TECHNISCHE FAKULTÄT
UNIVERSITÄT FÜR
MÜNCHEN INFORMATIK

Programming Languages

Concurrency: Memory Consistency

Dr. Michael Petter
Winter term 2019

Thread A

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

Thread B

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

Intuition: the assertion will never fail

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

⇝ in need of defining a *Memory Model*

Memory Models

Memory interactions behave differently in presence of

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

Memory Models are a product of negotiating

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

⇝ Modern Languages include the memory model in their language definition

---

Strict Consistency

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

Definition (Strict consistency)

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

Although idealistically desired, practically not existing

⚠ absolute global time problematic

⚠ physically not possible

⇝ strict consistency is too strong to be realistic

Abandoning absolute time

Thread A

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

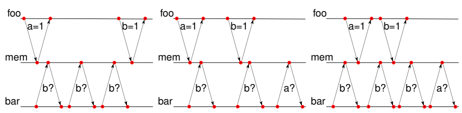
Thread B

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

- initial state of `a` and `b` is `0`
- A writes `a` before it writes `b`
- B should see `b` go to `1` before executing the `assert` statement
- the `assert` statement should always hold

⇝ here correctness means: writing a `1` to `a` *happens before* reading a `1` in `b`

Still, *any* of the following may happen:



⇝ Idea: state correctness in terms of what event *may* happen before another one

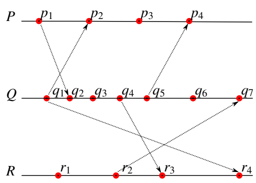
---

Happend-Before Relation and Diagram

Events in a Distributed System

A process as a series of events [2]: Given a distributed system of processes $P, Q, R, \ldots$, each process P consists of events $\bullet p_1, \bullet p_2, \ldots$.

Example:



- event $\bullet p_i$ in process P *happened before* $\bullet p_{i+1}$
- if $\bullet p_i$ is an event that sends a message to Q then there is some event $\bullet q_j$ in Q that receives this message and $\bullet p_i$ *happened before* $\bullet q_j$

The Happened-Before Relation

Definition

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

Observe:

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

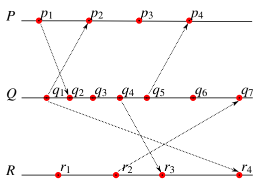
⇝ the $\rightarrow$ relation is a *strict partial order*

Concurrency in Happened-Before Diagrams

Let $a \not\rightarrow b$ abbreviate $\neg(a \rightarrow b)$.

Definition

Two distinct events p and q are said to be *concurrent* if $p \not\rightarrow q$ and $q \not\rightarrow p$.



- $p_1 \rightarrow r_4$ in the example
- p_3 and q_3 are, in fact, concurrent since $p_3 \not\rightarrow q_3$ and $q_3 \not\rightarrow p_3$

Ordering

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

Definition (Clock Condition)

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

$$p \rightarrow q \quad \Longrightarrow \quad C(p) < C(q)$$

For a distributed system the *clock condition* holds iff:

1. p_i and p_j are events of P and $p_i \rightarrow p_j$ then $C(p_i) < C(p_j)$
2. p is the sending of a message by process P and q is the reception of this message by process Q then $C(p) < C(q)$

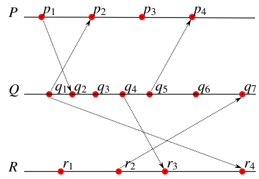
⇝ a logical clock C that satisfies the clock condition describes a *total order* $a < b$ (with $C(a) < C(b)$) that *embeds* the strict partial order $\rightarrow$

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

⇝ use the process model and $\rightarrow$ to define better consistency model

Defining C Satisfying the Clock Condition

Given:



e	p_1	p_2	p_3	p_4
$C(e)$	1	4	7	12

e	q_1	q_2	q_3	q_4	q_5	q_6	q_7
$C(e)$	2	3	5	6	11	13	14

e	r_1	r_2	r_3	r_4
$C(e)$	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

Memory Consistency Models based on the Happened-Before Relation

Happened-Before Based Memory Models

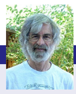
Idea: use happened-before diagrams to model more relaxed memory models.

Given a path through each of the threads of a program:

- consider the actions of each thread as events of a process
- use more processes to model memory
  - here: one process per variable in memory
- $\leadsto$ concisely represent *some* interleavings

$\leadsto$ 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
- of all processors joined were executed in some sequential order

Sequential Consistency applied to Multiprocessor Programs:
Given a program with n threads,

1. for fixed event sequences $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $p_0^n, p_1^n, \ldots$ keeping the program order,
2. executions obeying the clock condition on the p_j^i,
3. all executions have the same result

Yet, in other words:

- 1 defines the *execution path* of each thread
- each execution mentioned in 2 is one *interleaving* of processes
- 3 declares that the result of running the threads with these interleavings is always the same.

Working with Sequential Consistency

Sequential Consistency in Multiprocessor Programs:
Given a program with n threads,

1. for fixed event sequences $p_0^1, p_1^1, \ldots$ and $p_0^2, p_1^2, \ldots$ and $p_0^n, p_1^n, \ldots$ keeping the program order,
2. executions obeying the clock condition on the p_j^i,
3. 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 1 and a total ordering of all operations 2
- add extra processes to model other system components
- the original order 2 becomes a partial order $\rightarrow$
- show that total orderings C' exist for $\rightarrow$ for which the result differs

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

Definition (Sequential Consistency)

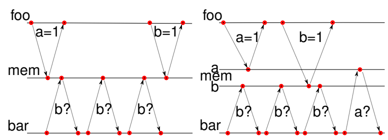
1. Memory operations in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\texttt{Op}_i[a] \leq \texttt{Op}_i[b]' \Rightarrow \texttt{Op}_i[a] \sqsubseteq \texttt{Op}_i[b]'$$
2. A load's value is determined by the latest write wrt. memory order
$$val(\texttt{Ld}_i[a]) = val(\texttt{St}_j[a] \mid \texttt{St}_j[a] = \max_{\sqsubseteq}(\{\texttt{St}_k[a] \mid \texttt{St}_k[a] \sqsubseteq \texttt{Ld}_i[a]\})$$

with

- $\texttt{Op}_i[a]$ any memory access to address a by CPU i
- $\texttt{Ld}_i[a]$ a load from address a by CPU i
- $\texttt{St}_i[a]$ a store to address a 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

Weakening the Model

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



Sequential consistency still obeyed:

- the accesses of `foo` to `a` occurs before `b`
- the first two read accesses to `b` are in parallel to `a=1`

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

Benefits of Sequential Consistency

- concisely represent *all* interleavings that are due to variations in timing
- synchronization using time is uncommon for software

$\leadsto$ a good model for correct behaviors of concurrent programs
$\leadsto$ program results besides SC results are undesirable (they contain *races*)

Realistic model for simple hardware architectures:

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

Not realistic for elaborate hardware with out-of-order stores:

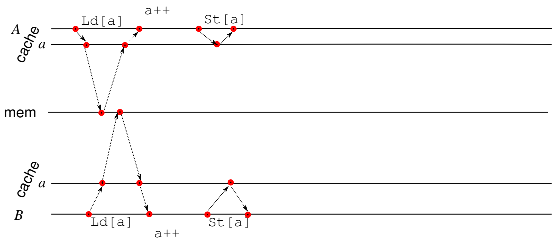
- what other processors see is determined by complex optimizations to cacheline management

$\leadsto$ internal workings of caches

Introducing Caches: The MESI Protocol

Introducing Caches

Idea: each cache line one process



Observations:

⚠ naive replication of memory in cache lines creates *incoherency*

Cache Coherency: Formal Spec [5, p. 14]

Definition (Cache Coherency)

1. Memory operations in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)

$$\texttt{Op}_i[a] \leq \texttt{Op}_i[a]' \Rightarrow \texttt{Op}_i[a] \sqsubseteq \texttt{Op}_i[a]'$$

2. A load's value is determined by the latest write wrt. memory order

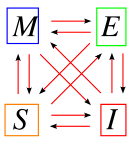
$$val(\texttt{Ld}_i[a]) = val(\texttt{St}_j[a] \mid \texttt{St}_j[a] = \max_{\sqsubseteq}(\{\texttt{St}_k[a] \mid \texttt{St}_k[a] \sqsubseteq \texttt{Ld}_i[a]\})$$

- 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 (⇝ *Cache Coherence Protocol*)

The MESI Cache Coherence Protocol: States [4]

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:

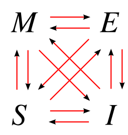
- I: it is *invalid* and is ready for re-use
- S: other caches have an identical copy of this cache line, it is *shared*
- E: the content is in no other cache; it is *exclusive* to this cache and can be overwritten without consulting other caches
- M: the content is exclusive to this cache and has furthermore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

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

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")
- *Writeback: Read Response* when in state M, as a side effect noticing main memory about modifications to the cacheline, changing sender's state to S



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

MESI Example

Consider how the following code might execute:

Thread A

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

Thread B

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

- in all examples, the initial values of variables are assumed to be 0
- suppose that `a` and `b` reside in different cache lines
- assume that a cache line is larger than the variable itself
- we write the content of a cache line as
  - Mx: modified, with value x
  - Ex: exclusive, with value x
  - Sx: shared, with value x
  - I: invalid

MESI Example (I)

Thread A

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

Thread B

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

statement	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
A.1	I	I	I	I	0	0	read invalidate of a from CPU A
	I	I	I	I	0	0	invalidate ack. of a from CPU B
	I	I	I	I	0	0	read response of a=0 from RAM
B.1	M 1	I	I	I	0	0	read of b from CPU B
	M 1	I	I	I	0	0	read response with b=0 from RAM
B.1	M 1	I	I	E 0	0	0	
A.2	M 1	I	I	E 0	0	0	read invalidate of b from CPU A
	M 1	I	I	E 0	0	0	read response of b=0 from CPU B
	M 1	S 0	I	S 0	0	0	invalidate ack. of b from CPU B
	M 1	M 1	I	I	0	0	

MESI Example (II)

Thread A

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

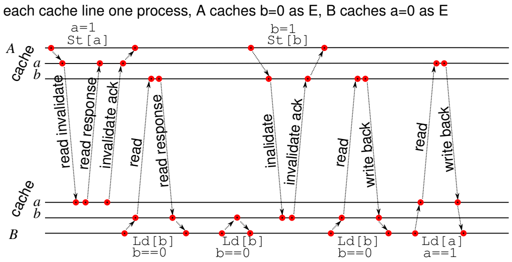
Thread B

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

statement	CPU A a	CPU A b	CPU B a	CPU B b	RAM a	RAM b	message
B.1	M 1	M 1	I	I	0	0	read of b from CPU B
	M 1	M 1	I	I	0	0	write back of b=1 from CPU A
B.2	M 1	S 1	I	S 1	0	1	read of a from CPU B
	M 1	S 1	I	S 1	0	1	write back of a=1 from CPU A
	S 1	S 1	S 1	S 1	1	1	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
A.1	S 1	S 1	S 1	S 1	1	1	invalidate of a from CPU A
	S 1	S 1	I	S 1	1	1	invalidate ack. of a from CPU B
	M 1	S 1	I	S 1	1	1	

MESI Example: Happened Before Model

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



Observations:

- each memory access must complete before executing next instruction ⇝ add edge
- second execution of test `b==0` stays within cache ⇝ no traffic

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

Introducing Store Buffers: Out-Of-Order Stores

Out-of-Order Execution

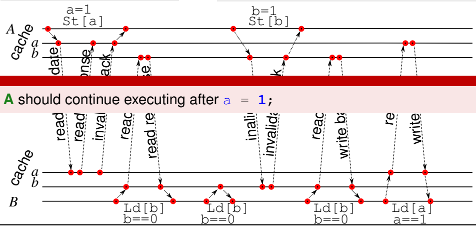
⚠ performance problem: writes always stall

Thread A

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

Thread B

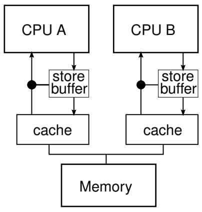
```
while (b == 0) {};    // B.1
assert(a == 1);       // B.2
```



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

Store Buffers

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



- put *each* store into a *store buffer* and continue execution
- Store buffers apply stores in various orders:
  - FIFO (Sparc/x86-*TSO*)
  - unordered (Sparc *PSO*)
- ⚠ program order still needs to be observed locally
  - store buffer snoops read channel and
  - on matching address, returns the youngest value in buffer

TSO Model: Formal Spec [6] [5, p. 42]

Definition (Total Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total
$$\forall_{a,b \in addr\ i,j \in CPU} \quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$
2. Stores in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{St}_i[a] \leq \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$
3. Loads preceding an other operation (wrt. program order $\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{Ld}_i[a] \leq \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$
4. A load's value is determined by the latest write as observed by the local CPU
$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \leq \mathtt{Ld}_i[a]\}))$$

Particularly, one ordering property from SC is not guaranteed:

$$\mathtt{St}_i[a] \leq \mathtt{Ld}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]$$

⚠ Local stores may be observed earlier by local loads then from somewhere else!

Happened-Before Model for TSO

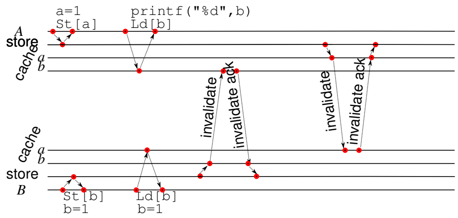
Thread A

```
a = 1;
printf("%d",b);
```

Thread B

```
b = 1;
printf("%d",a);
```

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



TSO in the Wild: x86

The x86 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

$$\mathtt{St}_i[a] \leq \mathtt{Ld}_i[b] \quad \not\Rightarrow \quad \mathtt{St}_i[a] \sqsubseteq \mathtt{Ld}_i[b]$$

  - modern x86 CPUs provide the `mfence` instruction
  - `mfence` orders all memory instructions:

$$\mathtt{Op}_i \leq mfence() \leq \mathtt{Op}_i' \quad \Rightarrow \quad \mathtt{Op}_i \sqsubseteq \mathtt{Op}_i'$$

- a fence between write and loads gives sequentially consistent CPU behavior (and is as slow as a CPU without store buffer)

⇝ use fences only when necessary

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

Definition (Partial Store Order)

1. The store order wrt. memory ($\sqsubseteq$) is total
$$\forall_{a,b \in addr\ i,j \in CPU} \quad (\mathtt{St}_i[a] \sqsubseteq \mathtt{St}_j[b]) \vee (\mathtt{St}_j[b] \sqsubseteq \mathtt{St}_i[a])$$
2. Fenced stores in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{St}_i[a] \leq \mathtt{sfence()} \leq \mathtt{St}_i[b] \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$
3. Stores to the same address in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{St}_i[a] \leq \mathtt{St}_i[a]' \Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[a]'$$
4. Loads preceding another operation (wrt. program order $\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{Ld}_i[a] \leq \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$
5. A load's value is determined by the latest write as observed by the local CPU
$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \leq \mathtt{Ld}_i[a]\}))$$

⚠ Now also stores are not guaranteed to be in order any more:

$$\mathtt{St}_i[a] \leq \mathtt{St}_i[b] \not\Rightarrow \mathtt{St}_i[a] \sqsubseteq \mathtt{St}_i[b]$$

⇝ What about sequential consistency for the whole system?

Happened-Before Model for PSO

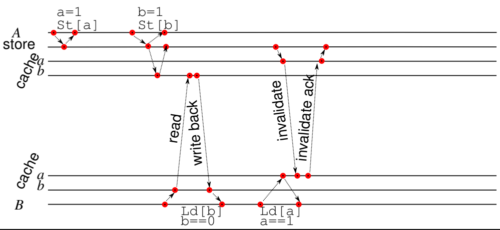
Thread A

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

Thread B

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

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



Explicit Synchronization: Write Barrier

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

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

Happened-Before Model for Write Barriers

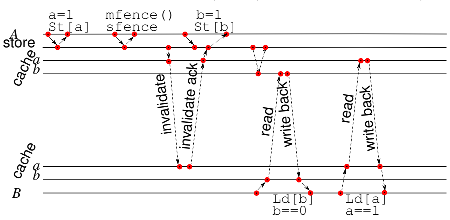
Thread A

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

Thread B

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

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

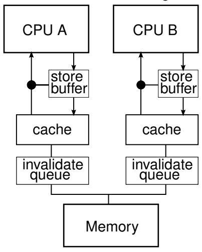


Further weakening the model: O-o-O Reads

Relaxed Memory Order

Communication of cache updates is still costly:

- a cache-intense computation can fill up store buffers in CPUs
- ⇝ waiting for invalidation acknoledgements may still happen
- invalidation acknoledgements are delayed on busy caches



- ⇝ immediately acknowledge an invalidation and apply it later
- put each invalidate message into an *invalidate queue*
- if a *MESI message* needs to be sent regarding a cache line in the invalidate queue then wait until the line is invalidated
- ⚠ local loads and stores do *not* consult the invalidate queue
- ⇝ What about sequential consistency?

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

Definition (Relaxed Memory Order)

1. Fenced memory accesses in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{Op}_i[a] \leq \mathtt{mfence()} \leq \mathtt{Op}_i[b] \Rightarrow \mathtt{Op}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$
2. Stores to the same address in program order ($\leq$) are embedded into the memory order ($\sqsubseteq$)
$$\mathtt{Op}_i[a] \leq \mathtt{St}_i[a]' \Rightarrow \mathtt{Op}_i[a] \sqsubseteq \mathtt{St}_i[a]'$$
3. Operations dependent on a load (wrt. *dependence* $\rightarrow$) are embedded in the memory order ($\sqsubseteq$)
$$\mathtt{Ld}_i[a] \rightarrow \mathtt{Op}_i[b] \Rightarrow \mathtt{Ld}_i[a] \sqsubseteq \mathtt{Op}_i[b]$$
4. A load's value is determined by the latest write as observed by the local CPU
$$val(\mathtt{Ld}_i[a]) = val(\mathtt{St}_j[a] \mid \mathtt{St}_j[a] = \max_{\sqsubseteq} (\{\mathtt{St}_k[a] \mid \mathtt{St}_k[a] \sqsubseteq \mathtt{Ld}_i[a]\} \cup \{\mathtt{St}_i[a] \mid \mathtt{St}_i[a] \leq \mathtt{Ld}_i[a]\}))$$

⚠ Now we need the notion of *dependence* $\rightarrow$:

- Memory access to the same address: $\mathtt{St}_i[a] \leq \mathtt{Ld}_i[a] \quad \Rightarrow \quad \mathtt{St}_i[a] \rightarrow \mathtt{Ld}_i[a]$
- Register reads are dependent on latest register writes:
$$\mathtt{Ld}_i[a]'' = \max_{\leq} (\mathtt{Ld}_i[a]' \mid targetreg(\mathtt{Ld}_i[a]') = srcreg(\mathtt{St}_i[b]) \wedge \mathtt{Ld}_i[a]' \leq \mathtt{St}_i[b]) \quad \Rightarrow \quad \mathtt{Ld}_i[a]'' \rightarrow \mathtt{St}_i[b]$$
- Stores within branched blocks are dependent on branch conditionals:
$$(\mathtt{Op}_i[a] \leq \mathtt{St}_i[b]) \wedge \mathtt{Op}_i[a] \rightarrow condbranch \leq \mathtt{St}_i[b] \quad \Rightarrow \quad \mathtt{Op}_i[a] \rightarrow \mathtt{St}_i[b]$$

Happened-Before Model for Invalidate Queues

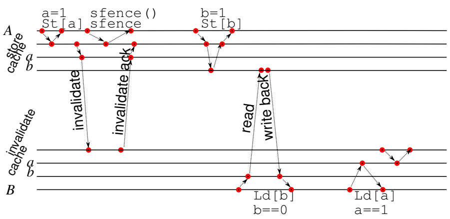
Thread A

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

Thread B

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

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



Explicit Synchronization: Read Barriers

Read accesses do not consult the invalidate queue.

- might read an out-of-date value
- need a way to establish sequential consistency between writes of other processors and local reads
- insert an explicit *read barrier* before the read access
  - a read barrier marks all entries in the invalidate queue
  - the next read operation is only executed once all marked invalidations have completed
- a read barrier *before* each read gives sequentially consistent read behavior (and is as slow as a system without invalidate queue)

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

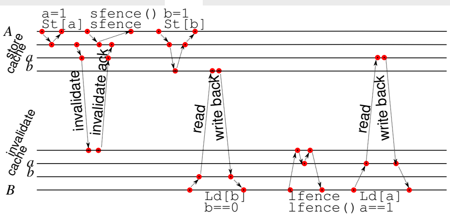
Happened-Before Model for Read Barriers

Thread A

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

Thread B

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



Example: The Dekker Algorithm on RMO Systems

Using Memory Barriers: the Dekker Algorithm

Mutual exclusion of *two* processes with busy waiting.

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

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

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

In process P_i:

- if P_{1-i} does not want to enter, proceed immediately to the critical section
- ⇝ `flag[i]` is a *lock* and may be implemented as such
- if P_{1-i} also wants to enter, wait for `turn` to be set to `i`
- while waiting for `turn`, reset `flag[i]` to enable P_{1-i} to progress

Dekker's Algorithm and RMO

Problem: Dekker's algorithm requires sequential consistency.
Idea: insert memory barriers between all variables common to both threads.

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

- insert a load memory barrier `lfence()` in front of every read from common variables
- insert a write memory barrier `sfence()` after writing a variable that is read in the other thread
- the `lfence()` of the first iteration of each loop may be combined with the preceding `sfence()` to an `mfence()`

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

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

After Optimization

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

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

Keeping semantics II

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

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

Summary

Learning Outcomes

1. Strict Consistency
2. Happened-before Relation
3. Sequential Consistency
4. The MESI Cache Model
5. TSO: FIFO store buffers
6. PSO: store buffers
7. RMO: invalidate queues
8. Reestablishing Sequential Consistency with memory barriers
9. Dekker's Algorithm for Mutual Exclusion

Future Many-Core Systems: NUMA

Many-Core Machines' Read Responses congest the bus

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

⚠ But in general, Symmetric multi-processing (SMP) has its limits:

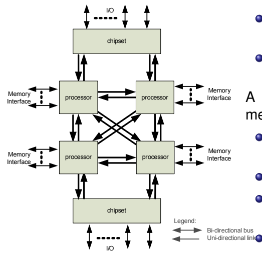
- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus

⇝ use a bus locally, use point-to-point links globally: *NUMA*

- *non-uniform memory access* partitions the memory amongst CPUs
- a directory states which CPU holds a memory region
- Interprocess communication between Cache-Controllers (*ccNUMA*): onchip on Opteron or in chipset on Itanium

Overhead of NUMA Systems

Communication overhead in a NUMA system.



source: [1]

- 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 acknowledge to A

References

[1] Intel.
An introduction to the intel quickpath interconnect.
Technical Report 320412, 2009.

[2] L. Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Commun. ACM, 21(7):558–565, July 1978.

[3] P. E. McKenny.
Memory Barriers: a Hardware View for Software Hackers.
Technical report, Linux Technology Center, IBM Beaverton, June 2010.

[4] M. S. Papamarcos and J. H. Patel.
A low overhead coherence solution for multiprocessors with private cache memories.
In *In Proc. 11th ISCA*, pages 348–354, 1984.

[5] D. J. Sorin, M. D. Hill, and D. A. Wood.
A Primer on Memory Consistency and Cache Coherence.
Morgan & Claypool Publishers, 1st edition, 2011.

[6] C. SPARC International, Inc.
The SPARC Architecture Manual: Version 8.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[7] C. SPARC International, Inc.
The SPARC Architecture Manual (Version 9).
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

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!

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Atomic Executions, Locks and Monitors

Dr. Michael Petter

Winter 2019

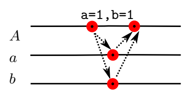
Why Memory Barriers are not Enough

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*:



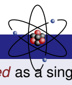
Atomic Executions

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 retained wrt. resources
  - e.g. a head and tail pointer must delimit 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.

Overview

We will address the *established* ways of managing synchronization. The presented techniques

- are available on most platforms
- likely to be found in most existing (concurrent) software
- provide solutions to common concurrency tasks
- are the source of common concurrency problems

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

Learning Outcomes

1. Principle of Atomic Executions
2. Wait-Free Algorithms based on Atomic Operations
3. Locks: Mutex, Semaphore, and Monitor
4. Deadlocks: Concept and Prevention

Wait-Free Atomic Executions

Wait-Free Updates

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

Program 1

```
i++;
```

Program 2

```
j = i;
i = i+k;
```

Program 3

```
int tmp = i;
i = j;
j = tmp;
```

Answer:

- none by default (even without store and invalidate buffers, *why?*)
- ⚠ The load and store (even i++'s) may be interleaved with a store from another processor.

All of the programs *can* be made atomic executions (e.g. on x86):

- i must be in memory
- Idea: *lock the cache bus* for an address for the duration of an instruction

Program 1

```
lock inc [addr_i]
```

Program 2 (fetch-and-add)

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

Program 3 (atomic-exchange)

```
lock xchg [addr_i],reg_j
```

Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a *bumper pointer* to allocated memory:

Bumper Pointer Allocation

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

char* alloc(int size) {
  char* start;
  asm("lock; xadd %0, %1" :"=r"(start),"=m"(firstFree):
       "0"(size),"m"(firstFree) :"memory");
  if (start+size>sizeof(heap)) garbage_collect();
  return start;
}
```

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

Thread-safe implementation:

- alloc's core functionality matches Program 2: fetch-and-add

⇝ inline assembler (GCC/AT&T syntax in the example)

Marking Statements as Atomic

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

Program 1

```
atomic {
  i++;
}
```

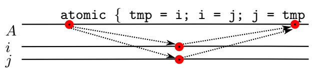
Program 2

```
atomic {
  j = i;
  i = i+k;
}
```

Program 3

```
atomic {
  int tmp = i;
  i = j;
  j = tmp;
}
```

The statements in an atomic block execute as *atomic execution*:



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

Wait-Free Synchronization

Wait-Free algorithms are limited to a single instruction:

- no control flow possible, no behavioral change depending on data
- often, there are instructions that execute an operation conditionally

Program 4

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

Program 5

```
atomic {
  r = b;
  b = 1;
}
```

Program 6

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

Operations *update* a memory cell and *return* the previous value.

- the first two operations can be seen as setting a flag b to $v \in \{0, 1\}$ and returning its previous state.
  - the operation implementing programs 4 and 5 is called *set-and-test*
- the third case generalizes this to setting a variable i to the value of j, if i's old value is equal to k's.
  - the operation implementing program 6 is called *compare-and-swap*

⇝ use as *building blocks* for algorithms that can *fail*

Lock-Free Algorithms

Lock-Free Algorithms

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

Common usage pattern for *compare and swap*:

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

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

General recipe for *lock-free* algorithms

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

⇝ 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

Locks

Definition (Lock)

A lock is a data structure that

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

⚠ may *deadlock* the program

Semaphores and Mutexes

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

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

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

A counting semaphore can track how many resources are still available.

- a thread *acquiring* a resource executes `wait()`
- if a resource is still available, `wait()` returns
- once a thread finishes using a resource, it calls `signal()` to *release*

Special case: initializing with $s = 1$ 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*

Implementation of Semaphores

A *semaphore* does not have to wait busily:

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

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

Busy waiting is avoided:

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

Practical Implementation of Semaphores

Certain optimisations are possible:

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

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

In general, the implementation is more complicated

- `wait()` may busy wait for a few iterations
  - avoids de-scheduling if the lock is released frequently
  - better throughput for semaphores that are held for a short time
- `wake(s)` informs the scheduler that `s` has been written to

⇝ using a semaphore with a single core reduces to

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

Mutexes

One common use of semaphores is to guarantee mutual exclusion.

⇝ in this case, a binary semaphore is also called a *mutex*

e.g. add a lock to the double-ended queue data structure

⚠ decide what needs protection and what not

Monitors: An Automatic, Re-entrant Mutex

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

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

Locking each procedure body that accesses a data structure:

1. is a re-occurring pattern, should be generalized
2. becomes problematic in recursive calls: it blocks

E.g. a thread t waits for a data structure to be filled

- t will call `pop()` and obtain `-1`
- t then has to call again, until an element is available

⇝ t is busy waiting and produces contention on the lock ⚠

Monitor: a mechanism to address these problems:

1. a procedure associated with a monitor acquires a lock on entry and releases it on exit
2. if that lock is already taken by the current thread, proceed

⇝ we need a way to release the lock after the return of the last recursive call

Implementation of a Basic Monitor

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

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

Define `monitor_enter` and `monitor_leave`:

- ensure mutual exclusion of accesses to `mon_t`
- track how many times we called a monitored procedure recursively

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

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

Condition Variables

✓ Monitors simplify the construction of thread-safe resources.

Still: Efficiency problem when using resource to synchronize:

E.g. a thread t waits for a data structure to be filled:

- t will call `pop()` and obtain `-1`
- t then has to call again, until an element is available

⇝ t is busy waiting and produces contention on the lock

Idea: create a *condition variable* on which to block while waiting:

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

Define these two functions:

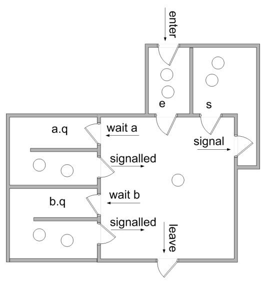
1. `wait` for the condition to become true
   - called while being *inside* the monitor
   - temporarily *releases* the monitor and blocks
   - when *signalled*, re-acquires the monitor and returns
2. `signal` waiting threads that they may be able to proceed
   - one/all waiting threads that called *wait* will be woken up, two possibilities:

     signal-and-urgent-wait : the *signalling* thread suspends and continues once the *signalled* thread has released the monitor

     signal-and-continue the *signalling* thread continues, any *signalled* thread enters when the monitor becomes available

Signal-And-Urgent-Wait Semantics

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



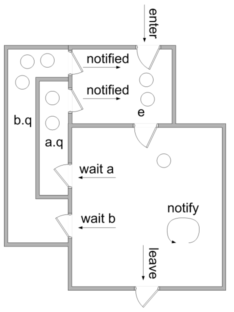
source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

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

⇝ queue s has priority over e

Signal-And-Continue Semantics

Here, the `signal` function is usually called `notify`.



- a call to `wait` on condition a adds thread to the queue $a.q$
- a call to `notify` for a adds one thread from $a.q$ to e (unless $a.q$ is empty)
- if a thread leaves, it wakes up one thread waiting on e

⇝ signalled threads compete for the monitor

- assuming FIFO ordering on e, threads who tried to enter between `wait` and `notify` will run first
- need additional queue s if waiting threads should have priority

Implementing Condition Variables

We implement the simpler *signal-and-continue* semantics for a single condition variable:

⇝ a *notified* thread is simply woken up and competes for the monitor

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

```
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:

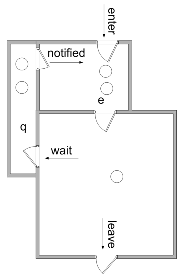
1. `notify`: wakes up exactly one thread waiting on condition variable
2. `notifyAll`: wakes up all threads waiting on a condition variable

⚠ 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#*:



source: http://en.wikipedia.org/wiki/Monitor_(synchronization)

```
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:

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

Deadlocks

Deadlocks with Monitors

Definition (Deadlock)

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

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

Consider this Java class:

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

and two instances:

```
  Foo a = new Foo(), b = new Foo();
  a.other = b; b.other = a;
  // in parallel:
  a.bar() || b.bar();
```

Sequence leading to a deadlock:

- threads A and B execute `a.bar()` and `b.bar()`
- `a.bar()` acquires the monitor of `a`
- `b.bar()` acquires the monitor of `b`
- A happens to execute `other.bar()`
- A blocks on the monitor of b
- B happens to execute `other.bar()`
- ⇝ both *block* indefinitely

How can this situation be avoided?

Treatment of Deadlocks

Observation: Deadlocks occur if the following four conditions hold [1]:

1. *mutual exclusion*: processes require exclusive access
2. *wait for*: a process holds resources while waiting for more
3. *no preemption*: resources cannot be taken away form processes
4. *circular wait*: waiting processes form a cycle

The occurrence of deadlocks can be:

1. *ignored*: for the lack of better approaches, can be reasonable if deadlocks are rare
2. *detection*: check within OS for a cycle, requires ability to *preempt*
3. prevention: design programs to be deadlock-free
4. *avoidance*: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

⇝ *prevention* is the only safe approach on standard operating systems

- can be achieved using *lock-free* algorithms
- but what about algorithms that require locking?

Deadlock Prevention through Partial Order

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

Definition (lock sets)

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

We require the transitive closure σ^+ of a relation σ:

Definition (transitive closure)

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

$$\begin{aligned} \sigma^0 &= \sigma \\ \sigma^{i+1} &= \{\langle x_1, x_3 \rangle \mid \exists x_2 \in X . \langle x_1, x_2 \rangle \in \sigma^i \wedge \langle x_2, x_3 \rangle \in \sigma^i\} \cup \sigma^i \end{aligned}$$

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

Definition (lock order)

Define $\lhd \subseteq L \times L$ such that $l \lhd l'$ iff $l \in \lambda(p)$ and the statement at p is of the form `wait(l')` or `monitor_enter(l')`. Define the lock order $\prec \; = \; \lhd^+$.

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.

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

Theorem (freedom of deadlock for monitors)

If $\forall a \in L_S . a \not\prec a$ and $\forall a \in L_M, b \in L . a \prec b \wedge b \prec a \Rightarrow a = b$ then the program is free of deadlocks.

Note: the set L contains *instances* of a lock.

- the set of lock instances can vary at runtime
- if we statically want to ensure that deadlocks cannot occur:
  - summarize every lock/monitor that may have several instances into one
  - a summary lock/monitor $\bar{a} \in L_M$ represents several concrete ones
  - thus, if $\bar{a} \prec \bar{a}$ then this might not be a self-cycle
  - ⇝ require that $\bar{a} \not\prec \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

Inferring locksets and lockset order in practice

⚠ fix a representation for locksets

⇝ in our case: L comprises all lines, where any object is created.

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

```
8:   void bar(this) {
       monitor_enter(this);
       if (*) {
         ...
12:      bar(&other);
13:      ...
14:    }
15:    mon
16:  }
```

$\mathtt{this} = \{\&a, \&b\}$

$\lambda(9) = \{\mathtt{l_0}, \mathtt{l_1}\}$

$\mathtt{other} = \{\&a, \&b\}$

Lockorder	$\lhd$	$\langle l_0, l_1 \rangle, \langle l_1, l_0 \rangle$

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_S: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists $l' \prec l$ is to be acquired
  ⇝ change program: release l, acquire l', then acquire l again
  ⚠ inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

Locks Roundup

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

stack: removal

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

- nested `atomic` blocks still describe one atomic execution

⇝ locks convey additional information over `atomic`

- locks cannot easily be recovered from `atomic` declarations

Outlook

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

Idea of mutexes: Implement `atomic` sections with locks:

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

⇝ creating locks automatically is non-trivial and, thus, not standard in programming languages

Concurrency across Languages

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

- the ability to use *atomic* operations

⇝ we can implement *wait-free* algorithms

In Java, C# and other higher-level languages

- provide monitors and possibly other concepts
- often simplify the programming but incur the same problems

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

(a) some pthread implementations allow a *reentrant* attribute

(b) newer API extensions (`java.util.concurrent.atomic.*` and `System.Threading.Interlocked` resp.)

(c) simulate semaphores using an object with two `synchronized` methods

Summary

Classification of concurrency algorithms:

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

Wait-free algorithms:

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

Lock-free algorithms:

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

Locking algorithms:

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

⇝ use algorithm that is best fit

References

[1] E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ACM Comput. Surv., 3(2):67–78, June 1971.

[2] T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Concurrency: Transactions

Dr. Michael Petter

Winter term 2019

Abstraction and Concurrency

Two fundamental concepts to build larger software are:

abstraction : an object storing certain data and providing certain functionality may be used without reference to its internals

composition : several objects can be combined to a new object without interference

Both, *abstraction* and *composition* are closely related, since the ability to compose depends on the ability to abstract from details.

Consider an example:

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

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

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

⇝ wrap the two calls in `insert()` in a mutex

- but other list operations can still be called ⇝ use the *same* mutex

⇝ unlike sequential algorithms, thread-safe algorithms cannot always be composed to give new thread-safe algorithms

Transactional Memory [2]

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

```
atomic {
  // code
  if (cond) retry;
  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

Semantics of Transactions

The goal is to use transactions to specify *atomic executions*.

Transactions are rooted in databases where they have the *ACID* properties:

- ***atomicity*** : a transaction completes or seems not to have run
  - ⇝ we call this *failure atomicity* to distinguish it from *atomic executions*
- ***consistency*** : each transaction transforms a consistent state to another consistent state
  - a consistent state is one in which certain *invariants* hold
  - invariants depend on the application
- ***isolation*** : among each other, transactions do not interfere
  - ⇝ coexisting with non-transactional memory, isolation is not so evident
- ***durability*** : the effects are permanent (w.r.t. main memory ✓)

Definition (Semantics of Transactions)

The result of running concurrent transactions must be identical to *one* execution of them in sequence. (⇝ Serialization)

Consistency During Transactions

Consistency during a transaction.

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
  ⇝ zombie transaction
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

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

⚠ critical for null pointer derefs or divisions by zero, e.g.

Definition (opacity)

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

⇝ failing transactions still see a consistent view of memory

Weak- and Strong Isolation

Can we mix transactions with code accessing memory non-transactionally?

- *strong isolation* retains order between accesses to TM and non-TM
- In *weak isolation*, guarantees are only given about memory accessed inside `atomic`
  - no conflict detection for non-transactional accesses
  - ⚠ standard *race problems*, e.g.

```
// Thread 1
atomic {                           // Thread 2
  x = 42;                          int tmp = x;
}
```

⇝ give programs with races the same semantics as if using a single global lock for all `atomic` blocks

Definition (SLA)

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

⇝ like *sequential consistency*, SLA is a statement about program equivalence

Disadvantages of the SLA model

The SLA model is *simple* but often too strong:

1. SLA has a weaker *progress* guarantee than a transaction should have

```
// Thread 1                        // Thread 2
atomic {                           atomic {
  while (true) {};                   int tmp = x; // x in TM
}                                  }
```

2. SLA correctness is too strong in practice

```
                                   // Thread 2
// Thread 1                        atomic {
data = 1;                            int tmp = data;
atomic {                             // Thread 1 not in atomic
}                                    if (ready) {
ready = 1;                             // use tmp
                                     }
                                   }
```

- under the SLA model, `atomic {}` acts as barrier
- intuitively, the two transactions should be independent rather than synchronize

⇝ need a weaker model for more flexible implementation of *strong isolation*

Transactional Sequential Consistency

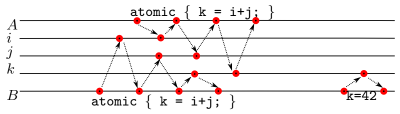
How about a more permissive view of transaction semantics?

- TM should not have the blocking behaviour of locks

⇝ the programmer cannot rely on synchronization

Definition (TSC)

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



- TSC is weaker: gives *strong isolation*, but allows parallel execution ✓
- TSC is stronger: accesses within a transaction may *not* be re-ordered ⚠

⇝ actual implementations use TSC with some *race free* re-orderings

Software Transactional Memory

Translation of `atomic`-Blocks

A TM system must track which shared memory locations are accessed:

- convert every read access `x` from a shared variable to `ReadTx(&x)`
- convert every write access `x=e` to a shared variable to `WriteTx(&x,e)`

Convert `atomic` blocks as follows:

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

- translation can be done using a pre-processor
  - determining a minimal set of memory accesses that need to be transactional requires a good static analysis
  - *idea*: translate all accesses to global variables and the heap as TM
  - more fine-grained control using manual translation
- an actual implementation might provide a `retry` keyword
  - when executing `retry`, the transaction aborts and re-starts
  - the transaction will again wind up at `retry` unless its *read set* changes
  - ⇝ block until a variable in the read-set has changed
  - similar to condition variables in monitors ✓

A Software TM Implementation

A software TM implementation allocates a *transaction descriptor* to store data specific to each `atomic` block, 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

Principles of TL2

The idea: obtain a version from the global counter on starting the transaction, the *read-version*, and watch out for accesses to newer versions throughout the transaction.

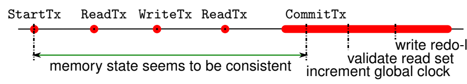
- A read `ReadTx` from a field at `offset` of object `obj` aborts,
  - when the objects version is younger than the transaction
  - when the object is locked at the moment of access

  or returns the read value and adds the accessed memory address to the *read-set*.
- `WriteTx` is simpler: add or update the location in the *redo-log*.
- `CommitTx` successively
  1. picks up locks for each written object
  2. increments the global version
  3. checks the read objects for being up to date

  before writing redo-log entries to memory while updating their version and realasing their locks

Properties of TL2

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



Other observations:

- read-only transactions just need to check that read versions are consistent (no need to increment the global clock)
- writing values still requires locks
  - deadlocks are still possible
  - since other transactions can be aborted, one can *preempt* transactions that are deadlocked
  - since lock accesses are generated, computing a lock order up-front might be possible
- there might be contention on the global clock

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:

```
// Thread 1                      // Thread 2
atomic { // clock=12
  ...                            atomic {
                                   WriteTx(&x,0) = 42; // clock=13
                                 }
  int r = ReadTx(&x,0);
} // tx.RV==12 != clock
```

- lock-based commits can cause contention
  - organize cells that participate in a transaction in one object
  - compute a new object as result of a transaction
  - atomically replace a pointer to the old object with a pointer to the new object if the old object has not changed
  - ⇝ idea of the original STM proposal
- TM system should figure out which memory locations must be logged
- danger of live-locks: transaction B might abort A which might abort B . . .

Integrating Non-TM Resources

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

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

Usual choice is one of the following:

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

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

Hardware Transactional Memory

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
  - if a cache-line in the read set is invalidated, the transaction aborts
  - if a cache-line in the write set must be written-back, the transaction aborts

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

Two principal implementation of HTM:

1. Explicit Transactional Memory: each access is marked as transactional
   - similar to `StartTx`, `ReadTx`, `WriteTx`, and `CommitTx`
   - requires separate transaction instructions
   - ⇝ a transaction has to be translated differently
   - ⚠ mixing transactional and non-transactional accesses is problematic
2. 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*

Example for HTM

AMD Advanced Synchronization Facilities (ASF):

- defines a logical *speculative region*
- `LOCK MOV` instructions provide *explicit* data transfer between normal memory and speculative region
- aimed to implement larger atomic operations

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

- *implicitely 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, ...)
- use a simple counter to support nested transactions
- may abort at any time due to lack of resources
- aborting in an inner transaction means aborting all of them

Intel provides two software interfaces to TM:

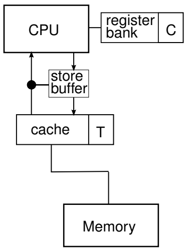
1. Restricted Transactional Memory (RTM)
2. Hardware Lock Elision (HLE)

Restricted Transactional Memory

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



⇝ additional transaction logic:

- `xbegin` increments C and, if $C = 0$, backs up registers and flushes buffer
  - subsequent read or write access to a cache line sets *T* if $C > 0$
  - applying an *invalidate* message to a cache line with *T* flag issues `xabort`
  - observing a *read* for a *modified* cache line with *T* flag issues `xabort`
- `xabort` clears all *T* flags and the store buffer, invalidates the former *TM* lines, sets $C = 0$ and restores CPU registers
- `xend` decrements C and, if $C = 0$, clears all *T* flags, flushes store buffer

Restricted Transactional Memory

Provides new instructions `xbegin`, `xend`, `xabort`, and `xtest`:

- `xbegin` *on transaction start* skips to the next instruction or *on abort*
  - continues at the given address
  - implicitely stores an error code in `eax`
- `xend` commits the transaction started by the most recent `xbegin`
- `xabort` aborts the whole transaction with an error code
- `xtest` checks if the processor is executing transactionally

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

```
_xbegin()

move   eax, 0xFFFFFFFF
xbegin _txnL1
_txnL1:
move   retval, eax
```

```
if(_xbegin()==_XBEGIN_STARTED) {
  // transaction code
  _xend();
} else {
  // non-transactional fall-back
}
```

⇝ user must provide *fall-back code*

Considerations for the Fall-Back Path

Consider executing the following code concurrently with itself:

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

Protecting the Fall-Back Path

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

```
int data[100]; // shared
int mutex;
void update(int idx, int value) {
  if(_xbegin()==_XBEGIN_STARTED) {
      if (!mutex>0) _xabort();
      data[idx] += value;
      _xend();
    } else {
      wait(mutex);
      data[idx] += value;
      signal(mutex);
    }
}
```

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

Happened Before Diagram for Transactions

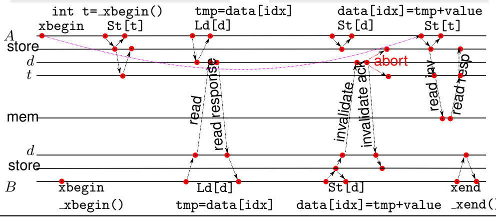
Augment MESI states with extra bit T. CPU A: d:E5 t:E0, CPU B: d:I, tmp/value registers

Thread A

```
int t = _xbegin();
int tmp = data[idx];
data[idx] = tmp + value;
_xend();
```

Thread B

```
_xbegin();
int tmp = data[idx];
data[idx] = tmp + value;
_xend();
```



Common Code Pattern for Mutexes

Using HTM in order to implement mutex:

```
int data[100]; // shared
int mutex;
void update(int idx, int val) {
  if(_xbegin()==_XBEGIN_STARTED) {
    if (!mutex>0) _xabort();
    data[idx] += val;
    _xend();
  } else {
    wait(mutex);
    data[idx] += val;
    signal(mutex);
  }
}
```

```
void update(int idx, int val) {
  lock(&mutex);
  data[idx] += val;
  unlock(&mutex);
}
void lock(int* mutex) {
  if(_xbegin()==_XBEGIN_STARTED)
  { if (!*mutex>0) _xabort();
    else return;
  } wait(mutex);
}
void unlock(int* mutex) {
  if (!*mutex>0) signal(mutex);
  else _xend();
}
```

- critical section may be executed without taking the lock (the lock is *elided*)
- as soon as one thread conflicts, it aborts, takes the lock in the fallback path and thereby aborts all other transactions that have read `mutex`

Hardware Lock Elision

Hardware Lock Elision

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

Idea: Hardware Lock Elision

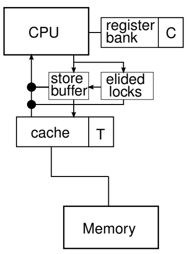
1. By default defer actual acquisition of the lock
2. Instead rely on HTM to sort out conflicting concurrent accesses
3. Fall back to actual locking only in case of conflicts
4. Support legacy lock code by locally acting as if semaphore value is actually modified

- requires annotations for lock instructions:
  - instruction that increments the semaphore must be prefixed with `xacquire`
  - instruction setting the semaphore to 0 must be prefixed with `xrelease`
  - these prefixes are ignored on older platforms
- for a successful elision, all signal/wait operations of a lock must be annotated

Implementing Lock Elision

Transactional operation:

- re-uses infrastructure for Restricted Transactional Memory
- add a buffer for elided locks, similar to store buffer



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

Transactional Memory: Summary

Transactional memory aims to provide `atomic` blocks for general code:

- frees the user from deciding how to lock data structures
- compositional way of communicating concurrently
- can be implemented using software (locks, atomic updates) or hardware

It is hard to get the details right:

- semantics of *explicit HTM* and *STM* transactions quite subtle when mixing with non-TM (*weak* vs. *strong isolation*)
- *single-lock atomicity* vs. *transactional sequential consistency* semantics
- STM not the right tool to synchronize threads without shared variables
- TM providing *opacity* (serializability) requires *eager conflict detection* or *lazy version management*

Pitfalls in *implicit* HTM:

- RTM requires a fall-back path
- no progress guarantee
- HLE can be implemented in software using RTM

TM in Practice

Availability of TM Implementations:

- GCC can translate accesses in `__transaction_atomic` regions into `libitm` library calls
- the library `libitm` provides different TM implementations:
  1. On systems with TSX, it maps atomic blocks to HTM instructions
  2. On systems without TSX and for the fallback path, it resorts to STM
- C++20 standardizes `synchronized`/`atomic_XXX` blocks
- RTM support slowly introduced to OpenJDK Hotspot monitors

Use of hardware lock elision is limited:

- allows to easily convert existing locks
- `pthread` locks in `glibc` use RTM `https://lwn.net/Articles/534758/`:
  - allows implementation of fallback mechanisms
  - HLE only special case of general lock
- implementing monitors is challenging
  - lock count and thread id may lead to conflicting accesses
  - in `pthreads`: error conditions often not checked anymore

Outlook

Several other principles exist for concurrent programming:

1. non-blocking message passing (the actor model)
   - a program consists of actors that send messages
   - each actor has a queue of incoming messages
   - messages can be processed and new messages can be sent
   - special filtering of incoming messages
   - *example:* Erlang, many add-ons to existing languages
2. blocking message passing (CSP, π-calculus, join-calculus)
   - a process sends a message over a channel and blocks until the recipient accepts it
   - channels can be send over channels (π-calculus)
   - *examples:* Occam, Occam-π, Go
3. (immediate) priority ceiling
   - declare *processes* with priority and *resources* that each process may acquire
   - each resource has the maximum (ceiling) priority of all processes that may acquire it
   - a process' priority at run-time increases to the maximum of the priorities of held resources
   - the process with the maximum (run-time) priority executes

References

[1] D. Dice, O. Shalev, and N. Shavit.
Transactional Locking II.
In *Distributed Computing*, LNCS, pages 194–208. Springer, Sept. 2006.

[2] T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.

Online resources on Intel HTM and GCC's STM:

1. `http://software.intel.com/en-us/blogs/2013/07/25/fun-with-intel-transactional-synchronization-extensions`
2. `http://www.realworldtech.com/haswell-tm/4/`
3. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf`

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Dispatching Method Calls

Dr. Michael Petter

Winter Term 2019

Dispatching - Outline

Dispatching

1. Motivation
2. Formal Model
3. Quiz
4. Dispatching from the Inside

Solutions in Single-Dispatching

1. Type introspection
2. Generic interface

Multi-Dispatching

1. Formal Model
2. Multi-Java
3. Multi-dispatching in Perl6
4. Multi-dispatching in Clojure

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;
}
```

Section 2

Overloading Function Names

Function Dispatching (ANSI C89)

```
#include <stdio.h>

void println(int i)   { print("%d\n",i); };
void println(float f) { print("%f\n",f); };

int main(){
  println(1.2);
  println(1);
  return 0;
}
```

⚠ Functions with same names but different parameters not legal

Generic Selection (C11)

generic-selection ↦ `_Generic`(*exp*, *generic-assoclist*)
generic-assoclist ↦ (*generic-assoc*,)* *generic-assoc*
generic-assoc ↦ `typename` : *exp* | `default` : *exp*

Example:

```
#include <stdio.h>

int main(){
  printf(_Generic((1.2),signed int: "%d",float: "%f"), 1.2), printf("\n");
  printf(_Generic(( 1),signed int: "%d",float: "%f"),   1), printf("\n");
  return 0;
}
```

Overloading (Java/C++)

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

public static void main() {
  D d = new D();
  D.p(d.f(2)+"\n");
  D.p(d.f(2.3)+"\n");
}
```

```
>$ javac Overloading.java; java Overloading
f(int): 3
f(double): 3.6
```

Overloading with Inheritance (Java)

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

public static void main() {
  D d = new D();
  B.p(d.f(2)+"\n");
  B.p(d.f(2.3)+"\n");
}
```

```
>$ javac Overloading.java; java Overloading
f(int): 3
f(double): 3.6
```

Overloading with Scopes (C++)

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

int main() {
  D* pd = new D;
  cout << pd->f(2) << '\n';
  cout << pd->f(2.3) << '\n';
}
```

```
>$ ./overloading
f(int): 3
f(double): 3.6
```

Overloading Hassles

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

public static void main() {
  D d = new D();
  D.p(d.f(2,2)+"\n");
}
```

```
>$ javac Overloading.java
Overloading.java:(?): error: reference to f is ambiguous
```

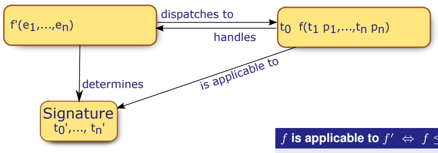
Static Methods are *Statically Dispatched*

Function Call Expression

Function to be dispatched

Concrete Method

Provides calling target for a call signature



Signature

- Function Name
- *Static* Types of Parameters
- Return Type

f is applicable to $f' \Leftrightarrow f \leq f'$:

$\leq$ is the *subtype relation*:

$$R\ f(T_1,\ldots,T_n) \leq R'\ f'(T'_1,\ldots,T'_n)$$

$$\implies R \leq R' \wedge T'_i \leq T_i$$

Inside the Javac – Predicates

Concept of methods being *applicable* for arguments:

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

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

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

Concept of method signatures being *more specific* then others:

```
// true if "more" is in every argument at least as specific as "less"
boolean isMoreSpecific(MemberDefinition more, MemberDefinition less) {
    Type moreType = more.getClassDeclaration().getType();
    Type lessType = less.getClassDeclaration().getType();
    return isMoreSpecific(moreType, lessType) // return type based comparison
        && isApplicable(less, more.getType().getArgumentTypes()); // parameter type based
}
```

Finding the Most Specific Concrete Method

```
MemberDefinition matchMethod(Environment env, ClassDefinition accessor,
                             Identifier methodName, Type[] argumentTypes) throws ... {
    // A tentative maximally specific method.
    MemberDefinition tentative = null;
    // A list of other methods which may be maximally specific too.
    List candidateList = null;
    // Get all the methods inherited by this class which have the name `methodName'
    for (MemberDefinition method : allMethods.lookupName(methodName)) {
        // See if this method is applicable.
        if (!env.isApplicable(method, argumentTypes)) continue;
        // See if this method is accessible.
        if ((accessor != null) && (!accessor.canAccess(env, method))) continue;
        if ((tentative == null) || (env.isMoreSpecific(method, tentative)))
            // `method' becomes our tentative maximally specific match.
            tentative = method;
        else {  // If this method could possibly be another maximally specific
                // method, add it to our list of other candidates.
                if (!env.isMoreSpecific(tentative,method)) {
                    if (candidateList == null) candidateList = new ArrayList();
                    candidateList.add(method);
    }   }     }
    if (tentative != null && candidateList != null)
        // Find out if our `tentative' match is a uniquely maximally specific.
        for (MemberDefinition method : candidateList )
            if (!env.isMoreSpecific(tentative, method))
                throw new AmbiguousMember(tentative, method);
    return tentative;
}
```

Section 3

Overriding Methods

Object Orientation

Emphasizing the *Receiver* of a Call

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

```
class Natural {
  int value;
}
void incBy(Natural n,int i) {
  n.value += Math.abs(i);
}

...
incBy(nat,42);
```

$\Longrightarrow$

```
class Natural {
  int value;

  void incBy(int i){
    this.value += Math.abs(i);
  }
}
...
nat.incBy(42);
```

- Associating the first parameter as *Receiver* of the method, and pulling it out of the parameters list
- Implicitely binding the first parameter to the fixed name *this*

Subtyping in 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.

```
class Integral {
  int i;
  void incBy(int delta){
      i += delta;
  }
}
class Natural extends Integral {
  int value;
  void incBy(int i){
    this.value += Math.abs(i);
  }
}
```

```
Integral i = new Integral(-5);
i.incBy(42);
Natural n = new Natural(42);
n.incBy(42);
i = n;
i.incBy(42);
```

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

⇝ Implicitely call the specialized method!

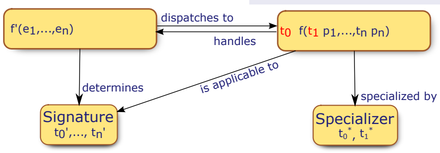
Methods are *dynamically dispatched*

Function Call Expression

Call expression to be dispatched.

Concrete Method

Provides calling target for a call signature



Signature

Static types of actual parameters.

Specializer

Specialized types to be matched at the call

How can we implement that?

Let's look at what Java does!

The Java platform as example for state of the art OO systems:

- Static Javac-based compiler
- Dynamic Hotspot JIT-Compiler/Interpreter

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

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

Bytecode

⇝ `matchMethod` returns the statically most specific signature

⇝ Codegeneration hardcodes `invokevirtual` with this signature

```
Code:
   0: new           #4                  // class Natural
   3: dup
   4: iconst_1
   5: invokespecial #5                  // Method "<init>":(I)V
   8: astore_1
   9: aload_1
  10: bipush        42
  12: invokevirtual #6                  // Method Integral.incBy:(I)V
  15: return
```

? What is the semantics of `invokevirtual`?

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

Inside the Hotspot VM

```
void LinkResolver::resolve_method(methodHandle& resolved_method, KlassHandle resolved_klass,
                                  Symbol* method_name, Symbol* method_signature,
                                  KlassHandle current_klass) {

  // 1. check if klass is not interface
  if (resolved_klass->is_interface()) ;//... throw "Found interface, but class was expected"

  // 2. lookup method in resolved klass and its super klasses
  lookup_method_in_klasses(resolved_method, resolved_klass, method_name, method_signature);
      // calls klass::lookup_method() -> next slide

  if (resolved_method.is_null()) { // not found in the class hierarchy
    // 3. lookup method in all the interfaces implemented by the resolved klass
    lookup_method_in_interfaces(resolved_method, resolved_klass, method_name, method_signature);

    if (resolved_method.is_null()) {
      // JSR 292:  see if this is an implicitly generated method MethodHandle.invoke(*...)
      lookup_implicit_method(resolved_method, resolved_klass, method_name, method_signature, current_klass);
    }

    if (resolved_method.is_null()) { // 4. method lookup failed
      // ... throw  java_lang_NoSuchMethodError()
  }  }

  // 5. check if method is concrete
  if (resolved_method->is_abstract() && !resolved_klass->is_abstract()) {
     // ... throw java_lang_AbstractMethodError()
  }

  // 6. access checks, etc.
}
```

Inside the Hotspot VM

The method lookup recursively traverses the super class chain:

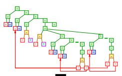
```
MethodDesc* klass::lookup_method(Symbol* name, Symbol* signature) {
  for (KlassDesc* klas = as_klassOop(); klas != NULL; klas = klass::cast(klas)->super()) {
    MethodDesc* method = klass::cast(klass)->find_method(name, signature);
    if (method != NULL) return method;
  }
  return NULL;
}
```

Inside the Hotspot VM

```
MethodDesc* klass::find_method(ObjArrayDesc* methods, Symbol* name, Symbol* signature) {
  int len = methods->length();
  // methods are sorted, so do binary search
  int i, l = 0 , h = len - 1;
  while (l <= h) {
    int mid = (l + h) >> 1;
    MethodDesc* m = (MethodDesc*)methods->obj_at(mid);
    int res = m->name()->fast_compare(name);
    if (res == 0) {
      // found matching name; do linear search to find matching signature
      // first, quick check for common case
      if (m->signature() == signature) return m;
      // search downwards through overloaded methods
      for (i = mid - 1; i >= l; i--) {
        MethodDesc* m = (MethodDesc*)methods->obj_at(i);
        if (m->name() != name) break;
        if (m->signature() == signature) return m;
      }
      // search upwards
      for (i = mid + 1; i <= h; i++) {
        MethodDesc* m = (MethodDesc*)methods->obj_at(i);
        if (m->name() != name) break;
        if (m->signature() == signature) return m;
      }
      return NULL; // not found
    } else if (res < 0) l = mid + 1;
           else         h = mid - 1;
  }
  return NULL;
}
```

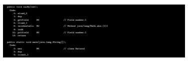
Single-Dispatching: Summary

Compile Time

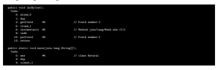


Javac

Matches a method call expression *statically* to the *most specific* method signature via `matchMethod( ... )`

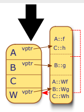


Runtime



Hotspot VM

Interprets `invokevirtual` via `resolve_method(...)`, scanning the superclass chain with `find_method(...)` for the statically fixed `signature`



Example: Sets of Natural Numbers

```
class Natural {
  Natural(int n){ number=Math.abs(n); }
  int number;
  public boolean equals(Natural n){
    return n.number == number;
  }
}
...
  Set<Natural> set = new HashSet<>();
  set.add(new Natural(0));
  set.add(new Natural(0));
  System.out.println(set);
```

```
>$ java Natural
[0,0]
```

⚠ Why? Is HashSet buggy?

⇝ Keep attention to exact signature!

Mini-Quiz: Java Method Dispatching

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

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

Section 4

Multi-Dispatching

Can we expect more than Single-Dispatching?

Mainstream languages support specialization of first parameter:

C++, Java, C#, Smalltalk, Lisp

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

1. introspection?
2. generic programming?
3. double dispatching?

Introspection

```
class Natural {
  Natural(int n) { number=Math.abs(n); }
  int number;
  public boolean equals(Object n){
    if (!(n instanceof Natural)) return false;
    return ((Natural)n).number == number;
  }
}
...
  Set<Natural> set = new HashSet<>();
  set.add(new Natural(0));
  set.add(new Natural(0));
  System.out.println(set);
```

```
>$ java Natural
[0]
```

✓ Works ⚠ but burdens programmer with type safety

⚠ and is only available for languages with type introspection

Generic Programming

```
interface Equalizable<T>{
  boolean equals(T other);
}
class Natural implements Equalizable<Natural> {
  Natural(int n){ number=Math.abs(n); }
  int number;
  public boolean equals(Natural n){
    return n.number == number;
  }
}
...
  EqualizableAwareSet<Natural> set = new MyHashSet<>();
  set.add(new Natural(0));
  set.add(new Natural(0));
  System.out.println(set);
```

⚠ needs another Set implementation and...

⚠ only works for one overloaded version in super hierarchy

```
>$ javac Natural.java
Natural.java:2: error: name clash: equals(T) in Equalizable and equals(Object)
in Object have the same erasure, yet neither overrides the other
```

Double Dispatching

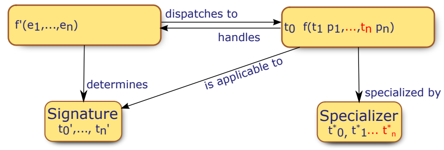
```
abstract class EqualsDispatcher{
  boolean dispatch(Natural) { return false };
  boolean dispatch(Object)  { return false };
}
class Natural {
  Natural(int n){ number=Math.abs(n); }
  int number;
  public boolean doubleDispatch(EqualsDispatcher ed) {
    return ed.dispatch(this);
  }
  public boolean equals(Object n){
    return n.doubleDispatch(
      new EqualsDispatcher(){
        boolean dispatch(Natural nat) {
          return nat.number==number;
      }; }
); }  }
```

✓ Works ⚠ but needs Dispatcher to know complete class hierarchies

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

- Typechecking families of concrete methods introduces checking the existance 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]

```
class Natural {
  public Natural(int n){ number=Math.abs(n); }
  private int number;
  public boolean equals(Object@Natural n){
    return n.number == number;
  }
}
...
  Set<Natural> set = new HashSet<>();
  set.add(new Natural(0));
  set.add(new Natural(0));
  System.out.println(set);
```

```
>$ java Natural
[0]
```

✓ Clean Code!

Natural Numbers Behind the Scenes

```
>$ javap -c Natural
```

```
public boolean equals(java.lang.Object);
 Code:
  0:   aload_1
  1:   instanceof      #2; //class Natural
  4:   ifeq    16
  7:   aload_0
  8:   aload_1
  9:   checkcast       #2; //class Natural
  12:  invokespecial   #28; //Method equals$body3$0:(LNatural;)Z
  15:  ireturn
  16:  aload_0
  17:  aload_1
  18:  invokespecial   #31; //Method equals$body3$1:(LObject;)Z
  21:  ireturn
```

⇝ Redirection to methods `equals$body3$1` and `equals$body3$0`

Section 5

Natively multidispatching Languages

Perl6

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

```
  Dispatch 1
  Dispatch base
```

Clojure

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

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

Principle of Multidispatching in Clojure

```
(derive ::child ::parent)

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

(pr (fun ::child ::child))
```

```
child equals
```

More Creative dispatching in Clojure

```
(defn salary [amount]
         (cond (< amount 600)    ::poor
               (>= amount 5000)  ::rich
               :else             ::average))

(defrecord UniPerson [name wage])

(defmulti  print (fn [person] (salary (:wage person)) ))
(defmethod print ::poor    [person](str "HiWi  " (:name person)))
(defmethod print ::average [person](str "Dr.   " (:name person)))
(defmethod print ::rich    [person](str "Prof. " (:name person)))

(pr (print (UniPerson. "Petter" 2000)))
(pr (print (UniPerson. "Stefan"  200)))
(pr (print (UniPerson. "Seidl" 16000)))
```

```
Dr. Petter
HiWi Stefan
Prof. Seidl
```

Multidispatching

Pro

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

Con

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

Other Solutions (extract)

- Dylan
- Scala

Lessons Learned

Lessons Learned

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

Section 6

Further materials

Further reading...

[1] hotspot/src/share/vm/interpreter/linkResolver.cpp.
OpenJDK 7 Hotspot JIT VM.
http://hg.openjdk.java.net/jdk7/jdk7.

[2] jdk/src/share/classes/sun/tools/java/ClassDefinition.java.
OpenJDK 7 Javac.
http://hg.openjdk.java.net/jdk7/jdk7.

[3] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers.
Multijava: Design rationale, compiler implementation, and applications.
ACM Transactions on Programming Languages and Systems (TOPLAS), May 2006.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha.
The Java Language Specification, Third Edition.
Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[5] S. Halloway.
Programming Clojure.
Pragmatic Bookshelf, 1st edition, 2009.

[6] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley.
The Java Virtual Machine Specification.
Addison-Wesley Professional, Java SE7 edition, 2013.

[7] R. Muschevici, A. Potanin, E. Tempero, and J. Noble.
Multiple dispatch in practice.
23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA), September 2008.

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Multiple Inheritance

Dr. Michael Petter

Winter term 2019

Outline

Inheritance Principles

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

C++ Object Heap Layout

1. Basics
2. Single-Inheritance
3. Virtual Methods

C++ Multiple Parents Heap Layout

1. Multiple-Inheritance
2. Virtual Methods
3. Common Parents

Excursion: Linearization

1. Ambiguous common parents
2. Principles of Linearization
3. Linearization algorithms

"Wouldn't it be nice to inherit from several parents?"

Interface vs. Implementation inheritance

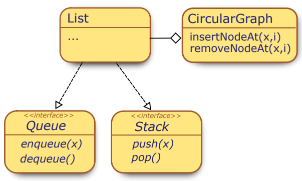
The classic motivation for inheritance is implementation inheritance

- *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

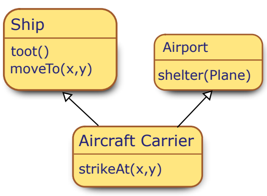
Code sharing in interface inheritance inverts this relation

- *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#

Interface Inheritance



Implementation inheritance



"So how do we lay out objects in memory anyway?"

Excursion: Brief introduction to LLVM IR

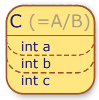
LLVM intermediate representation as reference semantics:

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

;(stack-) allocation of objects
%a = alloca %struct.A
;adress computation for selection in structure (pointers):
%1 = getelementptr %struct.A* %a, i64 0, i64 2
;load from memory
%2 = load i32(i32)* %1
;indirect call
%retval = call i32 (i32)* %2(i32 42)
```

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

Object layout

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

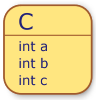



```
%class.C = type { %class.B, i32 }
%class.B = type { %class.A, i32 }
%class.A = type { i32 }
```

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

Translation of a method body

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




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

```
define i32 @_g(%class.B* %this, i32 %p) {
  %1 = getelementptr %class.B* %this, i64 0, i32 1
  %2 = load i32* %1
  %3 = add i32 %2, %p
  ret i32 %3
}
```

"Now what about polymorphic calls?"

Single-Dispatching implementation choices

Single-Dispatching needs runtime action:

1. Manual search run through the super-chain (Java Interpreter ⇝ last talk)

```
call i32 @__dispatch(%class.C* %c,i32 42,i32* "f(int,void)")
```

2. Caching the dispatch result (⇝ Hotspot/JIT)

```
; caching the recent result value of the __dispatch function
; call i32 @__dispatch(%class.C* %c,i32 42)
assert (%c type %class.D) ; verify objects class presumption
call i32 @_f_from_D(%class.C* %c, i32 42) ; directly call f
```

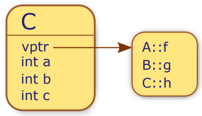
3. Precomputing the dispatching result in tables
   1. Full 2-dim matrix
   2. 1-dim Row Displacement Dispatch Tables
   3. Virtual Tables (⇝ LLVM/GNU C++,this talk)

A	B			F			
1	1	2	...	8	9		7

	f()	g()	h()	i()	j()	k()	l()	m()	n()
A	1								
B	1	2							
C	3		4						
D	3	2	4	5					
E						6		7	
F					8	9		7	

Object layout – virtual methods

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

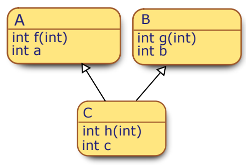



```
%class.C = type { %class.B, i32, [4 x i8] }
%class.B = type { [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
```

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

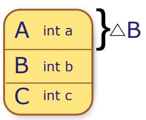
"So how do we include several parent objects?"

Multiple inheritance class diagram



Static Type Casts

```
class A {
  int a; int f(int);
};
class B {
  int b; int g(int);
};
class C : public A , public B {
  int c; int h(int);
};
...
B* b = new C();
```



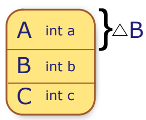

```
%class.C = type { %class.A, %class.B, i32 }
%class.A = type { i32 }
%class.B = type { i32 }
```

```
%1 = call i8* @_new(i64 12)
call void @_memset.p0i8.i64(i8* %1, i8 0, i64 12, i32 4, i1 false)
%2 = getelementptr i8* %1, i64 4          ; select B-offset in C
%b = bitcast i8* %2 to %class.B*
```

⚠ implicit casts potentially add a constant to the object pointer.
⚠ `getelementptr` implements ΔB as $4 \cdot i8$!

Keeping Calling Conventions

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




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

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

Ambiguities

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

C* pc;
pc->f(42);
```

⚠ Which method is called?

Solution I: Explicit qualification

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

Solution II: Automagical resolution

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

Linearization

Principle 1: Inheritance Relation

Defined by parent-child. Example: $C(A,B) \implies C \multimap A \wedge C \multimap B$

Principle 2: Multiplicity Relation

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

In General:

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

MRO via DFS

Leftmost Preorder Depth-First Search

$L[A] = ABWC$

⚠ Principle 1 *inheritance* is violated

Python: classical python objects (≤ 2.1) use LPDFS!

LPDFS with Duplicate Cancellation

$L[A] = ABCW$

✓ Principle 1 *inheritance* is fixed

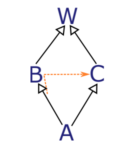
Python: new python objects (2.2) use LPDFS(DC)!

LPDFS with Duplicate Cancellation

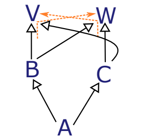
$L[A] = ABCWV$

⚠ Principle 2 *multiplicity* not fulfillable

⚠ However $B \rightarrow C \implies W \rightarrow V$??



$A(B,C)\ B(W)\ C(W)$



$A(B,C)B(V,W)C(W,V)$

MRO via Refined Postorder DFS

Reverse Postorder Rightmost DFS

$L[A] = ABFDCEGHW$

✓ Linear extension of inheritance relation

RPRDFS

$L[A] = ABCDGEF$

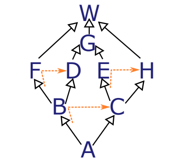
⚠ But principle 2 *multiplicity* is violated!

CLOS: uses Refined RPDFS [3]

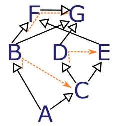
Refined RPRDFS

$L[A] = ABCDEFG$

✓ Refine graph with conflict edge & rerun RPRDFS!



$A(B,C)\ B(F,D)\ C(E,H)$
$D(G)\ E(G)\ F(W)\ G(W)\ H(W)$



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via Refined Postorder DFS

Refined RPRDFS

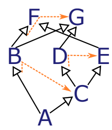
⚠ *Monotonicity* is not guaranteed!

Extension Principle: Monotonicity

If $C_1 \rightarrow C_2$ in C's linearization, then $C_1 \rightarrow C_2$ for every linearization of C's children.

$L[A] = A\ B\ C\ D\ E\ F\ G \implies F \rightarrow G$

$L[C] = C\ D\ G\ E\ F \implies G \rightarrow F$



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

MRO via C3 Linearization

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

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

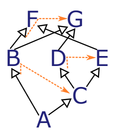
$$L[Object] = Object$$

with

$$\bigsqcup_i(L_i) = \begin{cases} c \cdot (\bigsqcup_i (L_i \setminus c)) & \text{if } \exists_{\min k} \forall_j\ c = head(L_k) \notin tail(L_j) \\ ⚠ \text{ fail} & \text{else} \end{cases}$$

MRO via C3 Linearization

$L[G]$	G
$L[F]$	F
$L[E]$	$E \cdot F$
$L[D]$	$D \cdot G$
$L[B]$	$B \cdot F \cdot G$
$L[C]$	$C \cdot D \cdot G \cdot E \cdot F$
$L[A]$	⚠ fail



$A(B,C)\ B(F,G)\ C(D,E)$
$D(G)\ E(F)$

C3 detects and reports a violation of *monotonicity* with the addition of A(B,C) to the class set.
C3 linearization [1]: is used in *Python 3*, *Perl 6*, and *Solidity*

Linearization vs. explicit qualification

Linearization

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

Qualification

- More flexible, fine-grained
- Linearization choices may be awkward or unexpected

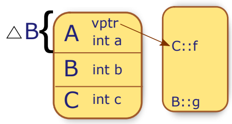
Languages with automatic linearization exist

- *CLOS* Common Lisp Object System
- *Solidity*, *Python 3* and *Perl 6* with C3
- Prerequisite for → Mixins

"And what about dynamic dispatching in Multiple Inheritance?"

Virtual Tables for Multiple Inheritance

```
class A {
  int a; virtual int f(int);
};
class B {
  int b; virtual int f(int);
         virtual int g(int);
};
class C : public A , public B {
  int c; int f(int);
};
...
C c;
B* pb = &c;
pb->f(42);
```

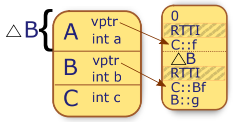



```
%class.C = type { %class.A, [12 x i8], i32 }
%class.A = type { i32 (...)**, i32 }
%class.B = type { i32 (...)**, i32 }
```

```
; B* pb = &c;
%0 = bitcast %class.C* %c to i8*       ; type fumbling
%1 = getelementptr i8* %0, i64 16      ; offset of B in C
%2 = bitcast i8* %1 to %class.B*       ; get typing right
store %class.B* %2, %class.B** %pb     ; store to pb
```

Virtual Tables for Multiple Inheritance

```
class A {
  int a; virtual int f(int);
};
class B {
  int b; virtual int f(int);
         virtual int g(int);
};
class C : public A , public B {
  int c; int f(int);
};
...
C c;
B* pb = &c;
pb->f(42);
```




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

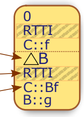
```
; pb->f(42);
%0 = load %class.B** %pb                                ;load the b-pointer
%1 = bitcast %class.B* %0 to i32 (%class.B*, i32)***   ;cast to vtable
%2 = load i32(%class.B*, i32)*** %1                     ;load vptr
%3 = getelementptr i32 (%class.B*, i32)** %2, i64 0     ;select f() entry
%4 = load i32(%class.B*, i32)** %3                      ;load function pointer
%5 = call i32 %4(%class.B* %0, i32 42)
```

Basic Virtual Tables (⇝ C++-ABI)

A Basic Virtual Table

consists of different parts:

1. *offset to top* of an enclosing objects memory representation
2. *typeinfo pointer* to an RTTI object (not relevant for us)
3. *virtual function pointers* for resolving virtual methods

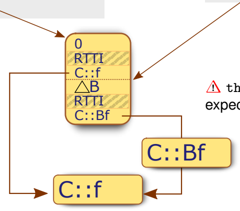


- Virtual tables are composed when multiple inheritance is used
- The `vptr` fields in objects are pointers to their corresponding virtual-subtables
- Casting preserves the link between an object and its corresponding virtual-subtable
- `clang -cc1 -fdump-vtable-layouts -emit-llvm code.cpp` yields the vtables of a compilation unit

Casting Issues

```
class A { int a; virtual int f(int); };
class B { virtual int f(int); };
class C : public A , public B {
  int c; int f(int);
};
C* c = new C();
c->f(42);
```

```
B* b = new C();
b->f(42);
```



⚠ `this`-Pointer for `C::f` is expected to point to `C`

Thunks

Solution: thunks

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

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

⇝ `B-in-C`-vtable entry for `f(int)` is the thunk `_f(int)`

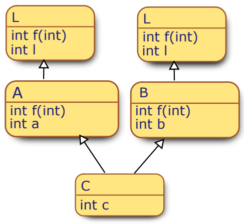
⇝ `_f(int)` adds a compiletime constant ΔB to `this` before calling `f(int)`

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

"But what if there are common ancestors?"

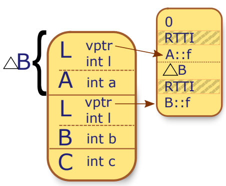
Common Bases – Duplicated Bases

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



Duplicated Base Classes

```
class L {
  int l; virtual void f(int);
};
class A : public L {
  int a; void f(int);
};
class B : public L {
  int b; void f(int);
};
class C : public A , public B {
  int c;
};
...
C c;
L* pl = (B*)&c;
pl->f(42); // where to dispatch?
C* pc = (C*)(B*)pl;
```

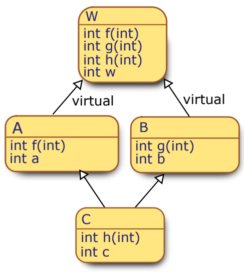



```
%class.C = type { %class.A, %class.B,
                  i32, [4 x i8] }
%class.A = type { [12 x i8], i32 }
%class.B = type { [12 x i8], i32 }
%class.L = type { i32 (...)**, i32 }
```

⚠ Ambiguity!

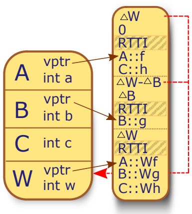
Common Bases – Shared Base Class

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



Shared Base Class

```
class W {
  int w; virtual void f(int);
  virtual void g(int);
  virtual void h(int);
};
class A : public virtual W {
  int a; void f(int);
};
class B : public virtual W {
  int b; void g(int);
};
class C : public A, public B {
  int c; void h(int);
};
...
C* pc;
pc->B::f(42);
((W*)pc)->h(42);
((B*)pc)->f(42);
```



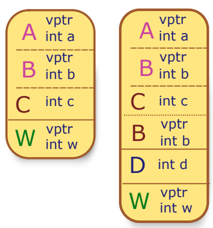
⚠ Ambiguities

⇝ e.g. overriding f in A *and* B

⚠ Offsets to virtual base

Dynamic Type Casts

```
class A : public virtual W {
...
};
class B : public virtual W {
...
};
class C : public A , public B {
...
};
class D : public C,
          public B {
...
};
...
C c;
W* pw = &c;
C* pc = dynamic_cast<C*>(pw);
```



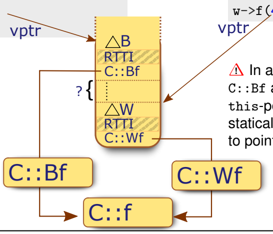
⚠ No guaranteed *constant* offsets between virtual bases and subclasses ⇝ No static casting!

⚠ *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 : public A , public B {
  int c; int f(int);
};
B* b = new C();
b->f(42);
```

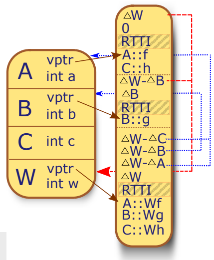
```
W* w = new C();
w->f(42);
```



⚠ In a conventional thunk `C::Bf` adjusts the `this`-pointer with a statically known constant to point to `C`

Virtual Thunks

```
class W { ...
virtual void g(int);
};
class A : public virtual W {...};
class B : public virtual W {
  int b; void g(int i){ };
};
class C : public A,public B{...};
C c;
W* pw = &c;
pw->g(42);
```




```
define void @__g(%class.B* %this, i32 %i) { ; virtual thunk to B::g
  %1 = bitcast %class.B* %this to i8*
  %2 = bitcast i8* %1 to i8**
  %3 = load i8** %2                        ; load W-vtable ptr
  %4 = getelementptr i8* %3, i64 -32 ; -32 bytes is g-entry in vcalls
  %5 = bitcast i8* %4 to i64*
  %6 = load i64* %5                        ; load g's vcall offset
  %7 = getelementptr i8* %1, i64 %6  ; navigate to vcalloffset+ Wtop
  %8 = bitcast i8* %7 to %class.B*
  call void @_g(%class.B* %8, i32 %i)
  ret void
}
```

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

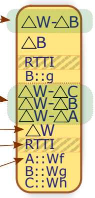
A Virtual Table for a Virtual Subclass

gets a *virtual base pointer*

A Virtual Table for a Virtual Base

consists of different parts:

1. *virtual call offsets* per virtual function for adjusting `this` dynamically
2. *offset to top* of an enclosing objects heap representation
3. *typeinfo pointer* to an RTTI object (not relevant for us)
4. *virtual function pointers* for resolving virtual methods



Virtual Base classes have *virtual thunks* which look up the offset to adjust the `this` pointer to the correct value in the virtual table!

Compiler and Runtime Collaboration

Compiler generates:

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

Runtime:

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

Polemics of Multiple Inheritance

Full Multiple Inheritance (FMI)

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

Multiple Interface Inheritance (MII)

- 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

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

Sidenote for MS VC++

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

Further reading...

[1] K. Barrett, B. Cassels, P. Haahr, D. Moon, K. Playford, and T. Withington.
A monotonic superclass linearization for dylan.
In *Object Oriented Programming Systems, Languages, and Applications*, 1996.

[2] CodeSourcery, Compaq, EDG, HP, IBM, Intel, R. Hat, and SGI.
Itanium C++ ABI.
URL: http://www.codesourcery.com/public/cxx-abi.

[3] R. Ducournau and M. Habib.
On some algorithms for multiple inheritance in object-oriented programming.
In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1987.

[4] R. Kleckner.
Bringing clang and llvm to visual c++ users.
URL: http://llvm.org/devmtg/2013-11/#talk11.

[5] B. Liskov.
Keynote address – data abstraction and hierarchy.
In *Addendum to the proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 17–34, 1987.

[6] L. L. R. Manual.
Llvm project.
URL: http://llvm.org/docs/LangRef.html.

[7] R. C. Martin.
The liskov substitution principle.
In *C++ Report*, 1996.

[8] P. Sabanal and M. Yason.
Reversing c++.
In *Black Hat DC*, 2007.
URL: https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf.

[9] B. Stroustrup.
Multiple inheritance for C++.
In *Computing Systems*, 1999.

Mini Seminars

1. SC=CC in Multicore Architectures with Cache (Meixner/Sorin 2006/2009)
2. Litmus Testing Memory Models: Herdtools 7
3. The Linux Kernel Memory Model
4. A Formal Analysis of the NVIDIA PTX Memory Consistency Model (2019)
5. GPU Concurrency: Weak Behaviours and Programming Assumptions (2015)
6. Transactional Memory Systems other than TSX: IBM Power 8 / BlueGene / zEnterprise
7. Lambda Calculus: Y Combinator and Recursion / SKI Combinator
8. Templates vs. Inheritance

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Mixins and Traits

Dr. Michael Petter

Winter 2019/20

What modularization techiques are there besides multiple implementation inheritance?

Outline

Design Problems

1. Inheritance vs Aggregation
2. (De-)Composition Problems

Inheritance in Detail

1. A Model for single inheritance
2. Inheritance Calculus with Inheritance Expressions
3. Modeling Mixins

Mixins in Languages

1. Simulating Mixins
2. Native Mixins

Cons of Implementation Inheritance

1. Lack of finegrained Control
2. Inappropriate Hierarchies

A Focus on Traits

1. Separation of Composition and Modeling
2. Trait Calculus

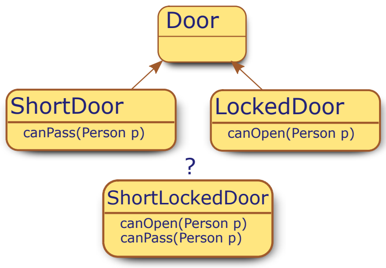
Traits in Languages

1. (Virtual) Extension Methods
2. Squeak

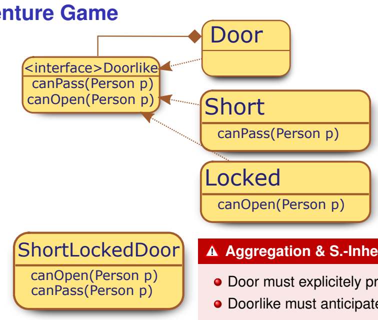
Reusability ≡ Inheritance?

- 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



The Adventure Game

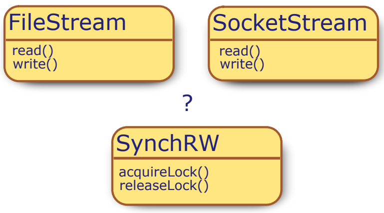


⚠ Aggregation & S.-Inheritance

- Door must explicitely provide chaining
- Doorlike must anticipate wrappers

⇒ **Multiple Inheritance** ✓

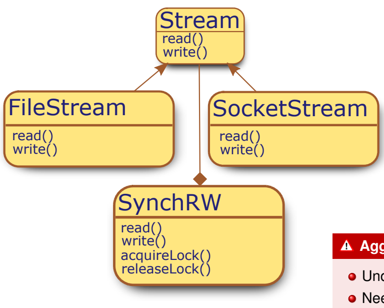
The Wrapper



⚠ Unclear relations

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

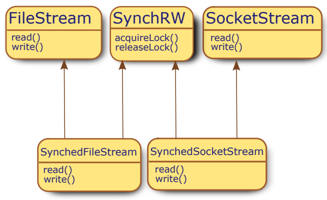
The Wrapper – Aggregation Solution



⚠ Aggregation

- Undoes specialization
- Needs common ancestor

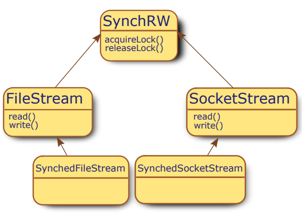
The Wrapper – Multiple Inheritance Solution



⚠ Duplication

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

Fragility



⚠ Inappropriate Hierarchies

Implemented methods (`acquireLock`/`releaseLock`) *to high*

(De-)Composition Problems

All the problems of

- Relation
- Duplication
- Hierarchy

are centered around the question

"How do I distribute functionality over a hierarchy"

⇝ *functional (de-)composition*

Classes and Methods

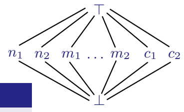
The building blocks for classes are

- a countable set of method *names* $\mathcal{N}$
- a countable set of method *bodies* $\mathbb{B}$

Classes map names to elements from the *flat lattice* $\mathcal{B}$ (called bindings), consisting of:

- method bodies $\in \mathbb{B}$ or classes $\in \mathcal{C}$
- $\perp$ *abstract*
- $\top$ *in conflict*

and the partial order $\perp \sqsubseteq b \sqsubseteq \top$ for each $b \in \mathcal{B}$



Definition (Abstract Class $\in \mathcal{C}$)

A general function $c : \mathcal{N} \mapsto \mathcal{B}$ is called a class.

Definition (Interface and Class)

A class c is called (with pre beeing the preimage)

interface iff $\forall_{n \in \text{pre}(c)} \,.\, c(n) = \perp$.
abstract class iff $\exists_{n \in \text{pre}(c)} \,.\, c(n) = \perp$.
concrete class iff $\forall_{n \in \text{pre}(c)} \,.\, \perp \sqsubset c(n) \sqsubset \top$.

Computing with Classes and Methods

Definition (Family of classes $\mathcal{C}$)

We call the set of all maps from names to bindings the family of classes $\mathcal{C} := \mathcal{N} \mapsto \mathcal{B}$.

Several possibilites for composing maps $\mathcal{C} \square \mathcal{C}$:

- the symmetric join $\sqcup$, defined componentwise:

$$(c_1 \sqcup c_2)(n) = b_1 \sqcup b_2 = \begin{cases} b_2 & \text{if } b_1 = \perp \text{ or } n \notin \text{pre}(c_1) \\ b_1 & \text{if } b_2 = \perp \text{ or } n \notin \text{pre}(c_2) \\ b_2 & \text{if } b_1 = b_2 \\ \top & \text{otherwise} \end{cases} \quad \text{where } b_i = c_i(n)$$

- in contrast, the asymmetric join $\sqcup\!\!\!\!\sqcup$, defined componentwise:

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

Example: Smalltalk-Inheritance

Smalltalk inheritance

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

Definition (Smalltalk inheritance ($\triangleright$))

Smalltalk inheritance is the binary operator $\triangleright : \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$, defined by
$c_1 \triangleright c_2 = \{\texttt{super} \mapsto c_2\} \sqcup\!\!\!\!\sqcup\, (c_1 \sqcup\!\!\!\!\sqcup\, c_2)$

Example: `Doors`

$$Door = \{canPass \mapsto \perp, canOpen \mapsto \perp\}$$
$$LockedDoor = \{canOpen \mapsto 0x4204711\} \triangleright Door$$
$$= \{\texttt{super} \mapsto Door\} \sqcup\!\!\!\!\sqcup\, (\{canOpen \mapsto 0x4204711\} \sqcup\!\!\!\!\sqcup\, Door)$$
$$= \{\texttt{super} \mapsto Door, canOpen \mapsto 0x4204711, canPass \mapsto \perp\}$$

Excursion: Beta-Inheritance

In *Beta*-style inheritance

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

Definition (Beta inheritance ($\triangleleft$))

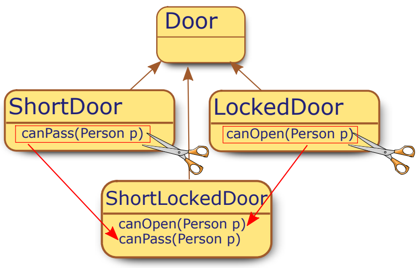
Beta inheritance is the binary operator $\triangleleft : \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$, defined by
$c_1 \triangleleft c_2 = \{\texttt{inner} \mapsto c_1\} \sqcup\!\!\!\!\sqcup\, (c_2 \sqcup\!\!\!\!\sqcup\, c_1)$

Example (equivalent syntax):

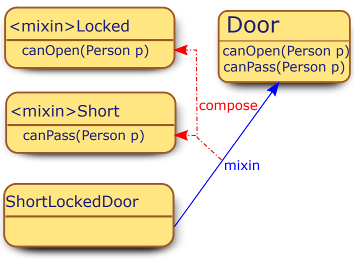
```
class Person {
  String name ="Axel Simon";
  public String toString(){ return name+inner.toString();};
};
class Graduate extends Person {
  public extension String toString(){ return ", Ph.D."; };
};
```

So what do we really want?

Adventure Game with Code Duplication



Adventure Game with Mixins



Adventure Game with Mixins

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

Back to the blackboard!

Abstract model for Mixins

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

Definition (Mixin)

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

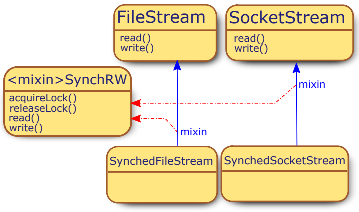
$$mixin(c) = \lambda x \,.\, c \triangleright x$$

⚠ Note: Mixins can also be composed $\circ$:

Example: `Doors`

$$Locked = \{canOpen \mapsto 0x1234\}$$
$$Short = \{canPass \mapsto 0x4711\}$$
$$Composed = mixin(Short) \circ (mixin(Locked)) = \lambda x \,.\, Short \triangleright (Locked \triangleright x)$$
$$= \lambda x \,.\, \{\texttt{super} \mapsto (Locked \triangleright x)\} \sqcup\!\!\!\!\sqcup\, (\{canOpen \mapsto 0x1234, canPass \mapsto 0x4711\} \triangleright x)$$

Wrapper with Mixins

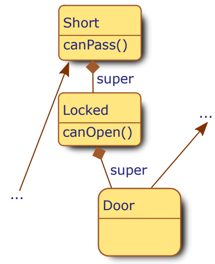


Mixins for wrappers

- 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 canPass(Person p)...
}
class ShortDoor
   = Short(Door);
class ShortLockedDoor
   = Short(Locked(Door));
...
ShortDoor d
   = new ShortLockedDoor();
```



⚠ *non-static* `super`-References

⇝ dynamic dispatching without precomputed virtual table

Surely multiple inheritance is powerful enough to simulate mixins?

Simulating Mixins in C++

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

Simulating Mixins in C++

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

True Mixins vs. C++ Mixins

True Mixins

- `super` natively supported
- 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

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

Ruby

```
class Person
  attr_accessor :size
  def initialize
    @size = 160
  end
  def hasKey
    true
  end
end
class Door
  def canOpen (p)
    true
  end
  def canPass(person)
    person.size < 210
  end
end
```

```
module Short
  def canPass(p)
    p.size < 160 and super(p)
  end
end
module Locked
  def canOpen(p)
    p.hasKey() and super(p)
  end
end

class ShortLockedDoor < Door
  include Short
  include Locked
end

p = Person.new
d = ShortLockedDoor.new
puts d.canPass(p)
```

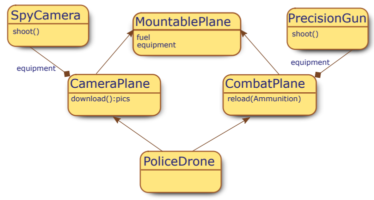
Ruby

```
class Door
  def canOpen (p)
    true
  end
  def canPass(person)
    person.size < 210
  end
end
module Short
  def canPass(p)
    p.size < 160 and super(p)
   end
end
module Locked
  def canOpen(p)
    p.hasKey() and super(p)
  end
end
```

```
module ShortLocked
  include Short
  include Locked
end
class Person
  attr_accessor :size
  def initialize
    @size = 160
  end
  def hasKey
    true
  end
end

p = Person.new
d = Door.new
d.extend ShortLocked

puts d.canPass(p)
```

Is Inheritance the Ultimate Principle in Reusability?

Lack of Control

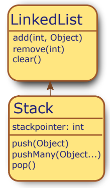


⚠ Control

- Common base classes are shared or duplicated at class level
- `super` as ancestor reference vs. qualified specification

⇝ No *fine-grained specification* of duplication or sharing

Inappropriate Hierachies



⚠ Inappropriate Hierarchies

- High up specified methods *turn obsolete*, but there is no statically safe way to remove them

⚠ Liskov Substitution Principle!

Is Implementation Inheritance even an *Anti-Pattern*?

Excerpt from the Java 8 API documentation for class `Properties`:

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

⚠ Misuse of Implementation Inheritance

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

The Idea Behind Traits

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

⚠ Central idea

Separate object *creation* from *modelling* hierarchies and *composing* functionality.

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

Traits – Composition

Definition (Trait $\in \mathcal{T}$)

A class t without attributes is called *trait*.

The *trait sum* $+ : \mathcal{T} \times \mathcal{T} \mapsto \mathcal{T}$ is the componentwise least upper bound:

$$(c_1 + c_2)(n) = b_1 \sqcup b_2 = \begin{cases} b_2 & \text{if } b_1 = \bot \lor n \notin \text{pre}(c_1) \\ b_1 & \text{if } b_2 = \bot \lor n \notin \text{pre}(c_2) \\ b_2 & \text{if } b_1 = b_2 \\ \top & \text{otherwise} \end{cases} \quad \text{with } b_i = c_i(n)$$

Trait-Expressions also comprise:

- *exclusion* $- : \mathcal{T} \times \mathcal{N} \mapsto \mathcal{T}$: $\quad (t - a)(n) = \begin{cases} \text{undef} & \text{if } a = n \\ t(n) & \text{otherwise} \end{cases}$
- *aliasing* $[\rightarrow] : \mathcal{T} \times \mathcal{N} \times \mathcal{N} \mapsto \mathcal{T}$: $\quad t[a \rightarrow b](n) = \begin{cases} t(n) & \text{if } n \neq a \\ t(b) & \text{if } n = a \end{cases}$

Traits t can be connected to classes c by the asymmetric join:

$$(c \mathbin{\triangleleft\!\sqcup} t)(n) = \begin{cases} c(n) & \text{if } n \in \text{pre}(c) \\ t(n) & \text{otherwise} \end{cases}$$

Usually, this connection is reserved for the last composition level.

Traits – Concepts

Trait composition principles

Flat ordering All traits have the same precedence under $+$
⇝ explicit disambiguation with aliasing and exclusion

Precedence Under asymmetric join $\triangleleft\!\sqcup$, class methods take precedence over trait methods

Flattening After asymmetric join $\triangleleft\!\sqcup$: Non-overridden trait methods have the same semantics as class methods

⚠ Conflicts ...

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

Conflict treatment

✓ Methods can be aliased ($\rightarrow$)
✓ Methods can be excluded ($-$)
✓ Class methods override trait methods and sort out conflicts ($\triangleleft\!\sqcup$)

Can we augment classical languages by traits?

Extension Methods (C#)

Central Idea:

Uncouple method definitions from class bodies.

Purpose:

- retrospectively add methods to complex types
  ⇝ *external definition*
- especially provide definitions of *interface methods*
  ⇝ poor man's multiple inheritance!

Syntax:

1. Declare a static class with definitions of static methods
2. Explicitely declare first parameter as receiver with modifier `this`
3. Import the carrier class into scope (if needed)
4. Call extension method in *infix form* with emphasis on the receiver

```
public class Person{
 public int size = 160;
 public bool hasKey() { return true;}
}
public interface Short {}
public interface Locked {}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
 public static bool canPass(this Short leftHand, Person p){
  return p.size<160;
 }
}
public class ShortLockedDoor : Locked,Short {
 public static void Main() {
  ShortLockedDoor d = new ShortLockedDoor();
  Console.WriteLine(d.canOpen(new Person()));
 }
}
```

Extension Methods as Traits

Extension Methods

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

. . . but not traits

- Interface declarations empty, thus kind of purposeless
- Flattening not implemented
- Static scope only

Static scope of extension methods causes unexpected errors:

```
public interface Locked {
  public bool canOpen(Person p);
}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
}
```

Virtual Extension Methods (Java 8)

Java 8 advances one step further:

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

Implementation

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

⚠ Precedence

Still, default methods do not override methods from *abstract classes* when composed

Traits as General Composition Mechanism

⚠ Central Idea

Separate class generation from hierarchy specification and functional modelling

1. model hierarchical relations with interfaces
2. compose functionality with traits
3. adapt functionality to interfaces and add state via glue code in classes

Simplified multiple Inheritance without adverse effects

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

Squeak

Smalltalk

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

Syntax:

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

Traits in Squeak

```
Trait named: #TRStream uses: TPositionableStream
  on: aCollection
    self collection: aCollection.
    self setToStart.
  next
    ^ self atEnd
      ifTrue: [nil]
      ifFalse: [self collection at: self nextPosition].
Trait named: #TSynch uses: {}
  acquireLock
    self semaphore wait.
  releaseLock
    self semaphore signal.

Trait named: #TSyncRStream uses: TSynch+(TRStream@(#readNext -> #next))
  next
    | read |
    self acquireLock.
    read := self readNext.
    self releaseLock.
    ^ read.
```

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

All different kinds of type expressions:

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

Explicitly: Traits differ from Mixins

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

Lessons learned

Mixins

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

Traits

- Implementation Inheritance based approaches leave room for improvement in modularity in real world situations
- Traits offer *fine-grained control* of composition of functionality
- Native trait languages offer *separation of composition* of functionality from *specification* of interfaces

Further reading...

[1] G. Bracha and W. Cook.
Mixin-based inheritance.
European conference on object-oriented programming on Object-oriented programming systems, languages, and applications (OOPSLA/ECOOP), 1990.

[2] J. Britt.
Ruby 2.1.5 core reference, Dec. 2014.

[3] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black.
Traits: A mechanism for fine-grained reuse.
ACM Transactions on Programming Languages and Systems (TOPLAS), 2006.

[4] M. Flatt, S. Krishnamurthi, and M. Felleisen.
Classes and mixins.
Principles of Programming Languages (POPL), 1998.

[5] B. Goetz.
Interface evolution via virtual extension methods.
JSR 335: Lambda Expressions for the Java Programming Language, 2011.

[6] A. Hejlsberg, S. Wiltamuth, and P. Golde.
C# Language Specification.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[7] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black.
Traits: Composable units of behaviour.
European Conference on Object-Oriented Programming (ECOOP), 2003.

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Prototypes

Dr. Michael Petter

Winter 2019/20

Outline

Prototype based programming

1. Basic language features
2. Structured data
3. Code reusage
4. Imitating Object Orientation

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

Motivation – Polemic

Bothersome features

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

Prototype based programming

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

"Let's go back to basic concepts – *Lua*"

Basic Language Features

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

```
s = 0;
i = 1          -- Single line comment
p = i+s p=42   --[[ Multiline
comment --]]
s = 1
```

Basic Types and Values

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

```
a = true
type(a)                -- boolean
type("42"+0)           -- number
type("Petter "..1)      -- string
type(type)             -- function
type(nil)              -- nil
type([[<html><body>pretty long string</body>
</html>
]])                    -- string
a = 42
type(a)                -- number
```

Functions for Code

✓ First class citizens

```
function prettyprint(title, name, age)
  return title.." "..name..", born in "..(2018-age)
end

a = prettyprint
a("Dr.","Petter",42)

prettyprint = function (title, name, age)
  return name..", "..title
end
```

Introducing Structure

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

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

Table Lifecycle

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

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

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

Table Behaviour

Metatables

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

```
meta = {}                          -- create as plain empty table
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
a = { prefix="Dr.",name="Petter"} -- create Michael
setmetatable(a,meta)               -- install metatable for a
print(a)                           -- print "Dr. Petter"
```

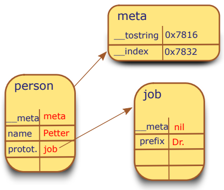
- Overload operators like `__add,__mul,__sub,__div,__pow,__concat,__unm`
- Overload comparators like `__eq,__lt,__le`

Delegation

- ⚠ reserved key `__index` determines *handling* of failed name lookups
- convention for signature: receiver table and key as parameters
- if dispatching to another table ⇝ *Delegation*

```
meta = {}
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
function meta.__index(tbl, key)
  return tbl.prototype[key]
end
job = { prefix="Dr." }
person = { name="Petter",prototype=job } -- create Michael
setmetatable(person,meta)                -- install metatable
print(person)                            -- print "Dr. Petter"
```

Delegation



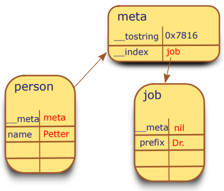

```
function meta.__tostring(person)  -- 0x7816
  return person.prefix .. " " .. person.name
end
function meta.__index(tbl, key) -- 0x7832
  return tbl.prototype[key]
end
```

Delegation 2

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

```
meta = {}
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
job = { prefix="Dr." }
meta.__index = job                   -- delegate to job
person = { name="Petter" }            -- create Michael
setmetatable(person,meta)            -- install metatable
print(person)                        -- print "Dr. Petter"
```

Delegation 2




```
function meta.__tostring(person)  -- 0x7816
  return person.prefix .. " " .. person.name
end
```

Delegation 3

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

```
meta = {}
function meta.__newindex(tbl,key,val)
  if (key == "title" and tbl.name=="Guttenberg") then
    error("No title for You, sir!")
  else
    tbl.data[key]=val
  end
end
function meta.__tostring(tbl)
  return (tbl.title or "") .. tbl.name
end
person={ data={} }           -- create person's data
meta.__index = person.data
setmetatable(person,meta)
person.name = "Guttenberg"    -- name KT
person.title = "Dr."          -- try to give him Dr.
```

Object Oriented Programming

⚠ so far no concept for multiple *objects*

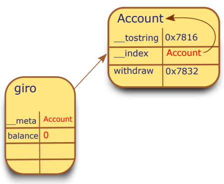
```
Account = { balance=0 }
function Account.withdraw (val)
  Account.balance=Account.balance-val
end
function Account.__tostring()
  return "Balance is "..Account.balance
end
setmetatable(Account,Account)
Account.withdraw(10)
print(Account)
```

Introducing Identity

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

```
function Account.withdraw (acc, val)
 acc.balance=acc.balance-val
end
function Account.tostring(acc)
  return "Balance is "..acc.balance
end
Account.__index=Account          -- share Account's functions
mikes = { balance = 0 }
daves = { balance = 0 }
setmetatable(mikes,Account)      -- delegate from mikes to Account
setmetatable(daves,Account)      -- del. from daves to Account
Account.withdraw(mikes,10)
mikes.withdraw(mikes,10)         -- withdraw independently
mikes:withdraw(10)
print(daves:tostring() .. " " .. mikes:tostring())
```

Introducing Identity




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

Introducing "Classes"

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

```
function Account:withdraw (val)
 self.balance=self.balance-val
end
function Account:tostring()
  return "Balance is "..self.balance
end
function Account:new(template)
 template = template or {balance=0}   -- initialize
 setmetatable(template,{__index=self})-- delegate to Account
 getmetatable(template).__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

```
LimitedAccount = { }
setmetatable(LimitedAccount,{__index=Account})
function LimitedAccount:new()
  instance = { balance=0,limit=100 }
  setmetatable(instance,{__index=self})
end
function LimitedAccount:withdraw(val)
  if (self.balance+self.limit < val) then
     error("Limit exceeded")
  end
  Account.withdraw(self,val)
end
specialgiro = LimitedAccount:new()
specialgiro:withdraw(90)
print(specialgiro)
```

Multiple Inheritance

⇝ Delegation leads to chain-like inheritance

```
function createClass (parent1,parent2)
  local c = {}                      -- new class, child of p1&p2
  setmetatable(c, {__index =
    function (t, k)                 -- search for each name
      local v = parent1[k]          -- in both parents
      if v then return v end
      return parent2[k]
    end}
  )
  c.__index = c                     -- c is prototype of instances
  function c:new (o)                -- constructor for this class
    o = o or {}
    setmetatable(o, c)              -- c is also metatable
    return o
  end
  return c                          -- finally return c
end
```

Multiple Inheritance

```
Doctor     = { postfix="Dr. "}
Researcher = { prefix=" ,Ph.D."}

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

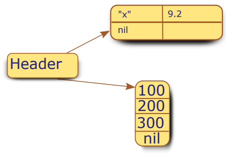
⇝ The special case of dual-inheritance can be extended to comprise multiple inheritance

Implementation of Lua

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

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

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



Further Topics in Lua

- Coroutines
- Closures
- Bytecode & Lua-VM

Lessons Learned

Lessons Learned

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

Further Reading...

[1] R. Ierusalimschy.
Programming in Lua, Third Edition.
Lua.Org, 2013.

[2] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes.
The implementation of lua 5.0.
Journal of Universal Computer Science, 2005.

[3] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho.
Lua-an extensible extension language.
Softw., Pract. Exper., 1996.

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Aspect Oriented Programming

Dr. Michael Petter
Winter 2019/20

"Is modularity the key principle to organizing software?"

Learning outcomes

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

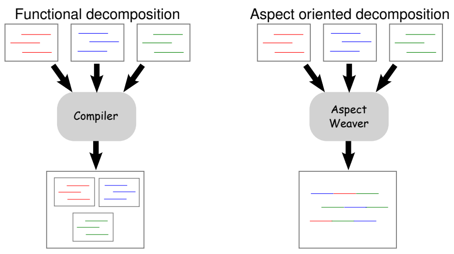
Motivation

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

⇝ Aspect *O*riented *P*rogramming

Aspect Oriented Programming

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



System Decomposition in Aspects

Example concerns:

- Security
- Logging
- Error Handling
- Validation
- Profiling

⇝ AspectJ

Static Crosscutting

Adding External Defintions

inter-type declaration

```
class Expr {}
class Const extends Expr {
  public int val;
  public Const(int val) {
    this.val=val;
} }
class Add extends Expr {
  public Expr l,r;
  public Add(Expr l, Expr r) {
   this.l=l;this.r=r;
} }

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

⇒

equivalent code

```
// aspectj-patched code
abstract class Expr {
  abstract int eval();
}
class Const extends Expr {
  public int val;
  public int eval(){ return val; };
  public Const(int val) {
    this.val=val;
} }
class Add extends Expr {
  public Expr l,r;
  public int eval() { return l.eval()
                         +  r.eval(); }
  public Add(Expr l, Expr r) {
   this.l=l;this.r=r;
  }
}
```

Dynamic Crosscutting

Join Points

Well-defined points in the control flow of a program

method/constr. call	executing the actual method-call statement
method/constr. execution	the individual method is executed
field get	a field is read
field set	a field is set
exception handler execution	an exception handler is invoked
class initialization	static initializers are run
object initialization	dynamic initializers are run

Pointcuts and Designators

Definition (Pointcut)

A pointcut is a *set of join points* and optionally some of the runtime values when program execution reaches a refered join point.

Pointcut designators can be defined and named by the programmer:

⟨*userdef*⟩ ::= '`pointcut`' ⟨*id*⟩ '(' ⟨*idlist*⟩? ')' ':' ⟨*expr*⟩ ';'

⟨*idlist*⟩ ::= ⟨*id*⟩ (',' ⟨*id*⟩)*

⟨*expr*⟩ ::= '!' ⟨*expr*⟩
| ⟨*expr*⟩ '&&' ⟨*expr*⟩
| ⟨*expr*⟩ '||' ⟨*expr*⟩
| '(' ⟨*expr*⟩ ')'
| ⟨*primitive*⟩

Example:

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

Advice

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

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

For example:

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

Binding Pointcut Parameters in Advices

Certain pointcut primitives add dependencies on the context:

- `args(arglist)`

This binds identifiers to parameter values for use in in advices.

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

`arglist` actually is a flexible expression:

⟨*arglist*⟩ ::= (⟨*arg*⟩ (',' ⟨*arg*⟩)*)?

⟨*arg*⟩ ::= ⟨*identifier*⟩	binds a value to this identifier
| ⟨*typename*⟩	filters only this type
| '*'	matches all types
| '..'	matches several arguments

Around Advice

Unusual treatment is necessary for

- `type around(formal)`

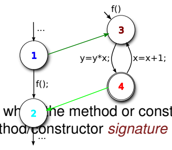
⚠ Here, we need to pinpoint, where the advice is wrapped around the join point – this is achieved via `proceed()`:

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

Pointcut Designator Primitives

Method Related Designators

- `call(signature)`
- `execution(signature)`



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

```
ResultTypeName RecvrTypeName.meth_id(ParamTypeName, ...)
NewObjectTypeName.new(ParamTypeName, ...)
```

⚠ Outdated: Adds a lot of joint points for call

https://www.eclipse.org/aspectj/doc/next/adk15notebook/join-point-signatures.html#method-call-join-point-signatures

Method Related Designators

```
class MyClass{
  public String toString() {
    return "silly me ";
  }
  public static void main(String[] args){
    MyClass c = new MyClass();
    System.out.println(c + c.toString());
} }
aspect CallAspect {
  pointcut calltostring() : call      (String MyClass.toString());
  pointcut exectostring() : execution(String MyClass.toString());
  before() :  calltostring() || exectostring() {
    System.out.println("advice!");
} }
```

```
advice!
advice!
advice!
silly me silly me
```

Field Related Designators

- `get(fieldqualifier)`
- `set(fieldqualifier)`

Matches field get/set join points at which the field accessed matches the signature. The syntax of a field qualifier is:

```
FieldTypeName ObjectTypeName.field_id
```

⚠ : However, set has an argument which is bound via `args`:

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

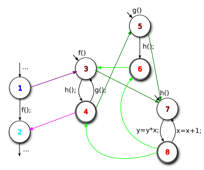
Type based

- `target(typeorid)`
- `within(typepattern)`
- `withincode(methodpattern)`

Matches join points of any kind which

- are refering to the receiver of type `typeorid`
- is contained in the class body of type `typepattern`
- is contained within the method defined by `methodpattern`

Flow and State Based



- `cflow(arbitrary_pointcut)`

Matches join points of *any kind* that occur strictly between entry and exit of each join point matched by `arbitrary_pointcut`.

- `if(boolean_expression)`

Picks join points based on a dynamic property:

```
aspect GuardedSetter {
  before(): if(thisJoinPoint.getKind().equals(METHOD_CALL)) && within(MyClass) {
    System.out.println("What an inefficient way to match calls");
} }
```

Which advice is served first?

Advices are defined in different aspects

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

Advices are defined in the same aspect

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

Implementation

Implementation

Aspect Weaving:

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

Woven JVM Code

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

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

```
...
 117: aload_1
 118: iconst_1
 119: dup_x1
 120: invokestatic  #73 // Method MyAspect.aspectOf:()LMyAspect;
 123: invokevirtual #79 // Method MyAspect.ajc$before$MyAspect$2$704a2754:()V
 126: putfield      #54 // Field Const.val:I
...
```

Woven JVM Code

```
Expr one = new Const(1);
Object e = new Add(one,one);
String s = e.toString();
System.out.println(s);
```

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

```
...
72: aload_2
73: instanceof    #1  // class Expr; pushes 1 if instance
76: ifeq          85  // jumps to 85 if 0 on stack
79: invokestatic  #67 // Method MyAspect.aspectOf:()MyAspect;
82: invokevirtual #70 // Method MyAspect.ajc$before$MyAspect$1$4c1f7c11:()V
85: aload_2
86: invokevirtual #33 // Method java/lang/Object.toString:()Ljava/lang/String;
89: astore_3
...
```

Poincut Parameters and Around/Proceed

Around clauses often refer to parameters and `proceed()`!

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

Now, imagine code like:

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

⚠ Injecting simple calls will not suffice!

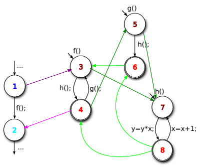
Around/Proceed – via Procedures

✓ injecting advice inplace of original call and outsource to an explicit method – all of it in JVM, disassembled to equivalent:

```
// aspectj patched code
public static void main(String[] args){
  C c = new C();
  foo_aroundBody1Advice(c,42,Doubler.aspectOf(),42,null);
}
private static final int foo_aroundBody0(C c, int i){
  return c.foo(i);
}
private static final int foo_aroundBody1Advice
    (C c, int i, Doubler d, int j, AroundClosure a){
      int temp = 2*i;
      int ret = foo_aroundBody0(c,temp);
      return ret / 2;
}
```

Property Based Crosscutting

```
after(int i) : call(void h()) &&
        cflow( call(void f(int)) &&
            args(i))
        { ... } ;
```



Idea 1: Stack based

- At each `call`-match, check runtime stack for `cflow`-match
- ⇝ Naive implementation
- ⇝ Poor runtime performance

Idea 2: State based

- Keep seperate stack of states
- ⇝ Only modify stack at `cflow`-relevant pointcuts
- ⇝ Check stack for emptyness

Even more optimizations in practice
⇝ state-sharing, ⇝ counters,
⇝ static analysis

Implementation – Summary

Translation scheme implications:

before/after Advice ... ranges from *inlined code* to distribution into *several methods and closures*

Joinpoints ... in the original program that have advices may get *explicitly dispatching wrappers*

Dynamic dispatching ... can require a *runtime test* to correctly interpret certain joinpoint designators

Flow sensitive pointcuts ... runtime penalty for the naive implementation, optimized version still *costly*

Aspect Orientation

Pro

- Un-tangling of concerns
- Late extension across boundaries of hierarchies
- Aspects provide another level of abstraction

Contra

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

Further reading...

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble.
Optimising aspectj.
SIGPLAN Not., 40(6):117–128, June 2005.

[2] G. Kiczales.
Aspect-oriented programming.
ACM Comput. Surv., 28(4es), 1996.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold.
An overview of aspectj.
ECOOP 2001 — Object-Oriented Programming, 2072:327–354, 2001.

[4] H. Masuhara, G. Kiczales, and C. Dutchyn.
A compilation and optimization model for aspect-oriented programs.
Compiler Construction, 2622:46–60, 2003.

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Metaprogramming

Dr. Michael Petter
Winter 2019/20

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

Learning outcomes

1. Compilers and Compiler Tools
2. Preprocessors for syntax rewriting
3. Reflection and Metaclasses
4. Metaobject Protocol
5. Macros

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

Metaprogramming

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

Codegeneration Tools

Codegeneration Tools

Compiler Construction

In Compiler Construction, there are a lot of codegeneration tools, that compile DSLs to target source code. Common examples are `lex` and `bison`.

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

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

⇝ generates 1.7k lines of C

Codegeneration via Preprocessor

Compiletime-Codegeneration

String Rewriting Systems

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

Example: C Preprocessor (CPP)

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

⚠ Nesting, Precedence, Binding, Side effects, Recursion, . . .

- Parts of Macro parameters can bind to context operators depending on the precedence and binding behaviour
- Side effects are recomputed for every occurance of the Macro parameter
- Any (indirect) recursive replacement stops the rewriting process
- Name spaces are not separated, identifiers duplicated

Compiletime-Codegeneration

Example application: Language constructs [3]:

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

```
#define ATOMIC(lock)    \
  acquire(&lock);\
  { /* user code */ }   \
  release(&lock);
```

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

⚠ How can we bind the block, following the `ATOMIC` to the usercode fragment?

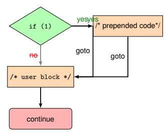
Particularly in a situation like this?

```
if (i>0)
  ATOMIC (mylock) {
    i--;
    i++;
  }
```

Compiletime-Codegeneration

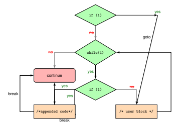
Prepend code to usercode

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



Append code to usercode

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



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

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

⚠ Reusability

labels have to be created dynamically in order for the macro to be reusable (→ `__LINE__`)

Homoiconic Metaprogramming

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

Reflection

Reflective Metaprogramming

Type introspection

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

Example: Java `instanceof`

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

Reflective Metaprogramming

Metaclasses (→ **code is data**)

Example: Java Reflection / Metaclass `java.lang.Class`

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

Metaobject Protocol

Metaobject Protocol

Metaobject Protocol (MOP [1])

Example: Lisp's CLOS metaobject protocol

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

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

Hygienic Macros

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

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

Remark: Established shortcut notation for lists:

$$(x_1\ x_2\ x_3) \quad \equiv \quad (x_1\ .\ (x_2\ .\ (x_3\ .\ ())))$$

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:

```
(def symbol doc? init?)
(do expr*)
(if test then else?)
(let [binding*] expr*)
(eval form)  ; evaluates the datastructure form
(quote form) ; yields the unevaluated form
(var symbol)
(fn name? ([params*] expr*)+)
(loop [binding*] expr*)
(recur expr*) ; rebinds and jumps to loop or fn
;...
```

Homoiconic Runtime-Metaprogramming

Macros

Macros are configurable syntax/parse tree transformations.

Language Implementation Idea: define advanced language features in macros, based very few special forms or other macros.

Example: While loop:

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

(macroexpand '(when a b))
;=> (if a (do b))
```

Homoiconic Runtime-Metaprogramming

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

```
(defmacro infix
  "converting infix to prefix"
  [infixed]
  (list (second infixed) (first infixed) (last infixed)))
```

...producing

```
(infix (1 + 1))
; => 2
(macroexpand '(infix (a + b)))
; => (+ a b)
```

⚠ Quoting

Macros and functions are directly interpreted, if not quoted via

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

Homoiconic Runtime-Metaprogramming

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

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

```clojure
(fac1 4)
; => 24
```

```clojure
(fac2 4)
; => 24
```

...produces

```clojure
(macroexpand '(fac1 4))
; => (* 4 (fac1 3))

(macroexpand-all '(fac1 4))
; => (* 4 (* 3 (* 2 (* 1 1))))
```

⇝ why bother?

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

Homoiconic Runtime-Metaprogramming

⚠ Macro Hygiene

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

```clojure
(def variable 42)
(macro mac [&stufftodo] `(let [variable 4711] ~@stufftodo))
(mac (println variable))
; => can't let qualified name: variable
```

```clojure
(macro mac [&stufftodo] `(let [variable# 4711] ~@stufftodo))
```

⇝ Symbol generation to avoid namespace collisions!

Further reading...

[1] R. P. Gabriel.
Gregor kiczales, jim des rivières, and daniel g. bobrow, the art of the metaobject protocol.
Artif. Intell., 61(2):331–342, 1993.

[2] D. Higginbotham.
Clojure for the Brave and True: Learn the Ultimate Language and Become a Better Programmer.
No Starch Press, San Francisco, CA, USA, 1st edition, 2015.

[3] S. Tatham.
Metaprogramming custom control structures in C.
https://www.chiark.greenend.org.uk/~sgtatham/mp/, 2012.
[Online; accessed 07-Feb-2018].

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

From Gotos to Continuations

Dr. Michael Petter
Winter 2019

"It's all about the program counter!"

Plain Old C goto Magic

```c
void myProcedure()
{
  int *a = malloc(sizeof(int)*15);
  ...
  for (int i=0;i<15;i++)
    for (int j=0;j<15;j++)
      if (a[i]-a[j] == 0) goto exit;
exit:
  free a;
  return;
}
```

⚠ gotos are scoped procedure-locally

Stack-Backward Control Flow

Stack Traversal with longjmp

performing control flow jumps across procedure boundaries is the domain of *setjmp/longjmp* (FreeBSD [4])

setjmp

```asm
;... signal blocking ...
movq    %rdi,%rcx
movq    0(%rsp),%rdx        ; return address
movq    %rdx, 0(%rcx)
movq    %rbx, 8(%rcx)
movq    %rsp,16(%rcx)
movq    %rbp,24(%rcx)
movq    %r12,32(%rcx)
movq    %r13,40(%rcx)
movq    %r14,48(%rcx)
movq    %r15,56(%rcx)
;... dealing with SSE / FPU
xorq    %rax,%rax
ret
```

longjmp

```asm
;... signal blocking / dealing with SSE Registers....
    movq    %rsi,%rax     ; 2nd param -> return value
    movq    0(%rdx),%rcx
    movq    8(%rdx),%rbx
    movq    16(%rdx),%rsp
    movq    24(%rdx),%rbp
    movq    32(%rdx),%r12
    movq    40(%rdx),%r13
    movq    48(%rdx),%r14
    movq    56(%rdx),%r15
    fldcw   64(%rdx)
    testq   %rax,%rax
    jnz     1f
    incq    %rax
1:  movq    %rcx,0(%rsp)  ; setjmp's return address
    ret
```

- control transfer by manipulating stackpointer and instruction pointer

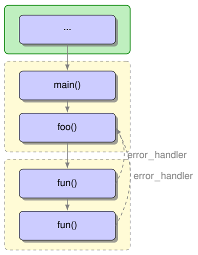
⇝ stack traversal only viable to enclosing stack frames, i.e. up the call hierarchy

Stack Traversal with longjmp

```c
int foo() {
  do {
    jmp_buf context;
    switch( setjmp(context) ) {  // TRY
      case 0: {
          printf("Main body\n");
          fun(&context, 0);
          printf("Not reachable\n");
      }
        break;
      case 1: { // CATCH 1
          printf("Caught 1!\n");
      }
        break;
      case 2: { // CATCH 2
          printf("Caught 2!\n");
      }
    }
  } while(0);
}
```

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

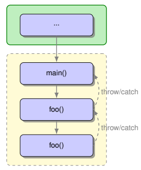
Stack Traversal with longjmp



⚠ heap objects might leak, after discarding several stack frames

Exceptions and Stack Unwinding [3]

```cpp
#include <iostream>
using namespace std;
int foo(int p){
  if (p>3) throw "Error!";
  else return foo(p+1);
}
int main(){
  try {
      return foo(1);
  } catch(const char* s){
      cerr << " Caught\n";
  }
}
```



✓ The compiler appends after the method's body a table of exceptions this method can catch and a cleanup table

1. The unwinder checks for each function in the stack which exceptions can be caught.
   - No catch for exception is found ⇝ `std::terminate`
   - Otherwise, the unwinder restarts on the top of the stack.
2. Again, the unwinder goes through the stack to perform a cleanup for this method. A so called ⇝ *personality routine* will check the cleanup table on the current method.
   - To run cleanup actions, it swaps to the current stack frame. This will run the destructor for each object allocated at the current scope.
   - Reaching the frame in the stack that can handle the exception, the unwinder jumps into the proper catch statement.

Same-Level Control Flow

Stack Switching with makecontext and swapcontext [9]

makecontext

```c
void makecontext(ucontext_t *ucp,
                 void (*func)(), int argc, ...);
```

- For preparation, the caller must
  - obtained a fresh context from a call to `getcontext()`
  - allocate a new stack for this context and assign its address to `ucp->uc_stack`
  - define a successor context and assign its address to `ucp->uc_link`
- `makecontext()` modifies the context pointed to by `ucp`
- On activation (using `swapcontext()`) the function `func` is called, and passed the `argc` many arguments of `int` type.
- When `func` returns, the successor context is activated. If the successor context pointer is NULL, the thread exits.

swapcontext

```c
int swapcontext(ucontext_t *oucp,const ucontext_t *ucp);
```

- `swapcontext()` saves the current context in `oucp`, and then activates `ucp`.

⇝ When successful, `swapcontext()` does not return. (But we may return later, in case `oucp` is activated, in which case it looks like `swapcontext()` returns 0.) On error, swapcontext() returns -1.

Stack Switching with makecontext and swapcontext

interleaved functions

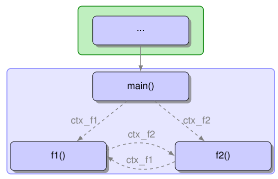
```c
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>
static ucontext_t ctx_m, ctx_f1, ctx_f2;
#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void f1(void) {
  printf("f1: started\n");
  printf("f1  --swapcontext--> f2\n");
  if (swapcontext(&ctx_f1, &ctx_f2) == -1) handle_error("swap");
  printf("f1: returning\n");
}
static void f2(void) {
  printf("f2: started\n");
  printf("f2  --swapcontext--> f1\n");
  if (swapcontext(&ctx_f2, &ctx_f1) == -1) handle_error("swap");
  printf("f2: returning\n");
}
```

startup platform

```c
int main(int argc, char *argv[]) {
   char f1_stack[16384];
   char f2_stack[16384];
   if (getcontext(&ctx_f1) == -1) handle_error("getcontext");
   ctx_f1.uc_stack.ss_sp = f1_stack;
   ctx_f1.uc_stack.ss_size = sizeof(f1_stack);
   ctx_f1.uc_link = &ctx_m;
   makecontext(&ctx_f1, f1, 0);
   if (getcontext(&ctx_f2) == -1) handle_error("getcontext");
   ctx_f2.uc_stack.ss_sp = f2_stack;
   ctx_f2.uc_stack.ss_size = sizeof(f2_stack);
   /* f2's successor context is f1(), unless argc > 1 */
   ctx_f2.uc_link = (argc > 1) ? NULL : &ctx_f1;
   makecontext(&ctx_f2, f2, 0);
   printf("main  --swapcontext--> f2\n");
   if (swapcontext(&ctx_m, &ctx_f2) == -1) handle_error("swap");
   printf("main: exiting\n");
   exit(EXIT_SUCCESS);
}
```

```
main --swapcontext--> f2
f2: started
f2  --swapcontext--> f1
f1: started
f1  --swapcontext--> f2
f2: returning
f1: returning
main: exiting
```

Stack Switching with makecontext and swapcontext



⚠ stack frame for subcontext
- size has to be known
- has to be allocated manually
- has to be allocated by parent frame

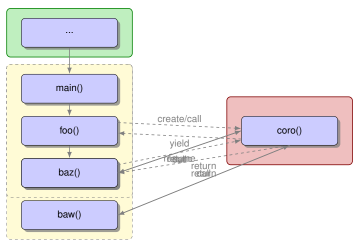
⚠ scheduling on termination depending on definition of a successor context

Stackless Coroutines

EcmaScript 6+:

```js
var genFn = function*(){
  var i = 0;
  while(true){
    yield i++;
  }
};
var gen = genFn();
while (true){
  var result = gen.next().value;
}
```

Stackless Coroutines

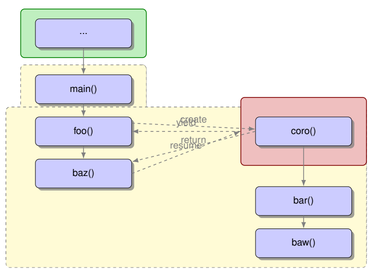


Stackful Coroutines

Lua:

```lua
function send (x)
  coroutine.yield(x)
end

local producer = coroutine.create(
  function ()
    while true do
      send(io.read())
    end
  end)
```

Stackful Coroutines



Generic Control Over the Program Counter

Continuations in Haskell

Abstracting Contexts

```
3 + 5*2 -1
3 + [5*2] -1
3 + [.] -1
```

- isolate the *context* of an expression within surrounding expression, i.e. `5*2`
- make the *context* a first level language construct

⇝ Continuations (Reynolds 1993[7])

Their counterpart

- is represented by already computed subexpressions
- is applicable to Continuations, yielding the final result

⇝ *Suspended Computations*

Continuation Passing Style (CPS) [8]

Transforming a function `f::a->b` into a CPS function `f'::a->((b->c)->c)` : `f'(k)`

- computes `f(k)` using only CPS styled functions and
- returns a function which, given a continuation `cont::b->c` returns `cont(f(k))`.
  ⇝ *suspended computation (`::(b->c)->c`)*

Direct style

```
square :: Int -> Int
square x = x * x

add :: Int -> Int -> Int
add x y = x + y

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

⇒

Continuation Passing Style

```
square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k ((*) x x)

add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k ((+) x y)

pyth_cps :: Int -> Int -> ((Int -> r) -> r)
pyth_cps x y = \k ->
  square_cps x (\x_squared ->
    square_cps y (\y_squared ->
      add_cps x_squared y_squared (k)))
```

Continuation Passing Style (CPS)

Higher order functions, that receive CPS styled functions as parameters

Direct style

```
trip :: (a -> a) -> a -> a
trip f x = f (f (f x))
```

⇒

Continuation Passing Style

```
trip_cps :: (a -> ((a -> r) -> r)) -> a -> ((a -> r) -> r)
trip_cps f_cps x = \k ->
   f_cps x (\fx ->
     f_cps fx (\ffx ->
       f_cps ffx (k)))
```

Function Parameter Signature: `(a->b)`

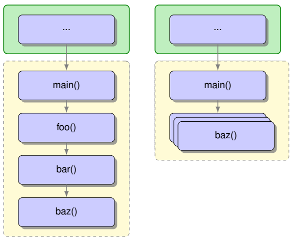
CPS Function Parameter Signature: `(a->((b->r)->r))`

Depending on how you where raised as a programmer (⇝ *functional* vs. *iterative*), this might look horrible to you – ⚠ is it even efficient at all?

Tail Call Optimization

Steele 1977 [6]

```
main :: IO
main = do
  print (foo(5))
foo :: Int -> Int
foo f = bar(10)

bar :: Int -> Int
bar b = baz(100)

baz :: Int -> Int
baz z = 100 + 100
```




```
main :: IO
main = do
  print (foo(5))
foo :: Int -> Int
foo f = 100+100
```

- Potentially generate new closure
- Reuse the existing stackframe
- Potentially shift actual parameters on stack
- *Jump* to called function

Composing Code by Continuations

Provide a function `compose`, that

- takes a suspended computation `s::(a->r)->r`
- takes a function in CPS style `f::a->((b->r)->r)`
- returns a composition of `f` to `s`, in form of another suspended computation `::(b->r)->r`

applying a CPS function to a *suspended computation*

```
compose :: ((a -> r) -> r) -> (a -> ((b -> r) -> r)) -> ((b -> r) -> r)
compose s f = \k -> s (\x -> f x (k))
```

The Cont Type Constructor

Data Constructor `Cont` represents suspended computations as a polymorphic Haskell data type, along with the functions:

⇝ `cont :: ((a -> r) -> r) -> Cont r a` creating a suspended computation

⇝ `runCont :: Cont r a -> (a -> r) -> r` computes the suspended computation with a given final function

Step by step introduce `Cont` into `compose`

```
compose' :: Cont r a -> (a -> Cont r b) -> Cont r b
compose' s f = cont (\k -> runCont s (\x -> runCont (f x) (k)))
```

⇕

Monadic bind: `(>>=) :: Monad m => m a -> (a -> m b) -> m b`

⇝ Can we constrain `Cont r` to a *Monad*?

Excursion: Monads

Essentials of Monads (Wadler 92 [10])

A monad is a ⇝ *type class* for arbitrary type constructors, defining at least a function called return, and a combinator function called bind or >>=

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Syntactic sugar: *do-notation*; allows to write monadic computations in a pseudo-imperative style

```
mothersPaternalGrandfather s =
 mother s >>= (\m ->
  father m >>= (\gf ->
    father gf))
```

⇒

```
mothersPaternalGrandfather s = do
   m  <- mother s
   gf <- father m
   father gf
```

Continuation Passing Style Monad

the Cont Monad

```
instance Monad (Cont r) where
    return x = cont (\k -> k x)
    s >>= f  = cont (\k -> runCont s (\x -> runCont (f x) k))
```

Continuation Passing Style

```
add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pyth_cps :: Int -> Int -> ((Int -> r) -> r)
pyth_cps x y = \k ->
 square_cps x (\x_squared ->
   square_cps y (\y_squared ->
     add_cps x_squared y_squared (k)))
```

Cont Monad Style

```
add_cont :: Int -> Int -> Cont r Int
add_cont x y = return (add x y)

square_cont :: Int -> Cont r Int
square_cont x = return (square x)

pythagoras_cont :: Int -> Int -> Cont r Int
pythagoras_cont x y = do
    x_squared <- square_cont x
    y_squared <- square_cont y
    add_cont x_squared y_squared
```

Call with Current Continuation

First implementation in Scheme

> *call/cc takes as 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 call/cc.*

Clinger et. al 1986[2]

callcc in CPS

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = cont (\h -> runCont (f (\a -> cont (\_ -> h a)) h ))

callCC' :: ((a->((b->r)->r)) -> ((a->r)->r)) -> ((a->r)->r)
callCC' f = (\h ->
  f (\a -> (\_ -> h a)) h
  )
```

- function parameter `f` is directly called by callcc with parameter `h` which
  - serves as direct continuation for `f`
  - is executable via a function call expression passed to `f` via some function parameter ignoring the continuation when called

Example: Control Structures with Call/CC

Loops with callcc

```
import Control.Monad.Trans.Class
import Control.Monad.Trans.Cont

main = flip runContT return $ do
  lift $ putStrLn "A"
  (k, num) <- callCC ( \c -> let f x = c (f, x)
                              in return (f, 0) )


  lift $ putStrLn "B"
  lift $ putStrLn "C"
  if num < 5
      then k (num + 1) >> return ()
      else lift $ print num
```

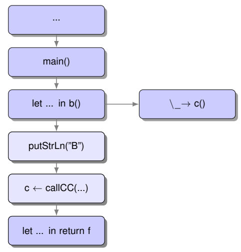
```
A
B
C
B
C
B
C
B
C
B
C
B
C
5
```

- Getting access to continuations may need a little monad trickery (⇝ lifting to Cont Monad)
- callCC now grants access to continuations (in this case *One-Shot* /*Escape Continuation* like exceptions)
- Continuations in Haskell via callCC are *Multi-Shot Continuations*

Implementation of Continuations [5]

```
main = flip runContT return $ do
  lift $ putStrLn "A"
  c <- callCC ( \k ->
        let f _ = k (f)
        in return (f)  )
  lift $ putStrLn "B"
  let b = \_ -> c() in b ()
```

- Continuations, returned from callcc may *escape the current context*/function frame
- calling continuations restarts execution at the original callcc site and function frame
- *Multi-Shot Continuations* may return to the same callcc site multiple times
- ⚠ traditional stack based frame management discards and overwrites old function frames



Roundup

Applications of call/cc

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

Lessons Learned

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

Further Topics

```
<3 + [.]> * 5
y = \f -> (\x -> f (x x)) (\x -> f (x x))
s = \f -> (\g -> (\x -> f x (g x)))
k = \x -> (\y -> x)
i = \x -> x
```

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

References

[1] K. Asai and O. Kiselyov.
Introduction to programming with shift and reset.
In *ACM SIGPLAN Continuation Workshop*, 2011.

[2] W. Clinger, D. P. Friedman, and M. Wand.
A Scheme for a Higher-Level Semantic Algebra, page 237–250.
Cambridge University Press, USA, 1986.

[3] Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI.
Itanium C++ ABI: Exception Handling.
https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html.

[4] FreeBSD.
setjmp implementation.
https://github.com/freebsd/freebsd/blob/master/lib/libc/amd64/gen/setjmp.S.

[5] R. Hieb, R. K. Dybvig, and C. Bruggeman.
Representing control in the presence of first-class continuations.
In B. N. Fischer, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 66–77. ACM, 1990.

[6] G. L. S. Jr.
Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: the ultimate GOTO.
In J. S. Ketchel, H. Z. Kriloff, H. B. Burner, P. E. Crockett, R. G. Herriot, G. B. Houston, and C. S. Kitto, editors, *Proceedings of the 1977 annual conference, ACM '77, Seattle, Washington, USA, October 16-19, 1977*, pages 153–162. ACM, 1977.

[7] J. C. Reynolds.
The discoveries of continuations.
Lisp and Symbolic Computation, 6(3-4):233–248, 1993.

[8] G. J. Sussman and G. L. Steele Jr.
Ai memo no. 349 december 1975.
contract, 14(75-C):0643.
http://www.laputan.org/pub/papers/aim-349.pdf.

[9] The IEEE and The Open Group.
The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition.
IEEE, New York, NY, USA, 2004.
https://pubs.opengroup.org/onlinepubs/009695399/functions/makecontext.html.

[10] P. Wadler.
The essence of functional programming.
In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, page 1–14, New York, NY, USA, 1992. Association for Computing Machinery.