Cache Consistency (I)

Motivation: consider the following code executed in two threads.

```
// Thread A:
void foo(void) {
        a = 1;
        b = 1;
}
// Thread B:
void bar(void) {
        while (b == 0) {};
        assert (a == 1);
}
```

Intuitively, the assertion should never fail. In practice, it might fail, given enough iterations, due to *cache consistency* issues! Therefore we need to define a memory model.

Memory Models

Problem: Memory interactions on the hardware level behave differently if multiple concurrent thread with shared data are involved, including deferred communication of updates.

Memory models trade off freedom of implementation against guarantees about race-related properties.

Strict Consistency

Strict consistency is the following "idealistic" model:

> **Strict Consistency**
> Independently of which process reads or writes, the value from the *most recent write* to a location is observable by reads *immediately after the write occurs*

Very nice, but too strong to be realistic:

- would need a concept of absolute global time
- physically not possible

Happened-Before Relations

"Abandoning absolute time": just require *some* pairs of events to happen in a certain order.

- A process P is seen as a series of events $p_1, p_2, \ldots$
- Event p_i *happened before* p_{i+1}
- If p_i in P sends a message to Q, there is an event q_j that receives this message, and p_i *happened before* q_j

> **Happened-Before Relation**
> p *happened before* q, i.e. $p \to q$, according to the above rules.
> The relation $\to$ is transitive, and furthermore irreflexive, asymmetric, and partial. Therefore it is a *strict partial order*.

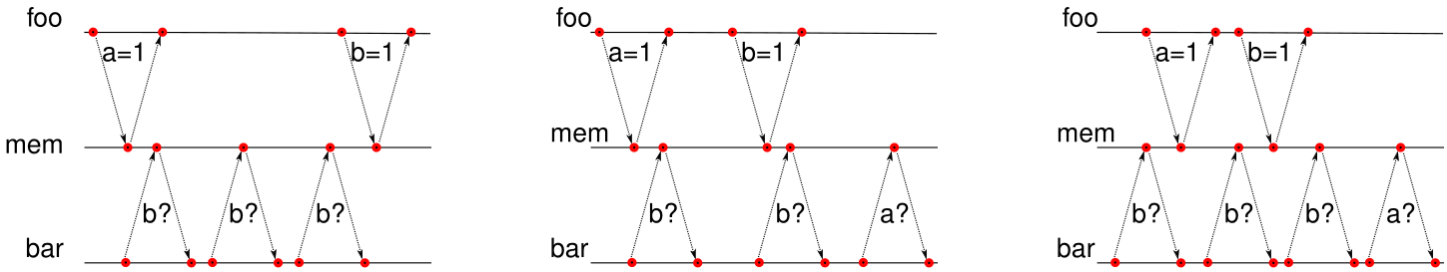
> **Concurrent Events**
> Two events p, q are *concurrent* if $p \not\to q$ and $q \not\to p$, i.e. they are $\to$-incomparable.

Example: Happened-Before

Consider the example from above: Require that

- Thread A stores to a before it stores to b
- Thread B sees b=1 before execute `assert`

But still, several outcomes are possible:



Event Ordering

A *logical clock* assigns a *logical global time stamp* $C(p)$ to each event p.

C satisfies the *clock condition* if for any p, q, $p \to q$, $C(p) < C(q)$.

For a distributed system, this means that

- if p_i, p_j are events of P, $p_i \to p_j$, then $C(p_i) < C(p_j)$
- if p is sending and q receiving of a message, then $C(p) < C(q)$

If C satisfies the clock condition and has unique timestamps, it induces a total order that embeds the strict partial order $\to$. The set of such C corresponds to the set of sequential executions that one can observe from the system.

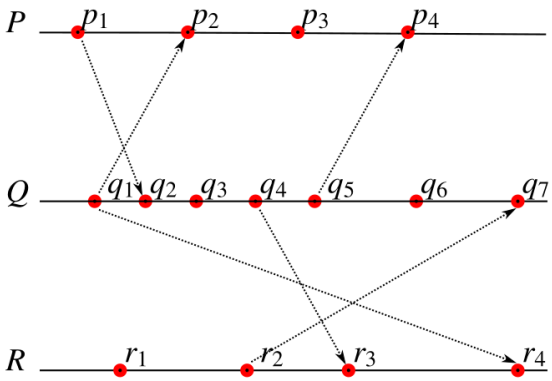
Models for Memory Coherence

Weakening the model: introduce *caches*.

- each process has a cache attached to it, that is quicker to access than main memory and is thought to store data that is accessed often

Each cache line is modeled as one process. (It seems like each variable is drawn as one line). QUESTION: Is that what is meant?

Obviously, a naive implementation of caches creates *races*, and therefore incoherence.



We see the following operations:

- $St[a]$ stores a value to a (from the process registers to the cache, apparently).
- $Ld[a]$ loads a value to a
- $Op[a]$ represents any operation on a

To denote the accessing CPU, use an index i: i.e. $St_i[a], Ld_i[a], Op_i[a]$.

Cache Coherence

Introduce the notion of a *memory order* $\sqsubseteq$: This orders the memory accesses for each adress a. Furthermore, the *program order* $\leq$ is specified by the control flow of the program for each CPU.

Cache Coherence

1. Operations in program order $\leq$ are embedded in $\sqsubseteq$:

$$Op_i[a] \leq Op_i[a]' \implies Op_i[a] \sqsubseteq Op_i[a]'$$

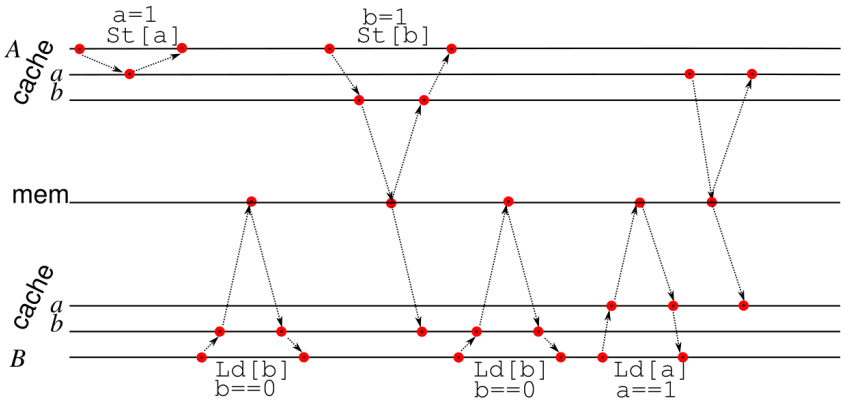
2. For each address a, $\sqsubseteq$ is a total order on all accesses to a
3. A load's value is determined by the latest store wrt. $\leq$ and $\sqsubseteq$:

$$val(Ld_i[a]) = val(St_j[a]), \text{ where } St_j[a] = \max_{\sqsubseteq}\{St_k[a] \mid St_k[a] \sqsubseteq Ld_i[a]\}$$

QUESTION: Doesn't $\leq$ depend on the actual CPU? Wouldn't $\leq_i$ be a more appropriate notation? $\Longrightarrow$ It appears as if $\leq = \bigcup_i (\leq_i)$.

Example: Coherent Cache

Consider this realization of the above example:



- A's cache write the value of b to memory before it writes the value of a
- B first sees `b == 1`, then `a == 0`, and the assertion fails!

However, the example *is* cache-coherent

- for memory location a, the read by B comes $\sqsubseteq$-before the write by A
- for memory location b, $\sqsubseteq$ first has a read by B, then a store by A, then a read by B
- $\leq$-embedding only needs to hold for pairs of accesses to the *same address*

$\Longrightarrow$ Cache Coherence is too weak!

Sequential Consistency

Sequential Consistency (Lamport, 1978): The result of any execution is the same as if the memory operations where executed in some sequential order, in which the operations of each individual processor appear in the right order.

To show a system is not sequentially consistent, find two executions (total orders $\sqsubseteq$) which both embed $\leq$, but which give different results.

Formally:

> **Sequential Consistency**
>
> 1. Memory operations in $\leq$-order are embedded in the total order $\sqsubseteq$:
>
> $$Op_i[a] \leq Op_i[b]' \implies Op_i[a] \sqsubseteq Op_i[b]'$$
>
> 2. A load's value is determined by the latest store wrt. $\sqsubseteq$:
>
> $$val(Ld_i[a]) = val(St_j[a]), \text{ where } St_j[a] = \max_{\sqsubseteq}\{St_k[a] \mid St_k[a] \sqsubseteq Ld_i[a]\}$$

Crucial difference to the Cache Coherence definition: The embedding covers all memory locations at once, instead of each one on its own!

To achieve sequential consistency, caches and memory need to collaborate in a *Cache Coherence Protocol*.

Benefits of Sequential Consistency

Sequential consistency covers *all interleavings* that are due to timing variations.

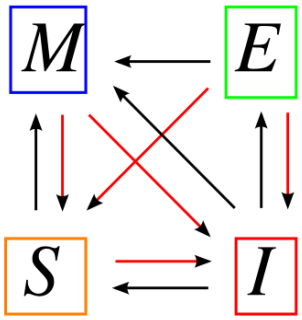
For simple hardware architectures, it is a realistic model: In particular, suitable for concurrent processors that acquire *exclusive memory access*. If processors use caches, they can still be made to maintain sequential consistency.

However for more elaborate hardware (with out-of-order stores): Complex cacheline management optimizations determine what other processors see. In this context, expecting sequential consistency is not realistic.

MESI Cache Coherence Protocol

The MESI protocol is a distributed system protocol to allow processors to keep track of the latest copy of a value.

Each cache line is attached a state machine:



Each cache line is in one of the states M, E, S, I:

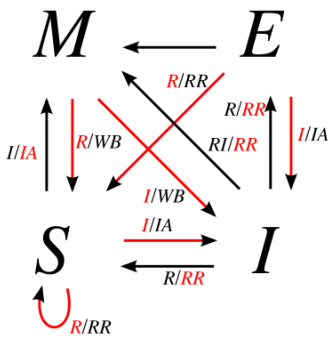
- I: it is *invalid* and is ready for re-use
- S: other caches have an identical copy of this cache line, it is *shared*
- E: the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches
- M: the content is exclusive to this cache and has furthermore been *modified*

- **I**nvalid: cache line is ready for re-use
- **S**hared: other caches have an *identical copy* of this cache line
- **E**xclusive: the content is in this cache only; can be overwritten without caring about other caches
- **M**odified: the content is exclusive *and* has been modified.

To keep the state of cache lines consistent, *messages* are being sent over a message bus.

(slightly simplified) messages of MESI:

- R (read = load from an address) and RR (read response = data for requested address)
- I (invalidate = ask others to evict a cache line) and IA (invalidate ack. = indicates that a line has been evicted)
- RI: Read + Invalidate ("read with intend to modify"), asks for the contents of a cache line *and* asks to invalidate it afterwards
- WB: Writeback, special form of RR or IA. When in state **M**, an R is received, WB is issued which notifies main memory about the modifications. The same happens if an **I** is received.



(black = sent; red = received)

Some interesting cases:

- if an **I** cache line requests data from memory, it sends out R, receives RR, and goes in the **E** state
- If an **E** or **S** cache receives an I message, it sends back IA and transitions to **I**
- If an **M** receives R, it transitions to **S**, issuing a WB
- If an **M** receives I, it transitions to **I**, issuing a WB
- **E** transitions into **M** when the CPU writes to a **E** cacheline; no messages need to be issued
- **S** transitions into **M** when an *atomic read-modify-write* is executed on a read-only item; then an I must be issued

An interesting detail: If R messages are sent out, this possibly is responded to quickly by the other CPUs which have the value in their cache; otherwise, it needs to be loaded from RAM, and the response takes longer.

Week 3 - Cache-Consistency II

Example: MESI on two threads

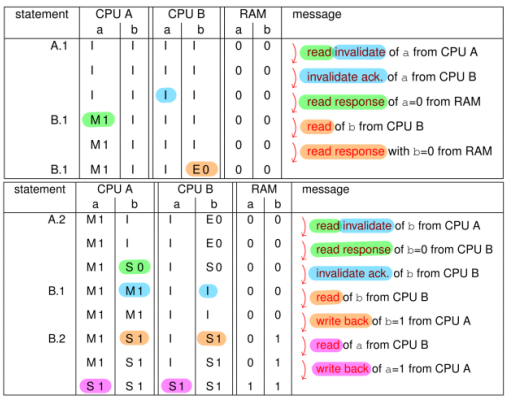
(Assumption: Cache lines larger than the variables itself - this requires us to do a bit of extra work)

Notation: Mx, Ex, Sx for M/E/S with value x, respectively, and I.

Code as before:

```
// Thread A:
void foo(void) {
        a = 1;                          // A.1
        b = 1;                  // A.2
}
// Thread B:
void bar(void) {
        while (b == 0) {}; // B.1
        assert (a == 1);   // B.2
}
```

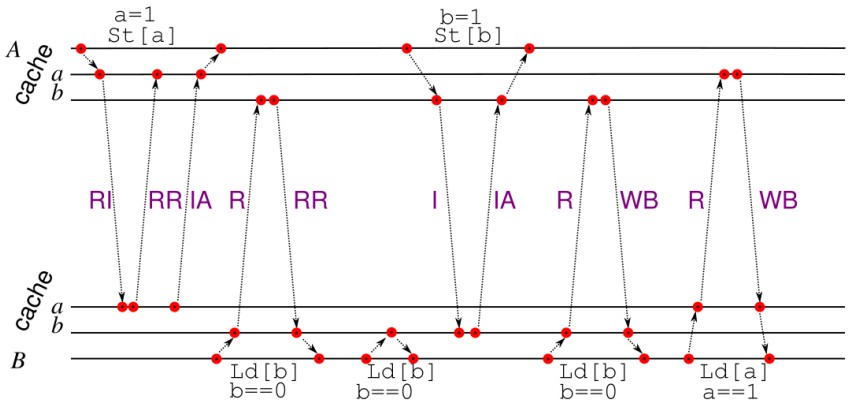
Mesi execution:



QUESTION: Why read (i.e. wait for RAM) when overwriting it anyway? (In step A.1)

Example: Happened-Before Diagram on two threads

For the same execution as above:

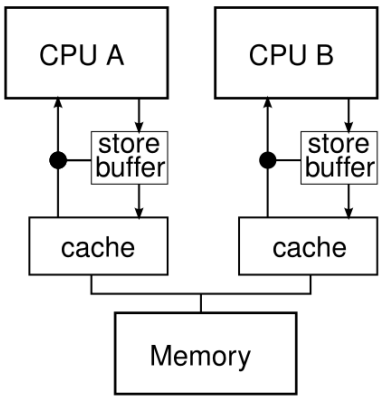


Out-Of-Order Stores

Now: Let's introduce some optimizations and see how they impact the behavior!

Problem so far: *stores always stall*, as after a store there is a let of MESI communication going on and the CPU cannot continue computing in the meanwhile.

Extend our so-far machine model to allow for *store buffers* between CPUs and caches:



- stores are not executed immediately; the CPU may continue working while the store buffer handles stores
- local CPU already sees the values of its still-pending stores (youngest if several stores for the same address)
- stores are written from store buffer to cache either FIFO (→ TSO) or unordered (→ PSO)

TSO: Total Store Order

Total Store Order

1. $\sqsubseteq$ is total *wrt. stores*: any $St_i[a], St_j[b]$ are $\sqsubseteq$-comparable
2. *Stores* in program order $\leq$ are embedded in $\sqsubseteq$:

$$St_i[a] \leq St_i[b] \implies St_i[a] \sqsubseteq St_i[b]$$

3. Loads *preceding another operation wrt.* $\leq$ are embedded in $\sqsubseteq$:

$$Ld_i[a] \leq Op_i[b] \implies Ld_i[a] \sqsubseteq Op_i[b]$$

4. A loads value is determined by the latest store *as observed by the local CPU*: $val(Ld_i[a]) = val(St_j[a])$, where

$$St_j[a] = \max_{\sqsubseteq}(\{St_k[a] \mid St_k[a] \sqsubseteq Ld_i[a]\} \cup \{St_i[a] \mid St_i[a] \leq Ld_i[a]\})$$

(The value of $St_j[a]$, where $St_j[a]$ is the $\sqsubseteq$-latest store which occurs before the load in $\sqsubseteq$-order or in $\leq$-order; i.e. a local store which is still in the store buffer is seen anyway on the same CPU).

Importantly, TSO does not satisfy one property which SC does satisfy: In TSO,

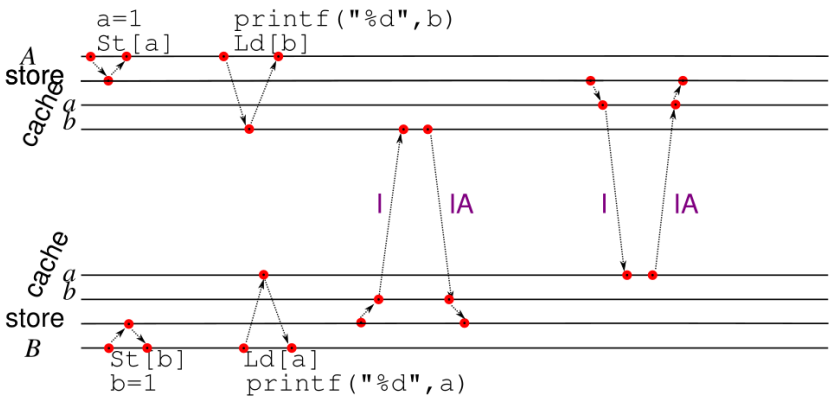
$$St_i[a] \leq Ld_i[b] \not\Longrightarrow St_i[a] \sqsubseteq Ld_i[b]$$

Example: Unexpected Result with TSO

Consider the two threads

```
// Thread A:
a = 1;
printf("%d", b);
// Thread B:
b = 1;
printf("%d", a);
```

Under sequential consistency assumptions, one would expect either of the outputs "10", "01", or "11". However under TSO assumptions, "00" is also a possible outcome! For example, in this happened-before-realization:



TSO in Practice: x86 and `mfence`

The x86 architecture uses a TSO memory model. To deal with the missing SC property, i.e. $St_i[a] \leq Ld_i[b] \not\Longrightarrow St_i[a] \sqsubseteq Ld_i[b]$, modern x86 CPUs provide the `mfence` instruction:

$$Op_i \leq mfence() \leq Op'_i \implies Op_i \sqsubseteq Op'_i$$

Therefore, a fence between stores and loads gives sequentially consistent CPU behavior (with the drawback of slowing down the CPU to store-buffer-less levels). These can be optionally used in cases where they are necessary.

PSO: Partial Store Order

Partial Store Order

1. $\sqsubseteq$ is total *wrt. stores*: any $St_i[a], St_j[b]$ are $\sqsubseteq$-comparable
2. Fenced stores in $\leq$-order are embedded in $\sqsubseteq$:

$$St_i[a] \leq sfence() \leq St_i[b] \implies St_i[a] \sqsubseteq St_i[b]$$

3. *Stores to the same address* in program order $\leq$ are embedded in $\sqsubseteq$:

$$St_i[a] \leq St_i[a]' \implies St_i[a] \sqsubseteq St_i[a]'$$

4. Loads *preceding another operation wrt.* $\leq$ are embedded in $\sqsubseteq$:

$$Ld_i[a] \leq Op_i[b] \implies Ld_i[a] \sqsubseteq Op_i[b]$$

5. A loads value is determined by the latest store *as observed by the local CPU*: $val(Ld_i[a]) = val(St_j[a])$, where

$$St_j[a] = \max_{\sqsubseteq}(\{St_k[a] \mid St_k[a] \sqsubseteq Ld_i[a]\} \cup \{St_i[a] \mid St_i[a] \leq Ld_i[a]\})$$

In contrast to TSO, PSO weakens the store embedding property: stores wrt. $\leq$ are now only embedded into $\sqsubseteq$ if they are fenced or if they go to the same address.

In particular, under PSO:

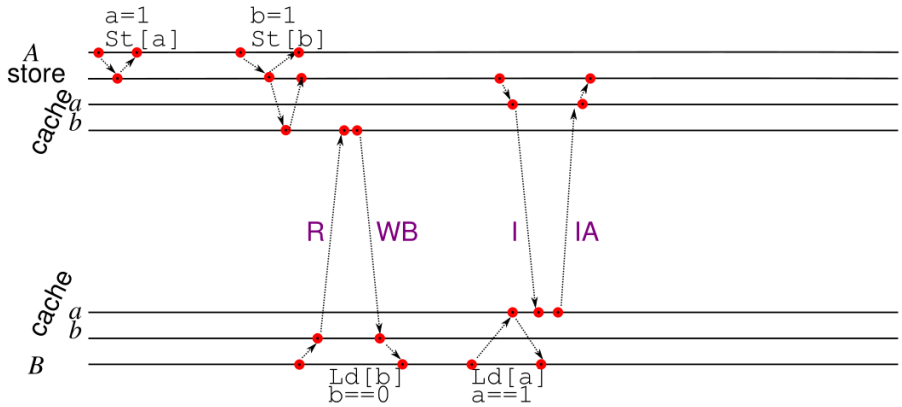
$$St_i[a] \leq St_i[b] \not\Longrightarrow St_i[a] \sqsubseteq St_i[b]$$

Example: Unexpected Result with PSO

PSO is now weak enough that our original example breaks:

```
// Thread A
a = 1;
b = 1;
// Thread B
while (b == 0) {};
assert(a == 1);
```

The assert will be false in the following happened-before realization:



What happens is that a=1 is written to A's store buffer before b=1, but b=1 is written to the cache earlier and hence B sees the state (b=1, a=0). Not sequentially consistent!

PSO in Practice: `sfence()`

In general, overtaking of store messages may be desirable; however if the program assumes sequential consistency between different CPUs, it is rendered incorrect.

The `sfence()` operation can be used to insert a write barrier, which forces stores currently in the store buffer to be completed before the next store can take place.

Adapted example where the assert cannot fail:

```
// Thread A
a = 1;
sfence();
b = 1;
// Thread B
while (b == 0) {};
assert(a == 1);
```

Out-of-Order Loads

A further way to weaken the memory model: *out-of-order Loads*.

Invalidate Queues

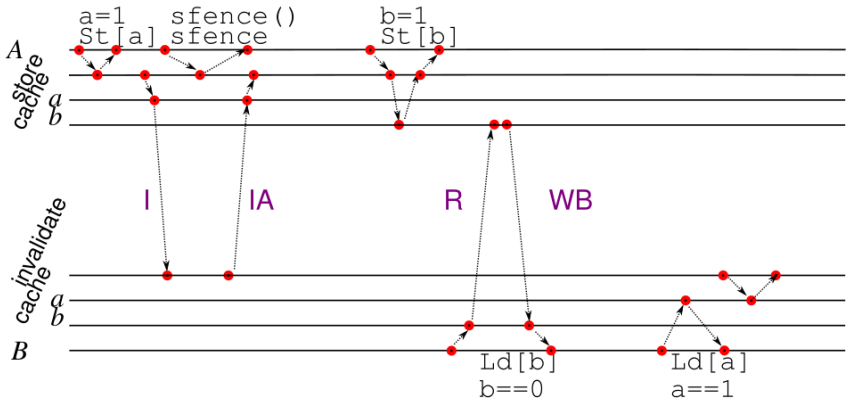
Introduce *invalidate queues*: Incoming "invalidate" messages from other CPUs are not executed immediately, but put into the queue and deferred. MESI messages regarding a cache line are not sent out while there is an invalidate pending for this line, however local loads and

stores on that line *are* executed. Therefore, *local loads may load an out-of-date value*.

Example: Unexpected Result with Invalidate Queues

With invalidate queues, even the improved original example with `sfence()` may fail:

```
// Thread A
a = 1;
sfence();
b = 1;
// Thread B
while (b == 0) {};
assert(a == 1);
```



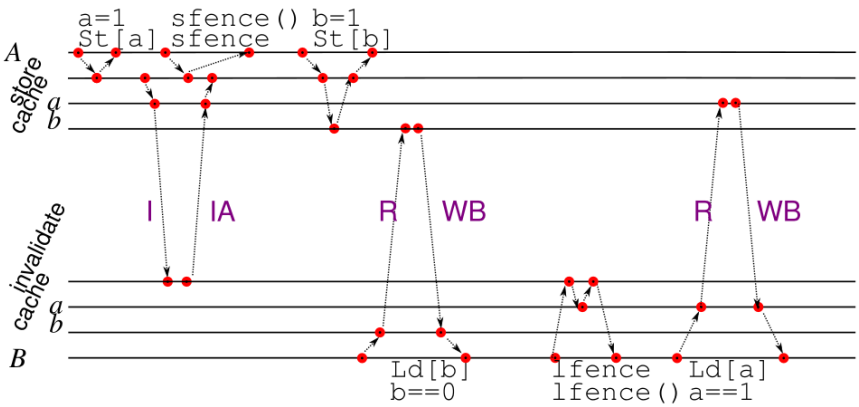
Load Barriers for Invalidate Queues

To enforce sequentially consistent loading behavior with invalidate queues, one can insert *load barriers* `lfence()`. A load barrier forces all invalidates waiting in the queue to be carried out before the next load is performed.

In practice: each *store barrier* in one process should be matched with a *load barrier* in another process.

Inserting a load barrier fixes the example again:

```
// Thread A
a = 1;
sfence();
b = 1;
// Thread B
while (b == 0) {};
lfence();
assert(a == 1);
```



Inserting load barriers *before each load* gives sequentially consistent behavior.

Load Buffers

Load buffers are somewhat of an alternative approach to invalidate queues, apparently.

Idea: *defer loads as well*. Namely, introduce a *load buffer* where pending loads are stored, and when the corresponding register is later accessed, the load may have been completed already (and otherwise is waited for). Motivation: For example, consider an expression where several values need to be loaded; by using the load buffer approach, some communication time is saved if the loads are not executed strictly one after another.

Load Barriers

Example (left out): Load buffers can lead to another kind of inconsistency. Namely, load accesses may compute "too late" and lead to loads "from the future".

Inserting load barriers *after each load* gives sequentially consistent behavior. Load barriers ensure that the next operation is only executed once all entries in the load buffer have completed.

RMO (Relaxed Memory Order)

Relaxed memory order: formal model to account for the kind of architectures we have seen.

Watch out: The specific specification depends on the manufacturer, but often manufacturers don't provide such a formal specification (but rather a specification by example).

Here: specification by SPARC architecture.

> **Relaxed Memory Order**
>
> 1. `mfence`d memory accesses in program order $\leq$ are embedded into memory order $\sqsubseteq$:
>
> $$Op_i[a] \leq \texttt{mfence()} \leq Op_i[b] \Rightarrow Op_i[a] \sqsubseteq Op_i[b]$$
>
> 2. *Stores to same address* in $\leq$ are embedded into $\sqsubseteq$:
>
> $$Op_i[a] \leq St_i[a]' \Rightarrow Op_i[a] \sqsubseteq St_i[a]'$$
>
> 3. Operations *dependent on a load* are embedded into $\sqsubseteq$:
>
> $$Ld_i[a] \rightarrow Op_i[b] \Rightarrow Ld_i[a] \sqsubseteq Op_i[b]$$
>
> 4. The value of a load is *determined by the latest store* as observed by the local CPU:
>
> $$val(Ld_i[a]) = val(\max_{\sqsubseteq}\{St_k[a] | St_k[a] \sqsubseteq Ld_i[a]\} \cup \{St_i[a] \mid St_i[a] \leq Ld_i[a]\})$$

The notion of *dependence* $\rightarrow$ is detailed in the specification:

- A dependence $Ld_i[a] \rightarrow St_i[b]$ holds if the load is the *latest load before the store* that has *the same target register that the store has as source register*.
- A dependence $Ld_i[a] \rightarrow St_i[b]$ holds if the a conditional branch that comes before the store depends on the load
- (doesn't apply here, but) a dependence $St_i[a] \rightarrow Ld_i[a]$ holds if $St_i[a] \leq Ld_i[a]$ (same address access)

Dekker Algorithm on RMO systems

Two processes want to ensure mutually exclusive access to a critical section. They share three variables: `flag[0]` (P0 has interest in critical section), `flag[1]` (P1 has interest in critical section), `turn` (the process which may enter the section right now).

Dekker's algorithm (assuming sequential consistency)

Algorithm for P0:

```
flag[0] = true;                         // P0 wants to enter
while (flag[1] == true) {       // While P1 also wants to enter:
        if (turn != 0) {                // If it's P1's turn:
                flag[0] = false;    // Signal "no interest", wait until it's P0's turn
                while(turn != 0) { /* busy wait */ }
                flag[0] = true;     // Take flag
        }
}

// critical section
{ /* ... (do the critical work) */

// After critical section: Let P1 work
turn = 1;
flag[0] = false;
```

(P1 symmetrically)

Dekker's algorithm in RMO

In RMO, the algorithm can fail to achieve the mutual exclusion. We need to insert `sfence()` and `lfence()` in some places:

- insert `lfence()` *before every load* of shared variables
- insert `sfence()` *after every store* of shared variables

```
flag[0] = true;
sfence();
while (lfence(), flag[1] == true) {
        if (lfence(), turn != 0) {
                flag[0] = false;
                sfence();
                while(lfence(), turn != 0) { /* busy wait */ }
                flag[0] = true;
                sfence();
        }
}

// critical section
{ /* ... (do the critical work) */ }

// After critical section: Let P1 work
turn = 1;
sfence();
flag[0] = false; sfence();
```

The `lfence()` placement in front of loads indicates that the invalidate buffer version of RMO is used in this example.

Conclusion

Memory barriers: lowest level of synchronization primitives.

- can be used for low-level protocol implementations
- hard to get right, best-suited for well-understood algorithms
- if store/invalidate buffers are a bottleneck, too many fences might be costly

Often better: (OS-offered) locks (better optimized, less low-level, easier to use).

Memory Models and Compilers

E.g. standard program optimization:

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

becomes (via *loop-invariant code motion* and *dead store elimination*)

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

But what if another process executes `x = 0`, and the intention was to always reset `x = 1` in the loop? → inconsistent behavior!

The compiler should also depend on consistency guarantees ⟶ demand for memory models on language model.

Avoiding compiler optimizations

Way 1: (sfence() keeps compiler from reordering across it)

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

Way 2: (volatile keeps C compiler from reordering accesses to this address)

```
volatile int x = 0;
for (int i = 0; i < 100; i++) {
        x = 1;
```

```
        printf("%d", x);
}
```

In Java, even further: barriers generated around accesses of `volatile` variables.

Week 4 - Atomic Execution

Introduction

Now: more high-level programming

Atomic Executions

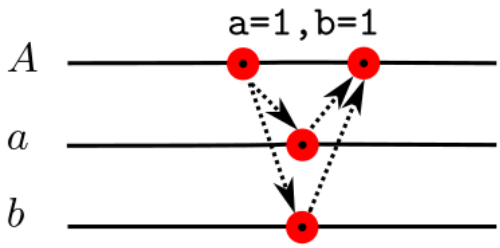
- Concurrent program: multiple resource-sharing threads (resources like memory locations or memory-mapped I/O).
- Invariants must be retained wrt. to the resources, which may be temporarily, locally broken during updates

A sequence of operations that temporally breaks invariants should be *atomic*: The invariant *never seems to be broken*.

Definition: a computation forms an *atomic execution* if its effect can only be *observerd* as a single transformation on the memory.

Relationship to Memory Barriers

Barriers are not enough: these act on *single* memory locations.



Use *barriers* to implement *automata that implement mutual exclusion*.

Wait-Free Algorithms

"Wait-Free": implementable without arbitrarily long waits.

Wait-Free Updates

Some examples that are not atomic:

```
// example 1
i++;

// example 2
j = i;
i = i+k;

// example 3
int tmp = i;
i = j;
j = tmp;
```

Reason: the load/stores, even in the simple `i++` case, might be interleaved with a store from another processor.

They *can be made atomic* by (e.g. on x86) locking the cache bus for an address. Examples:

```
// example 1
lock inc [addr_i]

// example 2
```

```
mov eax,reg_k
lock xadd [addr_i], eax
mov reg_j, eax

// example 3
lock xchg [addr_i],regj
```

Example: Wait-Free Bumper-Pointer Allocation

Simple garbage collector: system manages a contiguous chunk of memory that grows if more is needed; bumper pointer points to one of the edges of memory. Problem: multiple threads → might have concurrent `alloc` calls.

```
char heap[1<<20];
char* firstFree = &heap[0];

char* alloc(int size) {
        char* start;
        asm("lock; xadd %0, %1" :"=r"(start),"=m"(firstFree):
                "0"(size),"m"(firstFree) :"memory");

        if (start+size > sizeof(heap)) garbage_collect();
        return start;
}
```

Our Syntax for Atomic Statements

Instead of fiddling with inline assembly, we refer to examples 1-3 above and use the made-up keyword `atomic` to represent that actually atomic statements should be inserted.

```
// example 1 (atomic-increment)
atomic {
        i++;
}

// example 2 (fetch-and-add)
atomic {
        j = i;
        i = i+k;
}

// example 3 (atomic-exchange)
atomic {
        int tmp = i;
        i = j;
        j = tmp;
}

// example 4 (set-and-test)
atomic {
        // set flag b to 0, return its previous state
        r = b;
        b = 0;
}

// example 5 (set-and-test)
atomic {
        // set flag b to 1, return its previous state
        r = b;
        b = 1;
}

// example 6 (compare-and-swap)
atomic {
        // Update i to j if i's old value is equal to k
        r = (k==i);
        if (r) i = j;
}
```

Lock-Free Algorithms

If not wait-free, maybe lock-free?

Compare-and-Swap for Lock-Free Algorithms

The atomic *compare-and-swap* is commonly used like this:

```
do {
        k = i; // using memory barriers
        j = f(k);
}
// if i=k still holds, update i to j, else repeat
while (!atomic_swap(i, j, k));
```

(Assumption: $i = k$ implies that i was not updated.)

General Recipe

1. Group variables for which invariant must hold and read their n bytes atomically.
2. Compute a new value (using a pure function)
3. Perform compare-and-swap on these n bytes

Summary: Lock- and Wait-Free Algorithms

Severely limited: e.g. wait-free restricted to semantics of a single atomic operation. For lock-free algorithms, the n bytes of the invariant-abiding data must fit into the space provided by the compare-and-swap mechanism.

Also, the set of atomic operations varies by architecture.

Idea: use these low-level primitives as building blocks for *locks*.

Locks: Semaphores and Mutexes

A *lock* is a datastructure that can be acquired and released, ensures mutual exclusion, blocks other threads if they try to acquire it while already being locked. They are used to protect critical sections.

(Counting) Semaphores

A semaphore stores an integer `s` with a `signal` and a `wait` operation.

- `signal` increments the integer by one (releases the resource) atomically
- `wait` busy-waits for the the integer being greater than zero, in one atomic operation doing the check and if it passes, decrementing the integer (acquires the resource).

Special case: $s = 1$ ⇒ binary semaphore.

Implementation without busy waiting

Instead of performing busy-waiting, use the scheduling provided by the OS.

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

Here `de_schedule()` inserts the current thread into an OS-managed queue of threads that will be woken up once `*s` becomes non-zero. (Implementation by *monitoring writes* to s: `FUTEX_WAIT`)

The `signal` side has to call a `wake` function:

```
void signal(int *s) {
        atomic { *s = *s + 1; }
        wake(s);
}
```

Real-World Optimizations

- `wait()` might start off by busy-waiting for a few iterations before refraining to the schedule mechanism (good for locks with high throughput)

Semaphore with single core

... reduces to

```
if (*s) (*s)--; /* critical section */ (*s)++;
```

(apparently even with multiple threads, but only a single core, such a fine-grained atomicity is not required.)

Monitors

Monitors are automatic- re-entrant mutexes.

- locking procedure bodys is a pattern worth generalizing
- problematic: e.g. for recursive calls where the mutex blocks re-entry

Monitors: A procedure associated with a monitor acquires a lock on entry and releases it on exit. If the lock is already taken *by the current thread*, execution may proceed (i.e. no problems with recursive calls).

Still a problem: releasing lock again after last exit of function body by current thread.

Basic Monitor Implementation

This implementation stores

- a semaphore `count'
- the id `tid` of the occupying thread

```
typedef struct monitor mon_t;
struct monitor { int tid; int count; };
void monitor_init(mon_t* m) { memset(m, 0, sizeof(mon_t)); }
```

monitor_enter

- if already in thread's possession: increment the count
- otherwise more complicated: if monitor available (`tid == 0`), we take the monitor and increase the count. Otherwise, `de_schedule` waiting for the `tid` to change.

```
void monitor_enter(mon_t *m) {
        bool mine = false;
        while (!mine) {
                mine = thread_id() == m->tid;
                if (mine)
                        m->count++;
                else atomic {
                        if (m->tid == 0) {
                                m->tid = thread_id();
                                mine = true;
                                m->count = 1;
                        }
                };
                if (!mine) de_schedule(&m->tid);
        }
}
```

monitor_leave

- Decrement the count
- if count = 0 (i.e. monitor released by current thread), reset the thread ID and call `wake`

```
void monitor_leave(mon_t *m) {
        m->count--;
        if (m->count == 0) {
                atomic {
                        m->tid = 0;
                }
                wake(&m->tid);
```

```
        }
}
```

Condition Variables

With the constructions we have so far, there can still be efficiency problems: Say, a thread waiting for a queue to be filled, and busy-waiting, repeatedly calling `pop()`, until it can obtain an element. This way it produces contention on the lock (a producer might not even be able to `push()` into the queue).

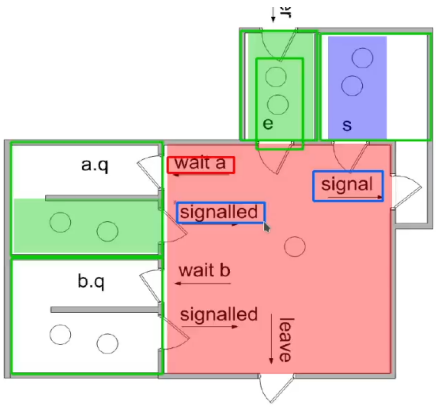
Monitor with Condition Variables

```
struct monitor { int tid; int count; int cond1; int cond2; ... };
```

with functions

- `wait`: Waiting for a condition to become true, *temporarily releases* monitor + blocks
  - when signalled: re-acquire monitor (now likely to succeed) and return
- `signal`: signal waiting threads that they may proceed
  - *signal-and-urgent-wait*: signalling thread blocks until signalled thread has released the monitor
  - *signal-and-continue*: signalling thread continues with its work

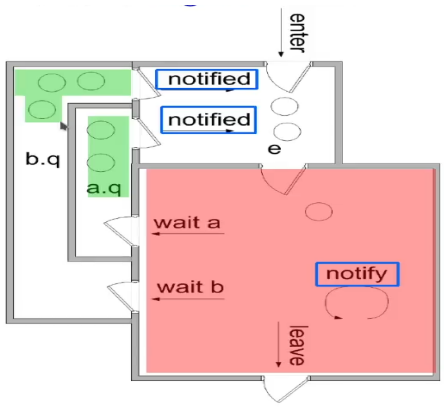
Signal-and-Urgent-Wait Semantics



- one queue for each condition
- one enter queue and one suspended queue
- signalled thread can immediately enter the monitor

Signal-and-Continue Semantics

(here, we say `notify` instead of `signal`)



- Call to `wait` for a → go into queue a.q
- Call to `notify` for a → first thread from a.q goes into e

Implementation (Simplified)

Here: only signal-and-continue for a single condition variable.

```
void cond_wait(mon_t *m) {
        assert(m->tid == thread_id()) // assert: we own the monitor (sometimes the compiler can do this)
        int old_count = m->count;
```

```
        atomic { m->tid = 0; }
        wait(&m->cond);
        bool next_to_enter;
        do {
                atomic {
                        next_to_enter = (m->tid == 0);
                        if (next_to_enter) {
                                m->tid = thread_id();
                                m->count = old_count;
                        }
                }
                if (!next_to_enter) de_schedule(&m->tid);
        } while (!next_to_enter);
}
```

- make sure the monitor is acquired, save the (own) count
- release the monitor and wait on the condition variable
- try to re-acquire the monitor, if unsuccessful `de_schedule` and repeat
- if successful: re-acquire monitor and reset count.

```
void cond_notify(mon_t *m) {
        // wake up other threads
        signal(&m->cond);
}
```

Notification Semantics

WIth signal-and-continue semantics, often two function exists:

- `notify` (wake up exactly one thread waiting on the condition variable)
- `notifyAll` (wake up all threads waiting on the condition variable)

Watch out: often `notify` actually has some "notify some" semantics which apparently simplifies the implementation. A thread cannot assume that it is the only one woken up → better recheck the condition.

Java and C# Monitors with Single Condition Variable

Monitors with a single cond. var. are built into Java and C#:

```
class C {
        public synchronized void f() {}
}
```

is equivalent to

```
class C {
        public void f() {
                monitor_enter(this);
                monitor_leave(this);
        }
}
```

and `Object` contains variables

```
private int mon_var;
private int mon_count;
private int cond_var;
protected void monitor_enter();
protected void monitor_leave();
```

The condition variable can be used within a `synchronized` method.

Week 5 - Deadlocks

Deadlocks

Deadlock: Two processes are waiting for the respective other to finish. More generally, a set of processes are cyclically waiting for each other to finish.

Example:

```
class Foo {
        public Foo other = null;
        public synchronized void bar() {
                other.bar();
        }
}

// in main:
Foo a = new Foo(), b = new Foo();
a.other = b; b.other = a;
// fictional syntax for "execute in parallel": ||
a.bar() || b.bar();
```

Four Condition for Deadlocks

Deadlocks occur $\Leftrightarrow$ following four conditions hold:

1. *mutual exclusion*: exclusive access to resources required
2. *wait for*: waiting for resources while holding on to others
3. *no preemption*: not possible to take resources away from processes
4. *circular wait*: processes wait in a cycle

Treatment of Deadlocks

- ignoring
- detecting (OS checks for a cycle; but needs ability to *preempt* to resolve)
- preventing (just design your programs to have no deadlocks)
- avoiding (additional information allows OS to schedule threads in a way s.t. no deadlocks occur)

$\Longrightarrow$ *Prevention* is the only feasible way on standards OSs.

Lockset Analysis: Deadlock Prevention through Partial Order

Idea: break the circular wait condition by partially ordering locks.

- denote by $\mathcal{L}$ the set of all locks
- for a program point p, denote by $\lambda(p) \subseteq \mathcal{L}$ the *lock set at p*: The set of all locks that may be in the acquired state at p
- for locks l, l' define $l \lhd l'$ iff $l \in \lambda(p)$ and the statement at p is a statement that acquires lock l'
  - here: `wait(l')` or `monitor_enter(l')`
- define the *lock order* $\prec$ as transitive closure of $\lhd$

Freedom of Deadlocks with $\prec$

Partition $\mathcal{L} = \mathcal{L}_S \cup \mathcal{L}_M$ into semaphore (mutex) locks and monitor locks. Now deadlock-freedom is related to irreflexivity of $\prec$.

Theorem (freedom of deadlocks, semaphores only). *If $\mathcal{L} = \mathcal{L}_S$ and for all $a \in \mathcal{L}_S : a \not\prec a$, the program is free of deadlocks.*

More generally for monitors:

Theorem (freedom of deadlocks, both semaphores and monitors). *The program is free of deadlocks if*

$$\forall a \in \mathcal{L}_S : a \not\prec a$$

and

$$\forall a \in \mathcal{L}_M, b \in \mathcal{L} : a \prec b \wedge b \prec a \Rightarrow a = b.$$

Summarizing instances of locks

It might be not statically possible to track all locks because there might be an unknown instances of locks (runtime-dependent). Lower-precision approximation: just summarize multiple instances of a class of locks into one *summarized lock/monitor*. Now if $\tilde{a} \not\prec \tilde{a}$ holds for all summarized locks, we are still fine. However if $\tilde{a} \prec \tilde{a}$, the program could be deadlock-free anyway (lower precision).

Deadlock Prevention Tactics

- keep an array of locks, only lock in increasing array index sequence
- or: if $l \in \lambda(p)$ and $l' \prec l$ is to be acquired, release l first, acquire l', reacquire l. This is quite inefficient.

Practical Example of Locksets/Lockset Order

Example here: lock instances whose object they lock on are created *on the same line* are summarized into one lock in $\mathcal{L}$.

```
0:  Foo * a = new Foo();
1:  Foo * b = new Foo();
2:  a->other = b;
3:  b->other = a;
4:
5:
6:  bar(a); || bar(b);
7:
```

```
8:  void bar(Foo* this) {
9:    monitor_enter(this);
10:   if (*) {
11:       ...
12:       bar(other);
13:       ...
14:   }
15:   monitor_leave(this);
16: }
```

- line 6: $\lambda(6) = \emptyset$. he synchronized bar calls
- line 8: `this` might be lock objects l_0 or l_1. $\lambda(8) = \emptyset$.
- line 9, monitor is being entered: $\lambda(9) = \{l_0, l_1\}$ because of that
- line 12: $\lambda(12) = \{l_0, l_1\}$
- line 8 again: $\lambda(8) = \{l_0, l_1\}$
- line 9: $l_0 \lhd l_1$ and $l_1 \lhd l_0$.

Summary: Locks

Atomic Execution $\iff$ Locks

The `atomic` blocks we introduced cannot be easily translated to locks. Some difficulties:

- an atomic block nested inside another atomic block should have no additional effect

It is non-standard in programming languages to have `atomic` blocks that automatically create locks, for a few reasons:

- use one global lock or several locks?
- some pairs of atomic blocks that access different variables would not require mutual exclusion
- locking on variables dangerous due to danger of deadlocks if one atomic block has another variable access order than another
- decrease in performance with many unnecessary locks

Concurrency Constructs in C/C++/Java/C#

language	barriers	wait-/lock-free	semaphore	mutex	monitor
C,C++	✓	✓	✓	✓	(a)
Java,C#	-	(b)	(c)	✓	✓

(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

Classification of Concurrent Algorithms

- **Wait-free**
  - never block, deadlock or starve and always succeed
  - very limited expressiveness
- **Lock-free**
  - never deadlock or starve, may starve and may fail
  - more expressive, but invariant must fit in little memory (Intel: 8 byte)
- **Locking**
  - may deadlock
  - can guard arbitrary code; several locks allow more fine-grained concurrency
  - can use semaphore (not re-entrant) or monitors (re-entrant)

Transactions

Abstraction and Composition vs. Concurrency

Two software concepts:

- abstraction: using an object without relying on its internals
- composition: combining several objects without interference

"The ability to compose relies on the ability to abstract from details."

Conflict Concurrency ⟷ Abstraction/Composition

Concurrency leads to problems with these principles. Example: a linked list that exposes `push()` and `forall()` and a set that provides `insert()` and internally relies on the linked list.

```
def insert(self, k):
        if self.list.forall(lambda x: x != k)
                push(self.list, k)
```

- assume the list is already threadsafe and uses some mutex internally
- the set now also needs to introduce a mutex because it makes *two* calls to the list
- it should use the *same mutex* as the list to avoid other list operations to be called

QUESTION: How is this an issue, given that the list is an internal of the set, not exposed?

The example shows: While sequential algorithms can always be composed, thread-safe algorithms that are composed may not yield other thread-safe algorithms.

Transactional Memory

Idea: convert `atomic` blocks ⟶ code that ensures atomic execution

- check that the block runs without conflicts with another thread - if another thread interferes, undo the computation done so far and restart the transaction
- provide `retry` keyword to manually restart

More in later sections.

Semantics of Transactions

We want the typical "ACID" properties:

- *atomicity*: transaction either runs completely or appears to not have run at all
- *consistency*: if a state is consistent (i.e. certain invariants holds) before execution, the state after the transaction was executed is consistent as well
- *isolation*: different transactions do not interfere with each other
- *durability*: effects of transactions are permanent
  - at least wrt. main memory!

> **Semantics of Transactions**.
> *The result of running concurrent transactions is identical to **one** execution of them in sequence (serializability).*

Consistency during Transactions, Opacity

ACID does not state what happens before a transactions commits: e.g. a transaction running on an inconsistent state may yield inconsistent states repeatedly (*zombie transaction*).

> **Opacity**.
> *A Transactional Memory system provides **opacity** if failing transactions are serializable wrt. comitting transactions (i.e. still see a consistent view of memory)*

Models of Isolation

Weak and Strong Isolation

- *strong isolation*: order between accesses to transactional and non-transactional memory retained
- *weak isolation*: guarantees only for transactional memory accesses (i.e. inside atomic blocks)

[TODO... Understand this better, find examples. What exactly is the problem in the code on the slides? Apparently, T2 can see intermediate states of T1. Likely the problem is that the value of x is visible before the transaction commits. The problem would be easier to see if x would be changed a second time in T1.]

```
// Thread 1:
atomic {
        x = 42;
}

// Thread 2:
tmp = x;
```

SLA: Single-Lock Atomicity

> **Single-Lock Atomicity (SLA).**
> *In the **single-lock atomicity** model, the execution is as if all transaction acquire a single program-wide mutex.*

The SLA model has some disadvantages:

- weak progress guarantees: two completely unrelated transactions that would not interfere at all cannot be executed concurrently.
  - extreme example: one transaction blocks indefinitely, no other transaction can be executed anymore
- SLA is stronger than one would like it to be in practice: e.g. empty atomic blocks can act as barriers and enable "spurious communication"
  - in the following example, the transactions should be independent, but synchronize

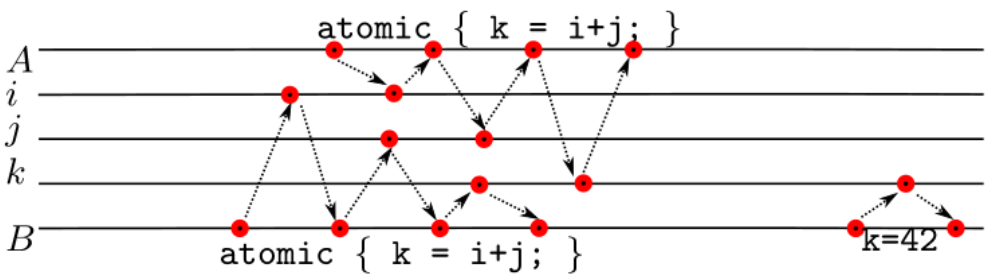
```
// Thread 1:
data = 1;
atomic {}
ready = 1;

// Thread 2:
atomic {
        int tmp = data;
        if (ready) { ... }
}
```

Transactional Sequential Consistency (TSC)

Goal: strong isolation should be implemented more flexibly.

> **Transactional Sequential Consistency (TSC).**
> In *transactional sequential consistency*, the accesses within each transaction are sequentially consistent.

TSC gives strong isolation, but allows some parallelism. Example:



TSC is not weaker than SLA, but stronger in some aspects, as accesses within a transaction may not be reordered! In practice, weakened versions of TLC are implemented where race-free reorderings are allowed.

QUESTION: What does this mean in detail? How does this ensure atomicity/isolation?

Software Transactional Memory

Now: how could such a transaction be implemented, in principle? How are the fictional `atomic` blocks translated, and how are there properties guaranteed in the implementation?

Translating `atomic` blocks

Convert transactional read/write accesses to `x` to calls `ReadTx(&x)`/`WriteTx(&x, val)`. Start transactions with `StartTx()` and commit them with `CommitTx()`.

```
do {
        StartTx();
        // code with ReadTx and WriteTx
} while (!CommitTx());
```

Translation with a preprocessor

Such a TM translation can be done by preprocessing or manually. Several ways:

1. try to minimize the set of transactional memory accesses (requires good static analysis)
2. simpler: translate all global variable and heap access to transactional memory accesses (we do that here)
3. Manual translation for more fine-grained control

`retry` keyword

A `retry` keyword might be provided: aborts and restarts the transaction. The transaction can wait with restarting until one of the variables it already read (the *read set*) changes.

Similar: Condition variables in monitors.

Implementation Techniques

Software TMs allocate *transaction descriptors* that store details about the transaction.

- *Undo logs* vs. *redo logs*:
  - *undo log*: all writes that have to be *undone* if the commit fails
  - *redo log*: all writes that are postponed until the commit
- *Read set* and *write set*:
  - Store the locations accessed (read from/written to) so far
- Read and write version
  - Timestamp (of logical clock) of when the value was accessed

One point made in the video: the system must decide on the granularity of memory locations. If it distinguishes every address as distinct memory location, the overhead of managing these locations might be too high. One alternative is to e.g. consider the set of adresses allocated by a single `malloc` as one location, as they are likely to belong to the same object in the program logic.

TL2: Example of an STM implementation

...from the Scalable Synchronization research group at Sun Microsystems Laboratories, as featured in the DISC 2006 article "Transactional Locking II".

Properties of TL2

The TL2 system

- ...provides *opacity*
- ...uses *lazy versioning*: writes stored in redo log and performed on commit
- ...*validating conflict detection*: accessing a modified address ⟶ abort

Functioning of TL2

1. on starting the transaction: obtain the *read version*, a global counter (timestamp)
2. reads `ReadTx` during the transaction:
   - watch out that no access to newer versions is performed
     - if an object version is younger during a read, abort the transaction
     - also abort if the object is locked
   - if the read goes through, add the accessed memory address to the read set.
3. writes `WriteTx` during the transaction:
   - add/update the location in the *redo log*
4. committing `CommitTx` the transaction:
   1. pick up locks for all the objects
   2. increment the global version
   3. check the read objects for being up to date
   4. if everything goes through, write redo log entries to memory and update their version; otherwise, the transaction fails

5. release all locks

Opacity in TL2

Opacity is guaranteed because transactions abort when read-accessing inconsistent values.

Locks and Preemption in TL2

Since writes need locks, deadlocks are still possible in principle. However

- transactions can be *preempted* ⟶ can break deadlocks
- a lock order might be enforced since the lock accesses are generated automatically ⟶ can prevent deadlocks

Locks still lead to the problem of contention on the global clock.

Challenges when using Software Transactional Memory

1. Unnecessary/suboptimal abortion of transactions: E.g., a very big transaction aborts on the last line because another very short transactions just changed a single value. Especially easy to happen if the granularity of transactions is too large.

```
// Thread 1
atomic {
        ... // 100 lines of code
        int r = ReadTx(&x);
}

// Thread 2
atomic {
        WriteTx(&x, 42);
}
```

2. Contention because of lock-based commits
3. Danger of live locks (A aborts B, B reruns and aborts A, A reruns and aborts B, etc.)

Integration of Non-Memory Resources

Problems when accessing other resources than memory inside atomic blocks. E.g. storage management, condition variables, `volatile` variables, I/O.

The semantics should be as if SLA or TSC semantics were implemented. There are some ways to handle this problem:

- "prohibit it": compiler should reject certain constructs
- "execute it": abort if actual IO happens (sometimes it might just go to a buffer, which possibly is no problem)
- "irrevocably execute it": to deal with operations that are not undoable: once such an operation is executed, enforce that its transaction must terminate by making all other transactions conflict.
- "integrate it": Rewrite resource accesses to be transactoinal (error logging; writing data to a file; etc...)

Currently: best to use TM for memory only, use irrevocable transactions for other resources.

Week 7 - Hardware Transactional Memory

Hardware Transactional Memory

Overview: Hardware Transactional Memory

Motivation: Maybe we can do perform transactions on hardware instead in software, and have them be more efficient! Rough idea:

- track read and write sets in hardware
- hook into the *cache* messaging functionalities that have to track accesses anyway:
    1. cache line in *read set* is *invalidated* ⟹ transaction aborts
        - because then someone else produced a newer version of a value that was read
    2. cache line in *write set* is *written back* ⟹ transaction aborts
        - because then someone else tried to access the value updated by the transaction

In principal, two approaches: Explicit and Implicit transactional memory.

Explicit Transactional Memory

Parallel to the operations in software transactional memory: Each transactional access is explicitly marked as transactional, similar to `StartTx`, `ReadTx`, `WriteTx`, `CommitTx`.

Problem when composing code: Calling some non-transactional library function from the transaction may break the transaction guarantees, because the library function fails to use the correct transactional commands!

Implicit Transactional Memory

Mark only beginning and end of transaction - the hardware internally keeps track if it is currently in *transactional mode* or not, and interprets the usual instructions in the corresponding mode.

Hardware access, OS calls, page table changes etc. all abort a transaction, because the go around the cache!

Implicit TM provides strong isolation

Example Implementations of Hardware TM

AMD Advanced Synchronization Facilities (ASF)

ASF was an AMD project which was never put into production.

Functioning: define *speculative region* in memory, which is treated transactionally. Instructions to explicitly transfer data between normal memory and the speculative region.

Intel TSX in Broadwell/Skywell architectures

TSX by Intel was put into production: However, soon correctness and security bugs appeared, which forced Intel to publish firmware updates and disable many features again. Today, only exists on server systems, not on desktop/mobile processors.

Functioning: Implicit Transactional Memory, tracks read/write sets using a *transaction bit* on cache lines. CPU state can be backed up to reserved space. Transactions may abort due to lack of resources.

Two Software Interfaces by Intel to TM, treated in detail in the next sections:

1. Restricted Transactional Memory (RTM)
2. Hardware Lock Elision (HLE)

Restricted Transactional Memory

Hardware Implementation of RTM

Only a few things have to be modified:

- a nesting counter C and a backup register set is added to the CPU registers
  - C counts how deeply nested the current transaction is
- to each cache line, a bit T is added, marking it as part or not part of a transaction

Transactional instructions:

- `xbegin` increments C, if $C = 0$ it backs up registers and flushes buffer
- `xabort` clears all T flags, the store buffers, invalidates TM lines, sets $C = 0$ and restores CPU state
- `xend` decrements C and, if $C = 0$ (i.e. outermost transaction), clears the T flags and flushes the store buffer

Different behavior of normal instructions inside a transaction:

- read or write accesses to a cache line set $T = 1$
- a MESI *invalidate* for a cache line that has $T = 1$ leads to `xabort`
- a MESI *read* for a *modified* cache line that has $T = 1$ leads to `xabort`

Software Interface to RTM

Instructions:

1. `xbegin`:
   - on transaction start: skips to next instruction
   - address can be specified where to continue at abort
   - on abort, error code is stored in eax
2. `xend`: commits the transaction started by the most recent `xbegin`
3. `xabort`: aborts whole transaction
4. `xtest`: checks if processor is in transaction mode

There is a library function `_xbegin()` that wraps the `xbegin` instruction:

- returns a code that indicates whether the transaction successfully started
- **very important**: if the transaction later aborted, *execution jumps back to the return from `_xbegin()`, now with an error code!*

Therefore, transactional code should be structured like this:

```
if(_xbegin() == _XBEGIN_STARTED) {
        // transaction code
        _xend();
} else {
        // non-transactional fall-back
        // is executed EITHER if transaction failed to start OR if it was aborted
}
```

Considerations for the Fallback Part

One should make sure that the fallback part is free of races as well, and in particular *also isolated from the actual transaction*. We can use a technique with a mutex:

```
if(_xbegin() == _XBEGIN_STARTED) {
        if (!mutex > 0) _xabort();
        // Do the transaction
        _xend();
} else {
        wait(mutex);
        // Do the fallback code
        signal(mutex);
}
```

The trick is: even though the transactional part does not actually *take the mutex*, the mutex is added to the transaction's *read set* and if someone else takes the mutex in the meanwhile, the transaction will abort.

Pattern for Transactional Mutexes

Idea: The pattern seen above can be generalized to allow transaction-based mutexes that fall back to regular mutexes in case of a race.

```
void update() {
        lock(&mutex);
        // Do the critical section
        unlock(&mutex);
}
void lock(int *mutex) {
        if(_xbegin() == _XBEGIN_STARTED) {
                if (!*mutex > 0) _xabort();
                else return;
        } wait(mutex);
}
void unlock(int* mutex) {
        if (!*mutex > 0) signal(mutex);
        else _xend();
}
```

Hardware Lock Elision

Idea: use the existing Restricted Transactional Memory infrastructure to do an optimistic optimization of mutex locking, resembling the optimization shown in Pattern for Transactional Mutexes.

By default, actual acquisition of the lock is deferred, and just tracked with the HTM system; in case there is a conflict, fall back on actual locking. This is supported by some hardware additions. Legacy code is supported: locally, it looks as if the semaphore value *was* modified.

General functioning:

- prefix lock-acquiring instructions with `xacquire`
- prefix lock-releasing instructions with `xrelease`

The codes for these annotations are chosen such that they map to NOPs on older platforms, i.e. there just the usual locking takes place.

Implementing Lock Elision

Addition: a buffer for elided locks that lives besides the store buffer.

1. `xacquire` ensures shared/exclusive cache line state with T bit set, issues `xbegin` *and keeps the modified lock value in the elided lock buffer*
   - if the T cache line is later invalidated, the transaction is aborted
   - the local CPU sees the modified value of the lock due to the elided lock buffer kicking in
2. The fallback path in case of a conflict leads to the actual locking instruction that was prefixed with `xacquire`: Fallback behavior = conventional locking
3. `xrelease` ends the transaction and clears the elided lock buffer

Transactional Memory in Practice

GCC has support:

- translates accesses in `__transaction_atomic` regions into library calls to `libitm`
- `libitm` provides TM implementations: uses HTM instructions on TSX systems, otherwise resorts to STM

C++20 standardizes `synchronized`/`atomic_...` blocks

OpenJDK support increasingly introduced (introduced in the VM implementation)

Hardware lock elision support is limited.

Outlook: Other principles for Concurrent Programming

1. non-blocking message passing (aka the actor model): Program consists of actors that send messages, actors have message queues.
   - example: Erlang. Also, add-ons to existing languages that support such a model
   - also partly in Rust with channels?
2. blocking message passing (CSP, π-calculus, join-calculus)
   - examples: Occam, Occam-π, Go
3. (immediate) priority ceiling
   - *processes* and *resources*, processes have *priority*
   - resource has maximum ("ceiling") priority of all processes that may acquire it
   - process priority at runtime = maximum of priorities of held resources
   - process with maximum run-time priority executes

Week 8 - Function Dispatching

Function Dispatching

The simplest model of dispatching is: each function name corresponds to exactly one function.

Example in C (ANSI C89): to dispatch, just use the function name to find the method, and check if the parameters match.

```c
void fun(int i) {}
void bar(int i, double j) {}

int main() {
        fun(1);
        bar(1, 1.2);
}
```

Overloading Function Names

One step further: allow multiple methods with the same name, but a different signature.

C11: Workaround for missing overloading

C11 still does not have overloading, but has a workaround: *Generic Selection*, which can be used e.g. in Macros. Depending on the static type of a variable, one of several expressions is evaluated. Syntax with `_Generic(...)`

Example:

```
int main() {
        printf(_Generic((1.2), signed int: "%d", float: "%f"), 1.2)
}
```

Java Overloading

In Java: can overload methods.

Example:

```
class D {
        public int f(int i) {}
        public double f(double d) {}
}
```

Overloading also works with inheritance, i.e. a child class can overload methods of the base class.

```
class D {
        public int f(int i) {}
}

class D extends B {
        public double f(double d) {}
}
```

C++ Overloading

Overloading per se works the same way as in Java.

Overloading with inheritance has one caveat to note: The method from the base class must be specified with `using MyParent::my_method`, else it is possibly treated as an *override* (e.g. in the case of doubles/ints where implicit casts are performed).

```
class B { public:
        int f(int i) {}
};
class D : public B { public:
        // If this is not present, f(double d) *overrides* f(int i)
        using B::f;
        double f(double d) {};
};
```

Overloading Ambiguities

Ambiguities may arise with overloading:

```
class D {
        public int f(int i, double j) {}
        public int f(double i, int j) {}
}

// ... in main:
(new D()).f(2, 2); // this fails!
```

Error message from java: "error: reference to f is ambiguous". Reason: there is not clear *most specific variant*.

Overloading with Static Dispatching

Overloading Theory

Three concepts involved:

- *function call expressions* $f(e_1, \ldots, e_n)$
- *call signatures* $t_0, \ldots, t_n$
- *function definitions* $t_0 f(t_1 p_1, \ldots, t_n p_n)$

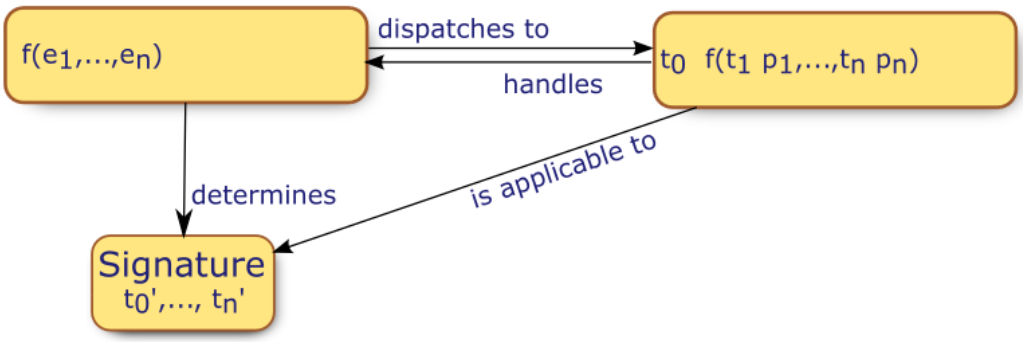
The call signature consists of the function name, the static parameter types and the return type. Function call expressions determine the signature and dispatch to function definitions. Function definitions are applicable to signatures.

(QUESTION: in the following, I think we define applicability of signatures to signatures, right?)

Definition (*Applicability*).
f is applicable to f', denoted $f \leq f'$ ($\leq$ is the subtype relation), if

$$T_0 \leq T_0' \wedge T_i \geq T_i' \forall i > 0$$

where the signatures are $T_0\ f(T_1, \ldots, T_n)$ and $T_0'\ f'(T_1', \ldots, T_n')$



Javac implementation of overloading

Javac defines

- a function `isApplicable(MemberDefinition m, Type args[])`
  - checks applicability of methods to an argument list, using the `isMoreSpecific` method to check $\leq$
- a function `isMoreSpecific(MemberDefinition more, MemberDefinition less)`
  - for member functions: checks that the classes that the methods are defined in are in $\leq$ relation and applicability of the arguments of one to the other method
- a function `matchMethod(Environment env, ClassDefinition accessor, Identifier methodName, Type[] argumentTypes)`
  - finds the most specific method that matches a signature
  - checks all methods in the class and takes note of those that match
  - then checks if there is a unique most specific one (otherwise: ambiguity exception)

```
// method m is applicable to actual parameter types in args
boolean isApplicable(MemberDefinition m, Type args[]) {
    // Sanity checks:
    Type mType = m.getType();
    if (!mType.isType(TC_METHOD))               return false;

    Type mArgs[] = mType.getArgumentTypes();
    if (args.length != mArgs.length)            return false;

    for (int i = args.length ; --i >= 0 ;)
        if (!isMoreSpecific(args[i], mArgs[i])) return false;
    return true;
}
// more<=less (implementation is type based)
boolean isMoreSpecific(Type moreSpec, Type lessSpec)
```

```
// more<=less implementation for Member Methods
boolean isMoreSpecific(MemberDefinition more, MemberDefinition less) {
    Type moreType = more.getClassDeclaration().getType();
    Type lessType = less.getClassDeclaration().getType();
    return isMoreSpecific(moreType, lessType) // return type based comparison
        && isApplicable(less, more.getType().getArgumentTypes()); // parameter type based
}
```

```
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 name `methodName'
    for (MemberDefinition method : allMethods.lookupName(methodName)) {
        // See if this method is applicable.
        if (!env.isApplicable(method, argumentTypes)) continue;
        // See if this method is accessible.
        if ((accessor != null) && (!accessor.canAccess(env, method))) continue;
        if ((tentative == null) || (env.isMoreSpecific(method, tentative)))
            // `method' becomes our tentative maximally specific match.
            tentative = method;
        else {  // If this method could possibly be another maximally specific
                // method, add it to our list of other candidates.
                if (!env.isMoreSpecific(tentative,method)) {
                    if (candidateList == null) candidateList = new ArrayList();
                    candidateList.add(method);
    }   }   }
    if (tentative != null && candidateList != null)
        // Find out if our `tentative' match is a uniquely maximally specific.
        for (MemberDefinition method : candidateList )
            if (!env.isMoreSpecific(tentative, method))
                throw new AmbiguousMember(tentative, method);
    return tentative;
}
```

Overriding methods

Recall: in OOP, procedures can strongly associate with objects (i.e. *instance methods* and *receivers*). Convention: associate first parameter ⟷ receiver of the method, bind it to `this` and have some extra syntax like `receiverParam.myMethod(secondParam, thirdParam)`.

Also, in OOP, a subtype should take responsibility for method calls with it as receiver. For example, it should ensure that a method call leaves its internal state consistent.

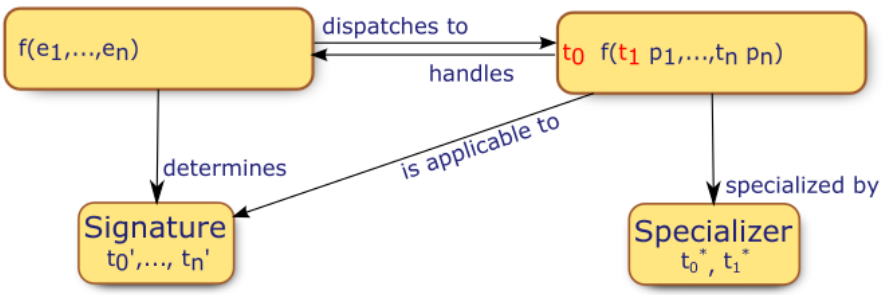
This leads to the *overriding* principle: if a subtype implements a method that exists in the supertype, the subtype method overrides this method, i.e. is always called instead of the supertype method. In particular, this also holds when an object statically is of the supertype, but dynamically is of the subtype.

```
class Integral { void incBy(int delta) {...} }
class Natural extends Integral { void incBy(int i) {...} }

// ...somewhere in the code
Integral i = new Natural(42);
i.incBy(43); // this calls the Integral::incBy method because of overriding
```

Dynamic Single-Dispatching

Dynamic dispatching with overriding theory

The Overloading Theory from above is slightly augmented: there is a *specializer* involved additionally which specializes a candidate function definition and does special dispatching based on the `this` parameter. During compiletime a more symbolic signature is generated instead of an address, which at runtime is matched to the most specific function.



Java Implementation of Dynamic Dispatching

Dynamic Dispatching in Bytecode

As before, *match method* is used to generate the *statically most specific signature*. In other words: a signature is generated which matches the static types of the variables and actually exists somewhere as a method. (I think) this signature consists of the name, return type and parameters of the function, but not the `this` type.

In the bytecode, an invocation consists of a call to `invokevirtual` with the computed signature.

Dynamic dispatching in the Hotspot VM

The Hotspot VM gets the statically computed signature, and dynamically resolves the method to call.

Multiple methods involved:

- `resolve_method` does some general checks and calls `lookup_method` with the *dynamic* type of the receiver
- `lookup_method` traverses the superclass chain and calls `find_method` until a match is found
- `find_method` searches for a method that matches the signature in a specific class

`find_method`

- finds all method of the right name available in the given class (via a binary search)
- searches all these methods and looks for an *exactly matching signature*
  - this is important: There are *no checks for "most specialized" etc.* done here!

The last point is what can lead to confusing behaviour, as we will see in the examples.

Examples: Dynamic Dispatching in Java

Overriding `equals` for a Set

Unexpected behavior when putting numbers in a set:

```
class Natural {
        // ...
        public boolean equals(Natural n) { return n.number == number }
}
// ... somewhere in the code
Set<Natural> set = new HashSet<>();
set.add(new Natural(0));
set.add(new Natural(0));
System.out.println(set); // [0, 0]
```

The `equals` implementation cannot be used by `HashSet` because it specializes the parameter! In other words: the statically generated signature (`Object.equals(Object)`) cannot be matched to this signature.

Visitor pattern with Single Dispatching

Recall the visitor pattern:

```
abstract class Exp {
        public abstract int accept(EvalVisitor v);
}

class Const extends Exp {
        public int value;
        // ...
        public int accept(EvalVisitor v) { return v.visit(this); }
}

class Sum extends Exp {
        public Exp left, right;
        // ...
        public int accept(EvalVisitor v) { return v.visit(this); }
}

class EvalVisitor {
        int visit(Const c) { return c.value; }
        int visit(Sum s) {
                return s.left.accept(this) + s.right.accept(this);
        }
        int visit(Exp e) { return e.accept(this); }
}
// ... somewhere in the code:
Exp e = new Sum(new Const(1), new Const(1));
System.out.println(new EvalVisitor().visit(e));
```

Why are the completely boilerplate `accept` methods needed? Because of Java's Single-Dispatching!

What if we would write:

```
int visit(Sum s) {
        return visit(s.left) + visit(s.right);
}
// no default accept
// int visit(Exp e) { return e.accept(this); }
```

Then there is an error "error: no suitable method found for visit(Exp)". Since `s.left` has static type Exp, `visit(s.left)` can only be dispatched to `visit(Exp)` at runtime (no specialization in the parameter). With the `accept` method however, dynamic dispatching is possible (specialization in the `this` parameter is possible).

The path taken in the visitor pattern: (simplified pseudo-correct notation)

- `visit(e: Exp)`
- `accept(e: Exp)`
- dynamically dispatched to `accept(e: Sum)`
- `visit(e: Sum)`
- `accept(e.left: Exp)`
  - dynamically dispatched to `accept(e.left: Const)`
  - `visit(e.left: Const)`
- `accept(e.right: Exp)`
  - dynamically dispatched to `accept(e.right: Const)`
  - `visit(e.right: Const)`

Mini-Quiz about Single Dispatching

```
class A {
        public void m1(A a) { System.out.println("m1(A) in A"); }
        public void m1()    { m1(new B()); }
        public void m2(A a) { System.out.println("m2(A) in A"); }
        public void m2()    { m2(this); }
}

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

1. `A a = new B(); a.m1(new B());`
2. `(new B()).m1(new B());`
3. `(new B()).m2();`
4. `(new B()).m1();`
5. `(new B()).m3();`

Answers:

1. "m1(A) in A"
   - statically, signature is A.m1(B) which does not exist, so is converted to A.m1(A)
   - dynamically, B does not have a method matching m1(A), so A.m1(A) is called
2. "m1(B) in B"
   - statically, signature is B.m1(B); this exists and also matches dynamically
3. "m2(A) in B"
   - statically, signature is B.m2() which does not exist, so is converted to A.m2()
   - dynamically, A.m2() is called
     - inside m2: statically, signature is A.m2(A)
     - dynamically, the receiver is specialized to B, and B has a method matching m2(A). So B.m2(A) is called.
4. "m1(A) in A"
   - A.m1() is called
     - inside m1: statically, signature is A.m1(B) which does not exist, so is converted to A.m1(A)
     - dynamically, the receiver is specialized to B, but B has no method m1(A). So A.m1(A) is called
5. "m1(A) in A"
   - B.m3() is called
     - inside m3: statically, signature is A.m1(B) which does not exist, so is converted to A.m1(A)
     - dynamically, the receiver still is A and A.m1(B) is called.

Multi-Dispatching

The examples have shown that single-dispatching isn't that great. Can we get anything better?

Workarounds for Multi-Dispatching in Java

How can we solve the `equals()` problem within the Java-Single-Dispatching framework with some workaround?

Multi-Dispatch via Introspection

The usual approach for `equals`: Keep your signatures general, perform the specialization manually via type introspection.

```
// ...
public boolean equals(Object n) {
        if (!(n instanceof Natural)) return false;
        return ((Natural)n).number == number;
}
// ...
```

Problems: this burdens the programmer with type saftey issues, and requires type introspection in the language.

Multi-Dispatch via Generics

Another idea: introduce an `Equalizable` interface and implement a set that is aware of this interface.

```
interface Equalizable<T> {
        boolean equals(T other);
```

```
}
class Natural implements Equalizable<Natural> {
        // ...
        public boolean equals(Natural n) { return n.number == number; }
}
// ... somewhere in the code:
EqualizableAwareSet<Natural> set = new EqualizableAwareHashSet<>();
```

Really not great: we need a new set implementation! Also, this only works for *one* overloaded version in the super hierarchy. (QUESTION: why?)

Double Dispatching

A bit similar to the Visitor Pattern example, rely on multiple single-dispatching calls to achieve what you want.

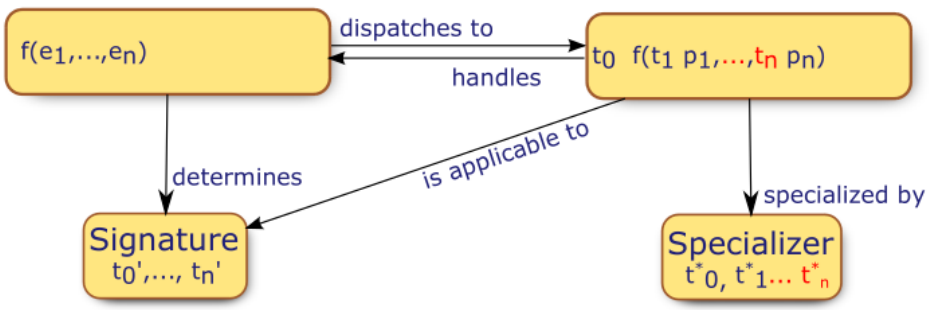
- `equals` calls a `doubleDispatch` method on the other object
- the current value is captured in an anonymous class, to which the other value is compared to later if it is a `Natural`

```
abstract class EqualsDispatcher {
        boolean dispatch(Natural n) { return false };
        boolean dispatch(Object o) { return false };
}
class Natural {
        // ...
        public boolean doubleDispatch(EqualsDispatcher ed) {
                return ed.dispatch(this);
        }
        public boolean equals(Object n) {
                return n.doubleDispatch(
                        // Create an anonymous class for the actual comparison
                        new EqualsDispatcher() {
                                boolean dispatch(Natural nat) {
                                        return nat.number == number;
                                }
                        }
                )
        }
}
```

Problems: class hierarchies need to be known for the Dispatcher interface; more problematically, `Object` needs to offer `doubleDispatch` as well (i.e. this only works for more specific baseclasses than `Object` where such a method can be coded in). Also, it looks very complicated and over-engineered.

Multi-Dispatching Theory

Same as the model for overriding, except that specialization works on all parameters.



Here, the dispatcher *dynamically* selects the *most specific concrete method*.
- not statically, but at runtime

Requirements for Implementing Multi-Dispatching

Implications on

- **Type checking**:
  - e.g. for static type safety, would need to check if there is a unique most specific method for all visible type *tuples*
  - when ambiguities arise at runtime (through MI/interfaces) there must be new runtime errors for these ambiguities (and it would be easy to overlook that something like this could occur!)
- **Performance**: the performance might suffer, especially with a high number of parameters
  - However usually, the penalty only occurs where multi-dispatching is used, and not elsewhere

Multi-Dispatching in Multi-Java

MultiJava is a Java extension that supports multi-dispatching annotations.

Back to our equals() example:

```
class Natural {
        public boolean equals(Object@Natural n) { return n.number == number; }
}
```

This works cleanly now!

(Note: the Multi-Java project seems to be abandoned, and was not updated since 2006.)

Multi-Java: Behind the Scenes

Looking at the bytecode, this more or less translates to a runtime-`instanceof` check, i.e. similar to the introspection workaround.

Languages with Native Multi-Dispatching

Multi-Dispatch in Raku

Raku (a Perl dialect, formerly known as *Perl 6*) has builtin multi-dispatching functionality. For methods that are marked as `multi`, multi-dispatch is used.

```
# type Cool is supertype of Int and Str
multi fun(Cool $one, Cool $two) { say "Dispatch base" }
multi fun(Cool $one, Str $two)  { say "Dispatch 1" }
multi fun(Str $one, Cool $two) { say "Dispatch base" }

my Cool $foo;
my Cool $bar;

# This gives: "Dispatch 1"
$foo = 1;
$bar = "bla";
fun($foo, $bar)

# This gives: "ambiguous call to fun(Str, Str)"
# (both the 2nd and 3rd signatures match, there is no most specific one)
$foo = "bla";
fun($foo, $bar);
```

Multi-Dispatch in Clojure

Clojure (a Lisp dialect) also has native multi-dispatching.

Clojure basic syntax

(copied from the slides, probably not important)

- Prefix notation
- () – Brackets for lists
- :: – Userdefined keyword constructor ::keyword
- [] – Vector constructor
- fn – Creates a lambda expression (`fn [x y] (+ x y)`)
- derive – Generates hierarchical relationships (`derive ::child ::parent`)
- defmulti – Creates new generic method (`defmulti name dispatch-fn`)
- defmethod – Creates new concrete method (`defmethod name dispatch-val &fn-tail`)

Clojure Multi-Dispatching Principles

Clojure allows more creative multi-dispatching as it allows to specify a dispatch function whose result determines the concrete variant used.

Example:

```
(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))
```

What happens here?

- `(derive ::child ::parent)` establishes a hierarchical relationship
- `(defmulti fun (fn [a b] [a b]))` means
  - when `fun` is called with parameters a, b, then the value `[a, b]` should be used to determined which implementation to dispatch to
- `(defmethod fun [::child ::child ] [a b] "child equals")` means
  - when the dispatching function returns `[::child ::child]`, the lambda expression `[a b] "child equals"` should be used

Elaborate Clojure Multi-Dispatch Example

```
; salary yields poor, rich or average label
(defn salary [amount]
        (cond (< amount 600)   ::poor
                  (>= amount 5000) ::rich
              :else            ::average))

(defrecord UniPerson [name wage])

; the multi method print dispatches to different variants
;       depending on the value of the salary function
(defmulti print (fn [person] (salary (:wage person)) ))
(defmethod print ::poor    [person] (str "HiWi  " (:name person)))
(defmethod print ::average [person] (str "Dr.   " (:name person)))
(defmethod print ::rich    [person] (str "Prof. " (:name person)))

(pr (print (UniPerson. "Petter" 2000)))  ; Dr. Petter
(pr (print (UniPerson. "Stefan" 200)))   ; Hiwi Stefan
(pr (print (UniPerson. "Seidl" 16000)))  ; Prof. Seidl
```

Takeaway: dispatching does not necessarily need to be tied to typing, there are also more creative approaches.

Summary: Multiple Dispatching

Pros:

- Generalizes the established technique, leads out of the weird single-dispatching story
- eliminates boilerplate code
- compatible with modular compilation/type checking

Cons:

- hard to combine with the "first parameter dispatching" which naturally comes from OOP
- some runtime overhead; new runtime exceptions
- notion of *most specific method* ambiguous

Some other languages which support multi-dispatching: *Dylan* and *Scala*.

Week 9 - Multi-Inheritance

Multi-Inheritance

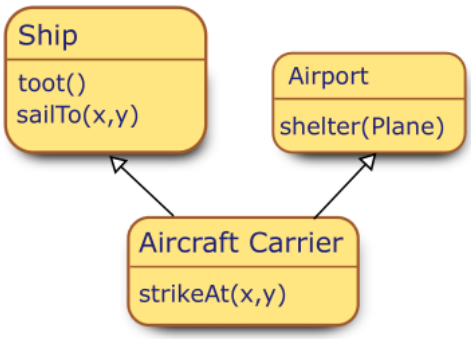
Implementation vs. Interface Inheritance

Implementation inheritance

Usual/classic motivation for inheritance: *code reusage*.

The parent implements common behaviours (acts as library of common behaviours), the child replaces particular behaviours with its own ones.

"An Aircraft Carrier is both a Ship and an Airport"

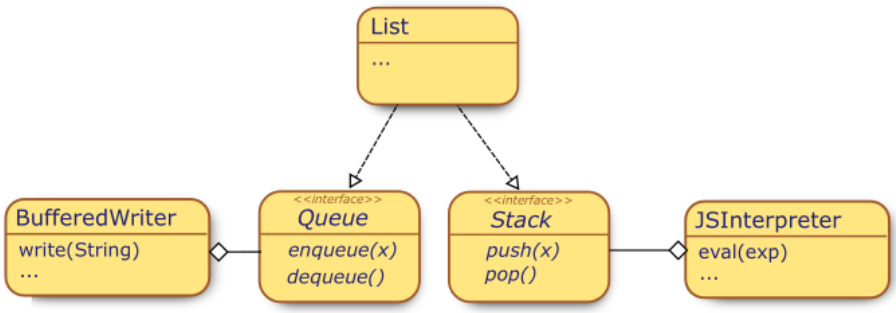


Interface inheritance

In interface inheritance, this relation is inverted: inheritance as *behavioural contract*.

The parent formalizes a type requirement to allow childs to *participate in shared code frames*. The child provides functionality for the method signatures specified by the parent.

"A List can be used both as a Queue and a Stack", and some other parts of the code (here BufferedWriter and JSInterpreter) use a Queue/Stack without specifying a specific implementing class.



Excursion: Intro to LLVM IR

LLVM IR = LLVM Intermediate Representation (LLVM used to be an acronym, but isn't anymore). LLVM is used for one common C++ compiler. We have a look at LLVM to look at how objects are layed out in memory.

LLLVM IR Example

Struct definitions: (LLVM IR has a concept of types, which makes it a little bit more abstract than assembly!)

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

For example, B has an i64 as first field, a 10×20 i32 array as second field, an i8 as third field. A has a struct B nested into it. The third field in A is a *function pointer taking i32 returning i32*.

Allocating objects:

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

Compute address to a struct field:

```
;adress computation for selection in structure (pointers):
%1 = getelementptr %struct.A* %a, i64 0, i64 2
```

The first "index by zero" `i64 0` dereferences the pointer. The second index `i64 2` indexes the result of that in position 2 (i.e. the function pointer in A)

Loading from an address:

```
;load from memory
%2 = load i32(i32)* %1
```

We know load the function pointer whose address we computed before from memory.

Indirect call:

```
;indirect call
%retval = call i32 (i32)* %2(i32 42)
```

The function at the pointer is called and the return value stored.

- the `%a`, `%1` etc. are local variables
- `getelementptr` abstracts away from the address computation (which may differ depending on the target). It takes a type context (e.g. `%struct.A*`) and a base pointer of that type (e.g. `%a`)

Retrieving LLVM Code

Retrieving IR code:

```
# the llvm-cxxfilt call at the end ungarbles the names
clang -O1 -S -emit-llvm source.cpp -fno-discard-value-names -o /dev/stdout | llvm-cxxfilt
```

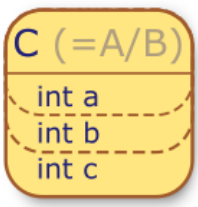
Retrieving memory layout:

```
clang -cc1 -x c++ -v -fdump-record-layouts -emit-llvm source.cpp
```

Object memory layout

Consider the simple example:

```
// Some class structure:
class A                     { public: int a; int f(int); };
class B : public A { public: int b; int g(int); };
class C : public B { public: int c; int h(int); };

// Some method implemenation:
int B::g(int p) {
        return p+b;
}

// ... somewhere in the code:
C c;
c.g(42);
```



LLVM IR compiles the class structure to

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

and the instructions to

```
%c = alloca %class.C
; compiler found B to have a matching method signature for g(42)
%1 = bitcast %class.C* to %class.B*
; receiver of the call becomes the first parameter; g is statically known
%2 = call i32 @_g(%class.B* %1, i32 42)
```

The method body is compiled to

```
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
}
```

The implicit contract is: whoever calls the method has to make sure that the `%this` parameter passed needs to be compatible with the B memory layout.

Implementing Single Dispatching

Multiple ways to do it:

Java Interpreter-Style: Manual Search through Super Chain

Requires a call to some `@__dispatch(...)` function with the concrete type and the method signature on every call.

This is inefficient and used to be quite a problem in the early Java days!

Java Hotspot/JIT Style: cache dispatch result

If we recently computed the required dispatch target, we obtain it from some cache and reuse it.

C++ style: Precomputing Dispatch results in Tables

The static way of doing things. keep a table where for each class and each possible method, the memory address to dispatch to is stored.

Several ways how to do this concretely:

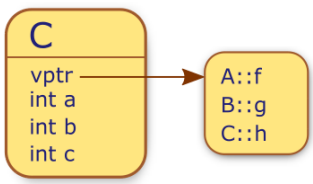
- full 2-dimensional table (all classes x all methods)
- 1-dimensional "row displacement dispatch" tables (TODO what is this exactly? Is it relevant)
  - this is a way to save some space
- Virtual tables: e.g. LLVM/GNU C++. Will be treated in more detail here.

Virtual Methods and Virtual Tables in C++

Consider the example like above, but with *virtual* functions `f`, `g`, `h`. Recall that in C++, methods need to be explicitly marked `virtual` to be overridable, i.e. for dynamic dispatch to work (TODO is this true?).

```
// Some class structure:
class A { public: int a; virtual int f(int);
                                                virtual int g(int);
                                                virtual int h(int); };

// B's g and C's h are *implicitly virtual* since A's g and h are virtual
class B : public A { public: int b; int g(int); };
class C : public B { public: int c; int h(int); };

// ... somewhere in the code:
C c;
c.g(42);
```

Memory layout now: In the class C layout, the first "field" is a pointer to the virtual table, called the `vptr`.



In LLVM: the %class.A struct has a pointer to an array of function pointers as first field.

```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 } ; the [12 x i8] stands for %class.A
%class.A = type { i32 (...)**, i32 }
```

The instructions get compiled to:

```
%c.vptr = bitcast %class.C* %c to i32 (%class.B*, i32)*** ; vtbl
%1 = load (%class.B*, i32)*** %c.vptr ; dereference vptr
%2 = getelementptr %1, i64 1              ; select g()-entry
%3 = load (%class.B*, i32)** %2           ; dereference g()-entry
%4 = call i32 %3(%class.B* %c, i32 42)
```

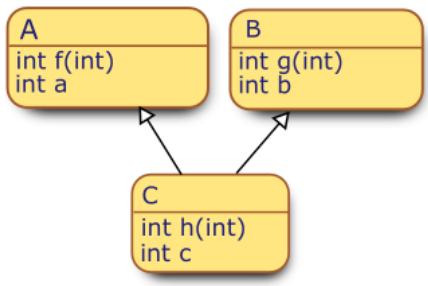
Step by step:

- convert C-pointer to vtable pointer
- dereference the pointer and select the second entry, corresponding to `g()`
- dereference the `g()` entry and call it with the `%c` interpreted as B-pointer

Comment: this code is longer than what Hotspot JIT produces; but *much* shorter than dynamic dispatch binary search.

Multi-Inheritance

Modify our example: `A` and `B` become independent, and `C` inherits from both.



Problem: Memory layout! We would like child class fields to be directly preceded by the parent class fields. But this is not possible with multiple parents.

Multiple Inheritance Layout difficulties

Example:

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

Now `B` has an offset in `C`: $\Delta B = |i32| = 4|i8|$.

We will now see: multi-inheritance makes the implementation harder on the compiler side.

Implicit call example

An implicit casts now potentially needs to add an offset:

```
; C++ code: B* b = new C();
%1 = ...                                                ; pointer to new C object
%2 = getelementptr i8* %1, i64 4 ; select B offset in C
%b = bitcast i8* %2 to %class.B*
```

Calling instance method example

Now create a `C` object and call a `B` method on it:

```
C c;
c.g(42);
```

LLVM casts the C pointer to an i8 pointer, selects the B offset, and calls the method with this pointer.

```
%c = alloca %class.C
%1 = bitcast %class.C* %c to i8*
%2 = getelementptr %i8* %1, i64 4
%3 = call i32 @_g(%class.B* %2, i32 42) ; g is statically known
```

Ambiguities from Multi-Inheritance

Now: Also for the users of the language, the implementation becomes harder: Multi-inheritance introduces ambiguities!

Easiest example:

```
class A { public: void f(int); };
class B { public: void f(int); };
class C : public A, public B {};

// ...somewhere in the code:
C* pc;
pc->f(42);
```

Which version of the `f` method is called?

Two approaches:

- explicit qualification (programmer must write e.g. `pc->A::f(42);` to make clear which method is meant)
- automagical resolution: linearization (the compiler has some rules on which parent classes take precedence)

Linearization

Goal: go back from the current set view of superclasses (due tu Multi-Inheritance) to a linear set view.

Define two relations: *inheritance* and *multiplicity*.

> Principle 1. *Inheritance*
> "⇾" points from childs to parents. Example: C(A, B) induces C ⇾ A and C ⇾ B

> Principle 2. *Multiplicity*
> "⇢" points from a parent with higher precedence to the next sibling with lower precedence. Example: C(A, B) induces A ⇢ B. (In the script also written as a simple arrow A → B)

Obeying the two principles means to find a linearization where for any A, B such that A ⇢ B or A ⇾ B, A comes before B in the linearization.

General rules:

- inheritance relation applies uniformly (i.e. not a different inheritance structure for, say, fields and methods)
- the method by which a linearization is created is also called *Method Resolution Order* (MRO)
- Linearization can only be best-effort

Simple Linearizations

Leftmost Preorder DFS as MRO

LPDFS = leftmost preorder depth-first search. More or less the simplest way to achieve linearization, but not very good!

Example: A(B, C) B(W) C(W).
Then $L[A] = ABWC$: Inheritance is violated, W comes before C!

Classical Python objects (before 2.1) use this approach.

LPDFS(DC) as MRO

LPDFS(DC) = LPDFS with Duplicate Cancellation. Idea: perform LPDFS, once you encounter a class for the second time you delete its first occurence and switch to the second one.

This fulfills inheritance. It has another odd behaviour though:

Example: A(B, C) B(V, W) C(W, V).
Then $L[A] = ABCWV$.

Multiplicity is not fulfilled, but that's fine since it's not fulfilable. However a bit odd: B comes before C, yet W comes before V (even though the other way around would work too!).

New Python 2 objects (after 2.2) use this approach.

Reverse Postorder DFS as MRO

Before: DFS leftmost preorder. Now: RPRDFS = rightmost postorder DFS, and reverse the result.

Example: A(B, C) B(F, D) C(E, H) D(G) E(G) F(W) G(W) H(W).
Then $L[A] = ABFDCEGHW$. Everything is fine!

Another example: A(B, C) B(F, G) C(D, E) D(G) E(F).
Then $L[A] = ABCDGEF$. Multiplicity violated! G comes before F, even though there is a clear precedence of F over G.

Refined RPRDFS as MRO

Idea: in clear cases as we just saw, introduce an additional *inheritance* edge where before there was only a multiplicity edge, if a conflict is detected. Then rerun RPRDFS on the refined graph.

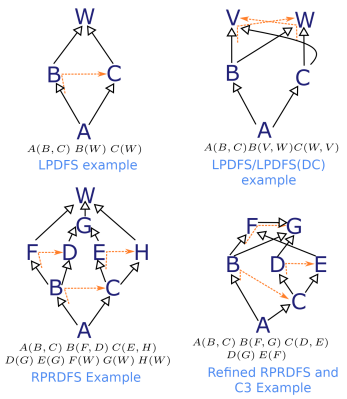
Example from above becomes:
$L(A) = ABCDEFG$

CLOS = Common Lisp Object System uses Refined RPDFS.

Another principle: ***Monotonicity***

> Principle 3. *Monotonicity*
> If C_1 comes before C_2 in C's linearization, then C_1 should come before C_2 in the linearization of every child of C.

Refined RPDFS violates this principle! Counterexample = the example above: Recall
$L(A) = ABCDEFG$. However, $L(C) = CDGEF$, where G comes before F!



C3 Linearization

Assume we have a class C with superclasses $B_1, \ldots, B_n$ in the *local precedence order* $C(B_1 \cdot \cdots \cdot B_n)$.

Define

$$L[C] = C \cdot \bigsqcup(L[B_1], \ldots, L[B_n], B_1, \ldots, B_n)$$

where $\bigsqcup$ is the *abstract inheritance join operator*, i.e. an operator which joins multiple inheritance relations, defined as follows:

$$\bigsqcup_i (L_i) = c \cdot \bigsqcup_i (L_i \setminus c), \text{ where } c = head(L_k), k = \min\{k \mid \forall j : head(L_k) \notin tail(L_j)\}$$

If no such k exists, the operator fails.

C3 Linearization Example

Again, the example from above: A(B, C) B(F, G) C(D, E) D(G) E(F).

Work your way backwards:

- $L[G] = G$, $L[F] = F$, $L[E] = EF$, $L[D] = DG$
- $L[B] = B \bigsqcup \{F, G, FG\} = BF \bigsqcup \{G, G\} = BFG$
- $L[C] = C \bigsqcup \{DG, EF, D, G\} = CD \bigsqcup \{G, EF, G\} = CDGEF$
- $L[A] = A \bigsqcup \{BFG, CDGEF, B, C\} = AB \bigsqcup \{FG, CDGEF, C\} = ABC \bigsqcup \{FG, DGEF\} = ABCD \bigsqcup \{FG, GEF\} = \bot$
  - the linearization fails: F is in the tail of GEF, G is in the tail of FG

$\Longrightarrow$ C3 Reports a *monotonicity violation* due to `A(B, C)`: we would not be able to guarantee compatible linearizations for children and parents.

C3 Linearization Adoption

C3 linearization is used, for example, by Python 3, Raku, and Solidity.

Linearization vs. Explicit Qualification

Qualification has the advantages of being

- more flexible
- and avoiding potentially awkward/unexpected linearization effects

Linearization has the advantages of

- not requiring switch/duplexer code to disambiguate between choices (QUESTION: What would be an example of that?)
- not requiring explicit naming of qualifiers
- given a *unique super reference*
- reduces the number of multi-dispatching conflicts

Some languages with automatic linearization are, as mentioned, CLOS, Solidity, Python 3, and Raku (formerly Perl 6) .

Automatic Linearization, is a prerequisite for Mixins in the presence of Multi-Inheritance, since these usually require a static super type.

Week 10 - C++ Multi-Inheritance and Dynamic Dispatching

C++: Multi-Inheritance and Dynamic Dispatching

What happens if we combine Multi-Dispatching and Virtual Tables?

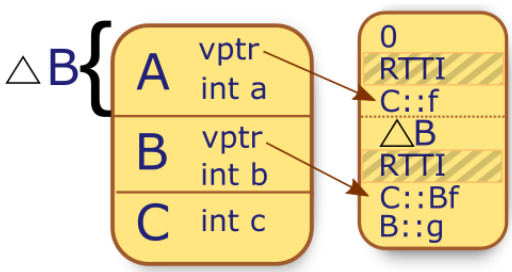
If $C(A, B)$, in the memory layout, we would need a vptr before A as well as before B!

C++ example

Recall this example, but now with a virtual overwritten method `f`.

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

// ...somewhere in the code:
C c;
B* pb = &c;
pb->f(42);
```

At the point where `pb` is of type `B*`, all the compiler is supposed to use is the fact that this points to an object of type `B`. So the vptr at the beginning of the memory layout of this `B` object must point to a vtable which contains the `C::f` version, not the `B::f` version.




```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

Basic Virtual Tables

Basic virtual tables are made up of

- the *offset to top*, i.e. how much higher you need to go to reach the enclosing object's top, starting at the vptr (e.g. ΔB)
- *typeinfo pointer* to an RTTI object (= Run-Time Type Information)
- multiple *virtual function pointers*, used to resolve virtual methods

When multiple inheritance is used, *virtual tables are composed*: The vptrs are pointing to the corresponding *virtual subtables*.

(QUESTION: Is this distinction into vtables/v-subtables actually relevant?)

With this setup, casting preserves the link between an object and its corresponding virtual subtable.

Finding the vtables of a program

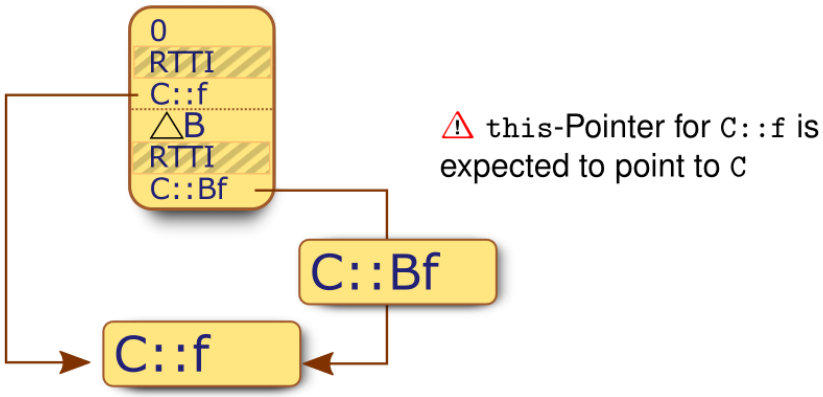
The vtables of a compilation unit are output by the command

```
clang -cc1 -fdump-vtable-layouts -emit-llvm source.cpp
```

Thunks

Casting issues when calling virtual methods

The following problem can occur when casting: Imagine the overwritten `C::f` method is called from an object of type `B`. Then the `C::f` methods expects as this reference an object of type `C`, so the `B` object must be cast to `C`. This information is not available, however, in the vtable as presented until now!



Thunks to the rescue

Thunks are "trampoline methods" that adapt the `this` reference before delegating to the original virtual method implementation.

In our example, in the B-in-C-vtable, `f(int)` is represented by the thunk `_f(int)`. It adds the compiletime constant ΔB to `this` and then calls `f(int)`.

Compiled code:

```
define i32 @__f(%class.B* %this, i32 %i) {  ; thunk definition
        %1 = bitcast %class.B* %this to i8*
        %2 = getelementptr i8* %1, i64 -16      ; subtract ΔB = size(A) = 16
        %3 = bitcast i8* %2 to %class.C*        ; interpret as C pointer...
        %4 = call i32 @_f(%class.C* %3, i32 %i) ; ...and call the f method
        ret i32 %4
}
```

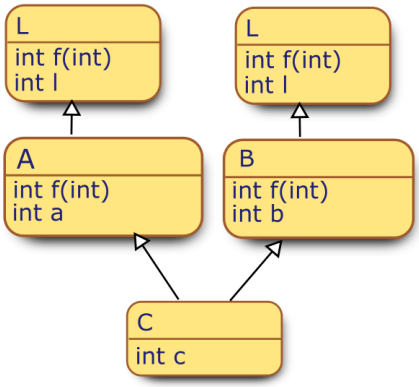
Common Ancestors

What if there are common ancestors? Where to place them in the memory layout? Classical example: Diamond problem.

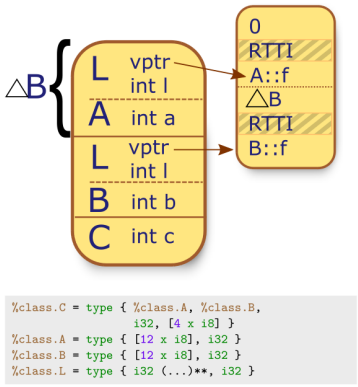
(Maybe this means you should rethink your class structure... But if the language allows it, the compiler must nevertheless handle that case.)

Standard C++ approach: Duplicated Bases

Standard C++ Multi-Inheritance: conceptually, duplicates of common ancestors.



Memory layout:




```
%class.C = type { %class.A, %class.B,
i32, [4 x i8] }
%class.A = type { [12 x i8], i32 }
%class.B = type { [12 x i8], i32 }
%class.L = type { i32 (...)**, i32 }
```

One can see that only `L` needs a vptr.

Examples of Ambiguities of Duplicated Bases

The following code fails to compile:

```
C c;
L* pl = &c; // 'L' is an ambiguous base of 'C'
```

There are two L objects stored inside `c`, the compiler wouldn't know to which to point!

This works:

```
C c;
L* pl = (B*)&c;
```

This also fails:

```
C c;
L* pl = (B*)&c;
C* pc = (C*)pl; // 'L' is an ambiguous base of 'C'
```

This works:

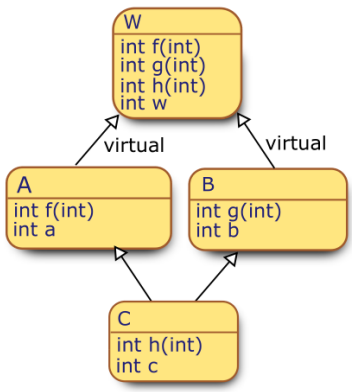
```
C c;
L* pl = (B*)&c;

// Even the call is allowed (other than c.f(42)): On an L object,
// calling f(...) is unambiguous (the ambiguity would already
// kick in during the cast to L if we didn't resolve it)
pl->f(42);

// For the cast back to C, give the compiler a hint (static casts
```

```
// need compile-time constant offsets!)
C* pc = (C*)(B*)pl;
```

Virtual base clases: Allow Common Bases

C++ allows diamond-pattern-style shared base classes with the `virtual` keyword:

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



Ambiguities can occur (e.g. if `f` is overwritten in both `A` and `B`), resolved by explicit qualification (`pc->B::f`).

Memory Layout with *Offset to Virtual Base* entries

In the memory layout of `C`, the shared base class `W` cannot be placed both

- directly above `A`
- and directly above of `B`

This violates the assumption that *the parent can be found within an offset of its child which is constant throughout each occurence of the particular parent in some inheritance relation*. We therefore have to drop this assumption to be able to layout shared base classes in memory.

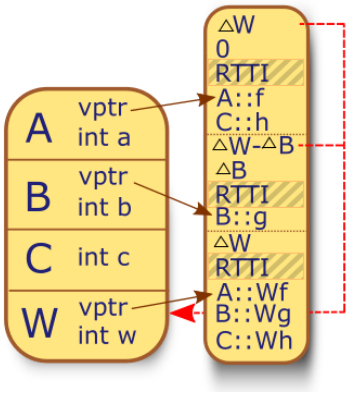
Each child of the virtual base class stores an *offset to virtual base* (`vbase_offset`) entry in the v-subtable.

Disadvantage: Each time you access a field of a virtual parent, you will have an indirect memory access!

Example Memory Layout with Offsets to Virtual Bases

Memory layout (in this particular case):

- place `W` at the end of the `C` representation
- In the vtables for "`A` in `C`" and "`B` in `C`" include offsets to the virtual base class `W` within `C` (in addition to the "offset to top" entries)
  - e.g. from `A`, the offset is $\Delta W = |A| + |B| + |C|$
  - from `B`, the offset is $\Delta W - \Delta B = \Delta W - |A| = |B| + |C|$



Some more details: see [Week 10.2 - VTable experiments]

Dynamic Casting

Since there is no guaranteed offset between virtual bases and their childs, *static casting* becomes impossible. Example: If `C(A, B)` and `D(C, B)`, then in the `C` layout we have $A|B|C|W$ and in the D layout, $A|B|C|B|D|W$. So a `W` pointer cannot be statically cast into a `C` pointer!

```
C c;
W* pw = &c;
C* pc = (C*)pw;                             // This gives a compiler error
C* pc = dynamic_cast<C*>(pw); // This works (uses offset-to-top fields)
```

Virtual Thunks

Recall that thunks added a constant (statically known) offset to the this reference before calling the original method.

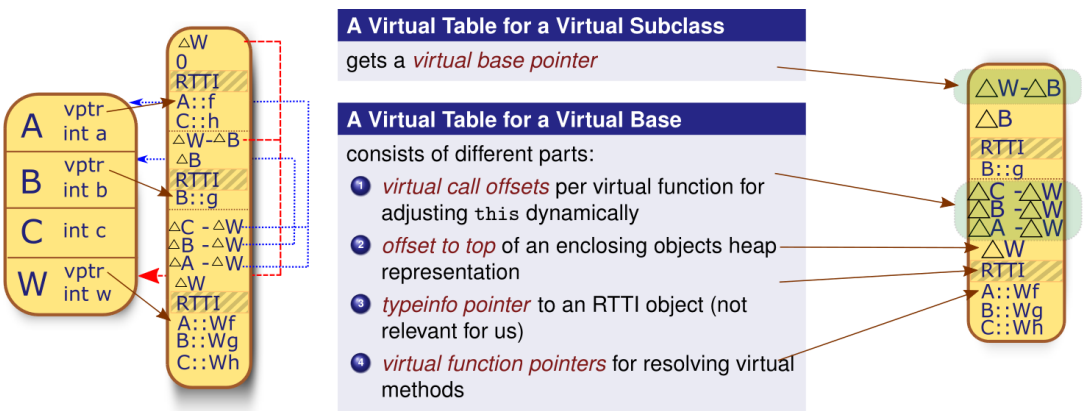
This doesn't work anymore in the virtual base class setting, due to casts being dynamic. Instead we need *virtual thunks*, which obtain the offset by which to translate the `this` pointer from the vtable.

The virtual table is extended by one additional entry for each method this is relevant for: The entry corresponding to a method pointer, e.g. `A::Wf`, contains by how much to shift the `this` pointer in the virtual thunk (e.g. `DeltaA-DeltaW` to get from W to the top, then down to A). These offset entries are called *virtual call offsets* (`vcall_offset`).

(Recall notation: `A::Wf` means calling the `A::f` method from a `W` pointer).

Complete convention for virtual subtable memory layout

- entries 0 to n: virtual function pointers
- entry -1: RTTI pointer
- entry -2: offset to top `offset_to_top`
- entry -3: offset to virtual base `vbase_offset` (in case there is a virtual base)
- entry $-4-i$ or $-3-i$, i from 0 to n: offsets for virtual thunks `vcall_offset`



C++ Memory Layout: Compiler/Runtime assumptions

Compiler generates:

- *one codeblock* per method
  - i.e. not different codeblocks depending on the type (like e.g. Rust?)
- *one virtual table* per class composition
  - referencing the *most recent* implementations of methods ("of a unique common signature"). I.e. single-dispatching.
  - containing sub-tables for the composed sub-classes,
  - top-of-object offsets per sub-table,

- virtual base offsets and virtual call offsets per method/subclass if needed

Runtime behaviour:

- globally create vtables at startup (copied in from binary)
- creating new object: allocate memory, call constructor; constructors stores vtable pointers in the objects
- method calls: call methods *statically* or *dynamically from vtables*; *unaware* of real class identity
- dynamic casts (usually downcasts): use offset-to-top fields.

Advantages and Disadvantages of Multi-Inheritance

Pros of *Full Multiple Inheritance* (FMI):

- Removes an (unneeded? inconvenient?) inheritance constraint
- Can be convenient in common cases
- Diamond patterns may occur, but are not as frequent as it seems from the discussion around it

Pros of *Multiple Interface Inheritance* (MII):

- simpler to implement
- already expressive (enough?)
- using FMI too frequently often considered a flaw in the class hierarchy design

Sidenote about Applicability (MS VC++)

The discussion about implementation of FMI applies to GNU C++ and LLVM.

In Microsoft's Visual C++, FMI is implemented a bit differently however:

- split virtual table into several smaller tables
- keep a *virtual base pointer* (`vbptr`) in the *object representation* which points to the virtual base of a child class

---

Week 11 - Mixins and Traits

Mixins and Traits

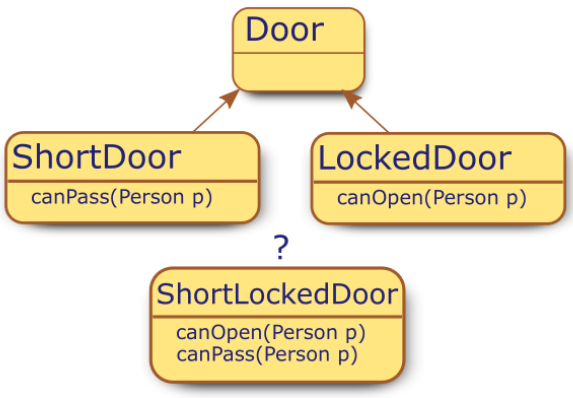
Discussion: What do we want to achieve?

Our ultimate goal: enable some kind of code sharing to avoid unnecessary code duplication.

- in OO systems: code sharing usually enabled via *inheritance* (which comes in different flavours)
- all inheritance-based systems tackle *composition* and *decomposition* problems

Example: ShortLockedDoor

Example: in an Adventure game, we have `ShortDoors` (that only small persons can pass), and `LockedDoors` (that only persons with a key can pass). Sometimes later in the game, we want to introduce `ShortLockedDoors` that combine these two characteristics.



- If multi-inheritance is available: trivial!
- Without MI: Aggregation + SI construction possible (quite involved)
  - have a `Doorlike` interface that provides both `canOpen()` and `canPass()`
  - `Door`, `Short`, `Locked` implement `Doorlike`: for the methods that are not relevant to them, they just always return true

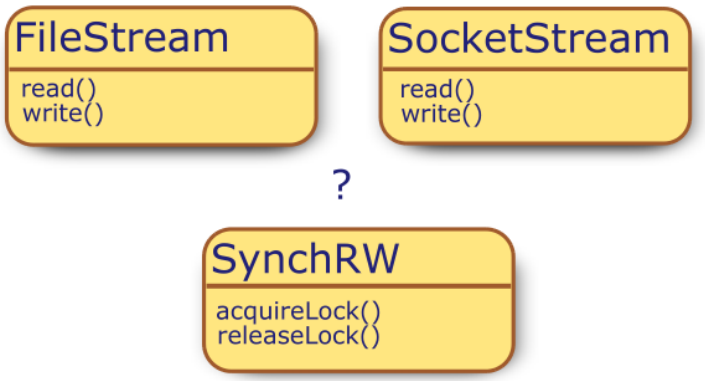
- A `Door` has a list of `Doorlikes` and aggregates their results

Problems:

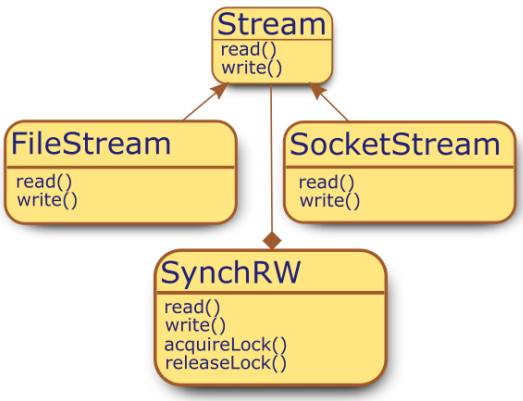
- `Door` must take care of chaining
- The `Doorlike` interface must anticipate all the different flavours of doors that are to come
- All the `Doorlike` implementing classes must return a default value for all types of door attributes not relevant to them

Example: Threadsafe Streams Wrapper

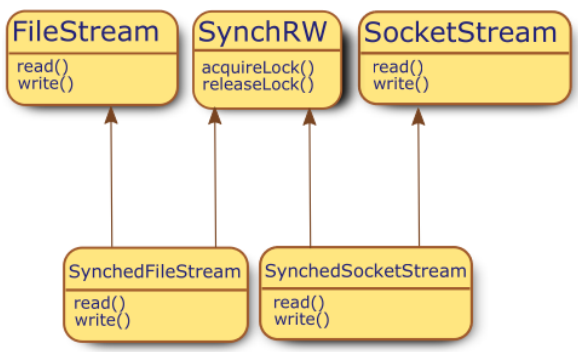
Example: we have a hierarchy of Stream classes, for example a `FileStream` and a `SocketStream` (both with `read`/`write` methods). We want to introduce a threadsafe version of streams `SynchRW` which adds some locking code around the `read`/`write` calls of a stream.



- Multi-Inheritance not directly applicable: many-to one instead of one-to many relation!
  - we want many streams to adapt the features of one wrapper class, not one wrapper class to adapt the features of many parents
- Aggregation solution: SynchRW has a `Stream` internally, which can then be specialized to `FileStream`, `SocketStream`, etc.
  - Undoes specialization, i.e. does not fit into the hierarchy: there is now no way to express that we want, e.g., a threadsafe `FileStream`
    - QUESTION: we could express this with a generic `<S: Stream>` parameter, right?
  - Only works here because there is a common ancestor `Stream` of `FileStream` and `SocketStream`

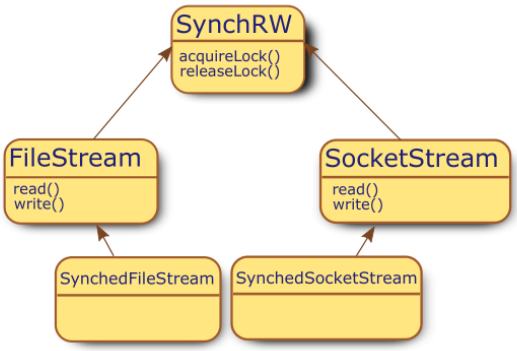


- Duplication MI solution
  - Have `SynchRW` independently from the stream classes
  - introduce child classes `SynchedFileStream`, `SynchedSocketStream` that inherit from both their respective stream type and `SynchRW`
  - Problem: code reusage limited, a lot of duplication is necessary!o
    - `read`/`write` need to be re-implemented identically for every wrapper



- SynchedRW as superclass solution
  - Place `SynchRW` as superclass of the two stream classes

- Make the streams implement read/write with and without locking; let them have some internal flag that indicates whether this feature is activated
- Have subclasses of the streams that simply activate said flag
- Problem: dodgy design, since the `SynchRW` methods are way too high up in the hierarchy; also, thread-safety functionality must be anticipated when writing the Stream classes



(De-)Composition problems as the key question

All the problems we saw are centered around

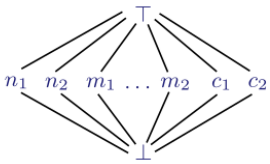
- relationships between subclasses/base classes
- how to set up a hierarchy of classes
- how to minimize code duplication

Key question: "How do I distribute functionality over a hierarchy".
$\Longrightarrow$ *functional decomposition*

Abstract Class Hierarchy Formalism

Assume we have the following sets:

- a countable set of *names* $\mathcal{N}$
- a countable set of *method bodies* $\mathbb{B}$
- a set of *classes* $\mathcal{C}$
- the "flat lattice" of *bindings* $\mathcal{B} = \mathbb{B} \cup \mathcal{C} \cup \{\top, \bot\}$
  - $\bot$ means "abstract" = no implementation
  - $\top$ means "in conflict" = more than one implementation
  - $\mathcal{B}$ is partially ordered by $\bot \sqsubseteq b \sqsubseteq \top$ (hence the name "flat lattice")



> **Definition** *Class* $\in \mathcal{C}$.
> A map $c : \mathcal{N} \to \mathcal{B}$ is called a *class*.
> I.e. a class maps names to bindings.

QUESTION: apparently, c only maps a subset of $\mathcal{N}$ to bindings, right? And $pre(c)$ denotes this subset?

> **Definition** *Interface/abstract/concrete classes*.
> A class c is called
>
> - *interface* if $c(n) = \bot$ for *all* $n \in pre(c)$
> - *abstract class* if $c(n) = \bot$ for *some* $n \in pre(c)$
> - *concrete class* if $c(n) \notin \{\bot, \top\}$ for all $n \in pre(c)$

> **Definition** *Family of classes* $\mathcal{C}$.
> The set of all mappings from names to bindings is called the *family of classes* $\mathcal{C}$

Binary operators on $\mathcal{C}$

Several possibilities to compose two elements of $\mathcal{C}$:

Symmetric join $\sqcup$

In $(c_1 \sqcup\ c_2)$, a name n is mapped to a binding if it defined in exactly one of the two classes, or if it is defined in both, and the definitions coincide. If it is defined differently in the two classes, the name is mapped to $\top$ = "in conflict".

$$(c_1 \sqcup c_2)(n) = \begin{cases} c_1(n) & \text{if } c_2(n) = \bot \text{ or } n \notin pre(c_2) \\ c_2(n) & \text{if } c_1(n) = \bot \text{ or } n \notin pre(c_1) \\ c_1(n) & \text{if } c_1(n) = c_2(n) \\ \top & \text{otherwise} \end{cases}$$

Asymmetric join $\Cup$

In $(c_1 \Cup c_2)$, the bound names of c_1 override any bound names of c_2. Only the names from c_2 which are not bound by c_1 are left intact.

$$(c_1 \Cup c_2) = \begin{cases} c_1(n) & \text{if } n \in pre(c_1) \\ c_2(n) & \text{otherwise} \end{cases}$$

Examples of Inheritance Models

Smalltalk Inheritance

Smalltalk inheritance is the archetype for how inheritance works in mainstream imperative languages like C# or Java.

Children's methods dominate parent's methods, and a `super` reference to the parent is established.

> **Definition**. *Smalltalk inheritance* $\triangleright$.
> Define the operator binary operator $\triangleright$ on $\mathcal{C}$ by
>
> $$c_1 \triangleright c_2 = \{\texttt{super} \mapsto c_2\} \Cup (c_1 \Cup c_2)$$
>
> Here c_1 is the child class and c_2 is the parent class.

Example: LockedDoor in Smalltalk Inheritance

$$Door = \{canPass \mapsto \bot, canOpen \mapsto \top\}$$

$$LockedDoor = \{canOpen \mapsto 0x4204711\} \triangleright Door$$

$$= \{super \mapsto Door, canOpen \mapsto 0x4204711, canPass \mapsto \bot\}$$

Beta-style Inheritance

In *Beta-style inheritance*, child methods do *not* replace parent methods. Instead, parent methods dominate, and child methods need to be explicitly qualified with the `inner` keyword. Design goal: provide security from replacement of one method by another.

> **Definition**. *Beta-style inheritance* $\triangleleft$.
> Define the operator $\triangleleft$ by
>
> $$c_1 \triangleleft c_2 = \{\texttt{inner} \mapsto c_1\} \Cup (c_2 \Cup c_1)$$

Example of Beta-style inheritance

...in pseudo-Java syntax with an added `inner`, `extension` keywords:

```
class Person {
        String name;
        public String toString() {
                return name + inner.toString();
        }
}

class Graduate extends Person {
        public extension String toString() { return ", Ph.D."; }
}
```

CRTP - Curiously Recurring Template Pattern

CRTP (the *Curiously Recurring Template Pattern*) is a C++ pattern that allows a base class to call a child class method statically (*static polymorphism*), by making a base class defined via a template where the child class can be filled in.

```
template <class D>
struct Base_class
```

```
{
        void base()
        {
                // Allows to call non-static methods of D
                static_cast<D*>(this)->derived();
        }
        static void static_base()
        {
                // Allows to call static methods of D
                D::static_derived();
        }
}
struct Derived
{
        void derived();
        static void static_derived();
}
```

Disadvantages are that

- the base class is defined as a template
- some spots are left open in an algorithm's definition (really a problem?)
- most problematic: struggles with repeatedly overwritten methods

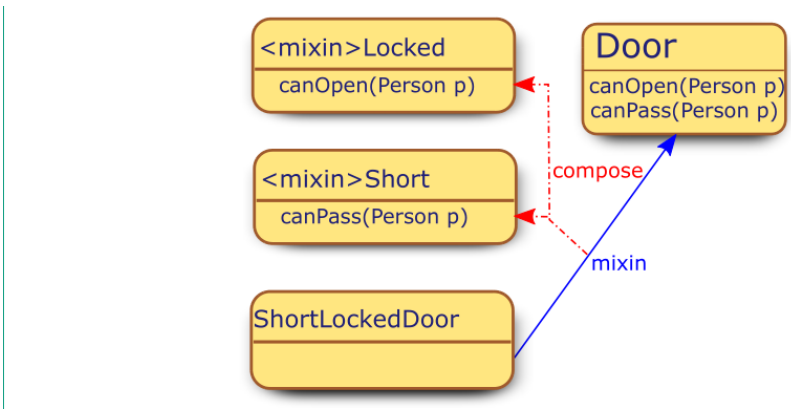
Mixins

Now that we saw some different inheritance models, let's find out what we really want to solve problems like in the above examples.

Example: ShortLockedDoor with Mixins

Idealistic goal: "cut out" the functionality from `LockedDoor` and `ShortDoor` and insert it in `ShortLockedDoor : Door`, *without creating a multi-inheritance diamond*.

Idea: have *mixins* `Locked` and `Short`, which can be composed to create the `ShortLockedDoor`.

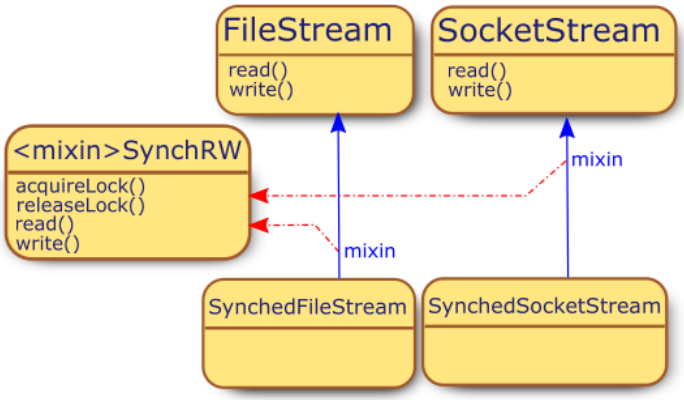


Code (pseudo-java with added mixin functionality):

```
class Door {
        boolean canOpen(Person p) { return true; };
        boolean canPass(Person p) { return p.size() < 210; };
}

// Can define mixins:
mixin Locked {  boolean canOpen(Person p){ ... } }
mixin Short {  canPass(Person p){ ... } }

// Can create the short and locked doors:
class ShortDoor = Short(Door);
class LockedDoor = Locked(Door);

// Can compose mixins to create short locked doors:
mixin ShortLocked = Short o Locked;
class ShortLockedDoor = Short(Locked(Door));
class ShortLockedDoor2 = ShortLocked(Door);
```

Threadsafe Stream wrappers with Mixins

Have `SynchRW` as mixin.



- avoids `read`/`write` code duplication
- keeps specialization (i.e. class hierarchy)
- does not need multi-inheritance

Abstract Mixin Formalism

> **Definition** *Mixin*.
> The mixin constructor $mixin : \mathcal{C} \to (\mathcal{C} \to \mathcal{C})$ is defined by
>
> $$mixin(c) = \lambda x.\, c \rhd x$$
>
> $mixin$ is a curried version of $\rhd$. Methods of the mixin class dominate over existing methods.

The mixin operator is a *unary higher-order type expression*, which in some languages can be created.

Mixin operators can be composed:

$$mixin(Short) \circ mixin(Locked) = \lambda x.\, Short \rhd (Locked \rhd x)$$

Mixins on Implementation Level: Non-Static `super`

Imagine you have

```
class ShortDoor = Short(Door);
class ShortLockedDoor = Short(Locked(Door));
```

Sometimes `Short` is "mixed into" `Door`, and sometimes into `Locked`; i.e. the `super` reference from `Short` certainly cannot be static.

Some dynamic `super` dispatching therefore has to be implemented for mixins to work.

Simulating Mixins with C++ Multi-Inheritance

Idea how to simulate mixins in C++:

- have a template for the mixin
- the template takes a generic `Super` class parameter

Example of C++ mixin simulation

```
// Mixin that makes a stream threadsafe:
template <class Super>
class SynchRw : public Super
{
        public:
        virtual int read()
        {
                acquireLock();
                int result = Super::read();
                releaseLock();
                return result
        }
        // ... rest of the implementation
};
// Mixin that makes a stream buffered:
template <class Super>
class BufStream: public Super { ... };

// Composed stream:
class BufSynchFileStream : public SynchRw<BufferedStream<FileStream>> {};
```

"True" Mixins vs. C++ Mixins

C++ mixins can be seen as a coding pattern; the C++ type system is not modular, hence the mixins have to stay source code.

With True Mixins, *super* is natively supported. Mixins are composable (but so they are in C++?)

Generally: mixins require a linearization, and the exact sequence of mixins matters.

Native Mixins in Ruby

Ruby is a language that supports native mixins

Example: ShortLockedDoor with Mixins in Ruby

Some setup:

```
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
```

Mixins are called `module`s in Ruby. Define `Short` and `Locked` mixins:

```
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
```

Mixins can be mixed into a class at class definition:

```
class ShortLockedDoor < Door
        include Short
        include Locked
end
```

Composed Mixins can be created:

```
module ShortLocked
        include Short
        include Locked
end
```

Objects can be extended with mixins even at runtime:

```
# At-runtime extension with a mixin
d = Door.new
d.extend ShortLocked

# Alternatively:
d = ShortLockedDoor.new
```

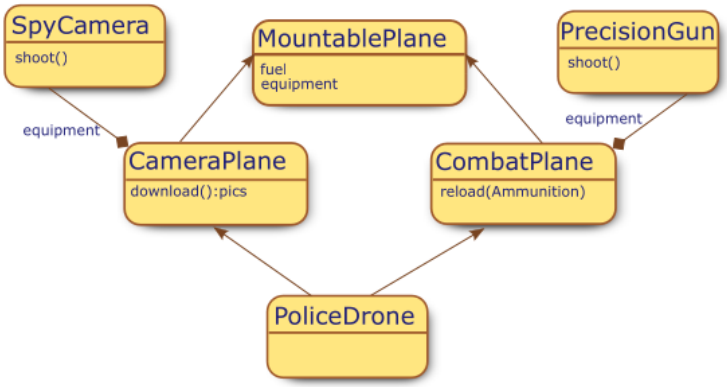
Discussion: Is Inheritance *The Principle* for Resuability?

Drawbacks of Inheritance

Lack of Control

Inheritance may not give sufficient control over which code to share and which not to share.

Example: `PoliceDrone`.



We would probably want to share the *fuel* attribute - but rather not share the *equipment* field, otherwise we might want to `shoot` a photo and instead `shoot` the precision gun!

In multi-inheritance, common base classes are usually either shared or duplicated. Such a fine-grained specification as we would want here is not possible.

Inappropriate Hierarchies

Inheritance does not allow to *remove* parent methods.

Example: a `LinkedList` which offers `add` and `remove` is extended by a `Stack` which offers `push` and `pop`. The `add` and `remove` methods should not belong to the methods offered by `Stack` - they turn obsolete. Yet there is no way to remove them (compare with *Liskov Substitution principle*: we should be able to insert a `Stack` everywhere where a `List` is used).

(Note: maybe it's not a good idea to have `Stack` extend `LinkedList`?)

Is Implementation Inheritance an *Anti Pattern*?

Example: In Java 8, `Properties` inherits `Hashtable` and therefore exposes `put` and `putAll`. However, this allows to insert non-`String` objects, which is not what the object should be doing.

The documentation says that *the use of `put` and `putAll` is strongly discouraged* and that *the `setProperty` method should be used instead*. However, there is just no way in Java to disallow this wrong usage statically!

Bottom line: *Implementation inheritance as a pattern for code reusage is often misused!*

Traits

The idea behind traits: maybe problems originate from the coupling of *implementation* and *modeling*?

- interfaces seem to be hierarchical
- functionality seems to be modular

Separate: *object creation* $\longleftrightarrow$ *hierarchy modeling* $\longleftrightarrow$ *functionality composition*

1. use interfaces for hierarchical *signature propagation*
2. use *traits* as modules for assembling functionality
3. use classes as frames for entities which can create objects

Abstract Traits Formalism

> **Definition** *Trait* $\in \mathcal{T}$.
> A class without attributes is called *trait*.

Note: for mixins, we also did not have attributes, but this is often done differently. For Traits, there are *explicitly* excluded.

- The *Trait sum* $+ : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ is just the symmetric join:

$$c_1 + c_2 = c_1 \sqcup c_2$$

- *Trait exclusion* $- : \mathcal{T} \times \mathcal{N} \to \mathcal{T}$ allows to exclude names from traits

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

- *Aliasing*: $[\to] : \mathcal{T} \times \mathcal{N} \times \mathcal{N} \to \mathcal{T}$
  allows to introduce an additional name for some binding.

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

- Connecting traits to classes: at the last composition level with the asymmetric join. Connect trait t to class c via $c \Cup t$.

Concepts and Principles of Traits

Trait Composition Principles

- all traits have same precedence (under +), due to $+ = \sqcup$. Disambiguation needs to be done explicitly with aliasing and exclusion. (*Flat ordering*)
- class methods take precedence over trait methods, i.e. classes can override the "default implementations" from traits (since $\Cup$ is used) (*Flattening*)

Conflict treatment:

- via *aliasing* or *exclusion*
- or by just overriding the conflicting method in the final class

Trait-like extensions in common languages

C#: Extension Methods

Idea behind extension methods: uncouple method definitions $\longleftrightarrow$ class bodies (methods can be defined outside classes).

Extension methods allow to

- retrospectively/externally add methods to types
- provide default implementations of interface methods ("poor man's multiple inheritance")

Syntax for extension methods is:

- declare extension methods as static methods within a static class
- declare first parameter as of type to be extended, with a `this` preceding the type, e.g. `this Locked lockedDoor`
- the extension method can now be called in infix form

```
public interface Short {}
public interface Locked {}
public static class DoorExtensions {
        public static bool canOpen(this Locked locked, Person p) { /*... */ }
        public static bool canPass(this Short locked, Person p) { /*... */ }
}
public class ShortLockedDoor : Locked, Short {}

// Somewhere in Main():
d = new ShortLockedDoor();
d.canOpen(new Person()); // calls the method defined above
```

Extension Methods vs. Traits

C# extension methods *can* extend types externally, *but not really traits*:

- no flattening: classes cannot override extension methods
- extension methods cannot be used to implement interface methods
  - e.g. if `Locked` would specify a `bool canOpen(Person p)` method, this would not be considered as implemented

Java 8: Virtual Extension Methods

Java has so-called *virtual extension methods*, which are basically default implementations for interfaces:

```
interface Door {
        boolean canOpen(Person p);
        boolean canPass(Person p);
}
```

```
interface Locked {
        default boolean canOpen(Person p) { /*...*/ }
}
interface Short {
        default boolean canPass(Person p) { /*...*/ }
}
public class ShortLockedDoor implements Short, Locked, Door {}
```

Note that the `Locked` and `Short` default implementations actually override the `Door`, at least that's claimed in the lecture... QUESTION: This does not seem to work as advertised on the slides!

```
HelloWorld.java:18: error: ShortLockedDoor is not abstract and does not override abstract method canPass(Person) in Door
class ShortLockedDoor implements Door, Short, Locked {}
^
```

Drawback: does not override methods from *abstract* classes.

Implementation of virtual extension methods: An extra interface phase is added to the name resolution of `invokevirtual`.

Traits as General Composition Mechanism

What we would actually want from traits: separate class generation ⟷ hierarchy specification ⟷ functional modeling via

1. modeling the hierarchical relations with interfaces
2. composing functionality with traits
3. in classes, adapt functionality to interfaces and add state via glue code

Goal: *simplified multiple inheritance without adverse effects*!

Important: if there is any *precedence* involved, what you have is probably more like mixins than traits (e.g. "Scala traits").

"Real Traits" in Squeak

Squeak is a Smalltalk implementation (dialect?) which is extended by a trait system.

Squeak basic syntax

Syntax:

```
"Declare method `someMethod` with two parameters `param1` and `param2`"
someMethod: param1 and: param2

"Declare variables `someVar1` and `someVar2`"
| someVar1 someVar2 |

"Assignment"
someVar1 := expr

"Send the message `someMessage` with content `someContent` to `someObject`"
"Think of method call: someObject.someMessage(someContent)"
someObject someMessage:someContent

"Line terminator: Dot"
.

"Return statement"
^ expr
```

Example: Threadsafe Streams Wrapper in Squeak

```
"Some stream base class"
Trait named: #TRStream uses: TPositionableStream
        on: aCollection
                self collection: aCollection.
                self setToStart.
        "`next` method that returns the next element (details unimportant)"
        next
                ^ self atEnd
                ifTrue [nil]
        ifFalse: [self collection at: self nextPosition].

"`TSynch` trait which implements acquiring/releasing a semaphore"
Trait named: #TSynch uses: {}
```

```
        acquireLock
                self semaphore wait.
        releaseLock
                self semaphore signal.

"Combining the two traits:
        - alias the #next method as #readNext so we can redefine #next
        - use trait sum + to add the two traits
"
Trait named: #TSyncRStream uses: TSynch+(TRStream@(#readNext -> #next))
        "Define a new `next` method which wraps reading with the lock"
        next
                | read |
                self.acquireLock.
                read := self readNext.
                self releaseLock.
                ^ read.
```

Traits vs. Mixins vs. Class Inheritance

Type expressions involved:

- Mixins: Second-order type operators + linearization
- Traits: Finegrained flat-ordered module composition
- Class inheritance: Defines (local) partial order on precedence of types wrt. MRO
- Combination of principles, e.g. in extension methods, templates, etc.

Traits and Mixins: Differences

- Traits are applied *in parallel*, mixins *sequentially*
  - and therefore also: trait composition is *unordered* and avoids linearization (and therefore e.g. unexpected/strange overloading/overriding)
- Traits *never contain attributes*: avoiding state conflicts
- Traits cause *glue code to be concentrated* in single classes

Summary: Mixins and Traits

Mixins = "low-effort alternative to multi-inheritance"
Mixins lift type expressions to second-order type expressions

- "just open up your type expressions to lambda expressions, and you already have mixins"

Traits:

- fine-grained control of composition of functionality
- in native trait languages: separation of composition of functionality ⟷ specification of interfaces

---

Week 12 - Prototybe-Based Programming in Lua

Prototype-Based Programming in Lua

Core idea: "why bother modeling types for my quick hack?"

Our example prototype-based language: *Lua*.

Basic Language Features

- implicitely defined global variables
- no type annotations: dynamic typing
- semicolons, or even new lines to separate statements, can also be left out

```
--[[set some
variables
--]]
```

```
s = 0
p = s+1 p = s+3 -- multiple statements in one line allowed
```

Types in Lua

Every value carries a type that can be retrieved with the `type` function.

```
type(true)         -- boolean
type(42)           -- number
type("Forty-Two")  -- string
type(type)         -- function
type(nil)          -- nil
type({})           -- table
-- two more basic types: `userdata` (arbitrary C data) and `thread`
```

Strings can be concatted:

```
s = "A " .. 1 .. " B " .. 2 -- "A 1 B 2"
```

Using a string as a number parses it:

```
a = "42" + "1"  -- 43
b = "42" + 1    -- 43
```

Types associated with a variable name are not static:

```
a = 1
type(a)   -- number
a = true
type(a)   -- boolean
```

All unitialized variables have the type `nil`:

```
type(some_uninit_variable) -- nil
```

Functions

Functions are first-class citizens, i.e. can be stored into variables, etc.

Syntax:

```
-- <--> python lambda expression style
fun1 = function(a, b)
        return a + b
end

-- syntactic sugar: <--> python def: style
function fun2(a, b)
        return a * b
end

-- syntactic sugar: associate function with a table
t = {}
function t.fun3(a, b)
        return a / b
end

fun1(3, 2) -- 5
fun2(3, 2) -- 6
t.fun3(3, 2) -- 1.5
```

Tables

The only basic complex datatype: the *table*, which stores key - value pairs. (Much like a python `dict`, or maybe rather a python `SimpleNamespace` since dot indexing is supported).

Keys can be arbitrary values expect `nil`.

Syntax:

```
a = {}
n = "name1"

-- equivalent ways to access table entries
a["name1"] = 1
a[n] = 1
a.name1 = 1

-- unitialized entries are nil:
print(t.name2) -- nil

-- delete an entry by setting it to nil
t.name1 = nil
```

Table constructors have the syntax:

```
a = {name1=2, name2=5}
```

Table variables have reference semantics:

```
a = {name1=2}
b = a
b.name1 = 100
print(a.name1) -- 100
```

Metatables

Attaching Behavior to Tables

A *metatable* is an ordinary table which usually contains implementations of functions with certain naming conventions. A table can carry a metatable, with the effect that some methods implemented in the metatable are used in the context of that table.

```
meta = {}
function meta.__tostring(person)
        return person.prefix .. " " .. person.name
end
a = {prefix="Dr.", name="Petter"}
setmetatable(a, meta) -- meta is now the metatable attached to a
print(a)                            --[[ print uses the __tostring() function from the
                                         metatable and prints "Dr. Petter" --]]
```

The metatable can be get and set via

```
setmetatable(a, meta)
getmetatable(a)
```

Among the functions with naming conventions that the metatable can override are

- operators like `__add`, `__mul`, `__sub`, `__div`, `__pow`, `__concat`, `__unm`
- comparators like `__eq`, `__lt`, `__le`

These methods are called *metamethods*.

Delegation

The `__index` method

Delegation is the core concept of prototype-based programming.

Failed name lookups can be handled dynamically via the `__index(tbl, key)` method of the metatable. For example: delegate the failed lookups to a *prototype*, i.e. a table with default values.

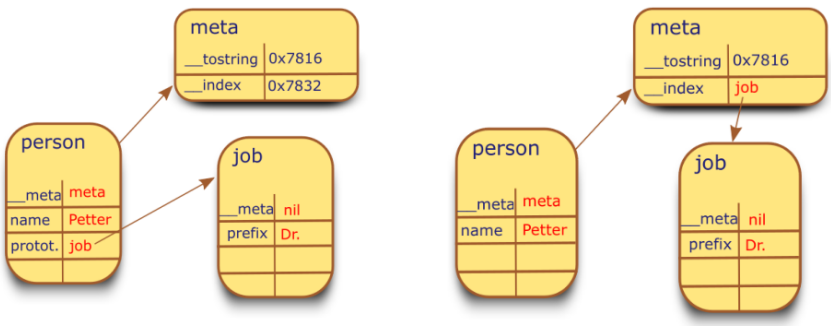
Example:

```
meta = {}
function meta.__tostring(person)
        return person.prefix .. " " .. person.name
end
function meta.__index(tbl, key)
        return tbl.prototype[key]                       -- delegate lookup to a prototype
end

job = { prefix="Dr." }
person = { name="Petter", prototype=job } -- use `job` as the prototybe
setmetatable(person, meta)
print(person)                                                   -- "Dr. Petter"
```

`__index` as table

For convenience, `__index` can be specified as table instead of as method. In this case, every failed name lookup is looked up directly in that table.

```
-- ...
job = { prefix = "Dr." }
meta.__index = job
setmetatable(person, meta)
-- ...
```



__index as method __index as table

The `__newindex` method

The `__newindex(tbl, key, val)` method allows to hook into unresolved table value *updates*.

Example:

```
-- ...
function meta.__newindex(tbl, key, val)
        if (key == "title" and tbl.name == "Guttenberg") then
                error("No title for you!")
        else
                -- have to keep an internal `data` table to avoid infinite recursion
                tbl.data[key] = val
        end
end

person = { data = {} }
meta.__index = person.data
setmetatable(person, meta)

person.name = "Guttenberg" -- this works
person.name = "Dr."                -- this fails!
```

Here, the `__newindex()` method is used to somewhat protect an object (not a very good protection, of course).

There also exists a method to hook into modifying existing fields.

Mimicking Object Oriented Programming

Mimicking Classes and Objects

We can mimick "classes" and "objects" by creating a table A that represents a class and tables $a, b, c, \ldots$ that have A as metatable, where `__index()` is delegated to A.

```
Account = {}
function Account.withdraw(acc, val)
        -- `acc` is used like a `this` reference here
        acc.balance = acc.balance - val
end

-- class/object setup
Account.__index = Account
mikes = { balance = 0 }
daves = { balance = 0 }
setmetatable(mikes, Account)
setmetatable(daves, Account)

-- using a "class" function: multiple ways to do the same thing
Account.withdraw(mikes, 10)
mikes.withdraw(mikes, 10) -- redundant "mikes"
daves.withdraw(mikes, 10) -- also withdraws from `mikes`
mikes:withdraw(10) -- better syntax: syntactic sugar for receiver param
```

Mimicking Constructors and Instance Methods

The previous example can be refined by

- providing a constructor
- rewriting `withdraw` with syntactic sugar for the receiver parameter

```
Account = {}
function Account:withdraw(val) -- the : syntax also works here
        self.balance = self.balance - val
end
function Account:new(template)
        -- if called like Account:new(..), `self` is the Account table
        template = template or {balance = 0} -- providing a default value

        -- set up the metatable to index into Account (to connect to withdraw method)
        setmetatable(template, {__index=self})
        return template
end

-- called with : syntax: hence the `self` is the Account table
giro1 = Account:new({balance=10})
giro1:withdraw(10)

 -- without passing the template parameter
giro2 = Account:new()
giro2:withdraw(10)
```

Mimicking Inheritance

Inheritance can be implemented by setting delegating via `__index` from the "subclass" table to the "superclass" table.

```
LimitedAccount = {}
setmetatable(LimitedAccount, {__index=Account})
function LimitedAccount:new()
        instance = { balance=0, limit=100 }
        setmetatable(instance, {__index=self})
        return instance
end
function LimitedAccount:withdraw(val)
        -- implement a different `withdraw()` for `LimitedAccount`
end
```

Mimicking Multi-Inheritance

We can create our own model of multi-inheritance: for example, one where a class can have two parents and conflicts are resolved by giving the first parent class precedence.

```
function createClass(parent1, parent2) -- parents = "class-like tables"
        local c = {} -- the new child class
        setmetatable(c, {__index =
                function (t, k)
```

```
                        local v = parent1[k]
                        if (v != nil) then return v end
                        return parent2[k]
                end
        })
        c.__index = c -- relevant for instances of c
        function c:new (o)
                o = o or {}
                setmetatable(o, c)
                return o
        end
        return c
end
```

Implementation of Lua and Further Topics

Implementation-wise, objects are represented by a type id and a value; values are just unions of the different low-level types.

```
typedef struct {
        int type_id;
        Value v;
} TObject;

typedef union {
        void *p;
        int b;
        lua_number n;
        GCObject *gc;
} Value;
```

Some optimization can be done for tables: these fork into

- a Hashmap (for dict-like indexing)
- and an integer-indexed array

Further Lua features are coroutines and closures.

Summary: Prototype-Based Programming

Lessons learned:

- we can possibly ease up the development by abandoning fixed inheritance (but can also have horrible runtime errors because of the dynamic nature of the language)
- OOP and MI can be implemented as special cases of delegation
- The minimal featureset of Lua makes it easy to implement a compiler/interpreter
- To find bugs before runtime, one could try to use some static analysis

---

Week 13 - Aspect-Oriented Programming in AspectJ

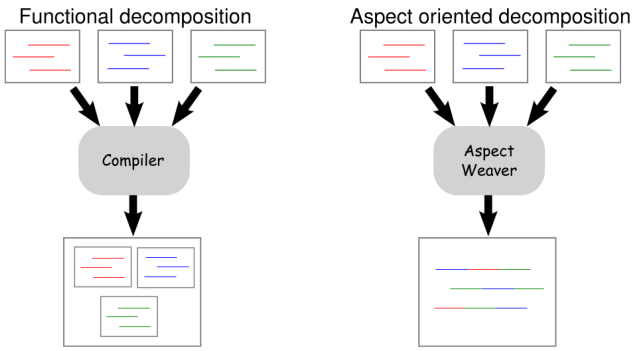
Aspect-Oriented Programming

Basic idea: allow to add new functionality to existing modules.

- via extension-method-like code (*static crosscutting*)
- via hooks into certain points in the code (*dynamic crosscutting*)

Examples where this might be helpful: Security, Logging, Error Handling, input validatino, profiling. The core idea is to keep the main code clean and simple, and add in all the other functionality (*aspects*) via AOP crosscutting.

Another way to view it: traditionally, code focusses on sequential execution. In AOP, one focusses on *aspects of concerns* which themselves are decoupled. The aspects are *woven* into the program by a *weaver*.



Here: we look at **AspectJ**, a superset of Java which allows to define aspects.

Static Crosscutting

The simpler version of crosscutting, which resembles extension methods: Add new methods to existing types.

Example:

```
class Expr {}
class Const extends Expr { /*...*/ }
class Add extends Expr { /*...*/ }

// Aspect for evaluating expressions
aspect ExprEval {
        abstract int Expr.eval();
        int Const.eval() { return val; }
        int Add.eval() {return l.eval() + r.eval(); }
}
```

This corresponds to the following plain Java code:

```
abstract class Expr { abstract int eval(); }
class Const extends Expr {
        /*...*/
        public int eval() { return val; }
}
class Add extends Expr {
        /*...*/
        public int eval() { return l.eval() + r.eval(); }
}
```

Dynamic Crosscutting

Dynamic crosscutting is the much more interesting and powerful part of AOP.

Idea: can define so-called *join points*, which correspond well-identifiable points in the byte code, that execution of new methods can hook into.

Overview: Types of Join Points in JAspect

- method *calls* (corresponds to syntactical method calls)
- method *executions* (corresponds to actual executions of a method)
- field *get* and *set*
- exception handler execution
- class initialization (corresponds to static initializers being run)
- object initialization (corresponds to dynamic initializers being run)

Pointcuts

> **Definition** (*Pointcut*).
> A *pointcut* is a set of join points, and optionally some specification of runtime values

A pointcut can combine multiple join points by boolean operations. Example:

```
pointcut dfs(): execution (void Tree.dfs()) ||
                           exeuction (void Leaf.dfs());
```

The pointcut `dfs` in the example combines the joinpoints for a `Tree.dfs()` and a `Leaf.dfs()` execution.

Advice

The callback code which should be executed at a joinpoint is called *advice*.

Advice code can be performed `before(...)` or `after(...)` a joinpoint, with additional syntax `after(...) returning (...)` and `after (...) throwing (...)`.

Example: before `C.foo(int)` is called, "About to call foo" should be printed.

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

Advice can bind certain parameters of the joinpoints, e.g. the method call parameters of a `call`: This is done via the `args(arglist)` syntax. The variables to bind to most be introduced in the `before(...)` or `after(...)` expression.

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

Instead of variable names, arguments in `arglist` can also be typenames to match, `*` for all types, `..` to match several arguments.

Around advice

In addition to `before` and `after`, there is `around`: A special function `proceed` is introduced that signifies the point where the `around` advice hands off to the actual call, after which control flow returns to after the proceed call.

Example:

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

Advice Precedence

The order of execution of mutiple pieces of advice for the same join point is governed by some rules, but not fully specified for all cases.

- precedence of one `aspect` over another can be declared via `declare precedence: A, B`
- if A is a subaspect of B, then advice from A takes precedence
- for pieces of advice from the *same* aspect, the following rules apply:
  - if both are `after` advice, the one that appears *later* in the code has precedence
    - QUESTION: why is that? seems weird. QUESTION: is "both" correct?
  - otherwise, the one that appears *earlier* has precedence

For all other cases, there is no specified precedence.

Types of Join Points in Detail

Method-related Designators

There are two method-related designators: `call(signature)` and `execution(signature)`.

`signature` has the syntax:

```
// method calls
ResultTypeName ReceiverTypeName.method_id(ParamTypeName, ...)
```

```
// constructor calls
NewObjectTypeName.new(ParamTypeName, ...)
```

Calls vs. Executions

Example usage:

```
class MyClass {
        public String toString() { "bla" }
        public static voide 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!");
        }
}
```

`call` and `execution` can differ in their behavior: In this example, one line from the call, but two lines from the execution are printed.

Output:

```
advice!
advice!
advice!
bla bla
```

Pointcuts that match multiple signatures

Once inheritance is involved, a `call` or `execution` specification might match multiple signatures. The behavior in this case is somewhat intricate.

Example:

```
interface Q          { int m(int i); }
class P implements Q { int m(int s) {return 0;} }
class S extends P    { int m(int s) {return 0;} }
class T extends S    {}
class U extends T    { int m(int s) {return 0;} }

// ... somewhere in the code:
(new T()).m();
```

Aspect that matches on `call` for `m` on all 5 types, i.e. `int Q.m(int)`, `int P.m(int)`, etc: The advice for Q, P, S, T is executed.

Aspect that matches on `execution` for `m` on all 5 types: The advice for Q, P, S is executed, i.e. not for T which does not implement its own variant of `m`.

Repeat the `execution` experiment, but now call `(new U()).m();`: The advice for Q, P, S, T, U is executed!

General rules of thumb:

- `call` matches for all matching signatures on supertypes
- `execution` *should* match methods declared in the specified type, overridden in that type, or inherited by that type; weirdly, in the second example, it doesn't match `T`.

Takeaway: all of this is maybe a bit ill-specified.

Field-related Designators

There are two designators to hook into setting and getting of fields: `get(fieldqualifier)` and `set(fieldqualifier`. Attention: this does *not* hook into the get/set methods, but rather in any use of the dot notation, `A.x = 1`.

The `fieldqualifier` has the syntax `FieldTypeName ObjectTypeName.field_id`. The argument of `set` can be bound by `args(arglist)`.

Example:

```
aspect GuardedSetter {
        before(int newval): set(static int MyClass.x) && args(newval) {
                if (Math.abs(newval - MyClass.x) > 100) { /*...*/ }
        }
}
```

Type-based Designators

There are three type-related designators:

- `target(typeorid)`
  - matches any join point where `typeid` is as receiver
- `within(typepattern)`
  - matches any join point contained in a particular class body
- `withincode(methodpattern)`
  - matches any join point contained in a particular method body

Flow and State-Based Designators

More complicated flow-based and state-based designators exist:

- `cflow(arbitrary_pointcut)`
  - matches *any join point* that occurs *between the entry and exit* of each join point matched by `arbitrary_pointcut`
- `if(boolean_expression)`
  - matches *any join point* that *matches a boolean expression*

Example of `if` (a very inefficient way to match method calls):

```
aspect GuardedSetter {
        before(): if(thisJoinPoint.getKind().equals(METHOD_CALL)) && within(MyClass) { /*...*/ }
}
```

Implementation of Aspect Weaving

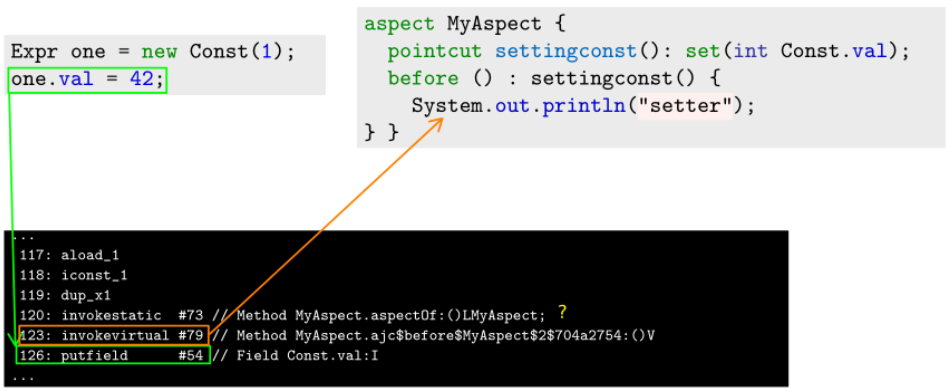
A lot of different methods are possible in principle to implement aspect weaving on the compiler level:

- via a preprocessor which transforms into valid Java sourcecode
- during compliation
- during runtime in the VM
- post-compile processor

The *post-compilation processor* method is the fashionable go-to method to date, and the others are not so relevant/vanishing. Here, the compiled byte code is modified to bring in the aspects.

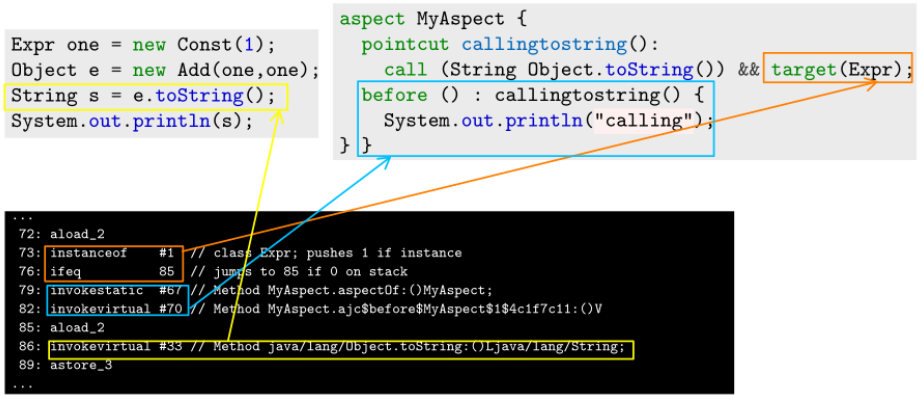
Weaving example: setter

A setter advice is implemented by invoking the advice method before the `putfield` command is executed.



Weaving example: method call

A call advice with an additional check is implemented by first performing an `instanceof` check, the invoking the advice method, then invoking the actual method.



Weaving of more complicated constructs

Around/Proceed Weaving

Weaving of around/proceed is more complicated. An advice method is created that first performs the part before `proceed`, then calls the actual method, then the part after `proceed`.

Property-Based Croscutting Weaving

More complicated: property-based, like `cflow` and `if`. At each join point, one would need to match the conditions of each property-based pointcut.

Idea to optimize this: keep a separate stack of states (for each `cflow` pointcut?) that is only modified at pointcuts relevant to the `cflow`, and just checked for emptiness at each join point. In practice, even more optimizations take place.

Summary: Weaving implementation

Translation scheme summary:

- `before`/`after`: ranges from inlined code to distribution to several methods/closures
- Joinpoints that have matching advices may get explictly dispatching wrappers
- dynamic dispatching may require runtime tests to interpret certain joinpoint designators
- flow-sensitive pointcuts like `cflow` incur a runtime penalty, even if optimized

Summary: Aspect Orientation

Pros of Aspect Orientation:

- concerns can be untangled
- late extension, across hierarchy boundaries, becomes possible
- another level of abstraction is provided by aspects

Cons:

- runtime overhead of weaving
- one loses transparency, since aspects can kick in without being apparent in the original code (especially: non-transparent interactions between aspects)
- IDE support needed, especially for debugging

Week 14 - Metaprogramming

Metaprogramming

Metaprogramming - guiding principle:

> "Let's write a program which writes a program"

In this category belong Compilers and Preprocessors, but also Reflection, the Metaobject Protocol, Macros and the most extreme form: Runtime Metaprogramming.

Some motivation: in Aspect-Oriented Programming, we saw how program code is programatically refined (i.e. existing code "treated as data that is processed"). Now we go one step further.

Codegeneration Tools

In compiler construction: tools that convert domain-specific languages to source code. This can be seen as the most simplistic way of metaprogramming.

Codegeneration Example: `lex`.

Example: `lex`, which generates a C code representation of a finite automaton corresponding to some regular expressions.

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

Compiletime Codegeneration: C Macros

C macros are expanded by the *C Preprocessor* (CPP), which is just a simple string rewriting system.

```
#define min(X,y) ((X < Y)? (X) : (Y))
x = min(5,x);
x = min(++x,y+5); /* ++x evaluated twice! */
```

Some drawbacks of C macros:

- parts of macros can bind to outside operators, if not properly parenthesized!
- unlike for function calls, side effects of parameter expressions might be recomputed several times
- no recursive behavior possible (rewriting stops)
- possibly duplicated identifiers (non-hygienic), since identifiers are hard-coded in the macros

Adding Language Constructs via Macro Hackery

With macros, one can in principle define new control structures, by exploiting the binding behavior of existing control structures. For example, one could syntax for the ATOMIC block:

```
ATOMIC (globallock) {
        /* user code */
}
// Should translate to
acquire(&lock);
{ /* user code */ }
release(&lock);
```

Macros don't allow something like this at first sight, due to their grammar; However, the following hack works. The key is to allow the usercode block that follows the macro to be executed after some preprended and before some appended code, with a combination of `if`/`while`/`break`/`goto`.

Preprending code:

```
if (1) {
        /* prepended code */
        goto body;
} else body:
        { /* usercode block */ }
```

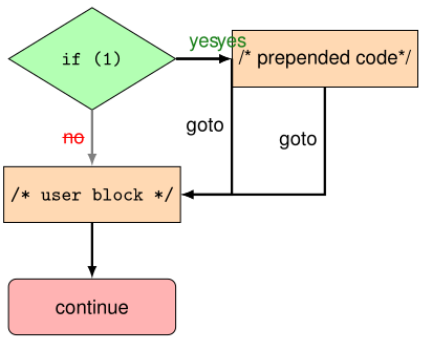
Appending code:

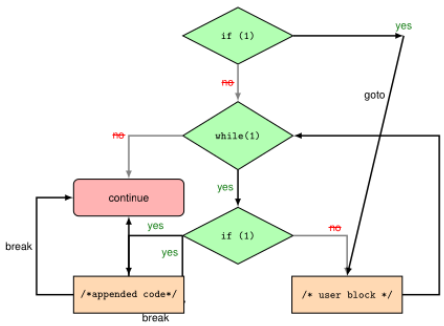
```
if (1)
        goto body;
else
```

```
        while (1)
                if (1) {
                        /* appended code */
                        break;
                }
        }
        else body:
        { /* usercode block */ }
```

Both techniques combined:

```
if (1) {
        /* prepended code */
        goto body;
} else
        while (1)
                if (1) {
                        /* appended code */
                        break;
                }
                else body:
                { /* usercode block */ }
```





Macro for custom control structures

```
#define concat_( a, b) a##b
#define label(prefix, lnum) concat_(prefix,lnum)
#define ATOMIC (lock)          \
if (1) {                                \
        acquire(&lock);                \
        goto label(body,__LINE__); \
} else                          \
        while (1)                       \
                if (1) {                \
                        release(&lock);     \
                        break;              \
                }                           \
                else                        \
                label(body,__LINE__):
```

One trick: the labels for the goto are generated dynamically using the line number `__LINE__`. Therefore, not two atomic blocks may be placed in the same line.

Homoiconononic Metaprogramming

Homoiconic = "with the same representation". More formally: in a homoiconic language, the primary representation of programs *is also* a data structure in a primitive type of the language itself.

> data is code ⟷ code is data

Reflection

Type introspection allows to examine types of objects at runtime. Java has type instrospection via `instanceof`:

```
public boolean isNatural(Object o) {
        return (o instanceof Natural);
```

```
}
```

Furthermore, the Java Reflection API allows to analyze the class structure at runtime:

```
static void fun(className) { // e.g. "Natural"
        // ... but the static type is of course Object
        Object incognito = Class.forName(className).newInstance();
        Class meta = incognito.getClass();
        Field[] fields = meta.getDeclaredFields();
        for (Field f : fields) {
                Class fieldClass = f.getType();
                Object fieldValue = f.get(incognito);
                if (fieldClass == boolean.class && Boolean.FALSE.equals(fieldValue)) {
                        return true;
                }
        }
        return false;
}
```

Metaobject Protocol (Lisp CLOS)

The Lisp CLOS *metaobject protocol* allows to manipulate the implementation of CLOS and change aspects of

- class, object, property and method creation
- causing inheritance relations
- creation of specializers (e.g. overwriting, multimethods)
- ...and many more aspects.

Homoiconic Metaprogramming in Lisp

Common Lisp (Clisp) offers homoiconic metaprogramming, and especially allows elaborate macro programming!

Lisp language basics

S-Expressions are either atoms, or concatenate S-expressions x, y in the form $(x \cdot y)$.

- Shortcut: $(x_1\ x_2\ x_3) = (x_1 \cdot (x_2 \cdot (x_3 \cdot ())))$

Forms are units that can be evaluated:

- numbers, keywords, empty lists, nil (fully evaluated)
- symbols (i.e. variables)
  - a symbol evaluates to the value it is bound to
- lists, and among these:
  - special forms (first element a certain keyword)
    - evaluates to a value
  - macro calls (first element a macro name)
    - is converted to another form
  - function calls (first element a function name)

Some *special forms*:

```
(if test then else?)
(let [binding*] expr*)           ; introduce a local binding
(eval-when (situation) form) ; evaluates the form in the situation
(progn form*)                            ; evaluates form and returns last value
(quote form)                             ; yields unevaluated form (quotes it)
(setq {var form}*)                       ; evaluates form and assigns it to variable var
```

Quoting in Lisp

Forms are directly interpreted, *unless they are quoted*! The syntax for quoting is `'` or `quote`, e.g. `(quote (let))` or `'(let)`.

Backquote-comma syntax to escape from quotes:

```
`((+ 1 2) ,(+ 1 2))
; gives ((+ 1 2) 3)
```

QUESTION: how does the comma syntax work exactly?

Lisp Macros

"Macros are configurable syntax/parse tree transformations". A guiding principle: Very few basic special forms, rest of language functionality is implemented in macros.

As an example, the for loop is implemented in a macro:

```
(macroexpand
        '(loop for y in '(1 2 3) do (print y))))
; this expands to multi-line list code that looks quite complicated
```

Example of how macros can be written, in this case a macro that converts an infix to a prefix expression:

```
(defmacro infix (fpar spar tpar)
        "converting infix to prefix"
        `(,spar ,fpar ,tpar))

(infix 1 + 2)                ; evaluates to 3
(macroexpand '(infix 1 + 2)) ; evaluates to (+ 1 2)
```

Example: Factorial Macro

```
(defmacro fac (n)
        if (= n 0)
                1
                `(* ,n (fac ,(- n 1)
))))
```

Example usage:

```
(macroexpand '(fac 4))
; gives (* 4 (fac 3))
(macroexpand-all '(fac 4))
; gives (* 4 (* 3 (* 2 (* 1 1))))
```

Takeaway: Macros expand at compile-time!

Macros: Why Bother?

Macros = static AST transformations ⟷ Functions = runtime control flow manipulations.

Some more differences of macros ⟷ functions:

- macro parameters are uninterpreted and not necessarily valid expressions
- function parameters evaluate once

Hygienic Macros in Lisp

In Lisp: to avoid *shadowing* of variables, lisp offers automatically generated symbols.

```
(defvar varia)
(defmacro mac (stufftodo) `(let ((varia 4711)) ,stufftodo))
(setf varia 42)
(mac (write varia))
```

Homoiconnic Runtime-Metaprogramming

Example: Factorial with runtime metaprogramming

(Warning: convoluted example ahead)

```
(defun fac (n)
        (let (
                  (lam `(lambda () (* ,@(loop for n from 1 below (+ n 1) by 1 collect n))))
                  (symb (gensym))
              )
          (compile symb lam)
          (disassemble symb)
```

```
          (funcall symb)
        )
)
```

Week 15 - Continuations

From Gotos to Continuations

Control Flow in C

In `C`, the simplest "irregular" way to change the control flow is via `goto`s:

```
for (int i=0;i<15;i++)
        for (int j=0;j<15;j++)
                if (a[i] == a[j]) goto exit;
exit:
        return;
```

This is similar to `break` or `continue`, but a little more "irregular" and frowned upon. Still, `goto`s are somewhat restricted as they only allow to jump around within the same method (*scoped procedure-locally*).

Stack-Backward Control Flow: `longjmp`

C also allows jumps accross procedure boundaries, in the sense of moving from child procedure calls to ancestor procedure calls, i.e. in the stack-upward direction: The methods for this are `setjmp` and `longjmp`

`longjmp` and exception handling

With these two commands we can mimick something similar to exception handling in higher-level languages. However this is not "proper" exception handling: For example, heap objects might be leaked since there is no mechanism to free them.

In other languages where this is properly implemented, e.g. C++: *stack unwinding*, and using tables of exceptions a method can catch and a cleanup table appended to the method body by the compiler.

In detail: if an exception occurs, the runtime first checks if some method up the call hierarchy can catch it; if yes, detailed unwinding starts, if not, the program terminates immediately.

If cleanup is necessary, a so-called *personality routine* uses the cleanup table for the current method and runs destructors for allocated objects.

Example of `longjmp`/`setjmp`

```
#include<setjmp.h>
#include<stdio.h>

int fun(jmp_buf *error_handler, int number) {
        printf(" fun num=%d |", num);
        if (number > 0)
        // Call is made, but the value will never be returned
                return fun(error_handler, number-1);
        longjmp(*error_handler, 2);
}

void main() {
        int x = 4;
        jmp_buf context;
        // 0 in the initial run, and 2 after each longjump
        int r = setjmp(context);
        x -= r;
        printf("In main: x = %d\n", x);
        if (x > 0) fun(&context, 5);
}
```

Output:

```
In main: x = 4
 fun num=5 | fun num=4 | fun num=3 | fun num=2 | fun num=1 | fun num=0 |
In main: x = 2
 fun num=5 | fun num=4 | fun num=3 | fun num=2 | fun num=1 | fun num=0 |
In main: x = 0
```

The second argument of `longjmp` is the return value which then the `setjmp` command returns.

"Same-Level" Control Flow: `swapcontext`

Other than the jumps restricted to the current stack we just saw, C also supports more flexible switching between "siblings":

- create a fresh context with `getcontext()`
- make context point to a particular function with `makecontext()`
- swap into the created context, thereby starting execution of the other function, with `swapcontext`
  - takes `ucontext_t *oucp` where it stores the current context, and `ucontext_t *ucp` which it activates
  - returns -1 on error, otherwise returns 0 as soon as `oucp` is activated again

Some drawbacks:

- a stack for the contexts has to be manually created, and its size be known
- scheduling of termination depends on definition of a successor context
- the functions are a bit fake: The behavior is not much different from having one big function body and `goto`-ing around within it.

Example: `swapcontext`

interleaved functions

```c
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>
static ucontext_t ctx_m, ctx_f1, ctx_f2;
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void f1(void) {
  printf("f1: started\n");
  printf("f1  --swapcontext--> f2\n");
  if (swapcontext(&ctx_f1, &ctx_f2) == -1) handle_error("swap");
  printf("f1: returning\n");
}
static void f2(void) {
  printf("f2: started\n");
  printf("f2  --swapcontext--> f1\n");
  if (swapcontext(&ctx_f2, &ctx_f1) == -1) handle_error("swap");
  printf("f2: returning\n");
}
```

startup platform

```c
int main(int argc, char *argv[]) {
   char f1_stack[16384];
   char f2_stack[16384];
   if (getcontext(&ctx_f1) == -1) handle_error("getcontext");
   ctx_f1.uc_stack.ss_sp = f1_stack;
   ctx_f1.uc_stack.ss_size = sizeof(f1_stack);
   ctx_f1.uc_link = &ctx_m;
   makecontext(&ctx_f1, f1, 0);
   if (getcontext(&ctx_f2) == -1) handle_error("getcontext");
   ctx_f2.uc_stack.ss_sp = f2_stack;
   ctx_f2.uc_stack.ss_size = sizeof(f2_stack);
   /* f2's successor context is f1(), unless argc > 1 */
   ctx_f2.uc_link = (argc > 1) ? NULL : &ctx_f1;
   makecontext(&ctx_f2, f2, 0);
   printf("main  --swapcontext--> f2\n");
   if (swapcontext(&ctx_m, &ctx_f2) == -1) handle_error("swap");
   printf("main: exiting\n");
   exit(EXIT_SUCCESS);
}
```

- main creates contexts for the other two functions
- main swaps to f2
- f2 swaps to f1
- f1 swaps to f2, which continues and returns to f1
- f1 continues and returns to main

Coroutines

Couroutines: technique implemented in several languages in some form (there is no "hard definition" of what Coroutines are, exactly).

Stackless Coroutines in EcmaScript 6+

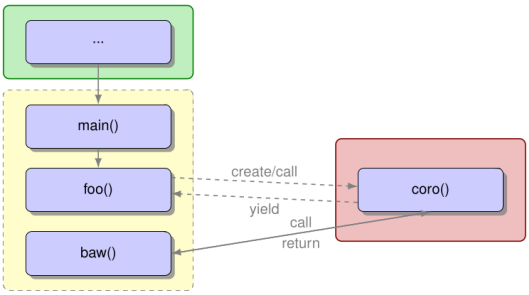
EcmaScript 6+ implements *generators*:

- `yield` returns a value, but later execution can restart from that point
- generator function bodies marked with a `*`, namely with `function*() {...}`

```js
var genFn = function*(){
        var i = 0;
        while (i < 10) {
                yield i++;
        }
};
var gen = genFn();
do {
        var result = gen.next();
        console.log(result.value);
} while (!result.done);
```

The coroutine lives besides the normal function call stack (*stackless coroutine*).

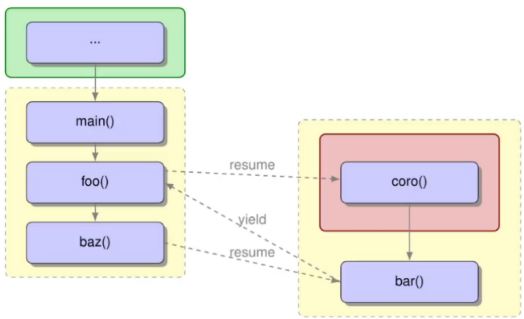
Only the function body can yield for a generator, i.e. we cannot simply call another function which then yields for this generator as well. This is good, as another function call would go on the stack, and yielding from there would mean that the function on the stack could be overwritten.



Stackful Coroutines in Lua

Lua is more liberal with its coroutines: `yield` can be called from any called function as well. In consequence, coroutines need their own stack.

```
function func1 ()
  coroutine.yield(1)
end

function func2 ()
  func1()
  coroutine.yield(2)
end

coro = coroutine.create(func2)
coroutine.resume(coro) -- yields value 1
coroutine.resume(coro) -- yields value 2
```



Continuations in Haskell

Haskell allows to change the "control flow" very freely via so-called *continuations*.

Continuation Passing Style

In *continuation passing style* (CPS), functions $f :: a \to b$ are transformed to functions $f' :: a \to ((b \to c) \to c)$, where $f'\ x\ cont$ computes $cont\ (f\ x)$, given a *continuation* $cont :: b \to c$.

$f'\ x$, with a continuation not yet specified, can be seen as a *suspended computation*.

Additionally to count as a CPS-style function, f' should internally rely only on CPS-style functions.

Example: Pythagoras in CPS

Direct style:

```
square x = x * x
add x y = x + y
pythagoras x y = add (square x) (square y)
```

Continuation Passing Style:

```
square_cps x = \k -> k ((*) x x)
add_cps x y = \k -> k ((+) x y)
pyth_cps x y = \k ->
        square_cps x (\x2 ->
                square_cps y (\y2 ->
                        add_cps x2 y2 k))
```

("After we obtain x^2, then after we obtain y^2, then after we add $x^2 + y^2$, we perform k with the result").

Example: CPS and Higher-Order Functions

Consider a function `trip` that applies a function f thrice:

```
trip f x = f (f (f x))
```

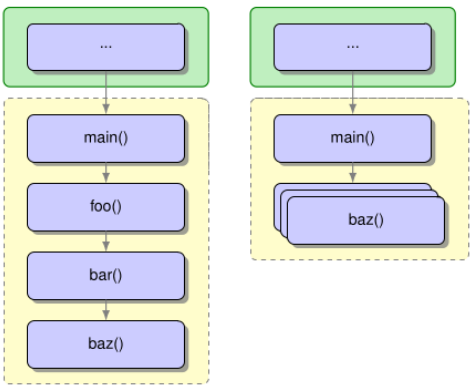
In CPS, we get:

```
trip_cps :: (a -> ((a -> r) -> r)) -> a ((a -> r) -> r)
trip_cps f_cps x = \k -> f_cps x (\fx ->
        f_cps fx (\ffx ->
                f_cps ffx k ))
```

Tail Call Optimization

Interlude: "Is this even remotely efficient?"

Functional compilers can make use of *tail call optimization* to make patterns with repeated function calls more efficient. Idea: If a function call is the last thing a procedure does, recycle the stack frame.



Something like

```
f 0 = 0
f n = n + f (n-1)
```

boils down to more or less a loop, which is of course quite efficient!

Composing Continuations

Define a type

```
newtype Cont r a = -- ...
-- roughly like this: Cont { runCont :: (a -> r) -> r }
-- but it seems like the actual definition is more complicated
```

(such a type is defined in `Control.Monad.Cont`).

The operations are $cont\ f$ which wraps a function $f :: (a \to r) \to r$ into a suspended computation $Cont\ r\ a$, and $runCont\ c\ k$ which executes the suspended computation $c :: Cont\ r\ a$ with the continuation $k :: a \to r$ and returns r.

One could now define a composition function to chain execution of continuations, like this:

```
compose' s f = cont (\k -> runCont s (\x -> runCont (f x) k))
```

```
-- could be used like this:
square x p = p (x * x)
runCont (compose' (square 3) square) id
```

In reality, this is implemented as a monadic bind! `Cont r` is instantiated as a `Monad`.

Interlude: Monad

Monads, i.e. instance of the `Monad` typeclass, define at least a function `return` and a function `>>=`:

```
class Monad m where
        return :: a -> m a
        (>>=)  :: m a -> (a -> m b) -> m b
```

Monads allow *do-notation*, a syntactic sugar to allow an imperative style:

```
mothersPaternalGrandfather s = do
        m <- mother s
        gf = father m
        father gf

-- or, desugared:
mothersPaternalGrandfather s =
        mother s >>= (\m ->
        father m >>= (\gf ->
        father gf))
```

`Cont` as Monad

Implementation as a monad:

```
instance Monad (Cont r) where
        return x = cont (\k -> k x)
        s >>= f  = cont (\k -> runCont s (\x -> runCont (f x) k))
```

Example: CPS Pythagoras in `Cont` Monad

```
add_cont x y = return (x+y)
square_cont x = return (x*x)
pythagoras_cont x y = do
        x2 <- square_cont x
        y2 <- square_cont y
        add_cont x2 y2
```

Call with Current Continuation

The function `callCC` (*call with current continuation*) calls a function with the current continuation as its argument: This allows to escape from a computation.

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = cont (\h -> runCont (f (\a -> cont (\_ -> ha a_)) h ))
```

Idea: *we ignore the current computation stream and reconnect to the original continuation point of callCC!*.

Simple `callCC` examples

```
import Control.Monad.Cont
main = runCont (do
        z <- callCC (\c -> do
                a <- (c 3)
                -- this part is skipped:
                return (a + 3)
        )
        return (z + 1)
        ) print
```

Here, the continuation c is used as an escape hatch: instead of returning $a + 3$, 3 is returned and assigned to z, the end result being 4.

While loops with `callCC`

We can simulate while loops: (some Monad hackery is needed to combine the `Cont` and `IO` Monad, hence)

```
import Control.Monad.Class
import Control.Monad.Cont

main = flip runContT return $ do
        lift $ putStrLn "A"
        (k, num) <- callCC (\c -> let f x = c (f, x)
                                  in return (f, 0))
        lift $ putStrLn "B"
        lift $ putStrLn "C"
        if num < 5
                then k (num+1) >> return ()
                else lift $ print num
```

This is basically a loop of the form

```
print("A");
int num = 0;
do {
        print("B");
        print("C");
} while(num++ < 5);
print(num);
```

What happens in detail:

- `k` is assigned the function `f` that calls the continuation with itself and its argument as output
- hence when `k` is called, we jump to the program point where callCC is called, and the argument of `k` is stored in `num+1`

The calling site of `callCC` marks the point of program continuation!

Implementing Continuations

Continuations pose some implementation difficulties:

- they can escape the current context (function frame): calling a continuation restarts execution at the original `callCC` site/function frame
- `callCC` continuations are multi-shot in principle: they may return to the same site multiple times

Traditional stack-based frame management is not good enough here, since old function frames are overwritten! Something more flexible, maybe more like a graph, is needed.

Outlook: More Techniques

We saw just a glimpse of what can be done with continuations. Further possibilities are

- Delimited/Partial Continuations
- Y Combinator
- SKI Calculus