance

```lua
function createClass (parent1,parent2)
local c = {}
setmetatable(c, {__index=Account})
function Account:new()
    instance = { balance=0,limit=100 }
    setmetatable(instance,{__index=self})
end
function Account:withdraw(val)
    if (self.balance+self.limit < val) then
        error("Limit exceeded")
    end
    self.balance=self.balance-val
end
function Account.tostring(acc)
    return "Balance is ".acc.balance
end
```

Introducing Identity

- Concept of an object's own identity via parameter
- Programming aware of multiple instances
- Share code between instances

```lua
function Account.withdraw (acc, val)
    acc.balance=acc.balance-val
end
function Account.tostring(acc)
    return "Balance is ".acc.balance
end
```
Multiple Inheritance

Doctor = { postfix="Dr. "}
Researcher = { prefix=" ,Ph.D."}
ResearchingDoctor = createClass(Doctor,Researcher)
axel = ResearchingDoctor:new( { name="Michael Petter" } )
print(axel.prefix..axel.name..axel.postfix)

⇝
The special case of dual-inheritance can be extended to comprise multiple inheritance

Implementation of Lua

typedef struct {
  int type_id;
  Value v;
} TObject;
typedef union {
  void *p;
  int b;
  lua_number n;
  GCObject *gc;
} Value;

Datatypes are simple values (Type+union of different flavours)
Tables at low-level fork into Hashmaps with pairs and an integer-indexed array part

Further Topics in Lua

- Coroutines
- Closures
- Bytecode & Lua-VM

Lessons Learned

1. Abandoning fixed inheritance yields ease/speed in development
2. Also leads to horrible runtime errors
3. Object-orientation and multiple-inheritance as special cases of delegation
4. Minimal featureset eases implementation of compiler/interpreter
5. Room for static analyses to find bugs ahead of time

Further Reading...

Roberto Ierusalimschy.
ISBN 859037985X.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho.
Lua—an extensible extension language.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes.
The implementation of lua 5.0.
“Is modularity the key principle to organizing software?”

Learning outcomes

1. AOP Motivation and Weaving basics
2. Bundling aspects with static crosscutting
3. Join points, Pointcuts and Advice
4. Composing Pointcut Designators
5. Implementation of Advices and Pointcuts

Motivation

- Traditional modules directly correspond to code blocks
- Aspects can be thought of separately but are smeared over modules
- Tangling of aspects
- Focus on Aspects of Concern

Aspect Oriented Programming

- Express a system’s aspects of concerns cross-cutting modules
- Automatically combine separate Aspects with a Weaver into a program

System Decomposition in Aspects

Example concerns:
- Security
- Logging
- Error Handling
- Validation
- Profiling
System Decomposition in Aspects

Example concerns:
- Security
- Logging
- Error Handling
- Validation
- Profiling

⇝ AspectJ

Aspect Oriented Programming

Introduction

Static Crosscutting

inter-type declaration
class Expr {}
class Const extends Expr {
    public int val;
    public Const(int val) {
        this.val=val;
    }
}
class Add extends Expr {
    public Expr l,r;
    public Add(Expr l, Expr r) {
        this.l=l;this.r=r;
    }
}

aspect ExprEval {
    abstract int Expr.eval();
    int Const.eval(){ return val; };
    int Add.eval() { return l.eval() + r.eval(); }
}

⇒ equivalent code
abstract class Expr {
    abstract int eval();
}
class Const extends Expr {
    public int val;
    public int eval(){ return val; };
    public Const(int val) {
        this.val=val;
    }
}
class Add extends Expr {
    public Expr l,r;
    public int eval() { return l.eval() + r.eval(); }
    public Add(Expr l, Expr r) {
        this.l=l;this.r=r;
    }
}

Adding External Definitions

Dynamic Crosscutting

Join Points

Well-defined points in the control flow of a program
- method/constr. call: executing the actual method-call statement
- method/constr. execution: the individual method is executed
- field get: a field is read
- field set: a field is set
- exception handler execution: an exception handler is invoked
- class initialization: static initializers are run
- object initialization: dynamic initializers are run

Pointcuts and Designators

Definition (Pointcut)
A pointcut is a set of join points and optionally some of the runtime values when program execution reaches a referred join point.

Pointcut designators can be defined and named by the programmer:
(userdef) ::= "pointcut" (def) ("(" idlist "\)\)\) ::= (expr) |
(idlist) ::= (id) ("," (id)) |
(expr) ::= "!" (expr) |
    "(" (expr) "\)" |
    "(" (expr) "\)" |
    "(" (expr) "\)"

Example:
pointcut dfs(): execution (void Tree.dfs()) || execution (void Leaf.dfs());
Advice

... are method-like constructs, used to define additional behaviour at joinpoints:

- before(formal)
- after(formal)
- after(formal) returning (formal)
- after(formal) throwing (formal)

For example:

```
aspect Doubler {
    before(): call(int C.foo(int)) {
        System.out.println("About to call foo");
    }
}
```

Aspect Oriented Programming Dynamic Crosscutting

Binding Pointcut Parameters in Advices

Certain pointcut primitives add dependencies on the context:

- args(arglist)

This binds identifiers to parameter values for use in advices.

```
aspect Doubler {
    before(int i): call(int C.foo(int)) && args(i) {
        int newi = proceed(i*2);
        return newi/2;
    }
}
```

Unusual treatment is necessary for

- type around(formal)

Here, we need to pinpoint, where the advice is wrapped around the join point – this is achieved via `proceed()`.

```
aspect Doubler {
    int around(int i): call(int C.foo(Object, int)) && args(i) {
        int newi = proceed(i*2);
        return newi/2;
    }
}
```

Method Related Designators

Matches call/execution join points at which the method or constructor called matches the given signature. The syntax of a method/constructor signature is:

```
ResultTypeName RecvrTypeName.meth
id(ParamTypeName, ...)
NewObjectTypeName.new(ParamTypeName, ...)
```

```
class MyClass{
    public String toString() {
        return "silly me ";
    }
    public static void main(String[] args){
        MyClass c = new MyClass();
        System.out.println(c + c.toString());
    }
}
```

```
aspect CallAspect {
    pointcut calltostring() : call(String MyClass.toString());
    pointcut exectostring() : execution(String MyClass.toString());
    before(): calltostring() || exectostring() {
        System.out.println("advice!");
    }
}
```

Method Related Designators
class MyClass{
    public String toString() {
        return "silly me ";
    }
    public static void main(String[] args){
        MyClass c = new MyClass();
        System.out.println(c + c.toString());
    }
}
advice!
advice!
advice!
silly me silly me

Field Related Designators
- get(fieldqualifier)
- set(fieldqualifier)
Matches field get/set join points at which the field accessed matches the signature. The syntax of a field qualifier is:
FieldTypeName ObjectTypeName.field

- target(typeorid)
- within(typepattern)
- withincode(methodpattern)
Matches join points of any kind which
are refering to the receiver of type typeorid
is contained in the class body of type typepattern
is contained within the method defined by methodpattern

Flow and State Based
- cflow(arbitrary_pointcut)
Matches join points of any kind that occur strictly between entry and exit of each join point matched by arbitrary_pointcut.
- if(boolean_expression)
Picks join points based on a dynamic property:

Which advice is served first?
Advices are defined in different aspects
- If statement declare precedence:A, B: exists, then advice in aspect A has precedence over advice in aspect B for the same join point.
- Otherwise, if aspect A is a subaspect of aspect B, then advice defined in A has precedence over advice defined in B.
- Otherwise, (i.e. if two pieces of advice are defined in two different aspects), it is undefined which one has precedence.

Advices are defined in the same aspect
- If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, the one that appears earlier in the aspect has precedence over the one that appears later.
Implementation

Aspect Weaving:
- Pre-processor
- During compilation
- Post-compile-processor
- During Runtime in the Virtual Machine
- A combination of the above methods

Around/Proceed – via Procedures
✓ inlining advices in main – all of it in JVM, disassembled to equivalent:

```java
public static void main(String[] args){
    new C().foo(42);
}
```

Escaping the Calling Context

However, instead of being used for a direct call, proceed() and its parameters may escape the calling context.

Woven JVM Code

```java
aspect MyAspect {
    pointcut settingconst():
        set (int Const.val);
    before () : settingconst() {
        System.out.println("setter");
    }
}
```

Poincut Parameters and Around/Proceed

Around clauses often refer to parameters and proceed() – sometimes across different contexts!

```java
class C {
    int foo(int i) { return 42+i; }
}
aspect Doubler {
    int around(int i): call(int *.foo(Object, int)) && args(i) {
        int newi = proceed(i*2);
        return newi/2;
    }
}
```

Woven JVM Code

```java
Expr one = new Const(1);
one.val = 42;
```

```java
public static void main(String[] args){
    new C().foo(42);
}
```
Pointcut parameters and Scope

- `proceed()` might not even be in the same scope as the original method!
- even worse, the scope of the exposed parameters might have expired!

```java
class C {
    int foo(int i) { return i*2+1; }
    public static void main(String[] str){ new C().foo(42); }
}

aspect Doubler {
    Executor executor;
    Future<Integer> f;
    int around(int i): call(void f(int)) && args(i) {
        Callable<Integer> c = () -> proceed(i*2)/2;
        f = executor.submit(c);
        return i/2;
    }
    public int getCachedValue() throws Exception {
        return f.get();
    }
}
```

Shadow Classes and Closures

- ✓ creates a shadow, carrying the advice
- ✓ creates a closure, carrying the context/parameters

```java
// aspectj patched code
public static void main(String[] str){
    int itemp = 42;
    Doubler shadow = Doubler.aspectOf();
    Object[] params = new Object[]{ new C(), Conversions.intObject(itemp) };
    f = executor.submit(c);
    return Conversions.intValue(c.run(params));
}
```

Property Based Crosscutting

```java
public static void main(String[] str){
    new C().foo(42); }
```

Shadow Classes and Closures

- ✓ creates a shadow, carrying the advice
- ✓ creates a closure, carrying the context/parameters

Implementation – Summary

Translation scheme implications:
- before/after Advice ... ranges from inlined code to distribution into several methods and closures
- Joinpoints ... in the original program that have advices may get explicitly dispatching wrappers
- Dynamic dispatching ... can require a runtime test to correctly interpret certain joinpoint designators
- Flow sensitive pointcuts ... runtime penalty for the naive implementation, optimized version still costly

Aspect Orientation

- Pro
  - Un-tangling of concerns
  - Late extension across boundaries of hierarchies
  - Aspects provide another level of abstraction
- Contra
  - Weaving generates runtime overhead
  - nontransparent control flow and interactions between aspects
  - Debugging and Development needs IDE Support

Even more optimizations in practice
- state-sharing, counters, static analysis
Further reading...


Motivation

- Aspect Oriented Programming establishes programmatic refinement of program code
- How about establishing support for program refinement in the language concept itself?
- Treat program \textit{code as data} \rightarrow \text{Metaprogramming}
In Compiler Construction, there are a lot of codegeneration tools, that compile DSLs to target source code. Common examples are lex and bison.

Example: lex:
lex generates a table lookup-based implementation of a finite automaton corresponding to the specified disjunction of regular expressions.

```
#include <stdio.h>

%{
   #include <stdio.h>
   
   /* Lexical Patterns */
   [0-9]+ { printf("%d\n", yytext); }
   }\n
int main(void) {
    yylex();
    return 0;
}%
```

→generates 1.7k lines of C

String Rewriting Systems
A Text Rewriting System provides a set of grammar-like rules (→ Macros) which are meant to be applied to the target text.

Example: C Preprocessor (CPP)

```
#define ATOMIC 
  acquire(&globallock);
/* user code */
release(&globallock);

ATOMIC {
  i--; 
  i++;
}
```

Prepend code to usercode
```
if (1) {
  /* prepended code */
  goto body;
} else 
  body:
{ /* block following the expanded macro */ }
```

Example application: Language constructs [3]:
```
ATOMIC {
  i--; 
  i++;
} 
```

How can we bind the block, following the ATOMIC to the usercode fragment? Particularly in a situation like this?
```
if (i>0) 
  ATOMIC {
    i--; 
    i++;
}
```
# Compiletime-Codegeneration

All in one

if (1) {
/* prepended code */
goto body;
} else
while (1)
if (1) {
/* appended code */
break;
}
else body:
{ /* block following the expanded macro */ }

---

## Homoiconic Metaprogramming

Homoiconicity

In a homoiconic language, the primary representation of programs is also a data structure in a primitive type of the language itself.

- Metaclasses and Metaobject Protocol
- *(Hygienic)* Macros

---

## Reflective Metaprogramming

Reflection

Type introspection

A language with *Type introspection* enables to examine the type of an object at runtime.

Example: Java `instanceof`

```java
class Natural {
    int value;
}

public boolean equals(Object o) {
    if (!(o instanceof Natural)) return false;
    return ((Natural)o).value == this.value;
}
```
Reflective Metaprogramming

Metaclasses (→ code is data)

Example: Java Reflection / Metaclass java.lang.Class

```java
static void fun(String param) {
    Object incognito = Class.forName(param).newInstance();
    Class meta = incognito.getClass(); // obtain Metaobject
    Field[] fields = meta.getDeclaredFields();
    for (Field f : fields) {
        Class t = f.getType();
        Object v = f.get(o);
        if (t == boolean.class && Boolean.FALSE.equals(v))
            // found default value
        else if (t.isPrimitive() && ((Number) v).doubleValue() == 0)
            // found default value
        else if (!t.isPrimitive() && v == null)
            // found default value
    }
}
```

Metaobject Protocol

Metaobject Protocol (MOP [1])

Example: Lisp's CLOS metaobject protocol

... offers an interface to manipulate the underlying implementation of CLOS to adapt the system to the programmer's liking in aspects of

- creation of classes and objects
- creation of new properties and methods
- causing inheritance relations between classes
- creation generic method definitions
- creation of method implementations
- creation of specializers (→ overwriting, multimethods)
- configuration of standard method combination (→ before,after,around, call-next-method)
- simple or custom method combinators (→ +,append,max,...)
- addition of documentation

Hygienic Macros

Clojure! [2]

Clojure programs are represented after parsing in form of symbolic expressions (S-Expressions), consisting of nested trees:

**S-Expressions**

S-Expressions are either

- an atom
- an expression of the form (x y) with x, y being S-Expressions

**Remark:** Established shortcut notation for lists:

\[(x_1, x_2, x_3) \equiv (x_1, (x_2, (x_3, \_)))\]
Macros are configurable syntax/parse tree transformations.

Example: While loop:

```clojure
(macroexpand '(while a b))
;=> (loop* [] (clojure.core/when a b (recur)))
```

Quoting

Macros and functions are directly interpreted, if not quoted via

```clojure
(quote keyword) ; or equivalently: 'keyword
; => keyword
```

Macro Hygiene

Shadowing of variables may be an issue in macros, and can be avoided by generated symbols:

```clojure
(defvariable 42)
(macro '(astufftodo) '(let [variable 4711] "@stufftodo))
;=> can't let qualified name: variable
```

```clojure
(defvariable 42)
(macro '(astufftodo) '(let [variable# 4711] "@stufftodo))
```

Symbol generation to avoid namespace collisions!

Macros vs. Functions

- Macros as static AST Transformations, vs. Functions as runtime control flow manipulations
- Macros replicate parameter forms, vs. Functions evaluate parameters once
- Macro parameters are uninterpreted, not necessarily valid expressions, vs. Functions parameters need to be valid expressions

Macros can be written by the programmer in form of S-Expressions:

```clojure
(defmacro fac1 [n]
(if (= n 0)
1
(* n (fac2 (- n 1)))))
```

```clojure
(defn fac2 [n]
(if (= n 0)
1
(* n (fac2 (- n 1)))))
```

(fac2 4)
; => 24