Memory Models
Memory interactions behave differently in presence of:
- multiple concurrent threads
- data replication in hierarchical and distributed memory systems
- deferred communication of updates

Memory Models are a product of negotiating:
- restrictions of freedom of implementation to guarantee race related properties
- establishment of freedom of implementation to enable program and machine model optimizations

Modern Languages include the memory model in their language definition.

Abandoning absolute time
Happend-Before Relation and Diagram

Events in a Distributed System
A process as a series of events [2]: Given a distributed system of processes P, Q, R, . . . , each process P consists of events eP1, eP2, . . .

Example:

```
P: eP1, eP2, . . .
Q: eQ1, eQ2, . . .
R: eR1, eR2, . . .
```

- A event eP in process P happened before eQ if:
  - If eP is an event that sends a message to Q then there is some event eQ in Q that receives this message and eP happened before eQ.

The Happened-Before Relation
Definition
If an event p happened before an event q then p → q.

Observes:
- p → q is partial (neither p → q nor q → p may hold)
- p → q is irreflexive (p → q never holds)
- p → q is transitive (p → q ∧ q → r ⇒ p → r)
- p → q is asymmetric (if p → q then q → p)

The → relation is a strict partial order.

Concurrency in Happened-Before Diagrams
Let p → q abbreviate (¬ r → q ∧ p → r).

Definition
Two distinct events p and q are said to be concurrent if p ∨ q and q ∨ p.

Function c assigns a globally unique time-stamp C(p) to each event p.

Definition (Clock Condition)
Function c satisfies the clock condition if for any events p, q:

\[ p \rightarrow q \implies C(p) < C(q) \]

For a distributed system the clock condition holds if:
- p and q are events of P and p → q then C(p) < C(q)
- p is the sending of a message by process P and q is the reception of this message by another process Q then C(p) = C(q)
- a logical clock c that satisfies the clock condition describes a total order \( a < b \) (with \( C(a) < C(b) \)) that encodes the strict partial order.

The set defined by all c that satisfy the clock condition describes a total order possible in the system.

≡ use the process model and to define better consistency model.

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

Definition (Strict consistency)
Indepedently of which processes reads or writes, the value from the most recent write to a location is observable by reads from the respective location immediately after the write occurs.

Although realistically desired, practically not existing
- absolute global time problematic
- physically not possible

⇒ strict consistency is too strong to be realistic.
Defining C Satisfying the Clock Condition

Given:

\[ C(e) = \{ 1, 4, 7, 12 \} \]
\[ C(e) = \{ 2, 3, 5, 6, 11, 13, 14 \} \]
\[ C(e) = \{ 8, 9, 10, 15 \} \]

Summing up Happened-Before Relations

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

Happened-Before Based Memory Models

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

Given a path through each of the threads of a program:
- consider the actions of each thread as events of a process
- use more processes to model memory
  - here: one process per variable in memory
  - to concisely represent some interleavings
- We establish a model for Sequential Consistency.

Sequential Consistency

Definition (Sequential Consistency Condition [2]):

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

Sequential Consistency applied to Multiprocessor Programs:

- Given a program with \( n \) threads,
  - for fixed event sequences \( p_1^0, p_2^0, \ldots, p_1^n, p_2^n, \ldots \)
  - keeping the program order,
  - executions obeying the clock condition on the \( p_r \),
  - all executions have the same result

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

Sequential Consistency: Formal Spec [5, p. 25]

Definition (Sequential Consistency):

- Memory operations in program order \( (\leq) \) are embedded into the memory order \( (\subseteq) \)
- A load’s value is determined by the latest write, memory order
  \[ w(\text{Load}, \text{Address}) = \text{Max}(\text{write}, \text{Address}, \text{comp}(\text{prevWrite}, \text{Address})) \]

with:
- \( \text{prevWrite} \) any memory access to address \( x \) by CPU \( i \)
- \( \text{Address} \) a load from address \( x \) by CPU \( i \)
- \( \text{comp} \) a store to address \( x \) by CPU \( i \)
- Program order \( \leq \) being specified by the control flow of the programs executed by their associated CPUs; only orders operations on the same CPU

Benefits of Sequential Consistency

- Concisely represent all interleavings that are due to variations in timing
- A synchronization using time is uncommon for software
  - a good model for correct behaviors of concurrent programs
- Programs results besides SC results are undesirable (may contain races)

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

Not realistic for elaborate hardware with out-of-order stores:
- what other processors see is determined by complex optimizations to cache line management
  - only a small number of actual operations are visible
  - internal workings of caches

Weakening the Model

Observation: more concurrency possible, if we model each memory location separately,
- i.e. as a separate process per variable in memory
  - concisely represent some interleavings
  - we establish a model for Sequential Consistency.

Working with Sequential Consistency

Sequential Consistency in Multiprocessor Programs:

- Given a program with \( n \) threads,
  - for fixed event sequences \( p_1^0, p_2^0, \ldots, p_1^n, p_2^n, \ldots \)
  - keeping the program order,
  - executions obeying the clock condition on the \( p_r \),
  - all executions have the same result

Idea for showing that a system is not sequentially consistent:
- pick a result obtained from a program run on a SC system
- pick an execution and a total ordering of all operations
- add extra processes to model other system components
- the original order becomes a partial order
- show that total orderings \( C' \) exist for \( \geq \) for which the result differs

Sequential Consistency still obeyed:
- the accesses of \( a \) to \( x \) occurs before \( b \)
- the first two read accesses to \( b \) are in parallel to \( a \)

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

Introducing Caches: The MESI Protocol
Introducing Caches

Idea: each cache line one process

a
Ld[a] A
B
a
Ld[a]
St[a]
St[a]
cache
cache
mem
a++
a++

Observations:
▲▲! naive replication of memory in cache lines creates incoherency


Definition (Cache Coherency)

1. Memory operations in program order \( \left[ \sigma \right] \) are embedded into the memory order \( \left[ \sigma \right] \)
2. A load’s value is determined by the latest write, memory order

- This definition superficially looks close to the definition of SC – except that it covers only singular memory locations instead of all memory locations accessed in a program
- Caches and memory can communicate using messaging, following some particular protocol to establish cache coherency


Processors use caches to avoid a costly round-trip to RAM for every memory access.

- programs often access the same memory area repeatedly (e.g. stack)
- keeping a local mirror image of certain memory regions requires bookkeeping about who has the latest copy

Each cache line is in one of the states M, E, S, I:
- M: It is invalid and is ready for re-use
- E: the content is in no other cache; it is exclusive to this cache and can be overwritten without consulting other caches
- S: the content is exclusive to this cache and has furthermore been modified
- I: invalid

⇝ the global state of cache lines is kept consistent by sending messages

Summary: MESI Cache Coherence Protocol

Sequential Consistency:
- specifies that the system must appear to execute all threads’ loads and stores to all memory locations in a total order that respects the program order of each thread
- a characterization of well-behaved programs
- a model for differing speed of execution
- for fixed paths through the threads and a total order between accesses to the same variables: executions can be illustrated by a happened-before diagram with one process per variable

Cache Coherency:
- A cache coherent system must appear to execute all threads’ loads and stores to a single memory location in a total order that respects the program order of each thread
- MESI cache coherence protocol ensures SC for processors with caches

The MESI Cache Coherence Protocol: Messages

Moving data between caches is coordinated by sending messages [3]:
- Read: sent if CPU needs to read from an address
- Read Response: when in state E or S, response to a Read message, carries the data for the requested address
- Invalidate: asks others to evict a cache line
- Invalidate Acknowledge: reply indicating that a cache line has been evicted
- Read Invalidate: like Read + Invalidate (also called “read with intend to modify”)
- Whiteback: Read Response when in state M, as a side effect notifying main memory about modifications to the cacheline, changing sender’s state to S

We mostly consider messages between processors. Upon Read Invalidate, a processor replies with Read Response/Whiteback before the Invalidate Acknowledge is sent.

Introducing Store Buffers: Out-Of-Order Stores
### Out-of-Order Execution

**Performance Problem:** Writes always stall

**Thread A:**

```c
a = 1; // A.1
b = 1; // A.2
```

**Thread B:**

```c
while (b == 0) {}; // B.1
assert(a == 1); // B.2
```

⇝ CPU A should continue executing after `a = 1`;


**Definition (Total Store Order)**

1. The store order with memory (`⊑`) is total.
2. Stores in program order (`≤`) are embedded into the memory order (`⊑`).
3. Loads preceding another operation (in program order `≤`) are embedded into the memory order (`⊑`).
4. A load's value is determined by the latest write as observed by the local CPU.

```
∀a.i ≤ Ld[b] ̸⇒ St[a] ⊑ Ld[b]
```

### Explicit Synchronization: Write Barrier

- Store buffers apply stores in various orders:
  - FIFO (Sparc/IBM-360)
  - unordered (Spec: PSO)
- Program order still needs to be observed locally
- Stores in program order (`≤`) are guaranteed to be in order any more:

```
St[a] ≤ St[b] ̸⇒ St[a] ⊑ St[b]
```

### Happened-Before Model for PSO

**Thread A:**

```c
a = 1;
b = 1;
```

**Thread B:**

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

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

### PSO Model: Formal Spec [6] [5, p. 58]

**Definition (Partial Store Order)**

1. The store order with memory (`⊑`) is total.
2. Fenced stores in program order (`≤`) are embedded into the memory order (`⊑`).
3. Stores to the same address in program order (`≤`) are embedded into the memory order (`⊑`).
4. Loads preceding another operation (in program order `≤`) are embedded into the memory order (`⊑`).
5. A load's value is determined by the latest write as observed by the local CPU.

```
∀a.i ≤ Ld[b] ̸⇒ St[a] ⊑ Ld[b]
```

### Explicit Synchronization: Write Barrier

Overtaking of messages may be desirable and does not need to be prohibited in general:
- Generalized stone buffers render programs incorrect that assume sequential consistency between different CPUs
- Whenever a store in front of another operation in one CPU must be observable in this order by a different CPU, an explicit write barrier has to be inserted
- A write barrier marks all current store operations in the store buffer
- The next store operation is only executed when all marked stores in the buffer have completed

### Store Buffers

**Abstract Machine Model:** defines semantics of memory accesses

- Put each store into a store buffer and continue execution
- Store buffers apply stores in various orders:
  - FIFO (Sparc/IBM-360)
  - Unordered (Spec: PSO)
- Program order still needs to be observed locally
- Store buffer snoops read channel
- On matching address, return the youngest value in buffer

### TSO in the Wild: x86

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

- FIFO store buffers keep quite strong consistency properties
  - The major obstacle to Sequential Consistency is:
    - `St[a] ≤ Ld[b] ̸⇒ St[a] ⊑ Ld[b]`
    - Modern x86 CPUs provide the `sfence` instruction
    - `sfence` orders all memory instructions:
      - `op1 ≤ spfence ≤ op2`  ⇒  `op1, op2`
  - A fence between write and loads gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)
- Use fences only when necessary

### Happened-Before Model for Write Barriers

**Thread A:**

```c
a = 1;
sfence();
b = 1;
```

**Thread B:**

```c
while (b == 0) {}; // B.1
assert(a == 1); // B.2
```

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

### Happened-Before Model for Write Barriers

**Thread A:**

```c
a = 1;
sfence();
b = 1;
```

**Thread B:**

```c
while (b == 0) {}; // B.1
assert(a == 1); // B.2
```

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

### Explicit Synchronization: Write Barrier

Overtaking of messages may be desirable and does not need to be prohibited in general:
- Generalized store buffers render programs incorrect that assume sequential consistency between different CPUs
- Whenever a store in front of another operation in one CPU must be observable in this order by a different CPU, an explicit write barrier has to be inserted
  - A write barrier marks all current store operations in the store buffer
  - The next store operation is only executed when all marked stores in the buffer have completed

### Further Weakening the Model: O-o-O Reads
Relaxed Memory Order
Communication of cache updates is still costly:
- A cache-intense computation can fill up store buffers in CPUs
- invalidation acknowledgments are delayed on busy caches
  → immediately acknowledge an invalidation and apply it later
  - put each invalidate message into an invalidate queue
  - If a MESI message needs to be sent regarding a cache line in the invalidate queue then wait until the line is invalidated
  - What about sequential consistency?

Happened-Before Model for Invalidate Queues

```
<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1;</td>
<td>b = 1;</td>
</tr>
<tr>
<td>sfence();</td>
<td></td>
</tr>
<tr>
<td>b = 1;</td>
<td></td>
</tr>
</tbody>
</table>

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

RMO Model: Formal Spec [7, p. 290]

- Fenced memory accesses in program order (≤) are embedded into the memory order (≤)
- Stores to the same address in program order (≤) are embedded into the memory order (≤)
- Operations dependent on a load (wi) are dependent on the local CPU
- Memory access to the same address: ws[i] ≤ wj[i] ⇒ ws[i] ≤ wj[i]
- Stores within branched blocks are dependent on branch conditions:
  - (ws[i] ≤ wj[i] ∧ op[i] ≤ op[j]) ⇒ wj[i] ≤ ws[i]

Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.
- Might read an out-of-date value
- Need a way to establish sequential consistency between writes of other processors and local reads
- Insert an explicit read barrier before the read access
- A read barrier marks all entries in the invalidate queue
- The next read operation is only executed once all marked invalidations have completed
- A read barrier before each read gives perfectly consistent read behavior (and is as slow as a system without invalidate queue)
  → match each write barrier in one process with a read barrier in another process

Happened-Before Model for Read Barriers

```
<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1;</td>
<td>b = 0;</td>
</tr>
<tr>
<td>sfence();</td>
<td></td>
</tr>
<tr>
<td>b = 1;</td>
<td></td>
</tr>
</tbody>
</table>
```

Example: The Dekker Algorithm on RMO Systems

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of two processes with busy waiting.

```
P0:  flag[0] = true;
    while (!flag[0]) { // critical section
      P1:  flag[1] = true;
           while (!flag[1]) {
             flag[0] = false;
             turn = 0;
             // busy wait
           }
           flag[1] = false;
           // critical section
           turn = 1;
           flag[0] = false;
```

The Idea Behind Dekker

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

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

Dekker’s Algorithm and RMO

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

Summary: Relaxed Memory Models

Highly optimized CPUs may use a relaxed memory model:
- Reads and writes are not synchronized unless requested by the user
- Many kinds of memory barriers exist with subtle differences
  → ARM, PowerPC, Alpha, ia-64, even x86 (→ SSE Write Combining)
- Memory barriers are the “lowest-level” of synchronization
Discussion

Memory barriers reside at the lowest level of synchronization primitives.
Where are they useful?
- when blocking should not de-schedule threads
- when several processes implement automata and coordinate their transitions via
  common synchronized variables
- protocol implementations
- OS provides synchronization facilities based on memory barriers
Why might they not be appropriate?
- difficult to get right, best suited for specific well-understood algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

Memory Models and Compilers

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

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

Standard Program Optimizations
comprises loop-invariant code motion and dead store elimination, e.g.
- having another thread executing `x = 0;` changes observable behaviour depending on
  optimizing or not
- Compiler also depends on consistency guarantees
- Demand for Memory Models on language level

Memory Models and C-Compilers

Keeping semantics I
```c
int x = 0;
for (int i=0;i<100;i++){
    x = 1;
    printf("%d",x);%
}
```
- Compilers may also reorder store instructions
- Write barriers keep the compiler from reordering across
- The specification of `volatile` keeps the C-Compiler from reordering memory
  accesses to this address
- Java-Compilers even generate barriers around accesses to `volatile` variables

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

Summary

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

Future Many-Core Systems: NUMA

Many-Core Machines: Read Responses congest the bus

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

Cache Coherence vs. Memory Consistency Models

- Sequential Consistency specifies that the system must appear to execute all threads’
  loads and stores to all memory locations in a total order that respects the program
  order of each thread
- A cache coherent system must appear to execute all threads’ loads and stores to a
  single memory location in a total order that respects the program order of each thread

All discussed memory models (SC, TSO, PSO, RMO) provide cache coherence!

Why Memory Barriers are not enough

Often, multiple memory locations may only be modified exclusively by one thread during a
computation.
- use barriers to implement automata that ensure mutual exclusion
- generalize the re-occurring concept of enforcing mutual exclusion

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

A concurrent program consists of several threads that share resources:
- resources can be memory locations or memory mapped I/O
- a file can be modified through a shared handle, e.g.
  - usually invariants must be maintained wrt. resources
  - e.g. a head and tail pointer must overlap a linked list
- an invariant may span multiple resources
- during an update, the invariant may be temporarily "locally broken"
  - multiple resources must be updated together to ensure the invariant

Ideally, a sequence of operations that update shared resources should be atomic [2]. This would ensure that the invariant never seems to be broken.

Definition (Atomic Execution)

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

Wait-Free Updates

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

Program 1
atomic { i++; }

Program 2
j = i; k = j; tmp = i;

Program 3
atomic {
  i = j;
  j = tmp;
}

Answer:
- none by default (even without store and invalidate buffers, why?)
- the load and store (even v=v) may be interleaved with a store from another processor.

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

Marking Statements as Atomic

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

Program 1
atomic { i++; }

Program 2 (fetch-and-add)
atomic { j = i; }

Program 3 (atomic-exchange)
atomic {
  i = j;
  j = tmp;
}

The statements in an atomic block execute as atomic execution:

atomic { i++; j = i; i = tmp; }

atomic only translatable when a corresponding atomic CPU instruction exist

the notion of requesting atomic execution is a general concept

Wait-Free Synchronization

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

Operations update a memory cell and return the previous value.
- the first two operations can be seen as setting a flag i to in [0,1] and returning its previous state.
- the operation implementing programs 4 and 5 is called test-and-set
- the third case generalizes this to setting a variable i to the value of j, if i's old value is equal to k.
- the operation implementing program 6 is called compare-and-swap
  - use as building blocks for algorithms that can fail?

Lock-Free Algorithms

If a wait-free implementation is not possible, a lock-free implementation might still be viable.
Common usage pattern for compare-and-swap:
- read the initial values in i into k (using memory barriers)
- compute a new value j = i/k
- update i to j if k = i still holds
- go to first step if k ≠ j meanwhile
- note: j = k must imply that no thread has updated i

General recipe for lock-free algorithms

- given a compare-and-swap operation for n bytes
- try to group variables for which an invariant must hold into n bytes
- read these bytes atomically
- compute a new value
- perform a compare-and-swap operation on these n bytes
- computing new value must be repeatable or pure

Limitations of Wait- and Lock-Free Algorithms

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

- Lock-Free instructions as building blocks for Locks
Locked Atomic Executions

Practical Implementation of Semaphores

Certain optimisations are possible:

```c
void signal(int *s) {
atomic { *s = *s + 1; /* s has been written to */
atomic { *s = *s - 1; }
while (!(*s));
}
```

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

Special case: initializing with $v = 1$ gives a binary semaphore:
- can be used to block and unblock a thread.
- can be used to protect a single resource.
- in this case the data structure is also called mutex.

Semaphores and Mutexes

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

```c
void signal(int *s) {
atomic { *s = *s + 1; /* critical section */ (*s)++; }
void wait(int *s) {
atomic { *s = *s - 1; }
while (!(*s));
}
```

Busy waiting is avoided:
- a thread waiting to decrease $s$ executes `wait()`.
- `de_schedule()` enters the operating system and inserts the current thread into a queue of threads that will be woken up when $s$ becomes non-zero, usually by monitoring events to $s$ (e.g. PUTEK mmap).
- once a thread calls `wake(s)`, the first thread waiting on $s$ is extracted.
- the operating system lets $t$ return from its call to `de_schedule()`.

Monitors: An Automatic, Re-entrant Mutex

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

Locking each procedure body that accesses a data structure:
- a procedure associated with a monitor acquires a lock on entry and releases it on exit.

When entering a monitored procedure recursively:
- we need a way to release the lock after the return of the last recursive call.

Condition Variables

E.g. a thread $t$ waits for a data structure to be filled:
- `$t$ will call `pop()` and obtain...
- it then has to call again, until an element is available.
- it is busy waiting and produces contention on the lock.

E.g. a thread $t$ waits for a data structure to be emptied:
- `$t$ will call `push()` and deposit...
- it then has to call again, until an element is available.
- it is busy waiting and produces contention on the lock.

Monitors simplify the construction of thread-safe resources.

Signal-And-Urgent-Wait Semantics

Requires one queue for each condition $c$ and a suspended queue $s$:

- a thread who wishes to enter a monitor is added to the queue $c$ if the monitor is occupied.
- a call to `wait` on condition $c$ adds thread to the queue.
- a call to `signal` for $c$ adds thread to queue $s$ (suspended).
- one thread from the $s$ queue is woken up.
- `signal` on a no-op if $c$ is empty.
- if the thread leaves, it wakes up one thread waiting on $c$.
- if $t$ is empty, it wakes up one thread from $s$.

May deadlock the program.
Signal-And-Continue Semantics
Here, the `signal` function is usually called `notify`.

Implementing Condition Variables
We implement the simpler signal-and-continue semantics for a single condition variable:

```c
void cond_wait(mon_t *m) {
    assert(&m->cond != NULL);
    int old_count = m->count;
    m->count = old_count;
    m->tid = tid;
    m->wait = monitor_search(m, cond);
    // body of f
    if (!next_to_enter) de_schedule(&m->tid);
    while (!next_to_enter);
}
void cond_notify(mon_t *m) {
    // wake up other threads
    signal(&m->cond);
}
```

A Note on Notify
With signal-and-continue semantics, two notify functions exist:

- `notify`: wakes up exactly one thread waiting on the condition variable
- `notifyAll`: wakes up all threads waiting on the condition variable

An implementation often becomes easier if `notify` means `notify some`
- Programmer should assume that thread is not the only one woken up

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

```java
class C {
    public synchronized void f() {
        // body of f
        } is equivalent to
    class C {
        public void f() {
            monitor_enter(this);
            // body of f
            monitor_leave(this);
        }
        with object containing:
        protected int mon_var;
        protected int mon_count;
        protected void monitor_enter();
        protected void monitor_leave();
    }
```
Avoiding Deadlocks in Practice

What to do when the lock order contains a cycle?
- determining which locks may be acquired at each program point is undecidable → lock sets are an approximation
- an array of locks in $L$: lock in increasing array index sequence
- if $l \in L(P)$ exists $l'$ to be acquired, → change program; release $l$, acquire $l'$, then acquire $l$ again
  inefficient
- if a lock set contains a summarized lock $a$ and $a$ is to be acquired, we're stuck

Locks Roundup

Transactional Memory [2]
- Idea: automatically convert atomic blocks into code that ensures atomic execution of the statements.

Atomic Execution and Locks
Consider replacing the specific locks with atomic annotations:

```java
void pop() {
  // code
  if (c) signal(q->s);
  // more code
  if (c) signal(q->t);
  // more code
}
```

Concurrency across Languages
In most systems programming languages (C, C++, etc.) we have
- the ability to use atomic operations
  → we can implement wait-free algorithms
In Java, C#, and other higher-level languages
- provide monitors and possibly other concepts
  → often simplify the programming but incur the same problems

<table>
<thead>
<tr>
<th>Language</th>
<th>Concurrent</th>
<th>wait-free</th>
<th>semaphore</th>
<th>mutex</th>
<th>monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, C++</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java, C#</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) some pthread implementations allow a `reentrant` attribute
(b) newer API extensions (Java 8, Concurrent Atomic) and `System.Threading.Interlocked` oper. (c) simulate semaphores using an object with two synchronized methods

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

Idea of muteness: Implement atomic sections with locks:
- a single lock could be used to protect all atomic blocks
- more concurrency is possible by using several locks
- some statements might modify variables that are never read by other threads → no lock required
- statements in one atomic block might access variables in a different order to another atomic block → deadlock possible with locks implementation
- creating too many locks can decrease the performance, especially when required to release locks in $L’$ when acquiring $l’$
  → creating locks automatically is non-trivial and, thus, not standard in programming languages

Summary
Classification of concurrency algorithms:
- wait-free, lock-free, locked
  → nert on the agenda: transactional
Wait-free algorithms:
- never block, always succeed, never deadlock, no starvation
- very limited in expressivity
Look-free algorithms:
- never block, may fail, never deadlock, may starve
- can use several locks to enable more fine-grained concurrency
- may deadlock
- semaphores are not re-entrant, monitors are
  → use algorithm that is best fit

References
System deadlocks.
Transactional memory, 2nd edition.

Abstraction and Concurrency
Two fundamental concepts to build larger software are:
- abstraction: an object storing certain data and providing certain functionality may be used without reference to its internals
- composition: several objects can be combined to a new object without interference

Both, abstraction and composition are closely related, since the ability to compose depends on the ability to abstract from details.
Consider an example:
- a linked list data structure exposes a fixed set of operations to modify the list structure, such as `push()` and `forAll()

The `insert()` operation uses the `forAll()` operation to check if the element already exists and uses `push()` if not.
Wrapping the linked list in a mutex does not help to make the sort thread-safe.
  → wrap the calls in `insert()` in a mutex
  → if other list operations can still be called → use the same mutex
  → unlike sequential algorithms, thread-safe algorithms cannot always be composed to give new thread-safe algorithms

Transactional Memory [2]
- Idea: automatically convert atomic blocks into code that ensures atomic execution of the statements.

```
atomic {
  // code
  if (c) signal(q->s);
  atomic {
    // more code
  }
  // code
}
```

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

Programing Languages
Concurrency: Transactions

Dr. Michael Petter
Winter term 2019

Technische Universität München
Fakultät für Informatik
Dr. Michael Petter
Winter term 2019
Semantics of Transactions

The goal is to use transactions to specify atomic executions. Transactions are rooted in databases where they have the ACID properties:
- atomicity: a transaction complete or seems not to have run
- consistency: each transaction transforms a consistent state to another consistent state
- isolation: among each other, transactions do not interfere
- durability: the effects are permanent (wrt main memory)

Definition (Atomicity)

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

A Software TM Implementation

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

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

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:

Consistency During Transactions

ACID states how committed transactions behave but not what may happen until a transaction commits.
- a transaction, run on an inconsistent state may continue yielding inconsistent states
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

Definition (opacity)

A Software TM Implementation

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

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

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:

Consistency During Transactions

ACID states how committed transactions behave but not what may happen until a transaction commits.
- a transaction, run on an inconsistent state may continue yielding inconsistent states
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

Definition (opacity)

A Software TM Implementation

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

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

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:

Consistency During Transactions

ACID states how committed transactions behave but not what may happen until a transaction commits.
- a transaction, run on an inconsistent state may continue yielding inconsistent states
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

Definition (opacity)

A Software TM Implementation

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

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

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:

Consistency During Transactions

ACID states how committed transactions behave but not what may happen until a transaction commits.
- a transaction, run on an inconsistent state may continue yielding inconsistent states
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

Definition (opacity)

A Software TM Implementation

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

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

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:

Consistency During Transactions

ACID states how committed transactions behave but not what may happen until a transaction commits.
- a transaction, run on an inconsistent state may continue yielding inconsistent states
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

Definition (opacity)

A Software TM Implementation

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

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

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:

Consistency During Transactions

ACID states how committed transactions behave but not what may happen until a transaction commits.
- a transaction, run on an inconsistent state may continue yielding inconsistent states
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

Definition (opacity)

A Software TM Implementation

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

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

Properties of TL2

Opacity is guaranteed by aborting on a read accessing an inconsistent value:
General Challenges when using STM

Executing `atomic` blocks by repeatedly trying to execute them non-atomically creates new problems:

- A transaction might unnecessarily be aborted
- the granularity of what is locked might be too large
- A TM implementation might impose restrictions:
  ```c
  #include <stm.h>
  atomic {
      // lock
      ...
  }
  ...
  ```
- A lock-based commit can cause contention
  - Locks can be shared across processes
  - A non-transactional commit requires a lock

Integrating Non-TM Resources

Allowing access to other resources than memory inside an `atomic` block poses problems:

- storage management, condition variables, volatile variables, input/output
- semantics should be as if `atomic` implements ISA or TSC semantics

Usual choice is one of the following:

- **Prohibit It.** Certain constructs do not make sense. Use compiler to reject these programs.
- **Execute It.** Operations may happen in any order (e.g., file writes usually go to a buffer). Abort if I/O happens.
- **Irrevocably Execute It.** Universal way to deal with operations that cannot be undone: enforce that this transaction terminates (possibly before starting) by making all other transactions conflict.
- **Integrate It.** Re-write code to be transactional: error logging, writing data to a file, ... ✓

→ currently best to use TM only for memory; check if TM supports irrevocable transactions

Hardware Transactional Memory

Transactions of a limited size can also be implemented in hardware:

- additional hardware to track read- and write-sets
- conflict detection is eager using the cache
- additional hardware makes it cheap to perform conflict detection
- A cache-line in the read set is invalidated, the transaction aborts
- A cache-line in the write set must be written-back, the transaction aborts

→ limited by fixed hardware resources, a software backup must be provided

Two principal implementations of HTM:

- **Explicit Transactional Memory:** Each access is marked as transactional
  - similar to `Atomic, Load, Volatile, and Count1`
  - requires separate transaction instructions
  - a transaction has to be translated differently
- **Implicit Transactional Memory:** Only the beginning and end of a transaction are marked
  - same instructions can be used, hardware interprets them as transactional
  - only instructions affecting memory that can be cached can be executed transactionally
  - hardware access, OS calls, page table changes, etc. all abort a transaction

→ provides strong isolation

Hardware Transactional Memory

Hardware transactional memory (HTM) offers strong isolation between transactions.

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical speculative region
- lock: `R5V` instructions provide explicit data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

Intel’s TSX in Broadwell/Skylake microarchitecture (since Aug 2014):

- implicitly transactional, can use normal instructions within transactions
- tracks read/write set using a single transaction bit on cache lines
- provides space for a backup of the whole CPU state (registers, ...) in the speculative region
- may abort at any time due to lack of resources
- aborting in an inner transaction means aborting all of them

Intel provides two software interfaces to TM:

- **Restricted Transactional Memory (RTM)**
- **Hardware Lock Elision (HLE)**

Implementing RTM using the Cache (Intel)

Supporting transactional operations:

- augment each cache line with an extra bit `T`
- introduce a nesting counter `C` and a backup register set `C`
- additional transaction logic:
  - `startTransaction`: increments `C` and, if `C = 0`, backs up registers and flushes buffer
  - subsequent read/write access to a cache line sets `T` to 0
  - applying an invalidate message to a cache line with `T` flags clears all
  - observing a read for a modified cache line with `T` flags issues abort
  - `abort` clears all `T` flags and the store buffer, invalidates the former TM lines, sets `C` to 0 and restores CPU registers
  - `commit` decrements `C` and, if `C = 0`, clears all `T` flags, flushes store buffer

Considerations for the Fall-Back Path

Consider executing the following code concurrently with itself:

```c
int data[100]; // shared
void update(int id, int value) {
    if (_begin() != _BEGIN_STARTED) {
        data[id] = value;
    } else {
        data[id] = value;
    }
}
```

```
```

Several problems:

- The fall-back code may execute racing itself
- The fall-back code is not isolated from the transaction
  → **First idea:** ensure that the fall-back path is executed atomically

→ user must provide fall-back code

Protecting the Fall-Back Path

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

```c
int data[100]; // shared
int mutex;
void update(int id, int value) {
    if (_begin() != _BEGIN_STARTED) {
        if (!mutex>0) _xabort();
    }
    data[id] = value;
    mutex++;
    _xend();
}
```

→ currently best to use TM only for memory; check if TM supports irrevocable transactions

```
```
Happened Before Diagram for Transactions
Augment MESI states with extra bit T. CPU A: d:E5 t:E0, CPU B: d:I, tmp/value

```
int tmp = data[idx];
data[idx] = tmp + value;
```

Common Code Pattern for Mutexes
Using HTM in order to implement mutex:

```
void insert(int id, int val) {
    lock(&mutex);
    data[id] += val;
    unlock(&mutex);
}
```

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

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

Implementing Lock Elision
Transactional operation:
* add a buffer for elided locks, similar to store buffer

```
if (_xbegin() == _XBEGIN_STARTED) {
    int id = _xbegin();
    int tmp = data[idx];
data[idx] = tmp + value;
_xend();
}
```

TM in Practice
Availability of TM Implementations:
GCC can translate accesses in transaction atomic regions into libitm library calls
The library libitm provides different TM implementations:
1. On systems with TSX, it maps atomic blocks to HTM instructions
2. C++03 standardizes synchronization using TSX blocks
3. RTM support slowly introduced to Open/RecHotDelta monitors

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

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

Common Code Pattern for Mutexes
Using HTM in order to implement mutex:

```
int data[100]; // shared
int mutex;
void insert(int id, int val) {
    lock(&mutex);
    data[id] += val;
    unlock(&mutex);
}
```

Implementing Lock Elision
Transactional operation:
* add a buffer for elided locks, similar to store buffer

```
if (_xbegin() == _XBEGIN_STARTED) {
    int id = _xbegin();
    int tmp = data[idx];
data[idx] = tmp + value;
_xend();
}
```

TM in Practice
Availability of TM Implementations:
1. GCC can translate accesses in transaction atomic regions into libitm library calls
2. The library libitm provides different TM implementations:
   1. On systems with TSX, it maps atomic blocks to HTM instructions
   2. C++03 standardizes synchronization using TSX blocks
   3. RTM support slowly introduced to Open/RecHotDelta monitors

Use of hardware lock elision is limited:
* allows to easily convert existing locks
* provides flexible use of HTM

```
if (id < 0) {
    _xbegin();
    int tmp = data[idx];
data[idx] = tmp + value;
_xend();
}
```

References
Transactional Locking II.

Transactional memory, second edition.

Online resources on Intel HTM and GCC’s STM:
  Fast-without-cost-transactional-synchronization-extensions
* http://www.realworldtech.com/xenwell-tm/4/
* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3541.pdf

Outlook
Several other principles exist for concurrent programming:
* non-blocking message passing (actor model)
  * a program consists of actors that send messages
  * each actor has a queue of incoming messages
  * messages can be processed and new messages can be sent
  * special handling of incoming messages
  * example: Erlang, many add-ons to existing languages
* blocking message passing (CSP = calculus, join-calculus)
  * a process sends a message over a channel and blocks until the recipient accepts it
  * channels can be send-only channels \((\rightarrow)\)
  * example: Occam, Occam-\(n\), Go
* (immediate) priority ceiling
  * declare processors with priority and resource that each process may acquire
  * each resource has the maximum ( ceiling ) priority of all processes that may acquire it
  * a process’ priority at run time increases to the maximum of the priorities of held resources
  * the process with the maximum (run-time) priority executes

Programming Languages
Dispatching Method Calls
Dr. Michael Petter
Winter Term 2019
Section 1

Direct Function Calls

Function Dispatching (ANSI C89)

```
#include <stdio.h>
void fun(int i) { }
void bar(int i, double j) { }
int main(){
fun(1);
bar(1,1.2);
void (*foo)(int);
foo = fun;
return 0;
}
```

Overloading Hassles

class D {
public static void p(Object o) { System.out.print(o); }
public int f(int i) { p(i); }
public double f(double d) { p(d); }
int main(){
D d = new D();
D.p(d.f(2)+"\n");
D.p(d.f(2.3)+"\n");
}
```

Section 2

Overloading Function Names

Function Dispatching (ANSI C89)

```
#include <stdio.h>
void fun(int i) { }
void bar(int i, double j) { }
int main(){
fun(1);
bar(1,1.2);
void (*foo)(int);
foo = fun;
return 0;
}
```

Overloading (Java/C++)

```
class D {
public static void p(Object o) { System.out.print(o); }
public int f(int i) { p(i); }
public double f(double d) { p(d); }
int main(){
D d = new D();
D.p(d.f(2)+"\n");
D.p(d.f(2.3)+"\n");
}
```

Overloading with Inheritance (Java)

```
class D {
public static void p(Object o) { System.out.print(o); }
public int f(int i) { p(i); }
public double f(double d) { p(d); }
int main(){
D d = new D();
D.p(d.f(2)+"\n");
D.p(d.f(2.3)+"\n");
}
```

Overloading with Scopes (C++)

```
#include<iostream>
using namespace std;
class B { public:
int f(int i) { cout << "f(int): " ; return i+1; }
public double f(double d) { cout << "f(double): " ; return d+1.3; }
int main(){
D d = new D();
D* pd = new D;
return 0;
}
```

Overloading Hassles

```
class D {
public static void p(Object o) { System.out.print(o); }
public int f(int i) { p(i); }
public double f(double d) { p(d); }
int main(){
D d = new D();
D.p(d.f(2)+"\n");
D.p(d.f(2.3)+"\n");
}
```

$javac Overloading.java
Overloading.java:(?): error: reference to f is ambiguous
▲▲!
Static Methods are Statically Dispatched

Signature
\[ f'(e_1, \ldots, e_n) \]
\[ \xrightarrow{t_0} f(t_1 p_1, \ldots, t_n p_n) \]
dispatches to \[ f' \] \iff \[ f \leq f' \]:
\[ \leq \] is the subtype relation:
\[ R f(T_1, \ldots, T_n) \leq R' f'(T'_1, \ldots, T'_n) \]
\[ \Rightarrow R \leq R' \land T'_i \leq T_i \]

Inside the Java Virtual Machine

Concept of method signatures being more specific to others:

- In Java, to check whether one or more methods are applicable:
  - `LocalMethodMatcher.isApplicable(resolved_method, resolved_klass, resolved_name, resolved_signature, resolved_environment, resolved_env)`
  - `JavaLangMethodMatcher.isApplicable(resolved_method, resolved_klass, resolved_name, resolved_signature, resolved_environment, resolved_env)`
- `java.lang.NoSuchMethodError` or `java.lang.AbstractMethodError`
- The Java platform as example for state of the art OO systems:
  - Concept of method signatures being more specific

Overriding Methods

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}

Object Orientation

Emphasizing the Receiver’s Responsibility

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

Methods are dynamically dispatched

Concrete Method

Provides calling target for a call signature

\[ f'(e_1, \ldots, e_n) \]
\[ \xrightarrow{t_0} f(t_1 p_1, \ldots, t_n p_n) \]
dispatches to \[ t_0 \] specialized by \[ t_1 \]

Signature

Static types of actual parameters:

Specializer

Specialized types to be matched at the call

Methods are dynamically dispatched

Concrete Method

Provides calling target for a call signature

\[ f'(e_1, \ldots, e_n) \]
\[ \xrightarrow{t_0} f(t_1 p_1, \ldots, t_n p_n) \]
dispatches to \[ t_0 \] specialized by \[ t_1 \]

Signature

Static types of actual parameters:

Specializer

Specialized types to be matched at the call

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}

Object Orientation

Emphasizing the Receiver’s Responsibility

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

Methods are dynamically dispatched

Concrete Method

Provides calling target for a call signature

\[ f'(e_1, \ldots, e_n) \]
\[ \xrightarrow{t_0} f(t_1 p_1, \ldots, t_n p_n) \]
dispatches to \[ t_0 \] specialized by \[ t_1 \]

Signature

Static types of actual parameters:

Specializer

Specialized types to be matched at the call

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}

Object Orientation

Emphasizing the Receiver’s Responsibility

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

Methods are dynamically dispatched

Concrete Method

Provides calling target for a call signature

\[ f'(e_1, \ldots, e_n) \]
\[ \xrightarrow{t_0} f(t_1 p_1, \ldots, t_n p_n) \]
dispatches to \[ t_0 \] specialized by \[ t_1 \]

Signature

Static types of actual parameters:

Specializer

Specialized types to be matched at the call

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}

Finding the Most Specific Concrete Method

MemberDefinition matchMethod(Environment env, ClassDefinition accessor, ...
if (!env.isMoreSpecific(tentative, method))
throw new AmbiguousMember(tentative, method);
return tentative;
}
Inside the Hotspot VM

The method lookup recursively traverses the super class chain:

MethodDesc* klass::find_method(ObjArrayDesc* methods, Symbol* name, Symbol* signature) {
    int len = methods->length();
    for (int i = 0; i < len; ++i) {
        MethodDesc* m = methods->at(i);
        if (m->signature() == signature) return m;
    }
    return NULL; // not found
}

Section 4

Multi-Dispatching

Inside the Hotspot VM

MethodDesc* klass::find_method(Symbol* name, Symbol* signature) {
    // For dynamic dispatch on signature (e.g. class C with method C::f()),
    // look in current class first.
    MethodDesc* method = klass::cast(klass)->find_method(name, signature);
    if (method != NULL) return method;
    // search upwards
    for (Class* k = klass->super_class(); k != NULL; k = k->super_class()) {
        method = klass::cast(k)->find_method(name, signature);
        if (method != NULL) return method;
    }
    // search downwards through overloaded methods
    for (Class* k = klass->super_class(); k != NULL; k = k->super_class()) {
        int res = m->name()->fast_compare(name);
        if (res == 0) return m;
        if (res > 0) l = mid + 1;
        else h = mid - 1;
    }
    return NULL;
}

Mini-Quiz: Java Method Dispatching

Can we expect more than Single-Dispatching?

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

So how do we solve the equals() problem?

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

Example: Sets of Natural Numbers

class Natural {
    Natural(int a) { number=Math.abs(a); }
    int number;
    public boolean equalsNatural(Natural a) {
        return a.number == number;
    }
}

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

¡Works ! but needs Dispatcher to know complete class hierarchies

Double Dispatching

Abstract class EqualsDispatcher

boolean dispatch(EqualsDispatcher ed) {
    return ed.dispatch(this);
}

class Natural {
    Natural(int a) { number=Math.abs(a); }
    int number;
    public boolean equals(Natural n) {
        return n.number == number;
    }
    public boolean equals(Object b) {
        return b.equals(this);
    }
}

New class MyHashSet<Natural> that implements the equals problem:

MyHashSet {
    public boolean contains(Natural a) {
        return set.contains(a);
    }
    public boolean contains(Object a) {
        return set.contains(a);
    }
}
### Formal Model of Multi-Dispatching [7]

**Idea**
- Introduce Specializers for all parameters

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

### Implications of the implementation

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

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

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

### Natural Numbers in Multi-Java [3]

```java
class Natural {
  public Natural(int n) {
    number = Math.abs(n);
  }
  // Other methods...
}
```

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

```bash
$ java Natural
✓ Clean Code!
```

### Section 5

**Natively multidispatching Languages**

- Perl 6
- Clojure
- More Creative dispatching in Clojure

### Perl 6

```perl
my Cool $foo;
my Cool $bar;
multi fun(Cool $one, Cool $two) { say "Dispatch 1" }
multi fun(Int $one, Str $two) { say "Dispatch 2" }
$foo=1;
$bar="blabla";
fun($foo,$bar);
$foo="bla";
fun($foo,$bar)
```

### Clojure

- is a REPL dialect for the JVM with:
  - Prefix notation
  - () - Brackets for lists
  - :: - User-defined keyword constructor
  - fn - Vector constructor
  - fn* - Creates a lambda expression
  - (fn [x y] (* x y))
  - derive - Generates Hierarchical relationships
  - (defmulti name dispatch-fn)
  - defmethod - Creates new concrete method
  - defmethod name dispatch-fn [val &fn-tail]

### Principle of Multidispatching in Clojure

```clojure
(defmulti fun (fn [a b] [a b]))
(defmethod fun ::child ::child) "child equals"
(defmethod fun ::parent ::parent) "parent equals"
```

### More Creative dispatching in Clojure

```clojure
(defn salary [amount]
  (cond (< amount 600) ::poor
        (>= amount 5000) ::rich
        :default :middle)
  (print (UniPerson. "Stefan" 2000))
  (print (UniPerson. "Petter" 2000))
  (print (UniPerson. "Seidl" 18000)))
```

### Multidispatching

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

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

### Other Solutions (extract)

- Dylan
- Scala
Lessons Learned

1. Dynamically dispatched methods are complex interaction of static and dynamic techniques.
2. Single Dispatching as in major O-O-Languages.

Section 6
Further materials

Outline

Inheritance Principles

1. Interface Inheritance
2. Implementation Inheritance
3. Dispatching implementation choices

Implementation inheritance

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

Implementation inheritance

- Queue
  - enqueue(x)
- List
  - enqueue(x)
- CircularGraph
  - insertNodeAt(x,i)
- Stack
  - pop()
  - push(x)

Further reading...

OpenJDK 7 Hotspot JIT VM.
ACM Transactions on Programming Languages and Systems (TOPLAS), September 2008.

OpenJDK 7 Javac.

Multiple Inheritance.


“Wouldn’t it be nice to inherit from several parents?”

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

“Implement inheritance

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

Interface inheritance

The classic motivation for inheritance is implementation inheritance:

- Code reusage
- Child specializes parents, replacing particular methods with custom ones
- Parent acts as library of common behaviours
- Implemented in languages like C++ or Lisp

Code sharing in interface inheritance inverts this relation:

- Behaviour contract
- Child provides methods, with signatures predetermined by the parent
- Parent acts as generic code frame with room for customization
- Implemented in languages like Java or C#
Excursion: Brief introduction to LLVM IR

LLVM intermediate representation as reference semantics:

```
;r defection struct definitions
struct B = type { i8, [12 x i8] }, i32 };
{stack} allocation of objects
B = alloca struct B

// call void @_printf('%s

... -emit-llvm source.cpp
```

Retrieve the IR Code of a compilation unit with:

```
clang -O1 -S -emit-llvm source.cpp -o IR.llvm
```

"Now what about polymorphic calls?"

Translation of a method body

class A {
int a; int f(int);
};
class B : public A {
int b; int g(int);
};
class C : public B {
int c
int b
int a
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

"So how do we include several parent objects?"

Object layout

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

Object layout – virtual methods

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

Multiple inheritance class diagram

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

Static Type Casts

```
A
int a
B
int b
C
int c
```

Keeping Calling Conventions

```
A
int a
B
int b
C
int c
```
And what about dynamic dispatching in Multiple Inheritance?

**Ambiguities**

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

Which method is called?

pc = A::f(42);

pc = B::f(42);

**Solution I: Explicit qualification**

Idea: The Compiler introduces a linear order on the nodes of the inheritance graph

**Solution II: Automatical resolution**

L = A B C D E F G =⇒ F → G

L = C D G E F =⇒ G → F

Linearization

1 Inheritance is a uniform mechanism, and its searches (→ total order) apply identically for every linearization of C.

Principle 1: Inheritance Relation

If C(B₁, . . . , Bₙ) =⇒ A → B

Principle 2: Multiplicity Relation

Defined by the succession of multiple parents. Example: C(A, B) =⇒ A → B

In General:

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

Virtual Tables for Multiple Inheritance

```
class A {
    int a; virtual int f(int); }
class B {
    int b; virtual int f(int); virtual g(int); }
class C : public A, public B {
    int c; int e; int f(int);
    C* pb = &b; pb->f(42);
}
```
### Virtual Tables for Multiple Inheritance

```cpp
class A {
    int a; virtual int f(int);
};
class B {
    int b; virtual int g(int);
};
class C {
    public A, public B {
        int c; int f(int);
    }
};
class W {
    int w; virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};
class A : public W, public B {
    ▲▲! Ambiguities
    ⇝ e.g. overriding f in A and B
    ▲▲! Offsets to virtual base
}

```

### Thunks

Solution: thunks

... are trampoline methods, delegating the virtual method to its original implementation

```cpp
f(int) addresses its locals relative to what it assumes to be a C pointer
```

### Common Bases – Duplicated Bases

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

```cpp
A
int f(int)
B
int f(int)
C
int f(int)
```

### Common Bases – Shared Base Class

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

```cpp
A
int f(int)
B
int f(int)
C
int f(int)
```

### Dynamic Type Casts

```cpp
A
int f(int)
B
int f(int)
C
int f(int)
W
int f(int)
```

No guaranteed constant offsets between virtual bases and subclasses — No static casting!

Dynamic casting makes use of offset-to-top
Again: Casting Issues

```
class W { virtual int f(int); };
class A : virtual W { int a; };
class B : virtual W { int b; };
class C : virtual W { int c; };
b = f(42);
w = new C();
\vptr{w} = \vptr{a} + \vptr{f(42)};
```

In a conventional thunk, \(\vptr{c}\) will adjust the \(\vptr{a}\) pointer with a statically known constant to point to \(\vptr{c}\)

Virtual Thunks

```
class W { ...
virtual void g(int);
};
class A : public virtual W {...};
class B : public virtual W ...
%6 ; navigate to vcalloffset+ Wtop
%8 = bitcast i8* %7 to %class.B*
call void @_g(%class.B* %8, i32 %i)
ret void
```

Virtual function pointers

Virtual call offsets per virtual function for adjusting this dynamically
- offset to top of an enclosing objects heap representation
- start pointer to an RTTI object (not relevant for us
- virtual function pointers for resolving virtual methods

Virtual Base classes have virtual thunks which look up the offset to adjust the \(\vptr{a}\) pointer to the correct value in the virtual table!

Polemics of Multiple Inheritance

Full Multiple Inheritance (FMI)

- Removes constraints on parents in inheritance
- More convenient and simple in the common cases
- Occurrence of diamond pattern not as frequent as discussions indicate

Multiple Interface Inheritance (MI)

- Simpler implementation
- Interfaces and aggregation already quite expressive
- Too frequent use of FMI considered as flaw in the class hierarchy design

Lessons Learned

Lessons Learned

- Different purposes of inheritance
- Heap Layouts of Hierarchically constructed objects in C++
- Virtual Table layout
- LLVM IR representation of object access code
- Linearization as alternative to explicit disambiguation
- Pitfalls of Multiple Inheritance

Further reading...

A monotonic superclass linearization for dylan.
```
Public: I. Object-Oriented Programming Systems, Languages, and Applications
```

Mini Seminars

- SCoCC in Multicore Architectures with Cache (Meiner/Sorin 2006/2009)
- Litmus Testing Memory Models: Hardware 7
- The Linux Kernel Memory Model
- A Formal Analysis of the NVIDIA PTX Memory Consistency Model (2019)
- Transactional Memory Systems other than TIS: IBM Power 8 / BlueGene / zEnterprise
- Lambda Calculus: Y Combinator and Recursion / SKI Combinator
- Templates vs. Inheritance

Programming Languages

- Hints and Traits

Dr. Michael Petter
Winter 2019/20
What modularization techniques are there besides multiple implementation inheritance?

**Outline**

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

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

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

- **Reusability ≡ Inheritance?**

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

- **The Adventure Game**

  - Door
    - ShortDoor
    - LockedDoor
    - ShortLockedDoor

  - Issues:
    - Aggregation & Inheritance
    - Door must explicitly provide chaining
    - Doorlike must anticipate wrappers

  - Solution:
    - Multiple Inheritance ✓

- **The Wrapper**

  - FileInputStream
  - SocketStream

  - Issues:
    - Unclear relations

  - Solution:
    - Cannot inherit from both in turn with Multiple Inheritance
    - (Many-to-One instead of One-to-Many Relation)

- **The Wrapper – Aggregation Solution**

  - FileInputStream
  - SocketStream

  - Issues:
    - Duplicate

  - Solution:
    - Needs common ancestor

- **The Wrapper – Multiple Inheritance Solution**

  - FileInputStream
  - SocketStream

  - Issues:
    - Inappropriate Hierarchies

- **Fragility**

  - Issues:
    - Inappropriate Hierarchies

  - Solution:
    - Implement methods (acquireLock/releaseLock) to high

- **(De-)Composition Problems**

  - All the problems of
    - Relation
    - Duplication
    - Inheritance

  - are centered around the question

    "How do I distribute functionality over a hierarchy?"

    → Functional (de-)composition
**Classes and Methods**

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

Classes map names to elements from the flat lattice \( R \) (called bindings), consisting of:
- method bodies \( b, B \) or classes \( c, C \)
- \( c \in C \) \iff \( \exists n \in \text{pre}(c) \) \( .c(n) = \bot \)
- \( c \in C \) \iff \( \forall n \in \text{pre}(c) .\bot \sqsubseteq c(n) \sqsubseteq \top \)

**Example: Smalltalk-Inheritance**

Smalltalk inheritance is the binary operator \( \circ : C \times C \rightarrow C \), defined by

\[
(c \circ b)(x) = \begin{cases} 
  c(x) & \text{if } x \in \text{pre}(c) \\
  b(x) & \text{otherwise}
\end{cases}
\]

Example: Doors

\[
\begin{align*}
\text{Door} & = \{\text{canPass} \mapsto \top, \text{canOpen} \mapsto \bot\} \\
\text{LockedDoor} & = \{\text{canOpen} \mapsto 04204711\} \circ \text{Door}
\end{align*}
\]

**Excursion: Beta-Inheritance**

In Beta-style inheritance
the design goal is to provide security wrt. replacement of a method by a different method.

**Definition (Beta Inheritance (\(\circ\)))**

Beta inheritance is the binary operator \( \circ : C \times C \rightarrow C \), defined by

\[
(c \circ b)(x) = \begin{cases} 
  c(x) & \text{if } x \in \text{pre}(c) \\
  b(x) & \text{otherwise}
\end{cases}
\]

Example (equivalent syntax):

```java
class Person {
  public String toString(){ return name+" , Ph.D."; }
};

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

**Adventure Game with Mixins**

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

**Definition (\(\text{mixin}\))**

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

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

**Example: Doors**

```java
Locked = \{ canOpen \mapsto 04204711 \}
Short = \{ canPass \mapsto 04204711 \}
...
```

**Computed with Classes and Methods**

**Definition (Family of classes (\(\square\)))**

We call the set of all maps from names to bindings the family of classes \( C \rightarrow R \).

Several possibilities for composing maps \( c : C \rightarrow R \):
- the symmetric join \( \sqcup \), defined componentwise:
  - \((c_1 \sqcup c_2)(x) = b \) if \( b = b_1 \) or \( n \notin \text{pre}(c_1) \)
  - \((c_1 \sqcup c_2)(x) = b_2 \) if \( b = b_2 \) or \( n \notin \text{pre}(c_2) \)

- in contrast, the asymmetric join \( \sqcup \), defined componentwise:
  - \((c_1 \sqcup c_2)(x) = c_1(x) \) if \( x \in \text{pre}(c_1) \)
  - \((c_1 \sqcup c_2)(x) = c_2(x) \) otherwise
Wrapper with Mixins

Mixins for wrappers
- avoids duplication of read/write code
- keeps specialization
- even compatible to single inheritance systems

Mixins on Implementation Level

class Door {
boolean canOpen(Person p)...
boolean canPass(Person p)...
}
mixin Locked {
boolean canOpen(Person p)...
}
mixin Short {
boolean canOpen(Person p)...
}

Simulating Mixins in C++
template <class Super>
class LogOpenClose : public Super {
public: virtual void close(){
Super::close();
};
private: virtual void log(char*s) { ... };
};
class MyDocument : public SyncRW<LogOpenClose<Document>> {}

Ok, ok, show me a language with native mixins!

Simulating Mixins in C++
template <class Super>
class SyncRW : public Super {
public: virtual int read(){
Super::read();
return result;
};
protected: virtual void log(char*s) { ... };
protected: virtual void close(){
Super::close();
};
};
class MyDocument : public SyncRW<LogOpenClose<Document>> {}

Surely multiple inheritance is powerful enough to simulate mixins?

True Mixins vs. C++ Mixins

True Mixins
- natively supported
- Composable mixins
- Hassle-free simple alternative to multiple inheritance

C++ Mixins
- Mixins reduced to templated superclasses
- Can be seen as coding pattern
- C++ Type system not modular
- Mixins have to stay source code

Common properties of Mixins
- Linearization is necessary
- Exact sequence of Mixins is relevant
Lack of Control
- Common base classes are shared or duplicated at class level
- Interfaces seem to be hierarchical
- Functionality seems to be modular

Control
- No fine-grained specification of duplication or sharing
- No explicit disambiguation with aliasing and exclusion
- Non-overridden trait methods have the same semantics as class methods

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

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

Central Idea
- Separate interface creation from modeling hierarchies and composing functionality
  - Use interfaces to design hierarchical signature propagation
  - Use traits as modules for assembling functionality
  - Use classes as frames for entities, which can create objects

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

Central Idea
- Separate interface creation from modeling hierarchies and composing functionality
  - Use interfaces to design hierarchical signature propagation
  - Use traits as modules for assembling functionality
  - Use classes as frames for entities, which can create objects

Traits – Concepts
Trait composition principles
- Flat ordering: All traits have the same precedence under +
- Precedence: Under asymmetric join \( \sqcup \), class methods take precedence over trait methods
- Flattening: After asymmetric join \( \sqcup \), class methods override trait methods and sort out conflicts (\( \sqcap \))

Conflicts
- arise if composed traits map methods with identical names to different bodies

Conflict treatment
- Methods can be aliased (\( \sqcup \))
- Methods can be excluded (\( \sqcap \))
- Class methods override trait methods and sort out conflicts (\( \sqcap \))

Extension Methods (C#)
Central idea:
Uncouple method definitions from class bodies.

Purpose:
- Retrospectively add methods to complex types
- Especially provide definitions of interface methods
- Poor man’s multiple inheritance!

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

Excerpt from the Java 8 API documentation for class Properties:
"Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not strings. The putProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-string key or value, the call will fail..."

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

Can we augment classical languages by traits?
Extension Methods as Traits

- Transparently extend arbitrary types externally
- Provide quick relief for plagued programmers
- Interface declarations empty, thus kind of purposeless
- Flattenning not implemented
- Static scope only

Static scope of extension methods causes unexpected errors:

```java
public interface Locked {
    public boolean canOpen(Person p);
    public boolean canPass(Person p);
}
```

Virtual Extension Methods (Java 8)

Java 8 advances one step further:

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

Traits as General Composition Mechanism

- Central Idea
Separate class generation from hierarchy specification and functional modelling
- Simplified multiple Inheritance without adverse effects

Traits in Squeak

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

Syntax:

- name: param1 and: param2
  declares method name with param1 and param2
- id: expr
  declares Variables id1 and id2
- expr: assign
  assignment
- object name: content
  sends message name with content to object (≡call: object.name(content))
- line terminator
- return statement

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

All different kinds of type expressions:

- Definition of curried second order type operators + Linearization
- Finegrained flat-ordered composition of modules
- Definition of (local) partial order on precedence of types wrt. MRO
- Combination of principles

Explicitly: Traits differ from Mixins

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

Further reading...

   Mixin-based inheritance.
   European conference on object-oriented... A. P. Black.
   Traits: Composable units of behaviour.

Simplified multiple Inheritance without adverse effects
"Let's go back to basic concepts – Lua"

Motivation – Polemic

Bothersome features

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

"Why bother with modelling types for my quick hack?"

Introducing Structure

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

Basic Types and Values

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

"So far nothing special – let's compose types"
Table Behaviour

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

- meta = {} -- create as plain empty table
  function meta.__tostring(person) -- 0x7816
    return person.prefix .. ' ' .. person.name
  end
  function meta.__index(tbl, key) -- 0x7832
    return tbl.prototype[key]
  end
- add, mul, sub, div, pow, concat, unm
- eq, lt, le

Overload operators like
Overload comparators like

Delegation

person
__index
__tostring 0x7816
__meta
job
__meta nilname Petter

Dr. prefix
job
meta
meta

function meta.__tostring(tbl) -- 0x7816
  return person.prefix .. ' ' .. person.name
end

Delegation

Function meta.__tostring(person) -- 0x7816
  return person.prefix .. ' ' .. person.name
end

Delegation 2

Function meta.__tostring(person) -- 0x7816
  return person.prefix .. ' ' .. person.name
end

Delegation 3

getmetatable(meta) -- install metatable

meta = {}

Account withdraw(mikes,10) -- withdraw independently
meta = {} -- create plain empty table

function Account.withdraw(acc, val)
  acc.balance=acc.balance-val
end

Account.__index=Account -- share Account's functions

function Account:toString()
  return 'Balance is '..Account.balance
end

Object Oriented Programming

▲▲! so far no concept for multiple objects

Account = { balance=0 }

function Account.withdraw (val)
  Account.balance=Account.balance-val
end

function Account:toString()
  return 'Balance is '..Account.balance
end

Introducing Identity

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

function Account.withdraw (acc, val)
  acc.balance=acc.balance-val
end

function Account:toString()
  return 'Balance is '..Account.balance
end

Account.__index=Account -- share Account's functions

function Account:withdraw(mikes)
  mikes.withdraw(mikes,10) -- withdraw independently
end

function Account:toString()
  return 'Balance is '..Account.balance
end

Introducing Identity

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

function Account.withdraw (acc, val)
  acc.balance=acc.balance-val
end

function Account:toString()
  return 'Balance is '..Account.balance
end

Account.__index=Account -- share Account's functions

function Account:withdraw(mikes)
  mikes.withdraw(mikes,10) -- withdraw independently
end

function Account:toString()
  return 'Balance is '..Account.balance
end

Introducing “Classes”

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

function Account.withdraw (val)
  self.balance=self.balance-val
end

function Account:toString()
  return 'Balance is '..self.balance
end

Account = Account.new(template) -- create instance

container.new(template) -- initialize
getmetatable(container.__index)=Account
metatable(Account).__tostring = Account:toString

return template
end

giro = Account.new({balance=10}) -- create instance

giro:withdraw(10)

print(giro)
**Inheriting Functionality**

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

```lua
class LimitedAccount
  balance: nil
  limit: 100
end

specialgiro = LimitedAccount:new()
specialgiro:withdraw(90)
print(specialgiro)
```

**Multiple Inheritance**

Delegation leads to chain-like inheritance

```lua
function LimitedAccount:withdraw(val)
  if (self.balance+self.limit < val) then
    error("Limit exceeded")
  end
  Account:withdraw(self, val)
end

specialgiro = LimitedAccount:new()
specialgiro:withdraw(90)
print(specialgiro)
```

**Differential description possible in child class style**

```lua
function LimitedAccount:withdraw(val)
  if (self.balance+self.limit < val) then
    error("Limit exceeded")
  end
  Account:withdraw(self, val)
end

specialgiro = LimitedAccount:new()
specialgiro:withdraw(90)
print(specialgiro)
```

**Easily creating particular singletons**

```lua
specialgiro = LimitedAccount:new()
specialgiro:withdraw(90)
print(specialgiro)
```

**Further Topics in Lua**

- Coroutines
- Closures
- Bytecode & Lua-VM

**Implementation of Lua**

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

```lua
Header
100
nil
200
300
nil
```

**Lessons Learned**

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

**Further Reading...**


**“Is modularity the key principle to organizing software?”**

**Motivation**

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

**Lessons Learned**

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

**Further Reading...**


**“Is modularity the key principle to organizing software?”**

**Motivation**

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

**Lessons Learned**

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

**Further Reading...**


**“Is modularity the key principle to organizing software?”**

**Motivation**

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

**Lessons Learned**

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

**Further Reading...**


**“Is modularity the key principle to organizing software?”**

**Motivation**

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

**Lessons Learned**

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

**Further Reading...**


**“Is modularity the key principle to organizing software?”**

**Motivation**

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

**Lessons Learned**

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

**Further Reading...**

Functional decomposition
Compiler
Aspect oriented decomposition
Aspect
Weaver

**System Decomposition in Aspects**

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

~~ AspectJ

**Adding External Definitions**

<table>
<thead>
<tr>
<th>inter-type declaration</th>
</tr>
</thead>
</table>
| class Expr {}
  class Const extends Expr {
    public int val;
    public Const(int val) {
      this.val=val;
    }
  }
  class Add extends Expr {
    public Expr l, r;
    public int eval() { return l.eval() + r.eval(); }
    public Add(Expr l, Expr r) {
      this.l=l;this.r=r;
    }
  }
  aspect ExprUtil {
    abstract int eval();
    abstract class Expr {
      // aspectj-patched code
      abstract int eval();
    }
    class Const extends Expr {
      public int val;
      public Const(int val) {
        this.val=val;
      }
    }
    class Add extends Expr {
      public Expr l, r;
      public int eval() { return l.eval() + r.eval(); }
      public Add(Expr l, Expr r) {
        this.l=l;this.r=r;
      }
    }
    class Add extends Expr {
      public Expr l, r;
      public int eval() { return l.eval() + r.eval(); }
      public Add(Expr l, Expr r) {
        this.l=l;this.r=r;
      }
    }
    }

<table>
<thead>
<tr>
<th>equivalent code</th>
</tr>
</thead>
</table>
| // aspectj-patched code
| abstract class Expr {
|   abstract int eval();
| }
| class Const extends Expr {
|   public int val;
|   public Const(int val) {
|     this.val=val;
|   }
| } |
| aspect ExprUtil {
|   abstract int eval();
|   abstract class Expr {
|     // aspectj-patched code
|     abstract int eval();
|   }
| } |

**Join Points**

Well-defined points in the control flow of a program
- method/constr. call
- method/constr. execution
- field get
- field set
- exception handler execution
- class initialization
- object initialization

executing the actual method-call statement
the individual method is executed
a field is read
a field is set
an exception handler is invoked
static initializers are run
dynamic initializers are run

**Advice**

... are method-like constructs, used to define additional behaviour at joinpoints:
- before(formal)
- after(formal)
- after(formal) returning (formal)
- after(formal) throwing (formal)

For example:

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

**Around Advice**

Unusual treatment is necessary for
- type around(formal)

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

```java
aspect Doubler {
  int around(int i): call(int C.foo(int)) & args(1) {
    int prev = proceed(i+2);
    return prev/2;
  }
}
```
### Method Related Designators

#### Class MyClass
```java
public String toString() {
    return "silly me ";
}
```
```java
public static void ... : calltostring() || exectostring() {
System.out.println("advice!");
} }
advice!
advice!
advice!
silly me silly me
```

#### Pointcut Designator Primitives

<table>
<thead>
<tr>
<th>Type based</th>
<th>Flow and State Based</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>target(typeorid)</code></td>
<td><code>cflow(arbitrary pointcut)</code></td>
<td>-</td>
</tr>
<tr>
<td><code>within(typepattern)</code></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td><code>withincode(methodpattern)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matches join points of any kind which are referring to the receiver of type `typeorid` is contained in the class body of type `typepattern` is contained within the method defined by `methodpattern`

#### Which advice is served first?

Advises are defined in different aspects

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

Advises are defined in the same aspect

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

#### Woven JVM Code

```java
aspect MyAspect {
    pointcut settingconst(): set(int Const.val);
    before () : settingconst() ... invokevirtual #79 // Method MyAspect.ajc$before$MyAspect$2$704a2754:()V
    ...
}
```
Woven JVM Code

```java
aspect MyAspect {
    pointcut callingtostring():
        call (String Object.toString()) && target(Expr);
    before () : callingtostring() {
        System.out.println("calling");
    }
}
```

### Implementation – Summary

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

---

**Further reading...**


---

**Motivation**

- Aspect Oriented Programming establishes programmatic refinement of program code
- How about establishing support for program refinement in the language concept itself?
- Treat program code as data

---

**Metaprogramming**

- Treat programs as data
- Read, analyse or transform (other) programs
- Program modifies itself during runtime

---

**Let’s write a program, which writes a program**

Learning outcomes:
- Compilers and Compiler Tools
- Preprocessors for syntax rewriting
- Reflection and Metaclasses
- Metadataset Protocol
- Macros

---

**Property Based Crosscutting**

- after(int i) : call(void h()) && Expr one = new Const(1);
- Proceed: return ret / 2;
- Poor runtime performance

---

**Around/Proceed – via Procedures**

- Injecting advice inline of original call and outsource to an exploit method — all of it in JVM, disassembled to equivalent:

```
// aspect patched code
public static void main(String[] args){
    C c = new C();
    int ret = foo_aroundBody0(c,temp);
}
```

---

**Aspects Orientation**

**Pre**
- Untangling of concerns
- Late extension across boundaries of hierarchies
- Aspects provide another level of abstraction

**Centra**
- Weaving generates runtime overhead
- Nontransparent control flow and interactions between aspects
- Debugging and Development needs IDE Support

---

**Programming Languages**

**Metaprogramming**

Dr. Michael Petter
Winter 2019/20
Codegeneration Tools

Compiletime-Codegeneration

All in one

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

Compiletime-Codegeneration

Prepend code to usercode
if (1)
/* prepended code */
goto body;
else
body:
{/* block ... (1)
while(1)
if (1)yes
/* user block */
no
no
yes
yes
goto
continue
break
no
/*appended code*/
yesbreak

Compiletime-Codegeneration

Append code to usercode
if (1)
/* appended code */
goto body;
else
body:
{/* block following the macro */}

Example: C Preprocessor (CPP)

#define ATOMIC (lock) \
{ acquire(lock); \ 
/* user code */ \ 
release(lock); 
}
#define label(prefix, lnum) concat_(prefix,lnum)
#define ATOMIC (globallock) {  
1--;  
1++;  
}
#define ATOMIC (mylock) {  
i--;  
i++;  
}

Example application: Language constructs

ATOMIC (globallock) {
  i--;  
  i++;  
}

How can we bind the block, following the ATOMIC to the usercode fragment? Particularly in a situation like this?

Compiletime-Codegeneration

String Rewriting Systems

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

Example: C Preprocessor (CPP)

#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))
#define ATOMIC (lock) \
{ acquire(lock); \ 
/* user code */ \ 
release(lock); 
}
#define label(prefix, lnum) concat_(prefix,lnum)
#define ATOMIC (globallock) {  
1--;  
1++;  
}
#define ATOMIC (mylock) {  
i--;  
i++;  
}

Homoiconic Programming

Homoiconicity

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

Metaclasses and Metaobject Protocol
(Hygienic) Macros

How can we bind the block, following the expanded macro?

We explicitly want to imitate constructs like while loops, thus we do not want to use round brackets for code block delimiters.
Reflection

Reflective Metaprogramming

Metaclasses (→ code is data)
Example: Java Reflection / Metaclass java.lang.Class
static void fun(String param){
    Object incognito = Class.forName(param).newInstance();
    Class meta = incognito.getClass(); // obtain Metaclass
    Field[] fields = meta.getDeclaredFields();
    for(Field f : fields){
        Class t = f.getType();
        Object v = f.get(o);
        if(t == boolean.class && Boolean.FALSE.equals(v))
            // found default value
        else if(t.isPrimitive() && v == null)
            // found default value
        } }

Metaobject Protocol (MOP[1])
Example: Lisp’s CLOS metaobject protocol
... offers an interface ...

Homoiconic Runtime-Metaprogramming

Clojure[2]

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

S-Expressions

S-Expressions are either

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

Remark: Established shortcut notation for lists:
\([x z z x]\) \(\equiv\) \([x. (z . (z . ())))\)

Macros

Macros are configurable syntax/parse tree transformations.

Example: While loop:
\(\text{macroexpand} \left(\text{while} \ a \ b)\)
\(\equiv\) \((\text{loop} \ [\text{binding}\*] \ e)\)
\(\text{recur} \ e\) ; rebinds and jumps to loop or fn

Metaobject Protocol

Metaobject Protocol (MOP[1])
Example: Lisp’s CLOS metaobject protocol
... offers an interface ...

Homoiconic Runtime-Metaprogramming

Special Forms

Special forms differ in the way that they are interpreted by the clojure runtime from the standard evaluation rules.

Language Implementation Idea: reduce every expression to special forms:

\(\text{def} \ x \ \text{don’t init})\)
\(\text{do \ expx}\)
\((\text{if test then else})\)
\((\text{let \ [bindings] \ expr})\)
\((\text{eval \ form})\) evaluates the datastructure form
\((\text{quote \ form})\) yields the unevaluated form
\((\text{var \ symbol})\)
\((\text{name? \ [\text{paramx}] \ expr}))\)
\((\text{loop \ [bindings] \ expr})\)
\((\text{recur \ expr})\) , rebinds and jumps to loop or fn

Macros

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

\(\text{defmacro \ x \ \{\text{quote \ \text{body}}\}}\)
\((\text{quote \ \text{body}})\) ; or equivalently:
\(\text{quote \ \text{body}}\)

Quoting

Macros and functions are directly interpreted, if not quoted via
\(\text{quote \ \text{body}}\)
Homoiconic Runtime-Metaprogramming

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

Further reading...


Programming Languages

From Gotos to Continuations

Dr. Michael Petter
Winter 2019

Plain Old C goto Magic

```c
define foo() {
  jmp_buf context;
  switch( setjmp(context) ) { // TRY
    case 0: // CATCH 0
      printf(“Not reachable”);
      return;
    case 1: { // CATCH 1
      if (number>=0)
        return fun(error_handler,number-1);
      longjmp(*error_handler, 1); // THROW
    }
    case 2: { // CATCH 2
      // handle error
    }
  }
}
```

Stack-Backward Control Flow

Stack Traversal with longjmp

performing control flow jumps across procedure boundaries is the domain of
`longjmp()` ([FreeBSD](https://www.freedesktop.org/software/))

```
int fun(jmp_buf *error_handler, int number) {
  if (number>=0)
    return fun(error_handler,number-1);
  longjmp(*error_handler, 1); // THROW
}
```

a control transfer by manipulating stackpointer and instruction pointer

→ stack traversal only viable to enclosing stack frames, i.e. up the call hierarchy
Exception Stack Traversal with longjmp
...
main()
foo()
fun()
fun()
errorhandler
errorhandler
▲▲! heap objects might leak, after discarding several stack frames

Stackless Coroutines
EcmaScript 6+
var genFx = function*(){
  var i = 0;
  while(true){
    yield i++;
  }
};
var gen = genFx();
while (true){
  var result = gen.next().value;
}

Stackful Coroutines
Luk:
function send (x)
coroutine.yield(s)
end
local producer = coroutine.create(
  function ()
    while true do
      send(io.read())
    end
  end)
Continuation Passing Style (CPS) [8]
Transforming a function \( f::a\rightarrow b \) into a CPS function \( f'::a\rightarrow ((b\rightarrow c)\rightarrow c) \) ... \( f'::a\rightarrow ((b\rightarrow r)\rightarrow r) \)

Continuation Passing Style

Abstracting Contexts

Continuation Passing Style (CPS)
Higher order functions, that receive CPS styled functions as parameters

Composing Code by Continuations
Provide a function compose, that takes a suspended computation \( s::(a\rightarrow r)\rightarrow r \) and a function \( f::(a\rightarrow r)\rightarrow (b\rightarrow r)\rightarrow r \)

Composing Code by Continuations

Excursion: Monads
Essentials of Monads (Wadler 92 [10])

Call with Current Continuation
First implementation in Scheme
callcc takes an argument an abstraction and passes to the abstraction another abstraction, that takes the role of a continuation. When this continuation abstraction is applied, it sends its argument to the continuation of the callcc.

Call with Current Continuation

Continuation Passing Style Monad
The Cont Monad

Continuation Passing Style Monad

Continuation Passing Style (CPS) [8]
Transforming a function \( f::a\rightarrow b \) into a CPS function \( f'::a\rightarrow ((b\rightarrow c)\rightarrow c) \) ... \( f'::a\rightarrow ((b\rightarrow r)\rightarrow r) \)

Continuation Passing Style

Abstracting Contexts

Continuation Passing Style (CPS)
Higher order functions, that receive CPS styled functions as parameters

Composing Code by Continuations
Provide a function compose, that takes a suspended computation \( s::(a\rightarrow r)\rightarrow r \) and a function \( f::(a\rightarrow r)\rightarrow (b\rightarrow r)\rightarrow r \)

Composing Code by Continuations

Excursion: Monads
Essentials of Monads (Wadler 92 [10])

Call with Current Continuation
First implementation in Scheme
callcc takes an argument an abstraction and passes to the abstraction another abstraction, that takes the role of a continuation. When this continuation abstraction is applied, it sends its argument to the continuation of the callcc.
Example: Control Structures with Call/CC

```haskell
Example: Control Structures with Call/CC
Loops with callcc
import Control.Monad.Trans.Class
import ...
Continuations in Haskell via callCC are Multi-Shot Continuations
```

Implementation of Continuations [5]

```haskell
main = flip runContT return $ do
i = \x -> x
k = \x -> (\y -> x)
s = \f -> (\g -> (\x -> f x (g x)))
main = flip runContT return $ do
i = \x -> x
k = \x -> (\y -> x)
s = \f -> (\g -> (\x -> f x (g x)))
```

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

Lessons Learned

1. Simple Gotos
2. Longjumps
3. Set-Swapcontext
4. Exception Handling
5. Stackful-less Continues
6. Single-/Multishot Continuations

Roundup

Applications of call/cc

- Standard Control Structures
- Exception Handling
- Coroutines
- Backtracking
- ...

```
Applications of call/cc

- Standard Control Structures
- Exception Handling
- Coroutines
- Backtracking

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus

References

Introduction to programming with shift and reset.
In ACM SIGPLAN ... Machinery.

Further Topics

- Delimited/Partial Continuations [1]
- Y Combinator
- SKI Calculus