Functional Programming + Verification

Helmut Seidl | TUM

Summer 2022

0 General

Contents of this lecture

- Correctness of programs
- Functional programming with OCaml

1 Correctness of Programs

- Programmers make mistakes !?
- Programming errors can be expensive, e.g., when a rocket explodes or a vital business system is down for hours ...
- Some systems must not have errors, e.g., control software of planes, signaling equipment of trains, airbags of cars ...

Problem

How can it be guaranteed that a program behaves as it should behave?

Approaches

- Careful engineering during software development
- Systematic testing

  $\Longrightarrow$ formal process model (Software Engineering)

- proof of correctness

  $\Longrightarrow$ verification

Tool: assertions

Example


```
public class GCD {
    public static void main (String[] args) {
        int x, y, a, b;
        a = read(); x = a;
        b = read(); y = b;
        while (x != y)
            if (x > y) x = x - y;
            else       y = y - x;

        assert (x == y);

        write(x);
    } // End of definition of  main();
}     // End of definition of class GCD;
```

Comments

- The static method `assert()` expects a Boolean argument.
- During normal program execution, every call `assert(e);` is ignored !?
- If Java is launched with the option: –ea (enable assertions), the calls of `assert` are evaluated:

  - ⇒ If the argument expression yields `true`, program execution continues.
  - ⇒ If the argument expression yields `false`, the error `AssertionError` is thrown.

Caveat

The run-time check should evaluate a property of the program state when reaching a particular program point.

The check should by no means change the program state (significantly) !!!
Otherwise, the behavior of the observed system differs from the unobserved system ???

In order to check properties of complicated data-structures, it is recommended to realize distinct inspector classes whose objects allow to inspect the data-structure without interference !

Problem

- In general, there are many program executions ...
- Validity of assertions can be checked by the Java run-time only for a specific execution at a time

$\Longrightarrow$

We require a general method in order to guarantee that a given assertion is valid ...

Robert W Floyd, Stanford U. (1936 – 2001)

Simplification

For the moment, we consider MiniJava only:

- only a single static method, namely, main
- only int variables
- only if and while.

Idea

- We annotate each program point with an assertion !
- At every program point, we argue that the assertion is valid ...

$\Longrightarrow$ logic

1.2 Background: Logic

Assertion: "All humans are mortal",
"Socrates is a human", "Socrates is mortal"

$\forall x.\,$human$(x) \Rightarrow$ mortal(x)
human(Socrates), mortal(Socrates)

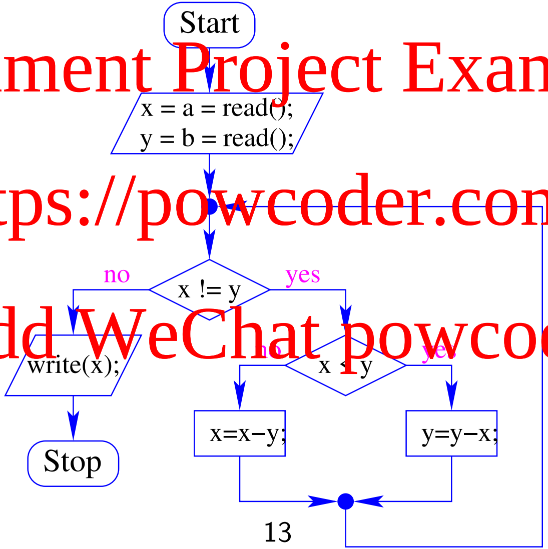
Deduction: If $\forall x.\, P(x)$ holds, then also $P(a)$ for a specific a !
If $A \Rightarrow B$ und A holds, then B must hold as well !

Tautology: $A \vee \neg A$
$\forall x \in \mathbb{Z}.\, x < 0 \;\vee\; x = 0 \;\vee\; x > 0$

1.2 Background: Logic

Laws:		
	$\neg\neg A \equiv A$	double negation
	$A \wedge A \equiv A$	idempotence
	$A \vee A \equiv A$	
	$\neg(A \vee B) \equiv \neg A \wedge \neg B$	De Morgan
	$\neg(A \wedge B) \equiv \neg A \vee \neg B$	
	$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$	distributivity
	$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$	
	$A \vee (B \wedge A) \equiv A$	absorption
	$A \wedge (B \vee A) \equiv A$	

Our Example



Discussion

- The program points correspond to the edges of the control-flow diagram !
- We require one assertion per edge ...

Background

$d \mid x$ holds iff $x = d \cdot z$ for some integer z.

For integers x, y, let $gcd(x, y) = 0$, if $x = y = 0$, and the greatest number d which both divides x and y, otherwise.

Then the following laws hold:

Discussion

$$
\begin{aligned}
gcd(x, 0) &= |x| \\
gcd(x, x) &= |x| \\
gcd(x, y) &= gcd(x, y - x) \\
gcd(x, y) &= gcd(x - y, y)
\end{aligned}
$$

Idea for the example

- Initially, nothing holds.
- After `a=read(); x=a;` $a = x$ holds.
- Before entering and during the loop, we should have:

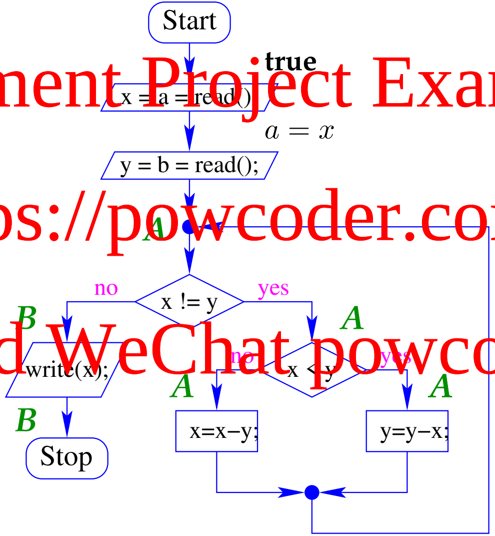
$$A \equiv gcd(a, b) = gcd(x, y)$$

- At program exit, we should have:

$$B \equiv A \wedge x = y$$

- These assertions should be locally consistent ...

Our Example



Start
true
x = a = read()
a = x
y = b = read();
A
no
x != y
yes
B
A
write(x);
no
x < y
yes
A
A
B
x=x−y;
y=y−x;
Stop

Question

How can we prove that the assertions are locally consistent?

Sub-problem 1: Assignments

Consider, e.g., the assignment: `x = y+z;`
In order to have after the assignment: $x > 0$, // post-condition
we must have before the assignment: $y + z > 0$. // pre-condition

General Principle

- Every assignment transforms a post-condition B into a minimal assumption that must be valid before the execution so that B is valid after the execution.
- In case of an assignment `x = e;` the weakest pre-condition is given by

$$\mathbf{WP}[\![\texttt{x = e;}]\!]\,(B) \quad \equiv \quad B[e/x]$$

This means: we simply substitute everywhere in B, x by e !!!

- An arbitrary pre-condition A for a statement s is valid, whenever

$$A \Rightarrow \mathbf{WP}[\![s]\!]\,(B)$$

// A implies the weakest pre-condition for B.

Example

assignment:	`x = x-y;`
post-condition:	$x > 0$
weakest pre-condition:	$x - y > 0$
stronger pre-condition:	$x - y > 2$
even stronger pre-condition:	$x - y = 3$

... in the GCD Program (1):

assignment: `x = x-y;`

post-condition: A

weakest pre-condition:

$$\begin{aligned} A[x-y/x] &\equiv gcd(a,b) = gcd(x-y,y) \\ &\equiv gcd(a,b) = gcd(x,y) \\ &\equiv A \end{aligned}$$

... in the GCD Program (2):

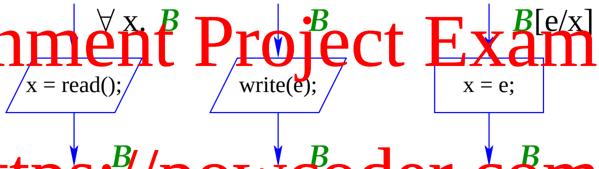
assignment: `y = y-x;`

post-condition: A

weakest pre-condition:

$$\begin{aligned} A[y-x/y] &\equiv gcd(a,b) = gcd(x, y-x) \\ &\equiv gcd(a,b) = gcd(x,y) \\ &\equiv A \end{aligned}$$

Wrap-up



$$
\begin{aligned}
\mathbf{WP}[\![\texttt{;}]\!](B) &\equiv B\\
\mathbf{WP}[\![\texttt{x = e;}]\!](B) &\equiv B[e/x]\\
\mathbf{WP}[\![\texttt{x = read();}]\!](B) &\equiv \forall x.\, B\\
\mathbf{WP}[\![\texttt{write(e);}]\!](B) &\equiv B
\end{aligned}
$$

Discussion

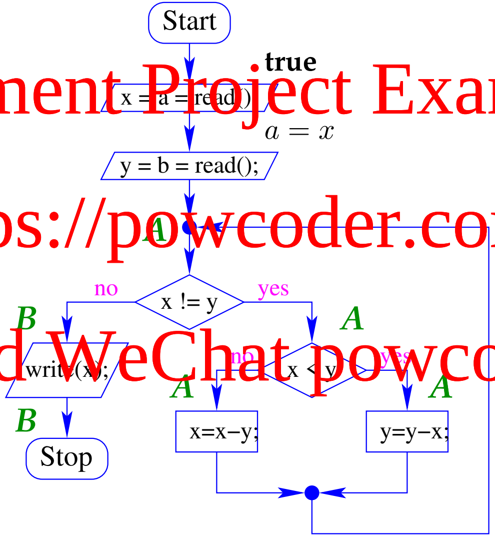
- For all actions, the wrap-up provides the corresponding weakest pre-conditions for a post-condition B.

- An output statement does not change any variable. Therefore, the weakest pre-condition is B itself.

- An input statement `x=read();` modifies the variable `x` unpredictably.

  In order B to hold after the input, B must hold for every possible `x` before the input.

Orientation

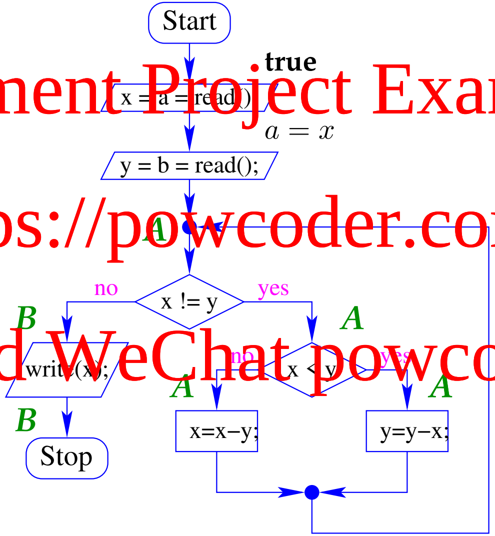


For the statements: `b = read(); y = b;` we calculate:

$$
\begin{aligned}
\mathbf{WP}[\![\texttt{y = b;}]\!]\;(A) &\equiv A[b/y] \\
&\equiv gcd(a,b) = gcd(x,b)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{WP}[\![\texttt{b = read();}]\!]\;(gcd(a,b) = gcd(x,b)) & \\
&\equiv \forall b.\, gcd(a,b) = gcd(x,b) \\
&\Leftarrow a = x
\end{aligned}
$$

Orientation

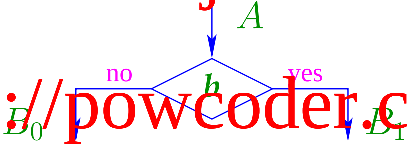


For the statements: `a = read(); x = a;` we calculate:

$$\begin{aligned} \mathbf{WP}[\![\texttt{x = a;}]\!]\,(a = x) &\equiv a = a \\ &\equiv \mathbf{true} \end{aligned}$$

$$\begin{aligned} \mathbf{WP}[\![\texttt{a = read();}]\!]\,(\mathbf{true}) &\equiv \forall\, a.\ \mathbf{true} \\ &\equiv \mathbf{true} \end{aligned}$$

Sub-problem 2: Conditionals



It should hold:

- $A \wedge \neg b \Rightarrow B_0$ and
- $A \wedge b \Rightarrow B_1$.

This is the case, if A implies the weakest pre-condition of the conditional branching:

$$\mathbf{WP}[\![b]\!]\,(B_0, B_1) \equiv ((\neg b) \Rightarrow B_0) \wedge (b \Rightarrow B_1)$$

The weakest pre-condition can be rewritten into:

$$\begin{aligned}\mathbf{WP}[\![b]\!]\,(B_0, B_1) &\equiv (b \vee B_0) \wedge (\neg b \vee B_1)\\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1) \vee (B_0 \wedge B_1)\\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1)\end{aligned}$$

Example

$$B_0 \equiv x > y \wedge y > 0 \qquad B_1 \equiv y > x \wedge x > 0$$

Assume that b is the condition $y > x$.

Then the weakest pre-condition is given by:

$$\begin{aligned} & (x \geq y \wedge x > y \wedge y > 0) \vee (y > x \wedge y > x \wedge x > 0) \\ \equiv \quad & (x > y \wedge y > 0) \vee (y > x \wedge x > 0) \\ \equiv \quad & x > 0 \wedge y > 0 \wedge x \neq y \end{aligned}$$

...for the GCD Example

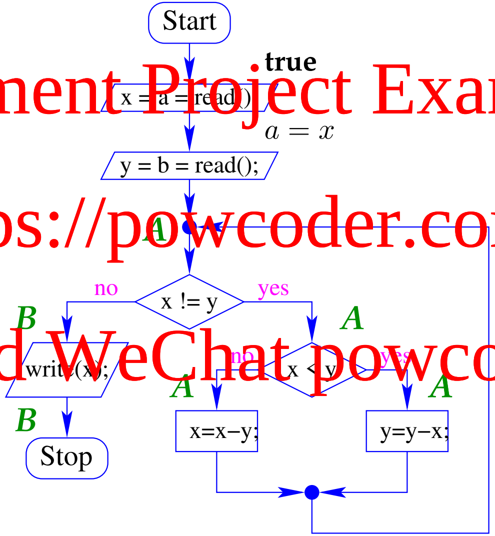
$$
\begin{aligned}
b &\equiv y > x \\
\neg b \wedge A &\equiv x \geq y \wedge gcd(a, b) = gcd(x, y) \\
b \wedge A &\equiv y > x \wedge gcd(a, b) = gcd(x, y)
\end{aligned}
$$

$\Longrightarrow$ The weakest pre-condition is given by

$$gcd(a, b) = gcd(x, y)$$

... i.e., exactly A

Orientation



The argument for the assertion before the loop is analogous:

$$b \equiv y \neq x$$

$$\neg b \wedge B \equiv A \wedge x = y$$

$$b \wedge A \equiv A \wedge x \neq y$$

$\Longrightarrow$ $A \equiv (A \wedge (x = y \vee x \neq y)$ is the weakest pre-condition for the conditional branching.

Summary of the Approach

- Annotate each program point with an assertion.
- Program start should receive annotation **true**.
- Verify for each statement s between two assertions A and B, that A implies the weakest pre-condition of s for B i.e.,

$$A \Rightarrow \mathbf{WP}[\![s]\!](B)$$

- Verify for each conditional branching with condition b, whether the assertion A before the condition implies the weakest pre-condition for the post-conditions B_0 and B_1 of the branching, i.e.,

$$A \Rightarrow \mathbf{WP}[\![b]\!]\,(B_0, B_1)$$

An annotation with the last two properties is called locally consistent.

1.3 Correctness

Questions

- Which program properties can be verified by means of locally consistent annotations ?
- How can we be sure that our method does not prove wrong claims ??

Recap (1)

- In MiniJava, the program state σ consists of a variable assignment, i.e., a mapping of program variables to integers (their values), e.g.,

$$\sigma = \{x \mapsto 5, y \mapsto -42\}$$

- A state σ satisfies an assertion A, if

$$A[\sigma(x)/x]_{x \in A}$$

// every variable in A is substituted by its value in σ

is a tautology, i.e., equivalent to **true**.

We write: $\sigma \models A$.

Example

$$
\begin{aligned}
\sigma &= \{x \mapsto 5, y \mapsto 2\} \\
A &\equiv (x > y) \\
A[5/x,\ 2/y] &\equiv (5 > 2) \\
&\equiv \textbf{true}
\end{aligned}
$$

$$
\begin{aligned}
\sigma &= \{x \mapsto 5, y \mapsto 12\} \\
A &\equiv (x > y) \\
A[5/x, 12/y] &\equiv (5 > 12) \\
&\equiv \textbf{false}
\end{aligned}
$$

Trivial Properties

$$\sigma \models \textbf{true} \quad \text{for every } \sigma$$
$$\sigma \models \textbf{false} \quad \text{for no } \sigma$$

$\sigma \models A_1$ and $\sigma \models A_2$ is equivalent to $\sigma \models A_1 \wedge A_2$

$\sigma \models A_1$ or $\sigma \models A_2$ is equivalent to $\sigma \models A_1 \vee A_2$

Recap (2)

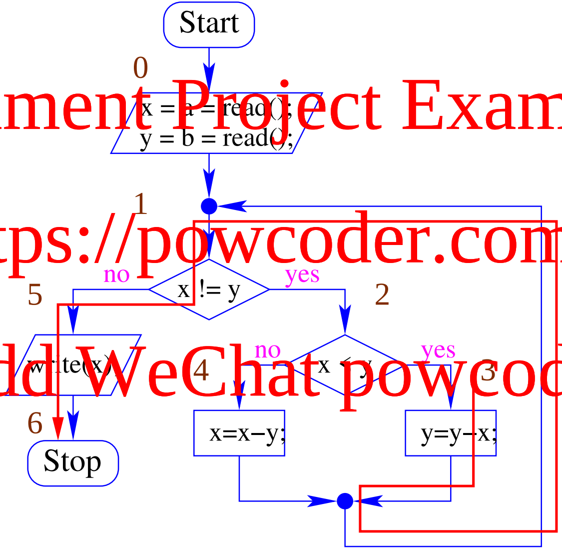
- An execution trace π traverses a path in the control-flow graph.
- It starts in a program point u_0 with an initial state σ_0 and leads to a program point u_m with a final state σ_m.
- Every step of the execution trace performs an action and (possibly) changes program point and state.

$\Longrightarrow$ The trace π can be represented as a sequence

$$(u_0, \sigma_0) s_1 (u_1, \sigma_1) \ldots s_m (u_m, \sigma_m)$$

where s_i are elements of the control-flow graph, i.e., basic statements or (possibly negated) conditional expressions (guards) ...

Example


Start
0
x = a = read();
y = b = read();
1
no
x != y
yes
5
2
write(x)
no
x < y
yes
4
3
6
Stop
x=x−y;
y=y−x;

Assume that we start in point 3 with $\{x \mapsto 6, y \mapsto 12\}$.

Then we obtain the following execution trace:

$$
\begin{array}{rcll}
\pi & = & (3, \{x \mapsto 6, y \mapsto 12\}) & \texttt{y = y-x;} \\
 & & (1, \{x \mapsto 6, y \mapsto 6\}) & \neg\texttt{(x != y)} \\
 & & (5, \{x \mapsto 6, y \mapsto 6\}) & \texttt{write(x);} \\
 & & (6, \{x \mapsto 6, y \mapsto 6\}) &
\end{array}
$$

Important operation: Update of of state

$$\sigma \oplus \{x \mapsto d\} = \{z \mapsto \sigma\, z \mid z \not\equiv x\} \cup \{x \mapsto d\}$$

$$\{x \mapsto 6, y \mapsto 12\} \oplus \{y \mapsto 6\} \; = \; \{x \mapsto 6, y \mapsto 6\}$$

Theorem

Let p be a MiniJava program, let π be an execution trace starting in program point u and leading to program point v.

Assumptions:

- The program points in p are annotated by assertions which are locally consistent.

- The program point u is annotated with A.
- The program point v is annotated with B.

Conclusion:

If the initial state of π satisfies the assertion A, then the final state satisfies the assertion B.

Remarks

- If the start point of the program is annotated with **true**, then every execution trace reaching program point v satisfies the assertion at v.
- In order to prove that an assertion A holds at a program point v, we require a locally consistent annotation satisfying:

  (1) The start point is annotated with **true**.

  (2) The assertion at v implies A.

- So far, our method does not provide any guarantee that v is ever reached !!!
- If a program point v can be annotated with the assertion **false**, then v cannot be reached.

Proof

Let $\pi = (u_0, \sigma_0)\, s_1\, (u_1, \sigma_1) \ldots s_m\, (u_m, \sigma_m)$

Assumption: $\sigma_0 \models A$.

Proof obligation: $\sigma_m \models B$.

Idea

Induction on the length m of the execution trace.

Base $\boxed{m = 0:}$

The endpoint of the execution equals the startpoint.

$\Longrightarrow$ $\sigma_0 = \sigma_m$ and $A \equiv B$

$\Longrightarrow$ the claim holds.

Important Notion: Evaluation of Expressions

Program State

$$\sigma = \{x \mapsto 5, y \mapsto -1, z \mapsto 21\}$$

Arithmetic Expression

$$t \equiv \mathtt{2 * z + y}$$

Evaluation

$$\begin{aligned} [\![t]\!]\,\sigma &= [\![\mathtt{2 * z + y}]\!]\,\{\mathtt{x} \mapsto \mathtt{5}, \mathtt{y} \mapsto \mathtt{-1}, \mathtt{z} \mapsto \mathtt{21}\} \\ &= 2 \cdot 21 + (-1) \\ &= 41 \end{aligned}$$

Proposition

For (arithmethic) expressions t, e,

$$[\![t]\!]\,(\sigma \oplus \{x \mapsto [\![e]\!]\,\sigma\}) = [\![t[e/x]]\!]\,\sigma$$

E.g.,, consider $t \equiv \mathtt{x} + \mathtt{y}$, $e \equiv 2 * \mathtt{z}$
for $\sigma = \{x \mapsto 5, y \mapsto -1, z \mapsto 21\}$

$$\begin{aligned}
[\![t]\!]\,(\sigma \oplus \{x \mapsto [\![e]\!]\,\sigma\}) &= [\![t]\!]\,(\sigma \oplus \{x \mapsto 42\}) \\
&= [\![t]\!]\,(\{x \mapsto 42, y \mapsto -1, z \mapsto 21\}) \\
&= 42 + (-1) \quad = \quad 41 \\
[\![t[e/\mathtt{x}]]\!]\,\sigma &= [\![(2 * \mathtt{z}) + \mathtt{y}]\!]\,\sigma \\
&= (2 \cdot 21) - 1 \quad = \quad 41
\end{aligned}$$

Proposition

$$\sigma \oplus \{x \mapsto [\![e]\!]\,\sigma\} \models t_1 < t_2 \quad \text{iff} \quad \sigma \models t_1[e/\mathtt{x}] < t_2[e/\mathtt{x}]$$

Proof

$$\begin{aligned}
\sigma \oplus \{x \mapsto [\![e]\!]\,\sigma\} &\models t_1 < t_2 \\
&\text{iff} \quad [\![t_1]\!]\,(\sigma \oplus \{x \mapsto [\![e]\!]\,\sigma\}) < [\![t_2]\!]\,(\sigma \oplus \{x \mapsto [\![e]\!]\,\sigma\}) \\
&\text{iff} \quad [\![t_1[e/\mathtt{x}]]\!]\,\sigma < [\![t_2[e/\mathtt{x}]]\!]\,\sigma \\
&\text{iff} \quad \sigma \models t_1[e/\mathtt{x}] < t_2[e/\mathtt{x}] \qquad \square
\end{aligned}$$

Proposition

for every formula A,

$$\sigma \oplus \{x \mapsto [\![e]\!]\,\sigma\} \models A \quad \text{iff} \quad \sigma \models A[e/\mathtt{x}]$$

Proof

Induction on the structure of formula A $\square$

Induction Proof of Correctness (cont.)

Step $\boxed{m > 0}$:

Induction Hypothesis: The statement holds already for $m - 1$.

Let B' denote the assertion at point u_{m-1}.

$\Longrightarrow \quad \sigma_{m-1} \models B'$

First, we deal with statements.

Case 1. $\quad s_m \equiv\ ;$

Then
- $\sigma_{m-1} = \sigma_m$
- $\mathbf{WP}[\![\,;\,]\!]\,(B) \equiv B$

$\Longrightarrow \quad B' \Rightarrow B$

$\Longrightarrow \quad \sigma_{m-1} = \sigma_m \models B \qquad \square$

Induction Proof of Correctness (cont.)

Case 2. $s_m \equiv$ `write(e);`

Then:

- $\sigma_{m-1} = \sigma_m$
- $\mathbf{WP}[\![\texttt{write(e);}]\!](B) \equiv B$

$\Longrightarrow \quad B' \Rightarrow B$

$\Longrightarrow \quad \sigma_{m-1} = \sigma_m \models B \quad \Box$

Case 3. $s_m \equiv$ `x = e;` Then we have:

- $\sigma_m = \sigma_{m-1} \oplus \{x \mapsto [\![e]\!]\,\sigma_{m-1}\}$
- $B' \Rightarrow \mathbf{WP}[\![\texttt{x = e;}]\!](B) \equiv B[e/x]$

$\Longrightarrow \quad \sigma_{m-1} \models B[e/x]$

$\Longrightarrow \quad \sigma_{m-1} \models B[e/x]$ iff $\sigma_m \models B$

$\Longrightarrow \quad \sigma_m \models B \quad \Box$

Induction Proof of Correctness (cont.)

Case 4. $s_m \equiv$ `x = read();`

Then

- $\sigma_m = \sigma_{m-1} \oplus \{x \mapsto c\}$ for some $c \in \mathbb{Z}$
- $\mathbf{WP}[\![$`x = read();`$]\!](B) \equiv \forall x.\, B$

$\Longrightarrow \quad B' \Rightarrow \forall x.\, B \Rightarrow B[c/x]$

$\Longrightarrow \quad \sigma_m \models B \qquad \square$

Induction Proof of Correctness (cont.)

Step $\boxed{m > 0}$:

Induction Hypothesis: The statement holds already for $m - 1$.

Let B' denote the assertion at point u_{m-1}.

$\Longrightarrow \sigma_{m-1} \models B'$

Finally, consider tests $\boxed{s_m \equiv b}$.

Then in particular $\sigma_{m-1} = \sigma_m$

Induction Proof of Correctness (cont.)

Case 1. $\sigma_m \models b$

$\Longrightarrow \quad B' \Rightarrow \mathbf{WP}[\![b]\!]\,(C, B)$ where

$\mathbf{WP}[\![b]\!]\,(C, B) \equiv (\neg b \Rightarrow C) \wedge (b \Rightarrow B)$

$\Longrightarrow \quad \sigma_m \models b \wedge (b \Rightarrow B)$

$\Longrightarrow \quad \sigma_m \models B \quad \square$

Case 2. $\sigma_m \models \neg b$

$\Longrightarrow \quad B' \Rightarrow \mathbf{WP}[\![b]\!]\,(B, C)$ where

$\mathbf{WP}[\![b]\!]\,(B, C) \equiv (\neg b \Rightarrow B) \wedge (b \Rightarrow C)$

$\Longrightarrow \quad \sigma_m \models \neg b \wedge (\neg b \Rightarrow B)$

$\Longrightarrow \quad \sigma_m \models B \quad \square$

This completes proof of the theorem.

Conclusion

- The method of Floyd allows us to prove that an assertion B holds whenever (or under certain assumptions) a program point is reached ...

- For the implementation, we require:

  - the assertion **true** at the start point
  - assertions for each further program point
  - a proof that the assertions are locally consistent

$\Longrightarrow$ Logic, automated theorem proving

1.4 Optimization

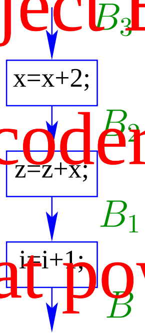
Goal: Reduction of the number of required assertions

Observation

If the program has no loops, a weakest pre-condition can be calculated for each program point !!!

Example


```
x = x + 2;
z = z + x;
i = i + 1;
```



Example (cont.)

Assume $B \equiv z = i^2 \wedge x = 2i - 1$
Then we calculate:

$$
\begin{aligned}
B_1 &\equiv \mathbf{WP}[\![\texttt{i = i+1;}]\!](B) &&\equiv z = (i+1)^2 \wedge x = 2(i+1) - 1 \\
& &&\equiv z = (i+1)^2 \wedge x = 2i + 1 \\
B_2 &\equiv \mathbf{WP}[\![\texttt{z = z+x;}]\!](B_1) &&\equiv z + x = (i+1)^2 \wedge x = 2i + 1 \\
& &&\equiv z = i^2 \wedge x = 2i + 1 \\
B_3 &\equiv \mathbf{WP}[\![\texttt{x = x+2;}]\!](B_2) &&\equiv z = i^2 \wedge x + 2 = 2i + 1 \\
& &&\equiv z = i^2 \wedge x = 2i - 1 \\
& &&\equiv B
\end{aligned}
$$

Idea

- For every loop, select one program point.

  Meaningful selections:

  → Before the condition

  → At the entry of the loop body

  → At the exit of the loop body ...

- Provide an assertion for each selected program point

  $\Longrightarrow$ loop invariant

- For all other program points, the assertions are obtained by means of $\mathbf{WP}[\![\ldots]\!]()$.

Example

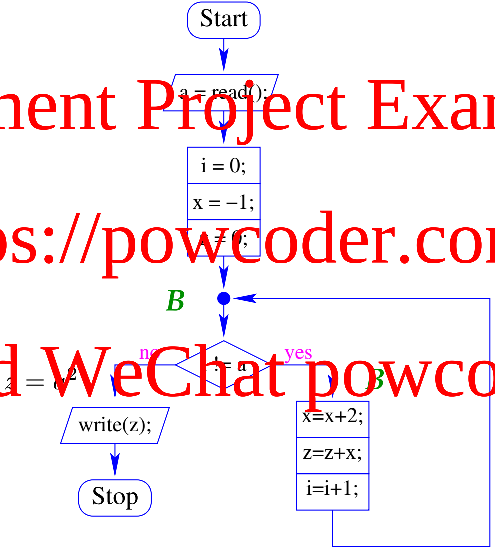
```
int a, i, x, z;
a = read();
i = 0;
x = -1;
z = 0;

while (i != a) {
        x = x + 2;
        z = z + x;
        i = i + 1;
}

assert(z == a * a);

write(z);
```

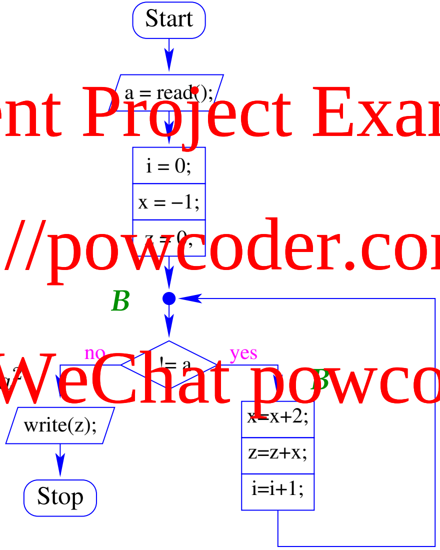
Example



We verify

$$
\begin{aligned}
\mathbf{WP}[\![\mathsf{i\ !=\ a}]\!](z = a^2, B) &\equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge B) \\
&\equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \\
&\Leftarrow (i = a \wedge z = i^2 \wedge x = 2i - 1) \vee (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \\
&\equiv z = i^2 \wedge x = 2i - 1 \ \equiv \ B
\end{aligned}
$$

Orientation



We verify:

$$
\begin{aligned}
\mathbf{WP}[\![\texttt{z = 0;}]\!](B) &\equiv 0 = i^2 \wedge x = 2i - 1 \\
&\equiv i = 0 \wedge x = -1 \\
\mathbf{WP}[\![\texttt{x = -1;}]\!](i = 0 \wedge x = -1) &\equiv i = 0 \\
\mathbf{WP}[\![\texttt{i = 0;}]\!](i = 0) &\equiv \mathbf{true} \\
\mathbf{WP}[\![\texttt{a = read();}]\!](\mathbf{true}) &\equiv \mathbf{true}
\end{aligned}
$$

1.5 Termination

Problem

- By our approach, we can only prove that an assertion is valid at a program point whenever that program point is reached !!!

- How can we guarantee that a program always terminates ?
- How can we determine a sufficient condition which guarantees termination of the program ??

Examples

- The GCD program only terminates for inputs a, b with $a = b$ or $a > 0$ and $b > 0$.
- The square program terminates only for inputs $a \geq 0$.
- `while(true);` never terminates.
- Programs without loops terminate always!

Can this example be generalized ??

Example

```
int i, j, t;
t = 0;
i = read();
while (i > 0) {
        j = read();
        while (j > 0) { t = t + 1; j = j - 1; }
        i = i - 1;
}
write(t);
```


- The read number i (if non-negative) indicates how often j is read.
- The total running time (essentially) equals the sum of all non-negative values read into j

$\Longrightarrow$ the program always terminates !!!

Programs with `for`-loops only of the form:

```
for (i=n; i>0; i--) {...}
// i is not modified in the body
```

... always terminate !

Question

How can we turn this observation into a method that is applicable to arbitrary loops ?

Idea

- Make sure that each loop is executed only finitely often . . .
- For each loop, identify an indicator value r, that has two properties

  (1) $r > 0$ whenever the loop is entered;

  (2) r is decreased during every iteration of the loop.

- Transform the program in a way that, alongside ordinary program execution, the indicator value r is computed.
- Verify that properties (1) and (2) hold!

Eample: Safe GCD Program

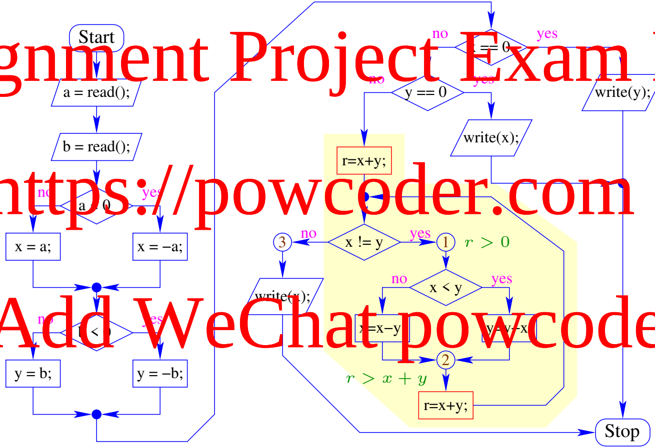

```
int a, b, x, y;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
     else {
        while (x != y)
           if (y > x) y = y - x;
           else       x = x - y;
        write(x);
}
```

We choose: $r = x + y$

Transformation

```
int a, b, x, y, r;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
     else { r = x + y;
        while (x != y) {
           if (y > x) y = y - x;
           else       x = x - y;
           r = x + y; }
        write(x);
}
```

Orientation



At program points 1, 2 and 3, we assert:

$$
\begin{array}{lll}
(1) & A & \equiv \quad x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y \\
(2) & B & \equiv \quad x > 0 \wedge y > 0 \wedge r > x + y \\
(3) & \textbf{true} &
\end{array}
$$

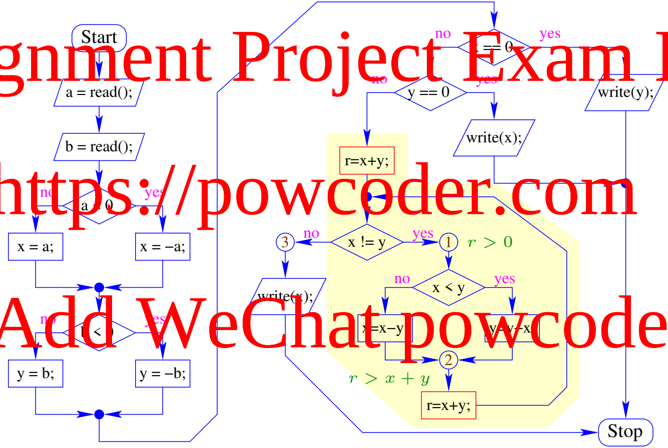
Then we have:

$$A \Rightarrow r > 0 \qquad \text{und} \qquad B \Rightarrow r > x + y$$

We verify:

$$
\begin{array}{rcl}
\mathbf{WP}[\![\texttt{x != y}]\!](\mathbf{true}, A) & \equiv & x = y \vee A \\
& \Leftarrow & x > 0 \wedge y > 0 \wedge r = x + y \\
& \equiv & C \\
\mathbf{WP}[\![\texttt{r = x+y;}]\!](C) & \equiv & x > 0 \wedge y > 0 \\
& \Leftarrow & B \\
\mathbf{WP}[\![\texttt{x = x-y;}]\!](B) & \equiv & x > y \wedge y > 0 \wedge r > x \\
\mathbf{WP}[\![\texttt{y = y-x;}]\!](B) & \equiv & x > 0 \wedge y > x \wedge r > y \\
\mathbf{WP}[\![\texttt{y > x}]\!](\ldots, \ldots) & \equiv & (x > y \wedge y > 0 \wedge r > x) \vee \\
& & (x > 0 \wedge y > x \wedge r > y) \\
& \Leftarrow & x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y \\
& \equiv & A
\end{array}
$$

Orientation



Further propagation of C through the control-flow graph completes the locally consistent annotation with assertions.

We conclude:

- At program points 1 and 2, the assertions $r > 0$ and $r > x + y$, respectively, hold.
- During every iteration, r decreases, but stays non-negative.
- Accordingly, the loop can only be iterated finitely often.

$\Longrightarrow$ the program terminates!

General Method

- For every occurring loop `while (b) s` we introduce a fresh variable `r`.
- Then we transform the loop into:


```
r = e0;
while(b) {
        assert(r > 0);
        s;
        assert(r > e1);
        r = e1;
}
```


for suitable expressions e0, e1.

1.6 Modular Verification and Procedures

Tony Hoare, Microsoft Research, Cambridge

Idea

- Modularize the correctness proof in a way that sub-proofs for replicated program fragments can be reused.
- Consider statements of the form:

$$\{A\} \quad p \quad \{B\}$$

... this means:

If before the execution of program fragment p, assertion A holds and program execution terminates, then
after execution of p assertion B holds.

A : pre-condition
B : post-condition

Examples

$\{x > y\}$ `z = x-y;` $\{z > 0\}$

$\{\textbf{true}\}$ `if (x<0) x=-x;` $\{x \geq 0\}$

$\{x > 7\}$ `while (x!=0) x=x-1;` $\{x = 0\}$

$\{\textbf{true}\}$ `while (true);` $\{\textbf{false}\}$

Modular verification can be used to prove the correctness of programs using functions/methods.

Simplification

We only consider

- procedures, i.e., static methods without return values;
- global variables, i.e., all variables are `static` as well.

  // will be generalized later

Example


```c
int a, b;
int x, y;

void main() {
    a = read();
    b = read();
    mm();
    write(x - y);
}
```

```c
void mm() {
    if (a > b) {
        x = a;
        y = b;
    } else {
        y = a;
        x = b;
    }
}
```

Comment

- for simplicity, we have removed all qualifiers static.
- The procedure definitions are not recursive.
- The program reads two numbers.
- The procedure `minmax` stores the larger number in `x`, and the smaller number in `y`.
- The difference of `x` and `y` is returned.
- Our goal is to prove:

$$\{a \geq b\}\ \texttt{mm();}\ \{a = x\}$$

Approach

- For every procedure `f()`, we provide a triple

$$\{A\}\ \texttt{f();}\ \{B\}$$

- Relative to this global hypothesis H we verify for each procedure definition `void f() { ss }` that

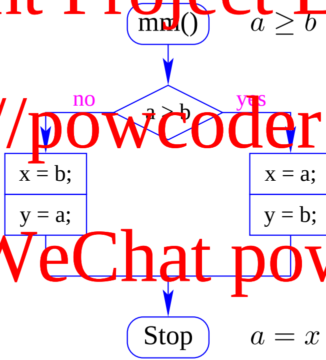
$$\{A\}\ \texttt{ss}\ \{B\}$$

holds.

- Whereever a procedure call occurs in the program, we rely on the triple from H ...

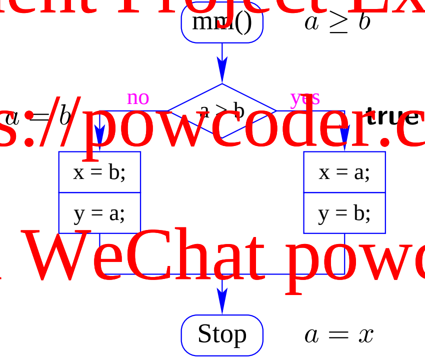
… in the Example

We verify:



... in the Example

We verify:



Discussion

- The approach also works in case the procedure has a return value: that can be simulated by means of a global variable `return` which receives the respective function results.

- It is not obvious, though, how pre- and post-conditions of procedure calls can be chosen if a procedured is called in multiple places ...
- Even more complicated is the situation when a procedure is recursive: the it has possibly unboundedly many distinct calls !?

Example


```c
int x, m0, m1, t;

void main () {
    x = read();
    m0 = 1; m1 = 1;
    if (x > 1) f();
    write(m1);
}
```



```c
void f() {
    x = x-1;
    if (x > 1)
        f();
    t = m1;
    m1 = m0 + m1;
    m0 = t;
}
```

Comment

- The program reads a number.
- If the number is at most 1, the program returns 1 ...
- Otherwise, the program computes the Fibonacci function fib.
- After a call to `f`, the variables `m0` and `m1` have the values fib$(i-1)$ and fib(i), respectively ...

Problem

- In the logic, we must be able to distinguish between the ith and the $(i+1)$th call.
- This is easier, if we have logical auxiliaries $\underline{l} = l_1, \ldots, l_n$ at hand to store (selected) values before the call ...

In the Example

$\{A\}$ `f();` $\{B\}$ where

$$\begin{array}{lcl} A & \equiv & x = l \wedge x > 1 \wedge m_0 = m_1 = 1 \\ B & \equiv & l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1} \end{array}$$

General Approach

- Again, we start with a global hypothesis H which provides a description

$$\{A\}\ \texttt{f();}\ \{B\}$$

// both A and B may contain l_i

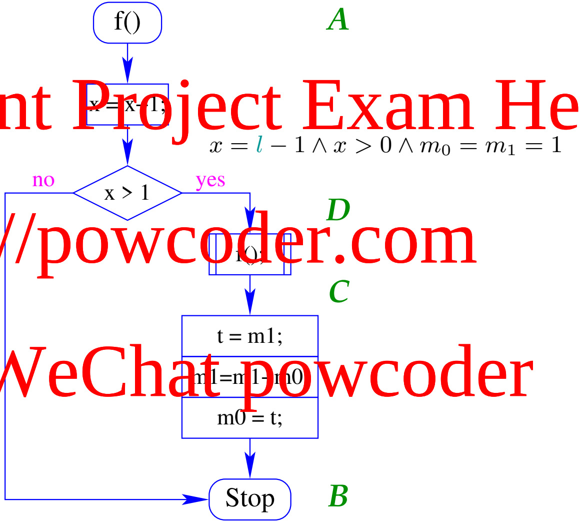
for each call of `f();`

- Given this global hypothesies H we verify for each procedure definition `void f() { ss }` that

$$\{A\}\ \texttt{ss}\ \{B\}$$

holds.

... in the Example



- We start with an assertion for the end point:

$$B \equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}$$

- The assertion C is obtained by means of $\mathbf{WP}[\![\ldots]\!]$ and weakening ...

$$\begin{aligned}
\mathbf{WP}[\![\texttt{t=m1; m1=m1+m0; m0=t;}]\!]\,(B) & \\
\equiv \quad & l - 1 > 0 \wedge m_1 + m_0 \leq 2^l \wedge m_1 \leq 2^{l-1} \\
\Leftarrow \quad & l - 1 > 1 \wedge m_1 \leq 2^{l-1} \wedge m_0 \leq 2^{l-2} \\
\equiv \quad & C
\end{aligned}$$

Question

How can the global hypothesis be used to deal with a specific procedure call ???

Idea

- The assertion $\{A\}$ `f();` $\{B\}$ represents a value table for `f()`.
- This value table can be logically represented by the implication:

$$\forall\, \underline{l}.\ \left(A[\underline{h}/\underline{x}] \Rightarrow B\right)$$

// $\underline{h}$ denotes a sequence of auxiliaries

The values of the variables $\underline{x}$ before the call are recorded in the auxiliaries.

Examples

Function:	`void double () { x = 2*x;}`
Specification:	$\{x = l\}$ `double();` $\{x = 2l\}$
Table:	$\forall\, l.(h = l) \Rightarrow (x = 2l)$
	$\equiv \quad (x = 2h)$

For the Fibonacci function, we calculate:

$$\begin{aligned} \forall\, l.\ \ (h > 1 \wedge h = l \wedge h_0 = h_1 = 1) \ \ &\Rightarrow \\ & l > 1 \wedge m_1 \leq 2^{l} \wedge m_0 \leq 2^{l-1} \end{aligned}$$

$$\equiv \quad (h > 1 \wedge h_0 = h_1 = 1) \ \Rightarrow\ m_1 \leq 2^{h} \wedge m_0 \leq 2^{h-1}$$

Another pair (A_1, B_1) of assertions forms a valid triple $\{A_1\}$ `f();` $\{B_1\}$, if we are able to prove that

$$\frac{\forall l.\ A[\underline{h}/\underline{x}] \longrightarrow B \qquad A_1[\underline{h}/\underline{x}]}{B_1}$$

Example: `double()`

$$\begin{array}{lcl} A & \equiv & x = l \\ A_1 & \equiv & x \geq 3 \end{array} \qquad \begin{array}{lcl} B & \equiv & x = 2l \\ B_1 & \equiv & x \geq 6 \end{array}$$

We verify:

$$\frac{x = 2h \qquad h \geq 3}{x \geq 6}$$

Remarks

Valid pairs (A_1, B_1) are obtained, e.g.,

- by substituting logical variables:

$$\frac{\{x = l\}\ \texttt{double();}\ \{x = 2l\}}{\{x = l - 1\}\ \texttt{double();}\ \{x = 2(l - 1)\}}$$

- by adding a condition C to the logical variables:

$$\frac{\{x = l\}\ \texttt{double();}\ \{x = 2l\}}{\{x = l \wedge l > 0\}\ \texttt{double();}\ \{x = 2l \wedge l > 0\}}$$

Remarks (cont.)

Valid pairs (A_1, B_1) are also obtained,

- if the pre-condition is strengthened or the post-condition weakened:

$$\frac{\{x = l\}\ \texttt{double();}\ \{x = 2l\}}{\{x > 0 \wedge x = l\}\ \texttt{double();}\ \{x = 2l\}}$$

$$\frac{\{x = l\}\ \texttt{double();}\ \{x = 2l\}}{\{x = l\}\ \texttt{double();}\ \{x = 2l \vee x = -1\}}$$

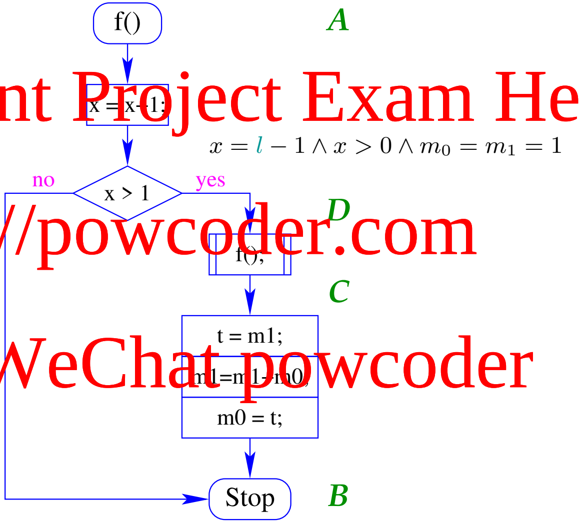
Application to Fibonacci

Our goal is to prove: $\{D\}$ `f();` $\{C\}$

$$
\begin{aligned}
A &\equiv x > 1 \land l = x \land m_0 = m_1 = 1 \\
A[(l-1)/l] &\equiv x > 1 \land l - 1 = x \land m_0 = m_1 = 1 \\
&\equiv D
\end{aligned}
$$

$$
\begin{aligned}
B &\equiv l > 1 \land m_1 \leq 2^{l} \land m_0 \leq 2^{l-1} \\
B[(l-1)/l] &\equiv l - 1 > 1 \land m_1 \leq 2^{l-1} \land m_0 \leq 2^{l-2} \\
&\equiv C
\end{aligned}
$$

Orientation



For the conditional, we verify:

$$
\begin{aligned}
\mathbf{WP}[\![\texttt{x>1}]\!]\ (B, D) \quad &\equiv \quad (x \le 1 \land l > 1 \land m_1 \le 2^{l} \land m_0 \le 2^{l-1}) \ \lor \\
& \quad (x > 1 \land x = l - 1 \land m_1 = m_0 = 1) \\
&\Leftarrow \quad x > 0 \land x = l - 1 \land m_0 = m_1 = 1
\end{aligned}
$$

1.7 Procedures with Local Variables

- Procedures `f()` modify global variables.
- The values of local variables of the caller before and after the call remain unchanged.

Example

```
{int y = 17; double(); write(y);}
```


Before and after the call of `double()` we have: $y = 17$.

- The values of local variables are automatically preserved, if the global hypothesis has the following properties:

  → The pre- and post-conditions: $\{A\}, \{B\}$ of procedures only speak about global variables !

  → The h are only used for global variables !!

- As a new specific instance of adaptation, we obtain:

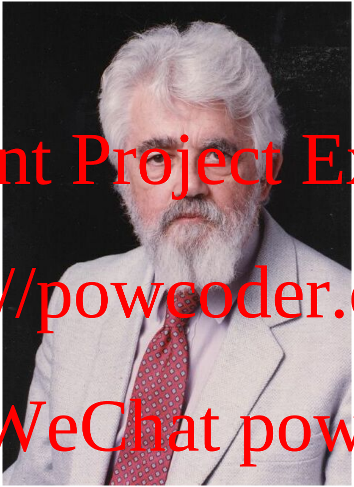
$$\frac{\{A\}\ \texttt{f();}\ \{B\}}{\{A \wedge C\}\ \texttt{f();}\ \{B \wedge C\}}$$

if C only speaks about logical variables or local variables of the caller.

Summary

- Every further language construct requires dedicated verification techniques.
- How to deal with dynamic data-structures, objects, classes, inheritance ?
- How to deal with concurrency, reactivity ??
- Do the presented methods allow to prove everything $\Longrightarrow$ completeness ?
- In how far can verification be automated ?
- How much help must be provided by the programmer and/or the verifier ?

Functional Programming



John McCarthy, Stanford

Robin Milner, Edinburgh

Xavier Leroy, INRIA, Paris

2 Basics

- Interpreter Environment
- Expressions
- Definitions of Values
- More Complex Datatypes
- Lists
- Definitions (cont.)
- User-defined Datatypes

2.1 The Interpreter Environment

The basic interpreter is called with `ocaml`.


```
seidl@linux:~> ocaml
        OCaml version 4.14.0
...
#
```


Definitions of variables, functions, ... can now immediately be inserted.
Alternatively, they can be read from a file:


```
# #use "Hello.ml";;
```

2.2 Expressions


```
# 3+4;;
- : int = 7
# 3+
  4;;
- : int = 7
#
```


→ At #, the interpreter is waiting for input.

→ The ;; causes evaluation of the given input.

→ The result is computed and returned together with its type.

Advantage: Individual functions can be tested without re-compilation !

Pre-defined Constants and Operators

Type	Constants: examples	Operators
int	0 3 -7	+ - * / mod
float	-3.0 7.0	+. -. *. /.
bool	true false	not || &&
string	"hello"	^
char	'a' 'b'	

Type	Comparison operators
int	`= <> < <= >= >`
float	`= <> < <= >= >`
bool	`= <> < <= >= >`
string	`= <> < <= >= >`
char	`= <> < <= >= >`


```
# -3.0 /. 4.0;;
- : float = -0.75
# "So" ^ " " ^ "it" ^ " " ^ "goes";;
- : string = "So it goes"
# 1 > 2 || not (2.0 < 1.0);;
- : bool = true
```

2.3 Definitions of Values

By means of `let`, a variable can be assigned a value.
The variable retains this value for ever!


```
# let seven = 3 + 4;;
val seven : int = 7
# seven;;
- : int = 7
```

Caveat: Variable names are start with a small letter !!!

Another definition of `seven` does not assign a new value to `seven`, but creates a new variable with the name `seven`.


```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```


The old variable is now hidden (but still there)!
Apparently, the new variable may even have a different type.

2.4 More Complex Datatypes

- Pairs


```
# (3 , 4);;
- : int * int = (3, 4)
# (1=2,"hello");;
- : bool * string = (false, "hello")
```


- Tuples


```
# (2, 3, 4, 5);;
- : int * int * int * int = (2, 3, 4, 5)
# ("hello", true, 3.14159);;
-: string * bool * float = ("hello", true, 3.14159)
```

Simultaneous Definition of Variables


```
# let (x,y) = (3,4.0);;
val x : int = 3
val y : float = 4.

# let (3,y) = (3,4.0);;
val y : float = 4.0
```

The latter use, though, will beforehand trigger the warning:

```
this pattern-matching is not exhaustive.
```

Records: Example

```
# type person = {given:string; sur:string; age:int};;
type person = { given : string; sur : string; age : int }

# let paul = { given="Paul"; sur="Meier"; age=24 };;
val paul : person = {given = "Paul"; sur = "Meier"; age = 24}

# let hans = { sur="kohl"; age=23; given="hans"};;
val hans : person = {given = "hans"; sur = "kohl"; age = 23}

# let hansi = {age=23; sur="kohl"; given="hans"};;
val hansi : person = {given = "hans"; sur = "kohl"; age = 23}

# hans = hansi;;
- : bool = true
```

Remark

- ... Records are tuples with named components whose ordering, therefore, is irrelevant.
- ... As a new type, a record must be introduced before its use by means of a `type` declaration.
- ... Type names and record components start with a small letter.

Access to Record Components

... via selection of components

```
# paul.given;;
- : string =  "Paul"
```

... with pattern matching

```
# let {given = x; sur = y; age = z} = paul;;
val x : string = "Paul"
val y : string = "Meier"
val z : int = 24
```


... and if we are not interested in everything:


```
# let {given = x; _} = paul;;
val x : string = "Paul"
```

Case Distinction: `match` and `if`


```
match n
with 0 -> "null"
   | 1 -> "one"
   | _ -> "uncountable!"

match e
with true  -> e1
   | false -> e2
```


The second example can also be written as

```
if e then e1 else e2
```

Watch out for redundant and incomplete matches!

```
# let n = 7;;
val n : int = 7

# match n with 0 -> "zero";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 5, -13).

# match n
    with 0 -> "zero"
       | 0 -> "one"
       | _ -> "uncountable!";;
Warning: this match case is unused.
- : string = "uncountable!
```

2.5 Lists

Lists are constructed by means of `[]` and `::`.

Short-cut: `[42; 0; 16]`


```
# let mt = [];;
val mt : 'a list = []

# let l1 = 1::mt;;
val l1 : int list = [1]

# let l = [1;2;3];;
val l : int list = [1; 2; 3]

# let l = 1::2::3::[];;
val l : int list = [1; 2; 3]
```

Caveat

All elements must have the same type:


```
# 1.0 :: 1 :: [];;
This expression has type int but is here used with type float
```


`alpha list` describes lists with elements of type `alpha`.

The type `'a` is a type variable:

`[]` denotes an empty list for arbitrary element types.

Pattern Matching on Lists


```
# match l
  with []    -> -1
     | x::xs -> x;;
-: int = 1
```

2.6 Definition of Functions


```
# let double x = 2 * x;;
val double : int -> int = <fun>

# (double 3, double (double 1));;
- : int * int = (6,4)
```


→ Behind the function name follow the parameters.

→ The function name is just a variable whose value is a function.

- Alternatively, we may introduce a variable whose value is a function.


```
# let double = fun x -> 2 * x;;
val double : int -> int = <fun>
```


- This function definition starts with `fun`, followed by the sequence of formal parameters.
- After `->` follows the specification of the return value.
- The variables from the left can be accessed on the right.

Caveat

Functions may additionally access the values of variables which have been visible at their point of definition:


```
# let factor = 2;;
val factor : int = 2

# let double x = factor * x;;
val double : int -> int = <fun>

# let factor = 4;;
val factor : int = 4

# double 3;;
- : int = 6
```

Caveat

A function is a value:


```
# double;;
- : int -> int = <fun>
```

Recursive Functions

A function is recursive, if it calls itself (directly or indirectly).


```
# let rec fac n = if n < 2 then 1 else n * fac (n - 1);;
val fac : int -> int = <fun>

# let rec fib = fun x -> if x <= 1 then 1
                         else fib (x - 1) + fib (x - 2);;
val fib : int -> int = <fun>
```

For that purpose, OCaml offers the keyword `rec`.

If functions call themselves indirectly via other other functions, they are called mutually recursive


```
# let rec even n = if n=0 then "even" else odd (n-1)
    and odd  n = if n=0 then "odd"  else even (n-1);;
val even : int -> string = <fun>
val odd : int -> string = <fun>
```


We combine their definitions by means of the keyword `and`

Definition by Case Distinction

```
# let rec length = fun l -> match l
    with [] -> 0
    | x::xs -> 1 + length xs;;
val length : 'a list -> int = <fun>
# length [1; 2; 3];;
- : int = 3
```


… can be shorter written as

```
# let rec length = function
    | [] -> 0
    | x::xs -> 1 + len xs;;
val length : 'a list -> int = <fun>
# length [1; 2; 3];;
- : int = 3
```

Case distinction for several arguments

```
# let rec app l y = match l with
    | [] -> y
    | x::xs -> x :: app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1; 2] [3; 4];;
- : int list = [1; 2; 3; 4]
```


... can also be written as

```
# let rec app = function [] -> fun y -> y
                      | x::xs -> fun y -> x::app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1; 2] [3; 4];;
- : int list = [1; 2; 3; 4]
```

Local Definitions

Definitions introduced by `let` may occur locally:

```
# let x = 5
    in let sq = x * x
    in sq + sq;;
- : int = 50

# let facit n = let rec
    iter m yet = if m > n then yet
                 else iter (m + 1) (m * yet)
    in iter 2 1;;
val facit : int -> int = <fun>
```

2.7 User-defined Datatypes

Example: playing cards

How to specify color and value of a card?

First Idea: pairs of strings and numbers, e.g.,

`("diamonds", 10)` $\equiv$ diamonds ten
`("clubs", 11)` $\equiv$ clubs jack
`("gras", 14)` $\equiv$ gras ace

Disadvantages

- Testing of the color requires a comparison of strings
  $\longrightarrow$ inefficient!
- Representation of Jack as `11` is not intuitive
  $\longrightarrow$ incomprehensible program!
- Which card represents the pair `("clubs", 9)`?
  (typos are not recognized by the compiler)

Better: Enumeration types of OCaml.

Example: Playing cards

2. Idea: Enumeration Types


```
# type color = Diamonds | Hearts | Gras | Clubs;;
type color = Diamonds | Hearts | Gras | Clubs
# type value = Seven | Eight | Nine | Jack | Queen | King |
               Ten | Ace;;
type value = Seven | Eight | Nine | Jack | Queen | King |
               Ten | Ace
# Clubs;;
- : color = Clubs
# let gras_jack = (Gras,Jack);;
val gras_jack : color * value = (Gras,Jack)
```

Advantages

→ The representation is intuitive.

→ Typing errors are recognized:

```
# (Culbs,Nine);;
Unbound constructor Culbs
```

→ The internal representation is efficient.

Remark

→ By `type`, a new type is defined.

→ The alternatives are called constructors and are separated by `|`.

→ Every constructor starts with a capital letter and is uniquely assigned to a type.

Enumeration Types (cont.)

Constructors can be compared:

```
# Clubs < Diamonds;;
- : bool = false;;
# Clubs > Diamonds;;
- : bool = true;;
```


Pattern Matching on constructors:

```
# let is_trump = function
  | (Hearts,_)  -> true
  | (_,Jack)   -> true
  | (_,Queen)   -> true
  | (_,_)      -> false

val is_trump : color * value -> bool = <fun>
```

By that, e.g.,

```
# is_trump (Gras,Jack);;
- : bool = true
# is_trump (Clubs,Nine);;
- : bool = false
```

Another useful function:

```
# let string_of_color = function
    | Diamonds -> "Diamonds"
    | Hearts   -> "Hearts"
    | Gras     -> "Gras"
    | Clubs -> "Clubs";;
val string_of_color : color -> string = <fun>
```


Remark

The function `string_of_color` returns for a given color the corresponding string in constant time (the compiler, hopefully, uses jump tables).

Now, OCaml can (almost) play cards:


```
# let takes c1 c2 = match (c1,c2) with
    | ((f1,Queen),(f2,Queen))   -> f1 > f2
    | ((_,Queen),_)             -> true
    | (_,(_,Queen))             -> false
    | ((f1,Jack),(f2,Jack))     -> f1 > f2
    | ((_,Jack),_)              -> true
    | (_,(_,Jack))              -> false
    | ((Hearts,w1),(Hearts,w2)) -> w1 > w2
    | ((Hearts,_),_)            -> true
    | (_,(Hearts,_))            -> false
    | ((f1,w1),(f2,w2))         -> if f1=f2 then w1 > w2
                                           else false;;
```

```
  ...
# let take card2 card1 =
    if takes card2 card1 then card2 else card1 ;;

# let trick card1 card2 card3 card4 =
    take card4 (take card3 (take card2 card1));;

# trick (Gras,Ace) (Gras,Nine) (Hearts,Ten) (Clubs,Jack);;
- : color * value = (Clubs,Jack)
# trick (Clubs,Eight) (Clubs,King) (Gras,Ten)
      (Clubs,Nine);;
- : color * value = (Clubs,King)
```

Sum Types

Sum types generalize of enumeration types in that constructors now may have arguments.

Example: Optional Values

```
type 'a option = None | Some of 'a
let is_some x = match x with
  | Some _ -> true
  | None -> false
...
```

```
...
let get x = match x with
    | Some y -> y
let value x a = match x with
    | Some y -> y
    | None -> a
let map f x = match x with
    | Some y -> Some (f y)
    | None -> None
let join a = match a with
    | Some a' -> a'
    | None -> None
```

Option is a module, which collects useful functions and values for option.
A constructor defined inside type t = Con of <type> | ...
has functionality Con : <type> -> t — must, however, always occur applied ...


```
# Some;;
The constructor Some expects 1 argument(s),
but is here applied to 0 argument(s)

# None;;
- : 'a option = None

# Some 10;;
- : int option = Some 10

# let a = Some "Hello!";;
val a : string option = Some "Hello!"
```

The type option is polymorphic – which means that it can be constructed for any type 'a, in particular int or string.

Polymorphic types with parameters 'a, 'b, 'c are then introduced by
`type ('a,'b,'c) t = ...`

The option type is useful for defining partial functions

```
let rec get_value a l = match l with
  | []            -> None
  | (b, z)::rest  -> if a = b then Some z
                     else get_value a rest
```

Datatypes can be recursive:

```
type sequence = End | Next of (int * sequence)

# Next (1, Next (2, End));;
- : sequence = Next (1, Next (2, End))
```


Note the similarity to lists.

A corresponding polymorphic type could be


```
type 'a sequence = End | Next of ('a * 'a sequence)

# Next (1, Next (2, End));;
- : int sequence = Next (1, Next (2, End))
```

Recursive datatypes lead to recursive functions:

```
# let rec nth n s = match (n, s) with
    | (_, End)               -> None
    | (0, Next (x, _))        -> Some x
    | (n, Next (_, rest)) -> nth (n-1) rest;;
val nth : int -> 'a sequence -> 'a option = <fun>

# nth 4 (Next (1, Next (2, End)));;
- : int = None

# nth 2 (Next (1, Next(2, Next (5, Next (17, End)))));;
- : int = Some 5
```

Another Example


```
# let rec down = function
    | 0 -> End
    | n -> Next (n, down (n-1));;
val down : int -> int sequence = <fun>

# down 3;;
- : int sequence = Next (3, Next (2, Next (1, End)));;

# down (-1);;
Stack overflow during evaluation (looping recursion?).
```

3 A closer Look at Functions

- Tail Calls
- Higher-order Functions
  - → Currying
  - → Partial Application

- Polymorphic Functions
- Polymorphic Datatypes
- Anonymous Functions

3.1 Tail Calls

A tail call in the body e of a function is a call whose value provides the value of e ...


```
let f x = x + 5

let g y = let z = 7
  in if y > 5 then f (-y)
     else z + f y
```


The first call is tail, the second is not.

$\Longrightarrow$ From a tail call, we need not return to the calling function.

$\Longrightarrow$ The stack space of the calling function can immediately be recycled !!!

A recursive function f is called tail recursive, if all (direct or indirect) calls to f and all functions mutually recursive with f in the right-hand sides of any of these functions are tail calls.

Examples


```
let fac x = let rec facit n acc =
  if n <= 1 then acc
  else facit (n - 1) (n * acc)
in facit x 1

let rec loop x = if x < 2 then x
  else if x mod 2 = 0 then loop (x / 2)
  else loop (3 * x + 1)
```

Discussion

- Tail-recursive functions can be executed as efficiently as loops in imperative languages.
- The intermediate results are handed from one recursive call to the next in accumulating parameters.
- From that, a stopping rule computes the result.
- Tail-recursive functions are particularly popular for list processing ...

Reversing a List – Version 1

```
let rec rev list = match list
    with [] -> []
       | x::xs -> app (rev xs) [x]
```


`rev [0;...;n-1]` calls function `app` with


```
[]
[n-1]
[n-1; n-2]
...
[n-1; ...; 1]
```


as first argument $\Longrightarrow$ quadratic running-time!

Reversing a List – Version 2


```
let rev list =
    let rec r2 a l =
      match l
      with [] -> a
         | x::xs -> r2 (x::a) xs
    in r2 [] list
```

$\Longrightarrow$ The local function r2 is tail recursive !

$\Longrightarrow$ it runs in linear running-time !!

3.2 Higher Order Functions

Consider the two functions


```
let f (a, b) = a + b + 1

let g a b    = a + b + 1
```


At first sight, `f` and `g` differ only in the syntax. But they also differ in their types:


```
# f;;
- : int * int -> int = <fun>

# g;;
- : int -> int -> int = <fun>
```

- Function `f` has a single argument, namely, the pair `(a,b)`. The return value is given by `a+b+1`.
- Function `g` has the one argument `a` of type `int`. The result of application to `a` is a function that, when applied to the other argument `b`, returns the result `a+b+1` :


```
# f (3, 5);;
- : int = 9

# let g1 = g 3;;
val g1 : int -> int = <fun>

# g1 5;;
- : int = 9
```

Haskell B. Curry, 1900–1982

In honor of its inventor Haskell B. Curry, this principle is called Currying.

→ g is called a higher order function, because its result is again a function.

→ The application of g to a single argument is called partial, because the result takes another argument, before the body is evaluated.

The argument of a function can again be a function.

```
# let curry f a b = f (a,b);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
...
```

```
...

# let plus (x,y) = x+y;;
val plus : int * int -> int = <fun>

# curry plus;;
- : int -> int -> int = <fun>

# let plus2 = curry plus 2;;
val plus2 : int -> int = <fun>

# let plus3 = curry plus 3;;
val plus3 : int -> int = <fun>

# plus2 (plus3 4);;
- : int = 9
```

3.3 Some List Functions


```
let rec map f = function
    [] -> []
  | x::xs -> f x :: map f xs

let rec fold_left f a = function
    [] -> a
  | x::xs -> fold_left f (f a x) xs

let rec fold_right f = function
    [] -> fun b -> b
  | x::xs -> fun b -> f x (fold_right f xs b)
```

```
let rec find_opt f = function
    [] -> None
  | x::xs -> if f x then Some x
    else find_opt f xs
```


Remarks

→ These functions abstract from the behavior of the function `f`. They specify the recursion according the list structure — independently of the elements of the list.

→ Therefore, such functions are sometimes called recursion schemes or (list) functionals.

→ List functionals are independent of the element type of the list. That type must only be known to the function `f`.

→ Functions which operate on equally structured data of various type, are called polymorphic.

3.4 Polymorphic Functions

The OCaml system infers the following types for the given functionals:


```
map : ('a -> 'b) -> 'a list -> 'b list

fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

find_opt : ('a -> bool) -> 'a list -> 'a option
```


→ 'a and 'b are type variables. They can be instantiated by any type (but each occurrence with the same type).

→ By partial application, some of the type variables may be instantiated:


```
# string_of_int;;
val : int -> string = <fun>

# map string_of_int;;
- : int list -> string list = <fun>

# fold_left (+);;
val it : int -> int list -> int = <fun>
```

→ If a functional is applied to a function that is itself polymorphic, the result may again be polymorphic:


```
# let cons_r xs x = x::xs;;
val cons_r : 'a list -> 'a -> 'a list = <fun>

# let rev l = fold_left cons_r [] l;;
val rev : 'a list -> 'a list = <fun>

# rev [1; 2; 3];;
- : int list = [3; 2; 1]

# rev [true; false; false];;
- : bool list = [false; false; true]
```

Some of the Simplest Polymorphic Functions

```
let compose f g x  = f (g x)
let twice f x      = f (f x)
let rec iter f g x = if g x then x else iter f g (f x);;

val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
val iter : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a = <fun>

# compose not not;;
- : bool -> bool = <fun>

# compose not not true;;
- : bool = true;;

# compose Char.chr plus2 65;;
- : char = 'C'
```

3.5 Polymorphic Datatypes

User-defined datatypes may be polymorphic as well:

```
type 'a tree = Leaf of 'a | Node of ('a tree * 'a tree)
```


→ `tree` is called *type constructor*, because it allows to create a new type from another type, namely its parameter `'a`.

→ In the right-hand side, only those type variables may occur, which have been listed on the left.

→ The application of constructors to data may instantiate type variables:

```
# Leaf 1;;
- : int tree = Leaf 1

# Node (Leaf ('a', true), Leaf ('b', false));;
- : (char * bool) tree = Node (Leaf ('a', true), Leaf ('b', false))
```

Functions for polymorphic datatypes are, typically, again polymorphic ...

```
let rec size = function
  | Leaf _      -> 1
  | Node(t,t') -> size t + size t'

let rec flatten = function
  | Leaf x      -> [x]
  | Node(t,t') -> flatten t @ flatten t'

let flatten1 t = let rec doit t xs = match t with
    | Leaf x      -> x :: xs
    | Node(t,t') -> let xs = doit t' xs
      in doit t xs
  in doit t []

...
```

```
...

val size : 'a tree -> int = <fun>
val flatten : 'a tree -> 'a list = <fun>
val flatten1 : 'a tree -> 'a list = <fun>

# let t = Node (Node (Leaf 1, Leaf 5), Leaf 3);;
val t : int tree = Node (Node (Leaf 1, Leaf 5), Leaf 3)

# size t;;
- : int = 3

# flatten t;;
val : int list = [1;5;3]

# flatten1 t;;
val : int list = [1;5;3]
```

3.6 Application: Queues

Wanted

Datastructure 'a queue which supports the operations


```
enqueue : 'a -> 'a queue -> 'a queue

dequeue : 'a queue  -> 'a option * 'a queue

is_empty : 'a queue -> bool

queue_of_list : 'a list -> 'a queue

list_of_queue : 'a queue -> 'a list
```

First Idea

- Represent the queue by a list:

```
type 'a queue = 'a list
```


The functions `is_empty`, `queue_of_list`, `list_of_queue` then are trivial.

- Extraction means access to the topmost element:


```
let dequeue = function
      []      -> (None, [])
    | x :: xs -> (Some x, xs)
```


- Insertion means append:

```
let enqueue x xs = xs @ [x]
```

Discussion

- The operator `@` concatenates two lists.
- The implementation is very simple.
- Extraction is cheap.
- Insertion, however, requires as many calls of `@` as the queue has elements.
- Can that be improved upon ??

Second Idea

- Represent the queue as two lists !!!


```
type 'a queue = Queue of 'a list * 'a list

let is_empty = function
                 Queue ([],[]) -> true
               | _ -> false

let queue_of_list list = Queue (list,[])

let list_of_queue = function
                 Queue (first,[]) -> first
               | Queue (first,last) ->
                       first @ List.rev last
```


- The second list represents the tail of the list and therefore in reverse ordering ...

Second Idea (cont.)

- Insertion is in the second list:


```
let enqueue x (Queue (first, last)) =
                  Queue (first, x::last)
```

- Extracted are elements always from the first list:
  Only if that is empty, the second list is consulted ...


```
let dequeue = function
    Queue ([],last) -> (match List.rev last
                        with [] -> (None, Queue ([],[]))
                        | x::xs -> (Some x, Queue (xs,[])))
  | Queue (x::xs,last) -> (Some x, Queue (xs,last))
```

Discussion

- Now, insertion is cheap!
- Extraction, however, can be as expensive as the number of elements in the second list ...
- Averaged over the number of insertions, however, the extra costs are only constant !!!

$\Longrightarrow$ amortized cost analysis

3.7 Anonymous Functions

As we have seen, functions are data. Data, e.g., [1;2;3] can be used without naming them. This is also possible for functions:

```
# fun x y z -> x + y + z;;
- : int -> int -> int -> int = <fun>
```


- `fun` initiates an abstraction.
  This notion originates in the λ-calculus.
- `->` has the effect of `=` in function definitions.
- Recursive functions cannot be defined in this way, as the recurrent occurrences in their bodies require names for reference.

Alonzo Church, 1903–1995

- Pattern matching can be used by applying `match ... with` for the corresponding argument.
- In case of a single argument, `function` can be considered ...


```
# function None -> 0
           | Some x -> x * x + 1;;
- : int option -> int = <fun>
```

Anonymous functions are convenient if they are used just once in a program. Often, they occur as arguments to functionals:


```
# map (fun x -> x * x) [1; 2; 3];;
- : int list = [1; 4; 9]
```

Often, they are also used for returning functions as result:


```
# let make_undefined () = fun x -> None;;
val make_undefined : unit -> 'a -> 'b option = <fun>

# let def_one (x,y) = fun x' -> if x = x' then Some y else None;;
val def_one : 'a * 'b -> 'a -> 'b option = <fun>
```

4 A Larger Application: Balanced Trees

Recap: Sorted Array

2	3	5	7	11	13	17

Properties

- Sorting algorithms allow to initialize with $\approx n \cdot \log(n)$ many comparisons.

  // $n ==$ size of the array

- Binary search allows to search for elements with $\approx \log(n)$ many comparisons.
- Arrays neither support insertion nor deletion of elements.

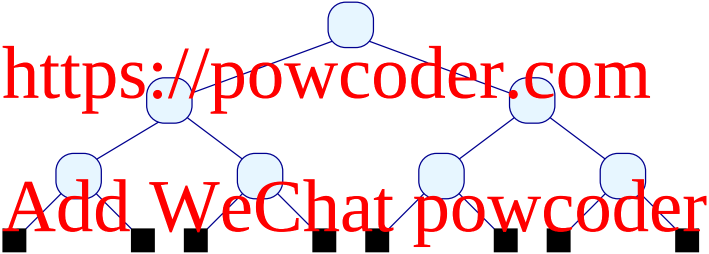
Wanted:

Datastructure 'a d which allows to maintain a dynamic sorted sequence of elements, i.e., which supports the operations


```
insert :      'a -> 'a d -> 'a d
delete :      'a -> 'a d -> 'a d
extract_min :'a d -> 'a option * 'a d
extract_max :'a d -> 'a option * 'a d
extract :     'a * 'a -> 'a d -> 'a list * 'a d
list_of_d :   'a d -> 'a list
d_of_list :   'a list -> 'a d
```

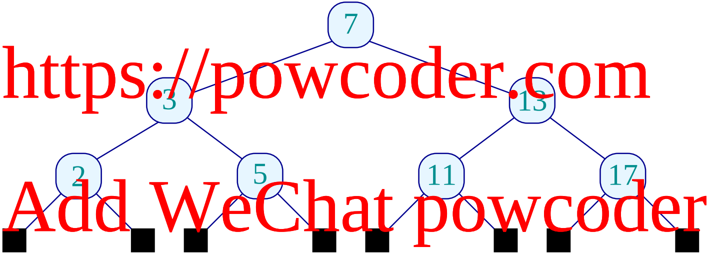
First Idea

Use balanced trees ...



First Idea

Use balanced trees ...



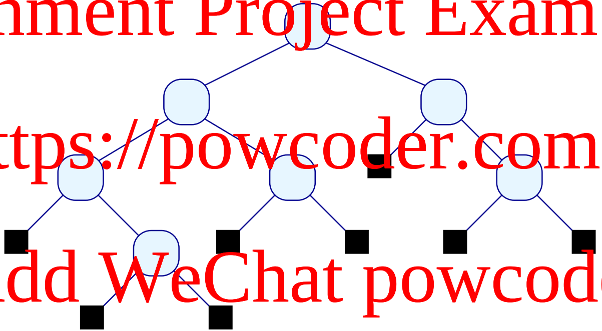
Discussion

- Data are stored at internal nodes!
- A binary tree with n leaves has $n-1$ internal nodes.
- In order to search for an element, we must compare with all elements along a path ...
- The depth of a tree is the maximal number of internal nodes on a path from the root to a leaf.
- A complete balanced binary tree with $n = 2^k$ leaves has depth $k = \log(n)$.
- How do we insert further elements ???
- How do we delete elements ???

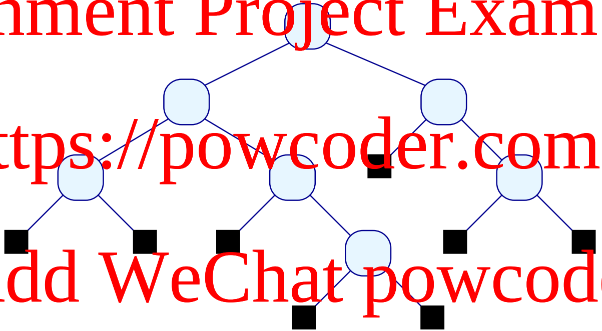
Second Idea

- Instead of balanced trees, we use almost balanced trees ...
- At each node, the depth of the left and right subtrees should be almost equal !
- An AVL tree is a binary tree where the depths of left and right subtrees at each internal node differs at most by 1 ...

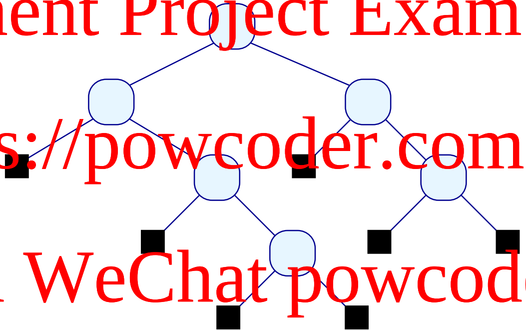
An AVL Tree



An AVL Tree



Not an AVL Tree



G.M. Adelson-Velskij, 1922

E.M. Landis, Moskau, 1921-1997

We prove:

(1) Each AVL tree of depth $k > 0$ has at least

$$\text{fib}(k) \geq A^{k-1}$$

nodes where $A = \frac{\sqrt{5}+1}{2}$ // golden cut

We calculate:

(1) Each AVL tree of depth $k > 0$ has at least

$$\text{fib}(k) \geq A^{k-1}$$

nodes where $A = \frac{\sqrt{5}+1}{2}$ // golden cut

(2) Every AVL tree with $n > 0$ internal nodes has depth at most

$$\frac{1}{\log(A)} \cdot \log(n) + 1$$

Proof: We only prove (1)

Let $N(k)$ denote the minimal number of internal nodes of an AVL tree of depth k.
Induction on the number $k > 0$ ...

$\boxed{k = 1 :}$ $\quad N(1) = 1 = \mathsf{fib}(1) = A^0$

$\boxed{k = 2 :}$ $\quad N(2) = 2 = \mathsf{fib}(2) \geq A^1$

$\boxed{k > 2 :}$ Assume that the assertion holds for $k-1$ and $k-2$ ...

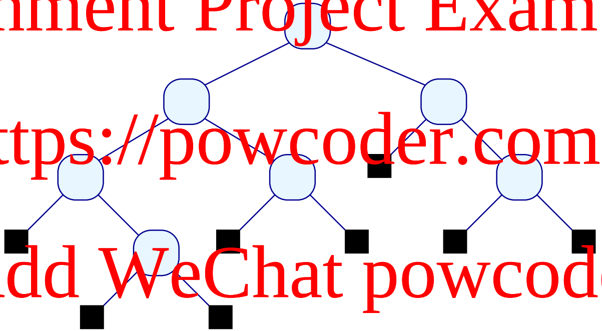
$$\Longrightarrow \quad \begin{aligned} N(k) &= N(k-1) + N(k-2) + 1 \\ &\geq \mathsf{fib}(k-1) + \mathsf{fib}(k-2) \\ &= \mathsf{fib}(k) \end{aligned}$$

$$\begin{aligned} \mathsf{fib}(k) &= \mathsf{fib}(k-1) + \mathsf{fib}(k-2) \\ &\geq A^{k-2} + A^{k-3} \\ &= A^{k-3} \cdot (A+1) \\ &= A^{k-3} \cdot A^2 \\ &= A^{k-1} \end{aligned}$$

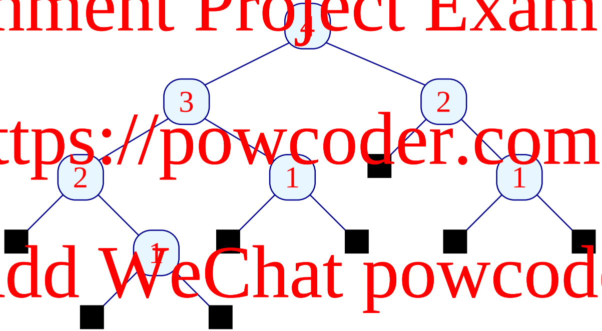
Second Idea (cont.)

- If another element is inserted, the AVL property may get lost !
- If some element is deleted, the AVL property may get lost !
- Then the tree must be re-structured so that the AVL property is re-established ...
- For that, we require for each node the depths of the left and right subtrees, respectively ...

Representation



Representation

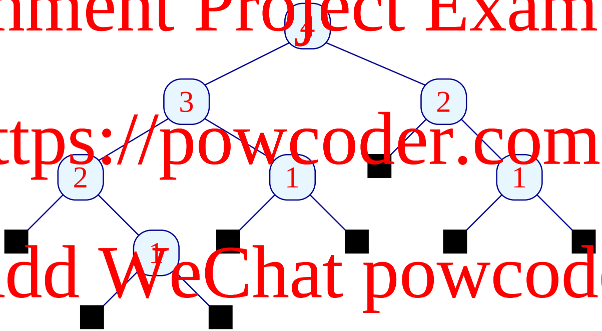


Third Idea

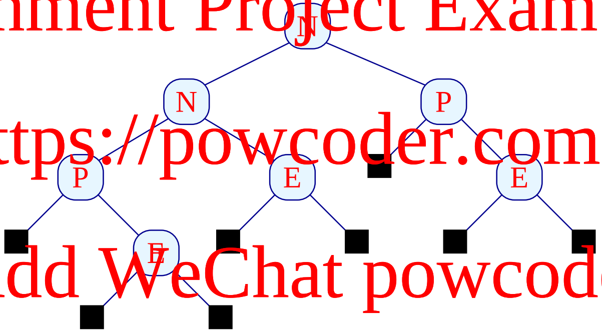
- Instead of the absolute depth, we store at each node only whether the difference in depth of the two subtrees is negative, positive or equal to zero !!!
- As datatype, we therefore define


```
type 'a avl = Null
            | Neg of 'a avl * 'a * 'a avl
            | Pos of 'a avl * 'a * 'a avl
            | Eq  of 'a avl * 'a * 'a avl
```

Representation



Representation



Insertion

- If the tree is a leaf, i.e., empty, an internal node is created with two new leaves.
- If the tree in non-empty, the new value is compared with the value at the root.
  - → If it is larger, it is inserted to the right.
  - → Otherwise, it is inserted to the left.

- Caveat: Insertion may increase the depth and thus may destroy the AVL property !
- That must be subsequently dealt with ...

```
let rec insert x avl = match avl
    with Null                  -> (Eq (Null,x,Null), true)
       | Eq (left,y,right) -> if x < y then
                                  let (left,inc) = insert x left
                                  in if inc then (Neg (left,y,right), true)
                                     else        (Eq (left,y,right), false)
                              else let (right,inc) = insert x right
                                  in if inc then (Pos (left,y,right), true)
                                     else        (Eq (left,y,right), false)
       ...
```


- Besides the new AVL tree, the function `insert` also returns the information whether the depth of the result has increased.
- If the depth is not increased, the marking of the root need not be changed.

```
| Neg (left,y,right) -> if x < y then
     let (left,inc) = insert x left
     in if inc then let (avl,_) = rotateRight (left,y,right)
                    in (avl,false)
        else        (Neg (left,y,right), false)
  else let (right,inc) = insert x right
     in if inc then (Eq (left,y,right), false)
        else        (Neg (left,y,right), false)
| Pos (left,y,right) -> if x < y then
     let (left,inc) = insert x left
     in if inc then (Eq (left,y,right), false)
        else        (Pos (left,y,right), false)
  else let (right,inc) = insert x right
     in if inc then let (avl,_) = rotateLeft (left,y,right)
                    in (avl,false)
        else     (Pos (left,y,right), false);;
```

Comments

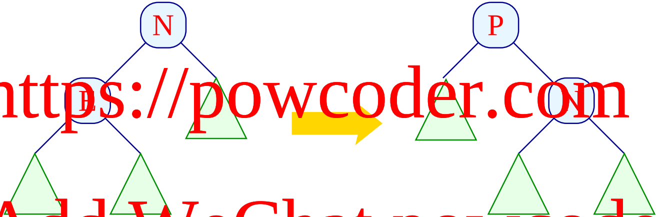
- Insertion into the less deep subtree never increases the total depth.

  The depths of the two subtrees, though, may become equal.

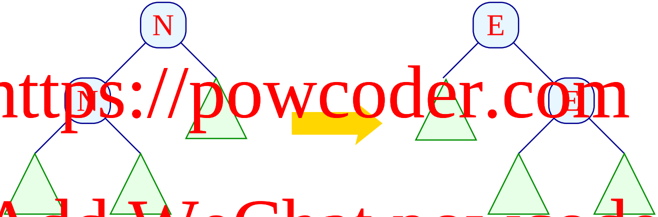
- Insertion into the deeper subtree may increase the difference in depth to 2.

  then the node at the root must be rotated in order to decrease the difference ...

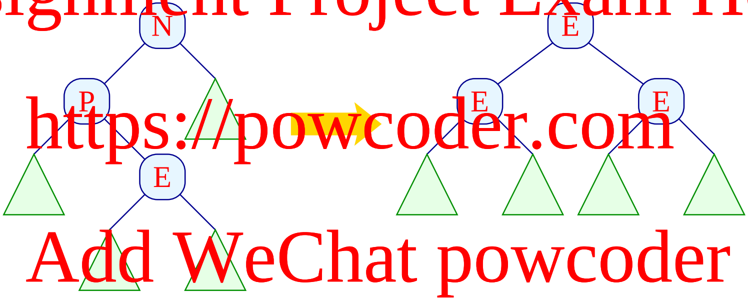
rotateRight



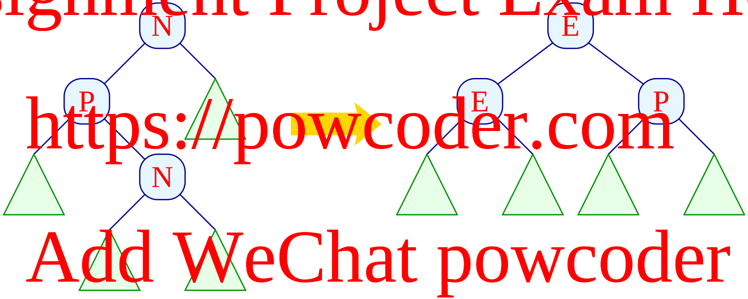
`rotateRight`



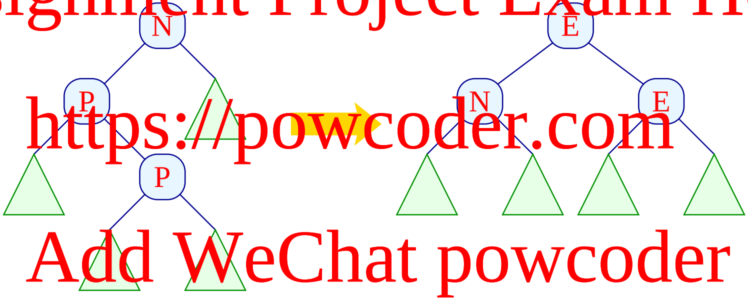
`rotateRight`



`rotateRight`



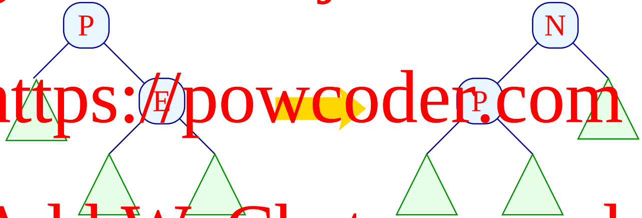
`rotateRight`



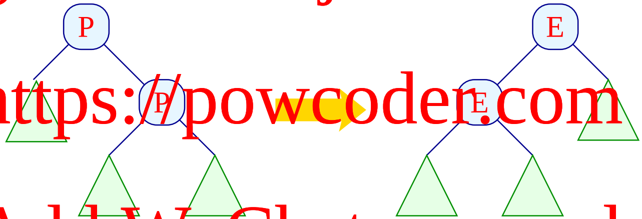
```
let rotateRight (left, y, right) = match left
    with Eq (l1,y1,r1) -> (Pos (l1, y1, Neg (r1,y,right)), false)
      | Neg (l1,y1,r1) -> (Eq (l1, y1, Eq (r1,y,right)), true)
      | Pos (l1, y1, Eq (l2,y2,r2)) ->
            (Eq (Eq (l1,y1,l2), y2, Eq (r2,y,right)), true)
      | Pos (l1, y1, Neg (l2,y2,r2)) ->
            (Eq (Eq (l1,y1,l2), y2, Pos (r2,y,right)), true)
      | Pos (l1, y1, Pos (l2,y2,r2)) ->
            (Eq (Neg (l1,y1,l2), y2, Eq (r2,y,right)), true)
```


- The extra bit now indicates whether the depth of the tree after rotation has decreased ...
- This is not the case only when the deeper subtree is of the form `Eq (...)` — which does never occur here.

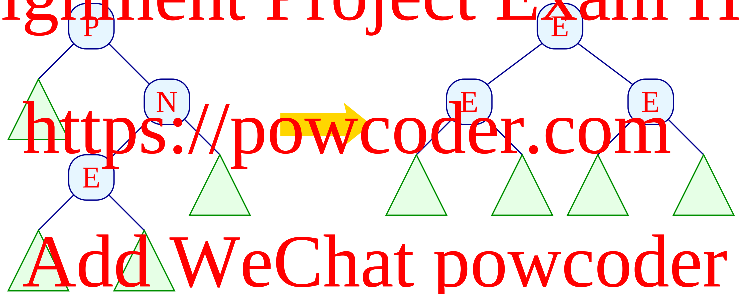
`rotateLeft`



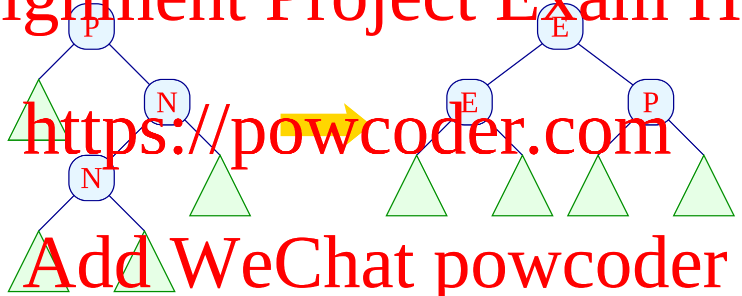
`rotateLeft`



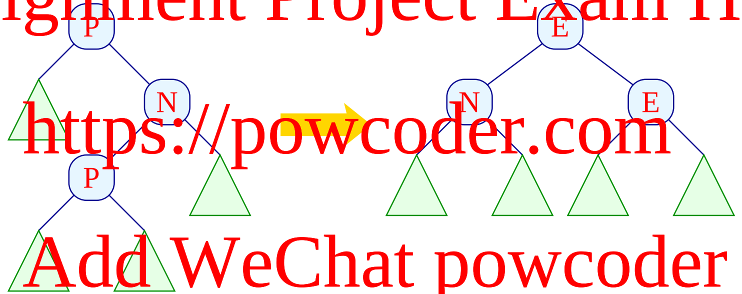
`rotateLeft`



`rotateLeft`



`rotateLeft`



```
let rotateLeft (left, y, right) = match right
    with Eq (l1,y1,r1) -> (Neg (Pos (left,y,l1), y1, r1), false)
       | Pos (l1,y1,r1) -> (Eq  (Eq  (left,y,l1), y1, r1), true)
       | Neg (Eq (l1,y1,r1), y2 ,r2) ->
                (Eq (Eq (left,y,l1),y1,  Eq (r1,y2,r2)), true)
       | Neg (Neg (l1,y1,r1), y2 ,r2) ->
                (Eq (Eq (left,y,l1),y1, Pos (r1,y2,r2)), true)
       | Neg (Pos (l1,y1,r1), y2 ,r2) ->
                (Eq (Neg (left,y,l1),y1, Eq (r1,y2,r2)), true)
```


- rotateLeft is analogous to rotateRight — only with the roles of Pos and Neg exchanged.
- Again, the depth shrinks almost always.

Discussion

- Insertion requires at most as many calls of `insert` as the depth of the tree.
- After returning from a call for a subtree, at most three nodes must be re-arranged.

- The total effort therefore is bounded by a constand multiple to $\log(n)$.
- In general, though, we are not interested in the extra bit at every call. Therefore, we define:


```
let insert x tree = let (tree,_) = insert x tree
                    in   tree
```

Extraction of the Minimum

- The minimum occurs at the leftmost internal node.
- It is found by recursively visiting the left subtree.
  The leftmost node is found when the left subtree equals `Null`.
- Removal of a leaf may reduce the depth and thus may destroy the AVL property.
- After each call, the tree must be locally repaired ...

```
let rec extract_min avl = match avl
    with Null              -> (None, Null, false)
       | Eq (Null,y,right) -> (Some y, right, true)
       | Eq (left,y,right) -> let (first,left,dec) = extract_min left
                   in if dec then (first, Pos (left,y,right), false)
                      else        (first,  Eq (left,y,right), false)
       | Neg (left,y,right) -> let (first,left,dec) = extract_min left
                    in if dec then (first, Eq (left,y,right), true)
                       else        (first, Neg (left,y,right), false)
       | Pos (Null,y,right) -> (Some y, right, true)
       | Pos (left,y,right) -> let (first,left,dec) = extract_min left
                in if dec then let (avl, b) = rotateLeft (left,y,right)
                               in (first,avl,b)
                   else        (first, Pos (left,y,right), false)
```

Discussion

- Rotation is only required when extracting from a tree of the form `Pos[illegible](...)` and the depth of the left subtree is decreased.

- Altogether, the number of recursive calls is bounded by the depth. For every call, at most three nodes are re-arranged.
- Therefore, the total effort is bounded by a constant multiple of $\log(n)$.
- Functions for maximum or last element from an interval are constructed analogously ...

5 Practical Features of OCaml

- Exceptions
- Input and Output as Side-effects
- Sequences

5.1 Exceptions

In case of a runtime error, e.g., division by zero, the OCaml system generates an exception.


```
# 1 / 0;;
Exception: Division_by_zero.

# List.tl (List.tl [1]);;
Exception: Failure "tl".

# Char.chr 300;;
Exception: Invalid_argument "Char.chr".
```

Here, the exceptions `Division_by_zero`, `Failure "tl"` and `Invalid_argument "Char.chr"` are generated.

Another reason for an exception is an incomplete match:


```
# match 1+1 with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 2, -9).
```


In this case, the exception `Match_failure ("", 2, -9)` is generated.

Pre-defined Constructors for Exceptions

`Division_by_zero`	division by 0
`Invalid_argument of string`	wrong usage
`Failure of string`	general error
`Match_failure of string * int * int`	incomplete match
`Not_found`	not found
`Out_of_memory`	memory exhausted
`End_of_file`	end of file
`Exit`	for the user ...

An exception is a first class citizen, i.e., a value from a datatype `exn` ...

```
# Division_by_zero;;
- : exn = Division_by_zero

# Failure "complete nonsense!";;
- : exn = Failure "complete nonsense!"
```


Own exceptions are introduced by extending the datatype exn ...


```
# exception Hell;;
exception Hell

# Hell;;
- : exn = Hell
```

```
# Division_by_zero;;
- : exn = Division_by_zero

# Failure "complete nonsense!";;
- : exn = Failure "complete nonsense!"
```


Own exceptions are introduced by extending the datatype exn ...


```
# exception Hell of string;;
exception Hell of string

# Hell "damn!";;
- : exn = Hell "damn!"
```

Handling of Exceptions

As in Java, exceptions can be raised and handled:


```
# let divide (n,m) = try Some (n / m)
     with Division_by_zero -> None;;

# divide (10,3);;
- : int option = Some 3
# divide (10,0);;
- : int option = None
```


In this way, the `member` function can, e.g., be re-defined as

```
let rec member x l = try if x = List.hd l then true
                         else member x (List.tl l)
  with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false
```


Following the keyword `with`, the exception value can be inspected by means of pattern matching for the exception datatype `exn` :


```
  try <exn>
  with <pat1> -> <exp1> | ... | <patN> -> <expN>
```

⟹ several exceptions can be caught (and thus handled) at the same time.

The programmer may trigger exceptions on his/her own
by means of the keyword `raise` ...


```
# 1 + (2 / 0);;
Exception: Division_by_zero.

# 1 + raise Division_by_zero;;
Exception: Division_by_zero.
```


An exception is an error value which can replace any expression.

Handling of an exception, results in the evaluation of another expression (of the correct type) — or raises another exception.

Exception handling may occur at any sub-expression, arbitrarily nested:

```
# let f (x, y) = x / (y - 1);;
# let g (x, y) = try let n = try f (x,y)
                                  with Division_by_zero ->
                                       raise (Failure "Division by zero")
                     in string_of_int (n * n)
                 with Failure str -> "Error: " ^ str;;

# g (6, 1);;
- : string = "Error: Division by zero"

# g (6, 3);;
- : string = "9"
```

5.2 Textual Input and Output

- Reading from the input and writing to the output violates the paradigm of purely functional programming !

- These operations are therefore realized by means of side-effects, i.e., by means of functions whose return value is irrelevant (e.g., `unit`).
- During execution, though, the required operation is executed

$\Longrightarrow$ now, the ordering of the evaluation matters !!!

- Naturally, OCaml allows to write to standard output:

```
# print_string "Hello World!\n";;
Hello World!
- : unit = ()
```

- Analogously, there is a function: `read_line : unit -> string` ...

```
# read_line ();;
Hello World!
- : string = "Hello World!"
```

In order to read from file, the file must be opened for reading ...

```
# let infile = open_in "test";;
val infile : in_channel = <abstr>

# input_line infile;;
- : string = "The file's single line ...";;

# input_line infile;;
Exception: End_of_file.
```

If there is no further line the exception `End_of_file` is raised.
If a channel is no longer required, it should be explicitly closed ...

```
# close_in infile;;
- : unit = ()
```

Further Useful Values


```
stdin             : in_channel
input_char        : in_channel -> char
in_channel_length : in_channel -> int
```


- `stdin` is the standard input as channel.
- `input_char` returns the next character of the channel.
- `in_channel_length` returns the total length of the channel.

Output to files is analogous ...

```
# let outfile = open_out "test";;
val outfile : out_channel = <abstr>

# output_string outfile "Hello ";;
- : unit = ()

# output_string outfile "World!\n";;
- : unit = ()
...
```

The words written seperately, may only occur inside the file, once the file has been closed ...

```
# close_out outfile;;
- : unit = ()
```

5.3 Sequences

In presence of side-effects, ordering matters.

Several actions can be sequenced by means of the sequence operator ; :


```
# print_string "Hello";
  print_string " ";
  print_string "world!\n";;
Hello world!
- : unit = ()
```

Often, several strings must be output !

Given a list of strings, the list functional List.iter can be used:


```
#  let rec iter f = function
         []     -> ()
       | x::[]  -> f x
       | x::xs  -> f x; iter f xs;;

val iter : ('a -> unit) -> 'a list -> unit = <fun>

# iter print_string ["Hello "; "world"; "!\n"];;
Hello world!
```

6 The Module System of OCaml

→ Modules

→ Signatures

→ Information Hiding

→ Functors

→ Separate Compilation

6.1 Modules

In order to organize larger software systems, OCaml offers the concept of modules:


```
module Pairs =
  struct
    type 'a pair      = 'a * 'a
    let pair (a,b)    = (a,b)
    let first (a,b)   = a
    let second (a,b)  = b
  end
```

On this input, the compiler answers with the type of the module, its signature:

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```


The definitions inside the module are not visible outside:

```
# first;;
Unbound value first
```

Access onto Components of a Module

Components of a module can be accessed via qualification:

```
# Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```


Thus, several functions can be defined all with the same name:


```
# module Triples =
    struct
      type 'a triple = Triple of 'a * 'a * 'a
      let first  (Triple (a, _, _)) = a
      let second (Triple (_, b, _)) = b
      let third  (Triple (_, _, c)) = c
    end;;
...
```

```
...
module Triples :
  sig
    type 'a triple = Triple of 'a * 'a * 'a
    val first : 'a triple -> 'a
    val second : 'a triple -> 'a
    val third : 'a triple -> 'a
  end

# Triples.first ;;
- : 'a Triples.triple -> 'a = <fun>
```

... or several implementations of the same function:


```
# module Pairs2 =
    struct
      type 'a pair = bool -> 'a
      let pair (a,b) = fun x -> if x then a else b
      let first ab = ab true
      let second ab = ab false
    end;;
```

Opening Modules

In order to avoid explicit qualification, all definitions of a module can be made directly accessible:


```
# open Pairs2;;

# pair;;
- : 'a * 'a -> bool -> 'a = <fun>

# pair (4,3) true;;
- : int = 4
```


the keyword `include` allows to include the definitions of another module into the present module ...

```
# module A = struct let x = 1 end;;
module A : sig val x : int end

# module B =
    struct
      open A
      let y = 2
    end;;
module B : sig val y : int end

# module C =
    struct
      include A
      include B
    end;;
module C : sig val x : int val y : int end
```

Nested Modules

Modules may again contain modules:


```
module Quads = struct
  module Pairs =
    struct
      type 'a pair    = 'a * 'a
      let pair (a,b) = (a,b)
      let first  (a,_) = a
      let second (_,b) = b
    end

    type 'a quad = 'a Pairs.pair Pairs.pair
    let quad (a,b,c,d) =
      Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
  ...
```

```
  ...
    let first  q = Pairs.first  (Pairs.first q)
    let second q = Pairs.second (Pairs.first q)
    let third  q = Pairs.first  (Pairs.second q)
    let fourth q = Pairs.second (Pairs.second q)
  end

# Quads.quad (1, 2, 3, 4);;
- : (int * int) * (int * int) = ((1, 2), (3, 4))

# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

6.2 Module Types or Signatures

Signatures allow to restrict what a module may export.

Explicit indication of the signature allows

- to restrict the set of exported variables;
- to restrict the set of exported types ...

... an Example

```
module Sort = struct
  let single lst = map (fun x -> [x]) lst

  let rec merge l1 l2 = match (l1, l2)
      with ([], _) -> l2
      | (_, []) -> l1
      | (x::xs,y::ys) -> if x < y then x :: merge xs l2
                                  else y :: merge l1 ys

  let rec merge_lists = function
      [] -> [] | [l] -> [l]
    | l1::l2::ll -> merge l1 l2 :: merge_lists ll

  let sort lst = let lst = single lst
      in let rec doit = function
        [] -> [] | [l] -> l
      | l -> doit (merge_lists l)
  in doit lst
end
```

The implementation allows to access the auxiliary functions single, merge and merge_lists from the outside:


```
# Sort.single [1;2;3];;
- : int list list = [[1]; [2]; [3]]
```


In order to hide the functions single and merge_lists, we introduce the signature

```
module type Sort = sig
  val merge : 'a list -> 'a list -> 'a list
  val sort : 'a list -> 'a list
end
```

The functions `single` and `merge_lists` are no longer exported:


```
# module MySort : Sort = Sort;;
module MySort : Sort

# MySort.single;;
Unbound value MySort.single
```

Signatures and Types

The types mentioned in the signature must be Instances of the types for the exported definitions.

In that way, these types are spezialized:


```
module type A1 = sig
  val f : 'a -> 'b -> 'b
end

module type A2 = sig
  val f : int -> char -> int
end

module A = struct
  let f x y = x
end
```

```
# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                      is not included in A1
Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b

# module A2 : A2 = A;;
module A2 : A2

# A2.f;;
- : int -> char -> int = <fun>
```

6.3 Information Hiding

For reasons of modularity, we often would like to prohibit that the structure of exported types of a module are visible from the outside.

Example


```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
        [] -> true | _ -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end
```

A signature allows to hide the implementation of a queue:


```
module type Queue = sig
    type 'a queue

    val empty_queue : unit -> 'a queue

    val is_empty : 'a queue -> bool

    val enqueue : 'a queue -> 'a -> 'a queue

    val dequeue : 'a queue -> 'a * 'a queue
end
```

```
# module Queue : Queue = ListQueue;;
module Queue : Queue

# open Queue;;

# is_empty [];;
This expression has type 'a list but is here used with type
'b queue = 'b Queue.queue
```


$\Longrightarrow$

The restriction via signature is sufficient to obfuscate the true nature of the type queue.

If the datatype should be exported together with all constructors, its definition is repeated in the signature:


```
module type Queue = sig
  type 'a queue = Queue of ('a list * 'a list)

  val empty_queue : unit -> 'a queue

  val is_empty : 'a queue -> bool

  val enqueue : 'a -> 'a queue -> 'a queue

  val dequeue : 'a queue -> 'a option * 'a queue
end
```

6.4 Functors

Since (almost) everything in OCaml is higher order, it is no surprise that there are modules of higher order: Functors.

- A functor receives a sequence of modules as parameters.
- The functor's body is a module where the functor's parameters can be used.
- The result is a new module, which is defined relative to the modules passed as parameters.

First, we specify the functor's argument and result by means of signatures:

```
module type Decons = sig
  type 'a t
  val decons : 'a t -> ('a * 'a t) option
end

module type GenFold = functor (X : Decons) -> sig
  val fold_left : ('b -> 'a -> 'b) -> 'b -> 'a X.t -> 'b
  val fold_right : ('a -> 'b -> 'b) -> 'a X.t -> 'b -> 'b
  val size : 'a X.t -> int
  val list_of : 'a X.t -> 'a list
  val iter : ('a -> unit) -> 'a X.t -> unit
end
...
```

```
...
module Fold : GenFold = functor (X : Decons) ->
  struct
    let rec fold_left f b t =
      match X.decons t with None -> b |
                            Some (x, t) -> fold_left f (f b x) t

    let rec fold_right f t b =
      match X.decons t with None -> b |
                            Some (x, t) -> f x (fold_right f t b)

    let size t = fold_left (fun a x -> a + 1) 0 t

    let list_of t = fold_right (fun x xs -> x :: xs) t []

    let iter f t = fold_left (fun () x -> f x) () t
  end
```


Now, we can apply the functor to the module to obtain a new module ...

```
module MyQueue = struct open Queue
  type 'a t = 'a queue

  let decons = function
    | Queue ([], xs) -> (
        match rev xs with [] -> None
          | x :: xs -> Some (x, Queue (xs, [])))
    | Queue (x :: xs, t) -> Some (x, Queue (xs, t))
end

module MyAVL = struct open AVL
  type 'a t = 'a avl

  let decons avl =
    match extract_min avl with
    | None, avl      -> None
    | Some (a, avl) -> Some (a, avl)
end
```

```
module FoldAVL = Fold (MyAVL)
module FoldQueue = Fold (MyQueue)
```


By that, we may define


```
let sort list = FoldAVL.list_of (
                    AVL.from_list list)
```


Caveat

A module satisfies a signature whenever it implements it !
It is not required to explicitly declare that !!

6.5 Separate Compilation

- In reality, deployed OCaml programs will not run within the interactive shell.
- Instead, there is a compiler ocamlc ...

```
> ocamlc Test.ml
```


that interpretes the contents of the file Test.ml as a sequence of definitions of a module Test.

- As a result, the compiler ocamlc generates the files

Test.cmo	bytecode for the module
Test.cmi	bytecode for the signature
a.out	executable program

- If there is already a file `Test.mli` this is interpreted as the signature for `Test`. Then we call

```
> ocamlc Test.mli Test.ml
```

- Given a module `A` and a module `B`, then these should be compiled by

```
> ocamlc B.mli B.ml A.mli A.ml
```

- If a re-compilation of `B` should be omitted, `ocamlc` may receive a pre-compiled file

```
> ocamlc B.cmo A.mli A.ml
```

- For practical management of required re-compilation after modification of files, Linux offers the tool `make`. The script of required actions then is stored in a `Makefile`.
- ... alternatively, dune can be used.

7 Formal Verification for OCaml

Question

How can we make sure that an OCaml program behaves as it should ???

We require:

- a formal semantics
- means to prove assertions about programs ...

7.1 MiniOCaml

In order to simplify life, we only consider a fragment of OCaml.

We consider ...

- only base types `int`, `bool` as well as tuples and lists
- recursive function definitions only at top level

We rule out ...

- modifiable datatypes
- input and output
- local recursive functions

This fragment of OCaml is called MiniOCaml.

Expressions in MiniOCaml can be described by the grammar

$$
\begin{array}{rcl}
E & ::= & \text{const} \mid \text{name} \mid \text{op}_1\; E \mid E_1\; \text{op}_2\; E_2 \mid \\
 & & \texttt{(}E_1, \ldots, E_k\texttt{)} \mid \texttt{let}\ \text{name} = E_1\ \texttt{in}\ E_0 \mid \\
 & & \texttt{match}\ E\ \texttt{with}\ P_1\ \texttt{->}\ E_1\ \texttt{|}\ \ldots\ \texttt{|}\ P_k\ \texttt{->}\ E_k \mid \\
 & & \texttt{fun}\ \text{name}\ \texttt{->}\ E \mid E\; E_1
\end{array}
$$

$$
P \;::=\; \text{const} \mid \text{name} \mid \texttt{(}P_1, \ldots, P_k\texttt{)} \mid P_1\ \texttt{::}\ P_2
$$

Short-cut

$$
\texttt{fun}\ x_1\ \texttt{->}\ \ldots\texttt{fun}\ x_k\ \texttt{->}\ e \quad \equiv \quad \texttt{fun}\ x_1\ \ldots\ x_k\ \texttt{->}\ e
$$

Caveat

- The set of admissible expressions must be further restricted to those which are well typed, i.e., for which the OCaml compiler infers a type ...

  `(1, [true; false])` well typed

  `(1 [true; false])` not well typed

  `([1; true], false)` not well typed

- We also rule out `if ... then ... else ...`, since it can be simulated by `match ... with true -> ... | false -> ...`.

- We could also have omitted `let ... in ...` (why?)

A program consists of a sequence of mutally recursive global definitions of variables $f_1, \ldots, f_m$:

$$\begin{array}{llll} \texttt{let rec} & f_1 & = & E_1 \\ \texttt{and} & f_2 & = & E_2 \\ & \ldots & & \\ \texttt{and} & f_m & = & E_m \end{array}$$

7.2 A Semantics for MiniOCaml

Question

Which value is returned for the expression E ??

A value is an expression that cannot be further evaluated.
The set of all values can be specified by means of the grammar

$$
\begin{array}{rcl}
V & ::= & \texttt{const} \;\mid\; \texttt{fun name}_1 \ldots \texttt{name}_k \texttt{ -> } E \;\mid \\
 & & (V_1, \ldots, V_k) \;\mid\; \texttt{[]} \;\mid\; V_1 \texttt{ :: } V_2
\end{array}
$$

A MiniOCaml Program ...

```
let rec comp = fun f g x -> f (g x)
and map = fun f list -> match list
                  with []     -> []
                     | x::xs -> f x :: map f xs
```


Examples of Values ...

```
1
(1, [true; false])
fun x -> 1 + 1
[fun x -> x + 1; fun x -> x + 2; fun x -> x + 3]
```

Idea

- We define a relation $e \Rightarrow v$ between expressions and their values $\Longrightarrow$ big-step operational semantics.
- The relation is defined by means of axioms and rules that follow the structure of e.
- Apparently, $v \Rightarrow v$ holds for every value v.

Tuples

$$\text{(TU)} \quad \frac{e_1 \Rightarrow v_1 \qquad \ldots \qquad e_k \Rightarrow v_k}{(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_k)}$$

Lists

$$\text{(LI)} \quad \frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Global definitions

$$\text{(GD)} \quad \frac{f = e \qquad e \Rightarrow v}{f \Rightarrow v}$$

Local definitions

$$(\text{LD})\quad \frac{e_1 \;\Rightarrow\; v_1 \qquad e_0[v_1/x] \;\Rightarrow\; v_0}{\texttt{let}\; x \;\texttt{=}\; e_1 \;\texttt{in}\; e_0 \;\Rightarrow\; v_0}$$

Function calls

$$(\text{APP})\quad \frac{e \;\Rightarrow\; \texttt{fun}\; x \;\texttt{->}\; e_0 \qquad e_1 \;\Rightarrow\; v_1 \qquad e_0[v_1/x] \;\Rightarrow\; v_0}{e\; e_1 \;\Rightarrow\; v_0}$$

By repeated application of the rule for function calls, a rule for functions with multiple arguments can be derived:

$$(\text{APP'}) \quad \frac{e_0 \Rightarrow \texttt{fun } x_1 \ldots x_k \texttt{ -> } e \quad e_1 \Rightarrow v_1 \ldots e_k \Rightarrow v_k \quad e[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{e_0\; e_1 \;\ldots\; e_k \;\Rightarrow\; v}$$

This derived rule makes proofs somewhat simpler.

Pattern Matching

$$(\text{PM})\quad \frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1,\ldots,v_k/x_k] \qquad e_i[v_1/x_1,\ldots,v_k/x_k] \Rightarrow v}{\texttt{match}\ e_0\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\ \texttt{->}\ e_m \Rightarrow v}$$

— given that v' does not match any of the patterns $p_1,\ldots,p_{i-1}$
;-)

Built-in operators

$$(\text{OP})\quad \frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v_1\ \mathsf{op}\ v_2 \Rightarrow v}{e_1\ \mathsf{op}\ e_2 \Rightarrow v}$$

Unary operators are treated analogously.

The built-in equality operator

$$v = v \quad \Rightarrow \quad \texttt{true}$$

$$v_1 = v_2 \quad \Rightarrow \quad \texttt{false}$$

given that v, v_1, v_2 are values that do not contain functions, and v_1, v_2 are syntactically different.

Example 1

$$\text{(OP)}\ \dfrac{\text{(OP)}\ \dfrac{\texttt{17} \Rightarrow \texttt{17} \quad \texttt{4} \Rightarrow \texttt{4} \quad \texttt{17} + \texttt{4} \Rightarrow \texttt{21}}{\texttt{17 + 4} \Rightarrow \texttt{21}} \quad \texttt{21} \Rightarrow \texttt{21} \quad \texttt{21} = \texttt{21} \Rightarrow \texttt{true}}{\texttt{17 + 4 = 21} \Rightarrow \texttt{true}}$$

The built-in equality operator

$$v = v \quad \Rightarrow \quad \texttt{true}$$

$$v_1 = v_2 \quad \Rightarrow \quad \texttt{false}$$

given that v, v_1, v_2 are values that do not contain functions, and v_1, v_2 are syntactically different.

Example 1 — omitting axioms $v \Rightarrow v$

$$\dfrac{\text{(OP)}\ \dfrac{\texttt{17 + 4} \Rightarrow \texttt{21}}{\texttt{17 + 4} \Rightarrow \texttt{21}} \qquad \texttt{21 = 21} \Rightarrow \texttt{true}}{\texttt{17 + 4 = 21} \Rightarrow \texttt{true}}\ \text{(OP)}$$

Example 2


```
let rec f = fun x -> x+1
    and s = fun y -> y*y
```


$$\text{(OP)}\ \dfrac{\text{(APP)}\ \dfrac{\text{(GD)}\ \dfrac{\texttt{f = fun x -> x+1}}{\texttt{f} \Rightarrow \texttt{fun x -> x+1}} \quad \text{(OP)}\ \dfrac{\texttt{16+1} \Rightarrow \texttt{17}}{\texttt{16+1} \Rightarrow \texttt{17}}}{\texttt{f 16} \Rightarrow \texttt{17}} \quad \text{(APP)}\ \dfrac{\text{(GD)}\ \dfrac{\texttt{s = fun y -> y*y}}{\texttt{s} \Rightarrow \texttt{fun y -> y*y}} \quad \text{(OP)}\ \dfrac{\texttt{2*2} \Rightarrow \texttt{4}}{\texttt{2*2} \Rightarrow \texttt{4}}}{\texttt{s 2} \Rightarrow \texttt{4}} \quad \texttt{17+4} \Rightarrow \texttt{21}}{\texttt{f 16 + s 2} \Rightarrow \texttt{21}}$$

// uses of $v \Rightarrow v$ have mostly been omitted

Example 3


```
let rec app = fun x y -> match x
        with []     -> y
           | h :: t -> h :: app t y
```


Claim: app (1 : : []) (2 : : []) $\Rightarrow$ 1 : : 2 : : []

$$
(\mathrm{APP'})\;\dfrac{
(\mathrm{GD})\;\dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}
\qquad
(\mathrm{PM})\;\dfrac{
(\mathrm{LI})\;\dfrac{
(\mathrm{APP'})\;\dfrac{
(\mathrm{GD})\;\dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}}
\qquad
(\mathrm{PM})\;\dfrac{\texttt{[]} \Rightarrow \texttt{[]} \quad \texttt{2::[]} \Rightarrow \texttt{2::[]}}{\texttt{match [] ...} \Rightarrow \texttt{2::[]}}
}{\texttt{app [] (2::[])} \Rightarrow \texttt{2::[]}}
}{\texttt{1 :: app [] (2::[])} \Rightarrow \texttt{1::2::[]}}
}{\texttt{match 1::[] ...} \Rightarrow \texttt{1::2::[]}}
}{\texttt{app (1::[]) (2::[])} \Rightarrow \texttt{1::2::[]}}
$$

// uses of $v \Rightarrow v$ have mostly been omitted

Discussion

- The big-step operational semantics is not well suited for tracking step-by-step how evaluation by MiniOCaml proceeds.

- It is quite convenient, though, for proving that the evaluation of a function for particular argument values terminates:

  For that, it suffices to prove that there are values to which the corresponding function calls can be evaluated ...

Example Claim

`app` l_1 l_2 terminates for all list values l_1, l_2.

Proof

Induction on the length n of the list l_1.

$\boxed{n = 0}$ I.e., l_1 `= [].` Then

$$(\text{APP'})\ \dfrac{(\text{GD})\ \dfrac{\texttt{app = fun x y -> } \cdots}{\texttt{app} \Rightarrow \texttt{fun x y -> } \cdots} \qquad (\text{PM})\ \dfrac{}{\texttt{match [] with [] -> } l_2 \texttt{ | ...} \Rightarrow l_2}}{\texttt{app [] } l_2 \Rightarrow l_2}$$

$\boxed{n > 0:}$ I.e., l_1 = `h::t`.

In particular, we assume that the claim already holds for all shorter lists. Then we have:

$$\texttt{app t } l_2 \Rightarrow l$$

for some l. We deduce

$$\text{(APP')}\ \dfrac{\text{(GD)}\ \dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \qquad \text{(PM)}\ \dfrac{\text{(LI)}\ \dfrac{\dfrac{\text{I.H.}}{\texttt{app t } l_2 \Rightarrow l}}{\texttt{h :: app t } l_2 \Rightarrow \texttt{h :: } l}}{\texttt{match h::t with } \cdots \Rightarrow \texttt{h :: } l}}{\texttt{app (h::t) } l_2 \Rightarrow \texttt{h :: } l}$$

Discussion (cont.)

- The big-step semantics also allows to verify that optimizing transformations are correct, i.e., preserve the semantics.
- Finally, it can be used to prove the correctness of assertions about functional programs !

- The big-step operational semantics suggests to consider expressions as specifications of values.
- Expressions which evaluate to the same values, should be interchangeable ...

Caveat

- In MiniOCaml, equality between values can only be tested if these do not contain functions !!
- Such values are called comparable. They are of the form

$$C \quad ::= \quad \texttt{const} \quad | \quad (C_1, \ldots, C_k) \quad | \quad \texttt{[]} \quad | \quad C_1 \;\texttt{::}\; C_2$$

- Apparently, a value of MiniOCaml is comparable if and only iff its type does not contain functions:

$$c \quad ::= \quad \texttt{bool} \quad | \quad \texttt{int} \quad | \quad \texttt{unit} \quad | \quad c_1 * \ldots * c_k \quad | \quad c\;\texttt{list}$$

Discussion

- For program optimization, we sometimes may want to exchange functions, e.g.,


```
comp (map f) (map g) = map (comp f g)
```


- Apparently, the functions to the right and left of the equality sign cannot be compared by OCaml for equality.

$\Longrightarrow$

Reasoning in logic requires an extended notion of equality!

Extension of Equality

The equality = of OCaml is extended to expression which may not terminate, and functions.

Non-termination

$$\frac{e_1, e_2 \quad \text{both not terminating}}{e_1 = e_2}$$

Termination

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1 = v_2}{e_1 = e_2}$$

Structured values

$$\frac{v_1 = v_1' \quad \dots \quad v_k = v_k'}{(v_1, \dots, v_k) = (v_1', \dots, v_k')}$$

$$\frac{v_1 = v_1' \qquad v_2 = v_2'}{v_1 :: v_2 = v_1' :: v_2'}$$

Functions

$$\frac{e_1[v/x_1] = e_2[v/x_2] \quad \text{for all} \quad v}{\texttt{fun}\ x_1\ \texttt{->}\ e_1 = \texttt{fun}\ x_2\ \texttt{->}\ e_2}$$

$\Longrightarrow$ extensional equality

We have:

$$\frac{e \Rightarrow v}{e = v}$$

Assume that the type of e_1, e_2 is functionfree. Then

$$\frac{e_1 = e_2 \qquad\qquad e_1 \ \text{terminates}}{e_1 = e_2 \ \Rightarrow\ \texttt{true}}$$

$$\frac{e_1 = e_2 \ \Rightarrow\ \texttt{true}}{e_1 = e_2 \qquad\qquad e_i \ \text{terminates}}$$

The crucial tool for our proofs is the ...

Substitution Lemma

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

We deduce for functionfree expressions e:

$$\frac{e_1 = e_2 \qquad e[e_1/x] \ \text{terminate}}{e[e_1/x] = e[e_2/x] \ \Rightarrow \ \texttt{true}}$$

Discussion

- The lemma tells us that in every context, all occurrences of the expression e_1 can be replaced by the expression e_2 — whenever e_1 and e_2 represent the same values.

- The lemma can be proven by induction on the depth of the required derivations (which we omit).
- The exchange of expressions proven equal, allows us to design a calculus for proving the equivalence of expressions ...

We provide us with a repertoir of rewrite rules for reducing the equality of expressions to the equality of, possibly simpler expressions ...

Simplification of local definitions

$$\frac{e_1 \text{ terminates}}{\texttt{let } x = e_1 \texttt{ in } e \;=\; e[e_1/x]}$$

Simplification of function calls

$$\frac{e_0 \;=\; \texttt{fun } x \texttt{ -> } e \qquad e_1 \text{ terminates}}{e_0\ e_1 \;=\; e[e_1/x]}$$

Proof of the let rule

Since e_1 terminates, there is a value v_1 with:

$$e_1 \Rightarrow v_1$$

Due to the Substitution Lemma, we have:

$$e[v_1/x] = e[e_1/x]$$

Case 1: $e[v_1/x]$ terminates.

Then a value v exists with

$$e[v_1/x] \Rightarrow v$$

Then

$$e[e_1/x] = e[v_1/x] = v$$

Because of the big step semantics, however, we have:

$$\begin{array}{lcll} \texttt{let}\ x = e_1\ \texttt{in}\ e & \Rightarrow & v & \text{and therefore,} \\ \texttt{let}\ x = e_1\ \texttt{in}\ e & = & e[e_1/x] & \end{array}$$

Case 2: $e[v_1/x]$ does not terminate.

Then $e[e_1/x]$ does not terminate and neither does $\texttt{let}\ x = e_1\ \texttt{in}\ e$.

Accordingly,

$$\texttt{let}\ x = e_1\ \texttt{in}\ e = e[e_1/x]$$

By repeated application of the rule for function calls, an extra rule for functions with multiple arguments can be deduced:

$$\frac{e_0 = \texttt{fun}\ x_1 \ldots x_k \texttt{ -> } e \qquad e_1, \ldots, e_k \text{ terminate}}{e_0\ e_1 \ldots e_k = e[e_1/x_1, \ldots, e_k/x_k]}$$

This derived rule allows to shorten some proofs considerably.

Rule for pattern matching

$$\frac{e_0 = []}{\texttt{match}\ e_0\ \texttt{with}\ [\,]\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\ \texttt{->}\ e_m \;=\; e_1}$$

$$\frac{e_0\ \text{terminates} \qquad e_0 = e_1' :: e_2'}{\texttt{match}\ e_0\ \texttt{with}\ \texttt{[]}\ \texttt{->}\ e_1\ \texttt{|}\ x :: xs\ \texttt{->}\ e_2 \;=\; e_2[e_1'/x, e_2'/xs]}$$

We are now going to apply these rules ...

7.3 Equational Proofs for MiniOCaml

Example 1:


```
let rec app = fun x -> fun y -> match x
                    with []    -> y
                       | h::t -> h :: app t y
```


We want to verify that

(1) `app x [] = x` for all lists `x`.

(2) `app x (app y z) = app (app x y) z`
for all lists `x`, `y`, `z`.

Idea: Induction on the length n of x

$\boxed{n = 0}$ Then x = [] holds.

We deduce:

$$
\begin{aligned}
\texttt{app x []} &\overset{\texttt{def x}}{=} \texttt{app [] []} \\
&\overset{\texttt{app}}{=} \texttt{match [] with [] -> [] | h::t -> h :: app t []} \\
&\overset{\texttt{match}}{=} \texttt{[]} \\
&\overset{\texttt{def x}}{=} \texttt{x}
\end{aligned}
$$

$\boxed{n > 0}$ Then: `x = h::t` where `t` has length $n - 1$.

We deduce:

$$
\begin{aligned}
\texttt{app x []} &\stackrel{\texttt{def x}}{=} \texttt{app (h::t) []} \\
&\stackrel{\texttt{app}}{=} \texttt{match h::t with [] -> [] | h::t -> h :: app t []} \\
&\stackrel{\texttt{match}}{=} \texttt{h :: app t []} \\
&\stackrel{\texttt{I.H.}}{=} \texttt{h :: t} \\
&\stackrel{\texttt{def x}}{=} \texttt{x}
\end{aligned}
$$

Analogously we proceed for assertion (2) ...

$\boxed{n = 0}$ Then: `x = []`

We deduce:

$$
\begin{aligned}
\texttt{app x (app y z)} &\stackrel{\texttt{def x}}{=} \texttt{app [] (app y z)} \\
&\stackrel{\texttt{app}}{=} \texttt{match [] with [] -> app y z | h::t -> ...} \\
&\stackrel{\texttt{match}}{=} \texttt{app y z} \\
&\stackrel{\texttt{match}}{=} \texttt{app (match [] with [] -> y | ...) z} \\
&\stackrel{\texttt{app}}{=} \texttt{app (app [] y) z} \\
&\stackrel{\texttt{def x}}{=} \texttt{app (app x y) z}
\end{aligned}
$$

$\boxed{n > 0}$ Then `x = h::t` where `t` has length $n - 1$.

We deduce:


```
                   def x
app x (app y z)      =    app (h::t) (app y z)
                    app
                     =    match h::t with [] -> app y z
                                   | h::t -> h :: app t (app y z)
                   match
                     =    h :: app t (app y z)
                   I.H.
                     =    h :: app (app t y) z
                 match,app
                     =    app (h :: app t y) z
                   match
                     =    app (match h::t with [] -> []
                                    | h::t -> h :: app t y) z
                    app
                     =    app (app (h::t) y) z
                   def x
                     =    app (app x y) z
```

Discussion

- For the correctness of our induction proofs, we require that all occurring function calls terminate.
- In the example, it suffices to prove that for all x, y, there exists some v such that:

$$\texttt{app x y} \Rightarrow v$$

... which we have already proven, as usual, by induction.

Example 2

```
let rec rev = fun x -> match x
          with [] -> []
             | h::t -> app (rev t) [h]

let rec rev1 = fun x -> fun y -> match x
          with [] -> y
             | h::t -> rev1 t (h::y)
```


Claim

`rev x = rev1 x []` for all lists `x`.

More generally,

`app (rev x) y = rev1 x y` for all lists `x, y`.

Proof: Induction on the length n of `x`

$\boxed{n = 0}$ Then: `x = [].` We deduce:

$$
\begin{aligned}
\texttt{app (rev x) y} &\stackrel{\texttt{def x}}{=} \texttt{app (rev []) y} \\
&\stackrel{\texttt{rev}}{=} \texttt{app (match [] with [] -> [] | ...) y} \\
&\stackrel{\texttt{match}}{=} \texttt{app [] y} \\
&\stackrel{\texttt{app,match}}{=} \texttt{y} \\
&\stackrel{\texttt{match}}{=} \texttt{match [] with [] -> y | ...} \\
&\stackrel{\texttt{rev1}}{=} \texttt{rev1 [] y} \\
&\stackrel{\texttt{def x}}{=} \texttt{rev1 x y}
\end{aligned}
$$

$\boxed{n > 0}$ Then `x = h::t` where `t` has length $n-1$.

We deduce (ommitting simple intermediate steps):

$$
\begin{aligned}
\texttt{app (rev x) y} &\overset{\texttt{def x}}{=} \texttt{app (rev (h::t)) y} \\
&\overset{\texttt{rev,match}}{=} \texttt{app (app (rev t) [h]) y} \\
&\overset{\texttt{by example 1}}{=} \texttt{app (rev t) (app [h] y)} \\
&\overset{\texttt{(app,match)}^2}{=} \texttt{app (rev t) (h::y)} \\
&\overset{\texttt{I.H.}}{=} \texttt{rev1 t (h::y)} \\
&\overset{\texttt{match,rev1}}{=} \texttt{rev1 (h::t) y} \\
&\overset{\texttt{def x}}{=} \texttt{rev1 x y}
\end{aligned}
$$

Discussion

- Again, we have implicitly assumed that all calls of `app`, `rev` and `rev1` terminate.
- Termination of these can be proven by induction on the length of their first arguments.
- The claim:


```
rev x = rev1 x []
```

follows from:

```
app (rev x) y = rev1 x y
```


by setting: `y = []` and assertion (1) from example 1.

Example 3

```
let rec sorted = fun x -> match x
    with h1::h2::t -> (
          match h1 <= h2
          with true  -> sorted (h2::t)
             | false -> false)
        | _ -> true

and merge = fun x -> fun y -> match (x,y)
    with ([],y) -> y
       | (x,[]) -> x
       | (x1::xs, y1::ys) -> (
          match x1 <= y1
          with true  -> x1 :: merge xs y
             | false -> y1 :: merge x ys)
```

Claim

$$\texttt{sorted x} \wedge \texttt{sorted y} \rightarrow \texttt{sorted (merge x y)}$$

for all lists `x`, `y`.

Proof: Induction on the sum n of lengthes of `x`, `y`.

Assume that $\texttt{sorted x} \wedge \texttt{sorted y}$ holds.

$\boxed{n = 0}$ Then: `x = [] = y`

We deduce:

$$\begin{aligned} \texttt{sorted (merge x y)} &\overset{\texttt{def x,y}}{=} \texttt{sorted (merge [] [])} \\ &\overset{\texttt{merge}}{=} \texttt{sorted []} \\ &\overset{\texttt{sorted}}{=} \texttt{true} \end{aligned}$$

$\boxed{n > 0}$

Case 1: `x = []`.

We deduce:

```
sorted (merge x y)   def x
                       =    sorted (merge [] y)
                     merge
                       =    sorted y
                by assumption
                       =    true
```

Case 2: `y = []` analogous.

Case 3: $\texttt{x = x1::xs} \wedge \texttt{y = y1::ys} \wedge \texttt{x1} \leq \texttt{y1}.$

We deduce:

$$
\begin{aligned}
\texttt{sorted (merge x y)} &\overset{\texttt{def x,y}}{=} \texttt{sorted (merge (x1::xs) (y1::ys))} \\
&\overset{\texttt{x1} \leq \texttt{y1?}}{=} \texttt{sorted (x1 :: merge xs (y1::ys))} \\
&\overset{\texttt{def y}}{=} \texttt{sorted (x1 :: merge xs y)} \\
&= \ \ldots
\end{aligned}
$$

Case 3.1: $\texttt{xs = []}$

We deduce:

$$
\begin{aligned}
\ldots &\overset{\texttt{def xs}}{=} \texttt{sorted (x1 :: merge [] y)} \\
&\overset{\texttt{merge}}{=} \texttt{sorted (x1 :: y)} \\
&\overset{\texttt{x1} \leq \texttt{y1}}{=} \texttt{sorted y} \\
&\overset{\texttt{by assumption}}{=} \texttt{true}
\end{aligned}
$$

Case 3.2: $\quad$ `xs = x2::xs'` $\wedge$ `x2` $\leq$ `y1`.

In particular: $\quad$ `x1` $\leq$ `x2` $\wedge$ `sorted xs`.

We deduce:

$$
\begin{aligned}
\texttt{...} &\overset{\texttt{def xs}}{=} \texttt{sorted (x1 :: merge (x2::xs') y)} \\
&\overset{\texttt{x2} \leq \texttt{y1}}{=} \texttt{sorted (x1 :: x2 :: merge xs' y)} \\
&\overset{\texttt{x1} \leq \texttt{x2}}{=} \texttt{sorted (x2 :: merge xs' y)} \\
&\overset{\texttt{merge}}{=} \texttt{sorted (merge xs y)} \\
&\overset{\texttt{I.H.}}{=} \texttt{true}
\end{aligned}
$$

Case 3.3: `xs = x2::xs'` $\wedge$ `x2` $>$ `y1`.

In particular: `x1` $\leq$ `y1` $<$ `x2` $\wedge$ `sorted xs`.

We deduce:

$$
\begin{array}{rcl}
\texttt{...} & \overset{\texttt{def xs/y}}{=} & \texttt{sorted (x1 :: merge (x2::xs') (y1::ys))} \\
 & \overset{\texttt{y1} < \texttt{x2}}{=} & \texttt{sorted (x1 :: y1 :: merge (x2::xs') ys)} \\
 & \overset{\texttt{def xs}}{=} & \texttt{sorted (x1 :: y1 :: merge xs ys)} \\
 & \overset{\texttt{x1} \leq \texttt{y1}}{=} & \texttt{sorted (y1 :: merge xs ys)} \\
 & \overset{\texttt{merge}}{=} & \texttt{sorted (merge xs y)} \\
 & \overset{\texttt{I.H.}}{=} & \texttt{true}
\end{array}
$$

Case 4: $\texttt{x = x1::xs} \wedge \texttt{y = y1::ys} \wedge \texttt{x1} > \texttt{y1}$.

We deduce:

$$
\begin{aligned}
\texttt{sorted (merge x y)} &\stackrel{\texttt{def x,y}}{=} \texttt{sorted (merge (x1::xs) (y1::ys))} \\
&\stackrel{\texttt{y1} < \texttt{x1}}{=} \texttt{sorted (y1 :: merge (x1::xs) ys)} \\
&\stackrel{\texttt{def x}}{=} \texttt{sorted (y1 :: merge x ys)} \\
&= \ \ldots
\end{aligned}
$$

Case 4.1: $\texttt{ys = []}$

We deduce:

$$
\begin{aligned}
\ldots &\stackrel{\texttt{def ys}}{=} \texttt{sorted (y1 :: merge x [])} \\
&\stackrel{\texttt{merge}}{=} \texttt{sorted (y1 :: x)} \\
&\stackrel{\texttt{y1} < \texttt{x1}}{=} \texttt{sorted x} \\
&\stackrel{\texttt{by assumption}}{=} \texttt{true}
\end{aligned}
$$

Case 4.2: $\texttt{ys = y2::ys'} \wedge \texttt{x1} > \texttt{y2}$.

In particular: $\texttt{y1} < \texttt{y2} \wedge \texttt{sorted ys}$.

We deduce:

$$
\begin{aligned}
\ldots &\overset{\texttt{def ys}}{=} \texttt{sorted (y1 :: merge x (y2::ys'))} \\
&\overset{\texttt{y2} < \texttt{x1}}{=} \texttt{sorted (y1 :: y2 :: merge x ys')} \\
&\overset{\texttt{y1} \leq \texttt{y2}}{=} \texttt{sorted (y2 :: merge x ys')} \\
&\overset{\texttt{merge}}{=} \texttt{sorted (merge x ys)} \\
&\overset{\texttt{I.H.}}{=} \texttt{true}
\end{aligned}
$$

Case 4.3: $\quad$ `ys = y2::ys'` $\wedge$ `x1` $\leq$ `y2`.

In particular: $\quad$ `y1` $<$ `x1` $\leq$ `y2` $\wedge$ `sorted ys`.

We deduce:

$$
\begin{array}{rl}
\ldots \stackrel{\texttt{def x,ys}}{=} & \texttt{sorted (y1 :: merge (x1::xs) (y2::ys'))} \\
\stackrel{\texttt{x1} \leq \texttt{y2}}{=} & \texttt{sorted (y1 :: x1 :: merge xs (y2::ys'))} \\
\stackrel{\texttt{def ys}}{=} & \texttt{sorted (y1 :: x1 :: merge xs ys)} \\
\stackrel{\texttt{y1} < \texttt{x1}}{=} & \texttt{sorted (x1 :: merge xs ys)} \\
\stackrel{\texttt{merge}}{=} & \texttt{sorted (merge x ys)} \\
\stackrel{\texttt{I.H.}}{=} & \texttt{true} \quad \text{by induction hypothesis}
\end{array}
$$

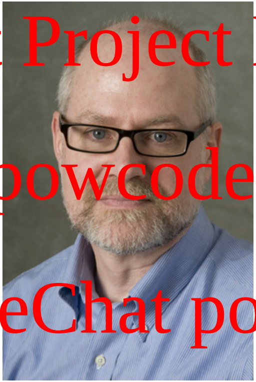
Discussion

- Again, we have assumed for the proof that all calls of the functions `sorted` and `merge` terminate.
- As an additional techniques, we required a thorough case distinction over the various possibilities for arguments in calls.
- The case distinction made the proof longish and cumbersome.

// The case $n = 0$ is in fact superfluous.

// since it is covered by the cases 1 and 2

8 Parallel Programming



John H. Reppy, University of Chicago

When your program requires multiple threads, use

```
ocamlc   -I +threads unix.cma  threads.cma  <my files>
ocamlopt -I +threads unix.cmxa threads.cmxa <my files>
```


When you want to play with it within utop, use the following sequence of commands:


```
#thread;;
#directory "+threads";;
#load "unix.cma";;
#load "threads.cma";;
```


Or load directly via:

```
utop -I +threads
```

Example


```
module Echo = struct
  open Thread

  let echo () = print_string (read_line () ^ "\n")

  let main =
    let t1 = create echo () in
    join t1;
    print_int (id (self ()));
    print_string "\n"
end
```

Comments

- The module `Thread` collects basic functionality for the creation of concurrency.
- The function `create: ('a -> 'b) -> 'a -> t` creates a new thread with the following properties:
  - The thread evaluates the function for its argument.
  - The creating thread receives the thread id as the return value and proceeds independently.
  - By means of the functions: `self : unit -> t` and `id : t -> int`, the own thread id can be queried and turned into an `int`, respectively.

Further useful Functions

- The function `join: t -> unit` blocks the current thread until the evaluation of the given thread has terminated.

- The function `kill: t -> unit` stops a given thread (not implemented);
- The function `delay: float -> unit` delays the current thread by a time period in seconds;

- The function `exit: unit -> unit` terminates the current thread.

... running the compiled code yields:

```
> ./a.out
Hello Echo!
Hello Echo!
0
>
```


- OCaml threads are only emulated by the runtime system.
- The creation of threads is cheap.
- Program execution terminates with the termination of the thread with the id 0 .

8.1 Channels

Threads communicate via channels.

The module `Event` provides basic functionality for the creation of channels, sending and receiving:


```
type 'a channel
type 'a event

new_channel : unit      -> 'a channel
always      : 'a        -> 'a event
sync        : 'a event   -> 'a
receive     : 'a channel -> 'a event
send        : 'a channel -> 'a          -> unit event
```

- Each call `new_channel()` creates another channel.
- Arbitrary data may be sent across a channel !!!
- `always` wraps a value into an event.
- Sending and receiving generates events ...
- Synchronization on evente returns their values.


```
module Exchange = struct open Thread open Event
  let thread ch = let x = sync (receive ch) in
                  print_string (x ^ "\n");
                  sync (send ch "got it!")

  let main = let ch = new_channel () in
             let _ = create thread ch in
             print_string "main is running ...\n";
             sync (send ch "Greetings!");
             print_string ("He " ^ sync (receive ch) ^ "\n")
end
```

Discussion

- `sync (send ch str)` exposes the event of sending to the outside world and blocks the sender, until another thread has read the value from the channel ...
- `sync (receive ch)` blocks the receiver, until a value has been made available on the channel. Then this value is returned as the result.
- Synchronous communication is one alternative for exchange of data between threads as well as for orchestration of concurrency $\Longrightarrow$ rendezvous
- In particular, it can be used to realize asynchronous communication between threads.

In the example, `main` spawns a thread. Then it sends it a string and waits for the answer. Accordingly, the new thread waits for the transfer of a `string` value over the channel. As soon as the string is received, an answer is sent on the same channel.

Caveat

If the ordering of `send` and `receive` is not carefully designed, threads easily get blocked ...

Execution of the program yields:


```
> ./a.out
main is running ...
Greetings!
He got it!
>
```

Example: A global memory cell

A global memory cell, in particular in presence of multiple threads, can be realized by implementing the signature `Cell`:


```
module type Cell = sig
  type 'a cell

  val new_cell : 'a      -> 'a cell
  val get      : 'a cell -> 'a
  val put      : 'a cell -> 'a      -> unit
end
```

The implementation must take care that the `get` and `put` calls are sequentialized.

This task is delegated to a server thread that reacts to get and put:


```
type 'a req  = Get of 'a channel | Put of 'a
type 'a cell = 'a req channel
```


The channel transports requests to the memory cell, which either provide the new value or the back channel ...

```
let get cell = let reply = new_channel () in
               sync (send cell (Get reply));
               sync (receive reply)
```


The function get sends a new back channel on the channel cell. If the latter is received, it waits for the return value.


```
let put cell x = sync (send cell (Put x))
```


The function put sends a Put element which contains the new value for the memory cell.

Of interest now is the implementation of the cell itself:

```
let new_cell x =
  let cell = new_channel () in
  let rec serve x =
    match sync (receive cell) with
    | Get reply ->
        sync (send reply x);
        serve x
    | Put y -> serve y
  in
  let _ = create serve x in
  cell
```

Creation of the cell with initial value `x` spawns a server thread that evaluates the call `serve x`.

Caveat

The server thread is possibly non-terminating!
This is why it can respond to arbitrarily many requests.
Only because it is tail-recursive, it does not successively consume the whole storage ...

```
let main =
  let cell = new_cell 1 in
  print_int (get cell);
  print_string "\n";
  put cell 2;
  print_int (get cell);
  print_string "\n"
```


Now, the execution yields


```
> ./a.out
1
2
>
```


Instead of `get` and `put`, also more complex query or update operations could be executed by the `cell` server ...

Example: Locks

Often, only one at a time out of several active threads should be allowed access to a given resource. In order to realize such a mutual exclusion, locks can be applied:


```
module type Lock = sig
    type lock
    type ack

    val new_lock  : unit -> lock
    val acquire   : lock -> ack
    val release   : ack  -> unit
end
```

Execution of the operation `acquire` returns an element of type `ack` which is used to return the lock:


```
type ack = unit channel
type lock = ack channel
```


For simplicity, `ack` is chosen itself as the channel by which the lock is returned.

```
let acquire lock = let ack = new_channel () in
                   sync (send lock ack);
                   ack
```

The unlock channel is created by `acquire` itself

```
let release ack = sync (send ack ())
```


… and used by the operation `release`.


```
let new_lock () =
  let lock = new_channel () in
  let rec acq_server () = rel_server (sync (receive lock))
  and rel_server ack =
    sync (receive ack);
    acq_server ()
  in
  let _ = create acq_server () in
  lock
```

Core of the implementation are the two mutually recursive functions `acq_server` and `rel_server`.

`acq_server` expects an element `ack`, i.e., a channel, and upon reception, calls `rel_server`.
`rel_server` expects a signal on the received channel indicated that the lock is released ...

Now we are in the position to realize a decent deadlock...

```
let dead =
  let l1 = new_lock () in
  let l2 = new_lock () in
  let th (l1, l2) =
    let a1 = acquire l1 in
    let _ = delay 1.0 in
    let a2 = acquire l2 in
    release a2; release a1;
    print_int (id (self ())); print_string " finished\n"
  in
  let t1 = create th (l1, l2) in
  let t2 = create th (l2, l1) in
  join t1
```


The result is

```
    > ./a.out
```

OCaml waits forever ...

Example: Semaphores

Occasionally there is more than one copy of a resource. Then semaphores are the method of choice ...

```
module type Sema = sig
    type sema

    new_sema : int -> sema
    up       : sema -> unit
    down     : sema -> unit
end
```

Idea

Again, a server is realized using an accumulating parameter, now maintaining the number of free resources or, if zero, the queue of waiting threads ...


```
module Sema = struct
  open Thread
  open Event

  type sema = unit channel option channel

  let up sema = sync (send sema None)

  let down sema =
    let ack = (new_channel () : unit channel) in
    sync (send sema (Some ack));
    sync (receive ack)
...
```

```
...
  let new_sema n = let sema = new_channel () in
    let rec serve (n, q) =
      match sync (receive sema) with
        | None -> (match dequeue q with
            | None, q -> serve (n + 1, q)
            | Some ack, q -> sync (send ack ());
                             serve (n, q))
        | Some ack ->
            if n > 0 then (
              sync (send ack ());
              serve (n - 1, q))
            else serve (n, enqueue ack q) in
    let _ = create serve (n, new_queue ()) in
    sema
end
```


Apparently, the queue does not maintain the waiting threads, but only their back channels.

8.2 Selective Communication

A thread need not necessarily know which of several possible communication rendezvous will occur or will occur first.
Required is a non-deterministic choice between several actions ...

Example: The function

```
add : int channel * int channel * int channel -> unit
```


is meant to read integers from two channels and send their sum to the third.

First Attempt

```
let forever f init =
  let rec loop x = loop (f x) in
  let _ = create loop init in
  ()

let add (in1, in2, out) =
  forever
    (fun () -> sync (send out (sync (receive in1) +
                          sync (receive in2))))
    ()
```


Disadvantage

If a value arrives at the second input channel first, the thread nonetheless must wait.

Second Attempt


```
let add (in1, in2, out) =
  forever
    (fun () ->
      let a, b = select [
          wrap (receive in1) (fun a -> (a, sync (receive in2)));
          wrap (receive in2) (fun b -> (sync (receive in1), b))
        ]
      in
      sync (send out (a + b)))
    ()
```


This program must be digested slowly ...

Idea

→ Initiating input or output operations, generates events.

→ Events are data objects of type `'a event`.

→ The function


```
wrap : 'a event -> ('a -> 'b) -> 'b event
```


applies a function a posteriori to the value of an event — given that it occurs.

The list thus consists of `(int*int)` events.

The functions


```
choose : 'a event list -> 'a event
select : 'a event list -> 'a
```

non-deterministically choose an event from the event list.

`select` synchronizes with the selected event, i.e., performs the corresponding communication task and returns the event:

```
let select = comp sync choose
```


Typically, that event occurs that finds its communication partner first.

Further Examples

The function

```
copy : 'a channel * 'a channel * 'a channel -> unit
```

is meant to copy a read element into two channels:

```
let copy (read, out1, out2) =
  forever
    (fun () ->
      let x = sync (receive read) in
      select
        [
          wrap (send out1 x) (fun () -> sync (send out2 x));
          wrap (send out2 x) (fun () -> sync (send out1 x));
        ])
    ()
```

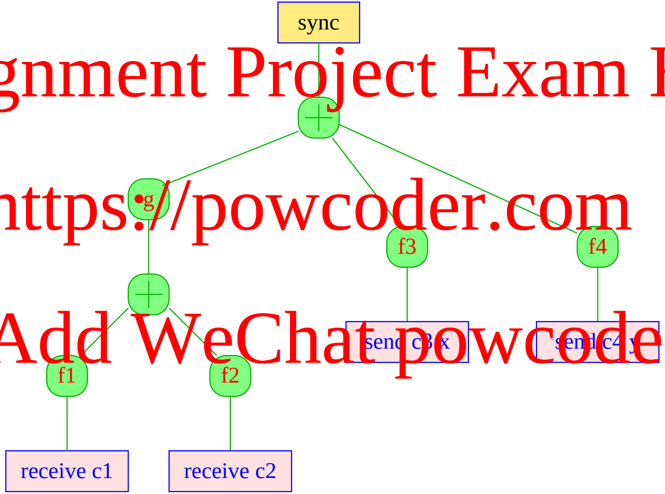
Apparently, the event list may also consist of send events — or contain both kinds.

```
type 'a cell = 'a channel * 'a channel

let get (get_chan, _) = sync (receive get_chan)
let put (_, put_chan) x = sync (send put_chan x)

let new_cell x =
  let get_chan = new_channel () in
  let put_chan = new_channel () in
  let rec serve x = select [
        wrap (send get_chan x) (fun () -> serve x);
        wrap (receive put_chan) serve;
      ]
  in
  let _ = create serve x in
  (get_chan, put_chan)
```

In general, there could be a tree of events:



- → The leaves are basic events.
- → A wrapper function may be applied to any given event.
- → Several events of the same type may be combined into a choice.
- → Synchronization on such an event tree activates a single leaf event. The result is obtained by successively applying the wrapper functions from the path to the root.

Example: A Swap Channel

Upon rendezvous, a swap channel is meant to exchange the values of the two participating threads. The signature is given by


```
module type Swap = sig
  type 'a swap

  val new_swap : unit    -> 'a swap
  val swap     : 'a swap -> 'a       -> 'a event
end
```


In the implementation with ordinary channels, every participating thread must offer the possibility to receive and to send.

As soon as a thread successfully completed to send (i.e., the other thread successfully synchronized on a `receive` event), the second value must be transmitted in opposite direction.

Together with the first value, we therefore transmit a channel for the second value:

```
module Swap = struct
  open Thread
  open Event

  type 'a swap = ('a * 'a channel) channel

  let new_swap () = new_channel ()

  let swap ch x =
    let c = new_channel () in
    choose [
        wrap (receive ch) (fun (y, c) -> sync (send c x); y);
        wrap (send ch (x, c)) (fun () -> sync (receive c));
      ]
end
```


A specific exchange can be realized by replacing `choose` with `select`.

Timeouts

Often, our patience is not endless.
Then, waiting for a send or receive event should be terminated ...


```
module type Timer = sig
    set_timer      : float        -> unit event
    timed_receive  : 'a channel -> float          -> 'a option event
    timed_send     : 'a channel -> 'a -> float -> unit option event
end
```

```
module Timer = struct open Thread open Event

  let set_timer t = let ack = new_channel () in
    let serve () = delay t; sync (receive ack) in
    let _ = create serve () in
    send ack ()

  let timed_receive ch time = choose [
      wrap (receive ch) (fun a -> Some a);
      wrap (set_timer time) (fun () -> None);
    ]

  let timed_send ch x time = choose [
      wrap (send ch x) (fun a -> Some ());
      wrap (set_timer time) (fun () -> None);
    ]
end
```

8.3 Threads and Exceptions

An exception must be handled within the thread where it has been raised.


```
module Explode = struct
  open Thread

  let thread x =
    x / 0;
    print_string "thread terminated regularly ...\n"

  let main =
    let _ = create thread 0 in
    delay 1.0;
    print_string "main terminated regularly ...\n"
end
```

... yields

```
> /.a.out
Thread 1 killed on uncaught exception Division_by_zero
main terminated regularly ...
```


The thread was killed, the OCaml program terminated nonetheless.

Also, uncaught exceptions within the wrapper function terminate the running thread:


```
module ExplodeWrap = struct open Thread open Event open Timer
  let main =
    try sync (wrap (set_timer 1.0) (fun () -> 1 / 0))
    with _ ->
      0;
      print_string "... this is the end!\n"
end
```

Then we have

```
> ./a.out
Fatal error: exception Division_by_zero
```


Caveat

Exceptions can only be caught in the body of the wrapper function itself, not behind the `sync` !

8.4 Buffered Communication

A channel for buffered communication allows to send without blocking. Receiving still may block, if no messages are available. For such channels, we realize a module `Mailbox`:

```
module type Mailbox = sig
    type 'a mbox

    val new_mailbox : unit    -> 'a mbox
    val receive     : 'a mbox -> 'a event
    val send        : 'a mbox -> 'a -> unit
end
```

For the implementation, we rely on a server which maintains a queue of sent but not yet received messages.

Then we implement:


```
module Mailbox = struct
    open Thread
    open Queue
    open Event

    type 'a mbox = 'a channel * 'a channel

    let send (in_chan, _) x = sync (send in_chan x)

    let receive (_, out_chan) = receive out_chan
...
```

```
...
  let new_mailbox () = let in_chan = new_channel ()
    and out_chan = new_channel () in
    let rec serve q =
      if is_empty q then serve (enqueue
                        (sync (Event.receive in_chan)) q)
      else select [
            wrap (Event.receive in_chan)
              (fun y -> serve (enqueue y q));
            wrap (Event.send out_chan (first q))
              (fun () -> let _, q = dequeue q in
                          serve q);
          ]
    in
    let _ = create serve (new_queue ()) in
    (in_chan, out_chan)
end
```


... where `first : 'a queue -> 'a` returns the first element in the queue without removing it.

8.5 Multicasts

For sending a message to `many` receivers, a module `Multicast` is provided that implements the signature `Multicast`:


```
module type Multicast = sig
  type 'a mchannel and 'a port

  val new_mchannel : unit         -> 'a mchannel
  val new_port     : 'a mchannel -> 'a port
  val receive      : 'a port     -> 'a event
  val multicast    : 'a mchannel -> 'a          -> unit
end
```

The operation `new_port` generates a fresh port where a message can be received.
The (non-blocking) operation `multicast` sends to all registered ports.

```
module Multicast = struct open Thread open Event
  module M = Mailbox

  type 'a port = 'a M.mbox

  type 'a mchannel = 'a channel * 'a port channel

  let new_port (_, req) = let m = M.new_mailbox () in
                          sync (send req m); m

  let multicast (send_ch, _) x = sync (send send_ch x)

  let receive port = M.receive port
...
```

The operation `multicast` sends the message on channel `send_ch`. The Operation `receive` reads from the mailbox of the port.

The multicast channel's server thread maintains the list of ports:


```
...
let new_mchannel () = let send_ch = new_channel () in
  let req = new_channel () in
  let send_port x port = M.send port x in
  let rec serve ports = select [
      wrap (Event.receive req) (fun p -> serve (p :: ports));
      wrap (Event.receive send_ch) (fun x ->
          let _ = create (List.iter (send_port x)) ports in
          serve ports);
    ]
  in
  let _ = create serve [] in
  (send_ch, req)
...
```

Note that the server thread must respond both to port requests over the channel `req` and to send requests over `send_ch`.

Caveat

Our implementation supports addition, but not removal of obsolete ports.
For an example run, we use a test expression `main`:

```
...
  let main = let mc = new_mchannel () in
    let thread i = let p = new_port mc in
      while true do
        let x = sync (receive p) in
        print_int i; print_string ": ";
        print_string (x ^ "\n")
      done
    in
    let _ = create thread 1 in
    let _ = create thread 2 in
    let _ = create thread 3 in
    delay 1.0;
    multicast mc "Hello!";
    multicast mc "World!";
    multicast mc "... the end.";
    delay 10.0
end
```

We obtain

```
- ./a.out
3: Hello!
2: Hello!
1: Hello!
3: World!
2: World!
1: World!
3: ... the end.
2: ... the end.
1: ... the end.
```

Summary

- The programming language OCaml offers convenient possibilities to orchestrate concurrent programs.
- Channels with synchronous communication allow to simulate other concepts of concurrency such as asynchronous communication, global variables, locks for mutual exclusion and semaphors.
- Concurrent functional programs can be as obfuscated and incomprehensible as concurrent Java programs.
- Methods are required in order to systematically verify the correctness of such programs ...

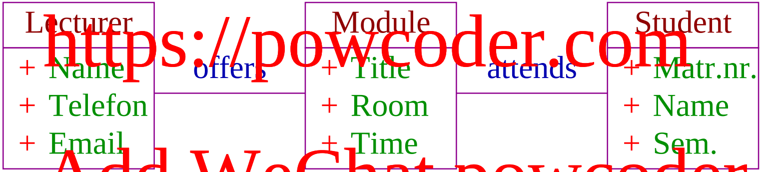
Perspectives

- Beyond the language concepts discussed in the lecture, OCaml has diverse further concepts, which also enable object oriented programming.
- Moreover, OCaml has elegant means to access functionality of the operating system, to employ graphical libraries and to communicate with other computers ...

$\Longrightarrow$ OCaml is an interesting alternative to Java.

9 Datalog: Computing with Relations

Example 1: The Study Program of a TU



$\Longrightarrow$ entity-relationship diagram

Discussion

- Many application domains can be described by entity-relationship diagrams.
- Entities in the example: lecturer, module, student.
- The set of all occurring entities, i.e., of all instances can be decribed by a table ...

Lecturer:

Name	Telefon	Email
Esparza	17204	esparza@in.tum.de
Nipkow	17302	nipkow@in.tum.de
Seidl	18155	seidl@in.tum.de

Module:

Title	Room	Time
Discrete Structures	MI 1	Thu 12:15-13, Fri 10-11:45
Pearls of Informatics III	MI 3	Thu 8:30-10
Funct. Programming and Verification	MI 1	Tue 16-18
Optimization	MI 2	Mon 12-14, Di 12-14

Student:

Matr.nr.	Name	Sem.
123456	Hans Dampf	03
007042	Fritz Schnurri	11
543345	Anna Blume	03
131175	Effi Briest	05

Discussion (cont.)

- The rows correspond to the instances.
- The columns correspond to the attributes.
- Assumption: the first attribute identifies the instance

  $\Longrightarrow$ primary key

  Consequence: Relationships are tables as well ...

offers:

Name	Title
Esparza	Discrete Structures
Nipkow	Pearls of Informatics III
Seidl	Funct. Programming and Verification
Seidl	Optimization

attends:

Matr.nr.	Title
123456	Funct. Programming and Verification
123456	Optimization
123456	Discrete Structures
543345	Funct. Programming and Verification
543345	Discrete Structures
131175	Optimization

Possible Queries

- In which semester are students attending the module "Discrete Structures" ?
- Who attends a module of lecturer "Seidl" ?
- Who attends both "Discrete Structures" and "Funct. Programming and Verification" ?

$\Longrightarrow$ Datalog

Idea: Table $\Longleftrightarrow$ Relation

A relation R is a set of tuples, i.e.,

$$R \subseteq \mathcal{U}_1 \times \ldots \times \mathcal{U}_n$$

where $\mathcal{U}_i$ is the set of all possible values for the ith component. In our example, there are:

`int`, `string`, possibly enumeration types

// unary relations represent sets.

Relations can be described by predicates ...

Predicates can be defined by enumeration of facts ...

... in the Example


```
offers ("Esparza", "Discrete Structures").
offers ("Nipkow", "Pearls of Informatics III").
offers ("Seidl", "Funct. Programming and Verification").
offers ("Seidl", "Optimization").

attends (123456, "Optimization").
attends (123456, "Funct. Programming and Verification").
attends (123456, "Discrete Structures").
attends (543345, "Funct. Programming and Verification").
attends (543345, "Discrete Structures").
attends (131175, "Optimization").
```

Rules can be used to deduce further facts ...

... in the Example


```
has_attendant (X,Y) :- offers (X,Z), attends (M,Z),
                       student (M,Y,_).
semester (X,Y) :- attends (Z,X), student (Z,_,Y).
```


- `:-` represents the logical implication "$\Leftarrow$".
- The comma-separated list collects the assumptions.
- The left-hand side, the head of the rule, represents the conclusion.
- Variables start with a capital letter.
- The anonymous variable `_` refers to irrelevant values.

The knowledge base consisting of facts and rules now can be queried ...

... in the Example


```
?- has_attendant ("Seidl", Z).
```


- Datalog finds all values for Z, so that the query can be deduced from the given facts by means of the rules.
- In our examples these are:


```
Z = "Hans Dampf"
Z = "Anna Blume"
Z = "Effi Briest"
```

Further Queries


```
?- semester ("Discrete Structures", X)
      X = 2
      X = 4

?- attends (X, "Funct. Programming and Verification"),
      attends (X, "Discrete Structures").
      X = 123456
      X = 543345
```

Caveat

A query may contain none, one or several variables.

An Example Proof

The rule

```
has_attendant (X,Y) :- offers (X,Z), attends (M,Z),
                                      student (M,Y,_).
```


holds for all `X`, `M`, `Y`, `Z`.

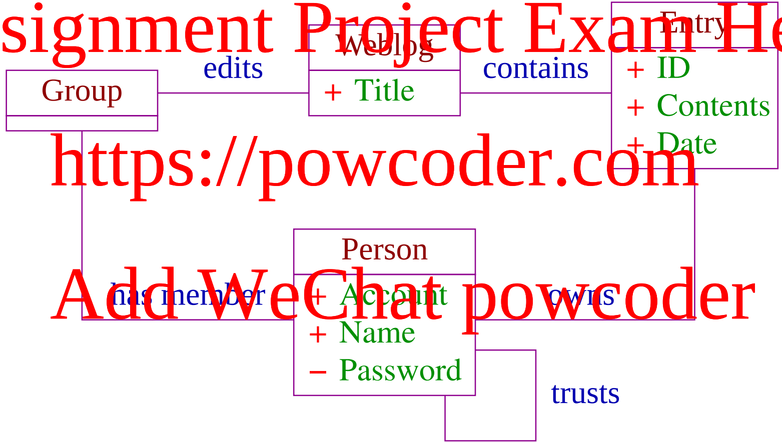
By means of the substitution


```
"Seidl"/X   "Funct. Programming ..."/Z
543345/M   "Anna Blume"/Y
```

we prove


```
offers ("Seidl", "Funct. Programming ...")
    attends (543345, "Funct. Programming ...")
        student (543345, "Anna Blume", 3)
-------------------------------------------------
has_attendant ("Seidl", "Anna Blume")
```

Example 2: A Weblog



Task: Specification of access rights

- Every member of the group of editors is entitled to add an entry.
- Only the owner of an entry is allowed to delete it.
- Everybody trusted by the owner, is entitled to modify.
- Every member of the group as well as everybody directly or indirectly trusted by a member of the group, is allowed to read ...

Specification in Datalog


```
may_add (X,W) :- edits (Z,W),
                 has_member (Z,X).
may_delete (X,E) :- owns (X,E).
may_modify (X,E) :- owns (X,E).
may_modify (X,E) :- owns (Y,E),
                    trusts (Y,X).
may_read (X,E) :- contains (W,E),
                  may_add (X,W).
may_read (X,E) :- may_read (Y,E),
                  trusts (Y,X).
```

Remark

- All available predicates or even fresh auxiliary predicates can be used for the definition of new predicates.
- Apparently, predicate definitions may be recursive.
- Together with a person `X` owning an entry, also all persons are entitled to modify trusted by `X`.
- Together with a person `Y` entitled to read, also all persons are entitled to read trusted by `Y`.

9.1 Answering a Query

Given: a set of facts and rules

Wanted: the set of all provable facts

Problem

```
equals (X,X).
```


$\Longrightarrow$ the set of all provable facts is infinite.

Theorem

Assume that W is a finite set of facts and rules with the following properties:

(1) Facts do not contain variables.

(2) Every variable in the head, also occurs in the body.

Then the set of provable facts is finite.

Proof Sketch

For every provable fact `p(a1,...,ak)`, it is shown that each constant `ai` already occurs in W.

Calculation of All Provable Facts

Successively compute the sets $R^{(i)}$ of all facts having proofs of depth at most i ...

$$R^{(0)} = \emptyset \qquad R^{(i+1)} = \mathcal{F}(R^{(i)})$$

where the operator $\mathcal{F}$ is defined by

$$\begin{array}{lll} \mathcal{F}(M) & = & \{h[\underline{a}/\underline{X}] \mid \exists\, h \,\text{:-}\, l_1, \ldots, l_k. \in W : \\ & & \quad l_1[\underline{a}/\underline{X}], \ldots, l_k[\underline{a}/\underline{X}] \in M\} \end{array}$$

// $[\underline{a}/\underline{X}]$ a substitution of the variables $\underline{X}$

// k can be equal to 0.

We have: $R^{(i)} = \mathcal{F}^i(\emptyset) \subseteq \mathcal{F}^{i+1}(\emptyset) = R^{(i+1)}$

The set R of all implied facts is given by

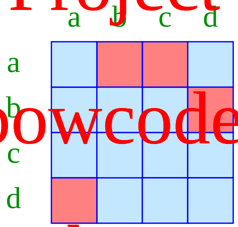
$$R = \bigcup_{i \geq 0} R^{(i)} = R^{(n)}$$

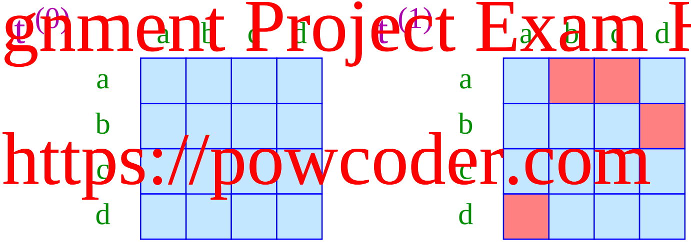
for a suitable n — since R is finite.

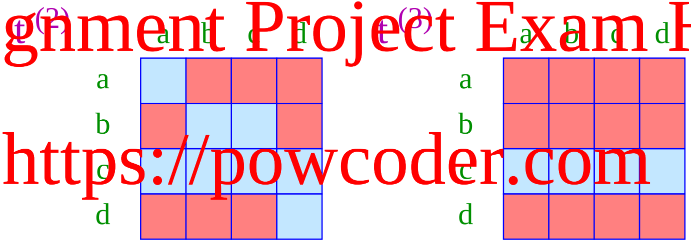
Example


```
edge (a,b).
edge (a,c).
edge (b,d).
edge (d,a).
t (X,Y) :- edge (X,Y).
t (X,Y) :- edge (X,Z), t (Z,Y).
```

Relation edge :




$t^{(0)}$
a b c d
a
b
c
d
$t^{(1)}$
a b c d
a
b
c
d


$t^{(2)}$
a b c d
a
b
c
d
$t^{(3)}$
a b c d
a
b
c
d

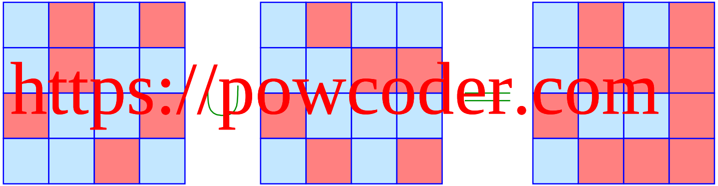
Discussion

- Our considerations are strong enough to calculate all facts implied by a Datalog program.
- From that, the set of answer substitutions can be extracted.
- The naive approach, however, is hopelessly inefficient.
- Smarter approaches try to avoid multiple calculations of the ever identical same facts ...
- In particular, only those facts need be proven which are useful for answering the query $\Longrightarrow$ compiler construction, databases

9.2 Operations on Relations

- We use predicates in order to describe relations.
- There are natural operations on relations which we would like to express in Datalog, i.e., define for predicates.

1. Union



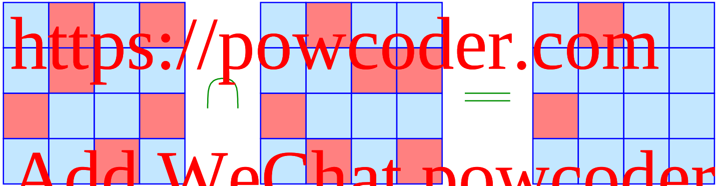
... in Datalog:

$$
\begin{aligned}
r(X_1, \ldots, X_k) &\;\text{:-}\; s_1(X_1, \ldots, X_k).\\
r(X_1, \ldots, X_k) &\;\text{:-}\; s_2(X_1, \ldots, X_k).
\end{aligned}
$$

Example

```
attends_Esparza_or_Seidl (X) :- has_attendant ("Esparza", X).
attends_Esparza_or_Seidl (X) :- has_attendant ("Seidl", X).
```

2. Intersection



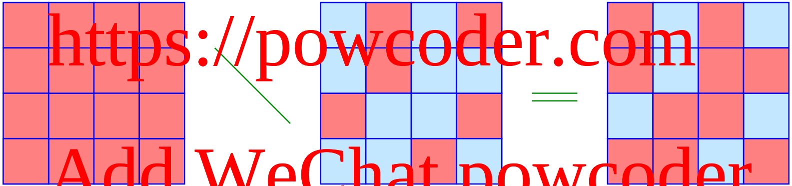
... in Datalog:

$$\begin{aligned} r(X_1, \ldots, X_k) \;\; :- \;\; & s_1(X_1, \ldots, X_k), \\ & s_2(X_1, \ldots, X_k). \end{aligned}$$

Example

```
attends_Esparza_and_Seidl (X) :- has_attendant ("Esparza", X),
                                 has_attendant ("Seidl", X).
```

3. Relative Complement



... in Datalog:

$$r(X_1, \ldots, X_k) \;\texttt{:-}\; s_1(X_1, \ldots, X_k), \texttt{not}(s_2(X_1, \ldots, X_k)).$$

i.e., $r(a_1, \ldots, a_k)$ follows when $s_1(a_1, \ldots, a_k)$ holds but $s_2(a_1, \ldots, a_k)$ is not provable.

Example


```
does_not_attend_Seidl (X) :- student (_, X, _),
                             not (has_attendant ("Seidl", X)).
```

Caveat

The query


```
p("Hello!").
?- not (p(X)).
```

results in infinitely many answers.

$\Longrightarrow$ we allow negated literals only if all occurring variables have already occurred to the left in non-negated literals.


```
p("Hello!").
q("Damn ...").
?- q(X), not (p(X)).
   X = "Damn ..."
```

Caveat (cont.)

Negation is only meaningful when s does not recursively depend on r ...


```
p(X) :- not (p(X)).
```


... is not easy to interpret.

$\Longrightarrow$ We allow $\texttt{not}(s(\ldots))$ only in rules for predicates r of which s is independent.

$\Longrightarrow$ stratified negation

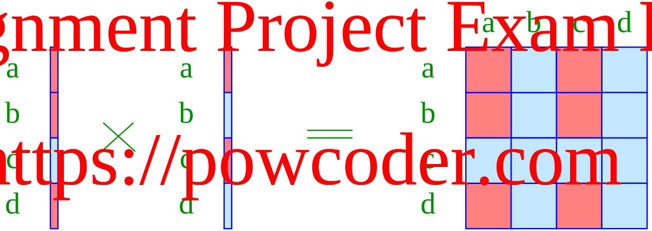
// Without recursive predicates, every negation is stratified.

4. Cartesian Product

$$S_1 \times S_2 = \{(a_1, \ldots, a_k, b_1, \ldots, b_m) \mid \ (a_1, \ldots, a_k) \in S_1, \ (b_1, \ldots, b_m) \in S_2 \ \}$$

... in Datalog:

$$r(X_1, \ldots, X_k, Y_1, \ldots, Y_m) \ \ :- \ \ s_1(X_1, \ldots, X_k), s_2(Y_1, \ldots, Y_m).$$


a
b
c
d
a
b
c
d
a b c d
a
b
c
d

Example


```
lecturer_student (X,Y) :- lecturer (X,_,_),
                          student (_,Y,_).
```


Comments

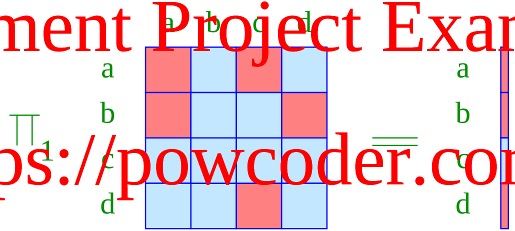
- The product of independent relations is very expensive.
- It should be avoided whenever possible ;-)

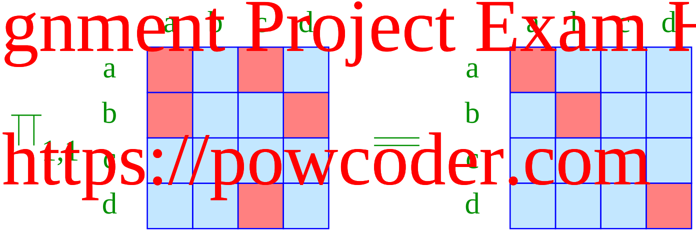
5. Projection

$$\pi_{i_1,\ldots,i_k}(S) = \{(a_{i_1}, \ldots, a_{i_k}) \mid (a_1, \ldots, a_m) \in S\}$$

... in Datalog:

$$r(X_{i_1}, \ldots, X_{i_k}) \quad \texttt{:-} \quad s(X_1, \ldots, X_m).$$


a b c d
a
b
c
d
$\prod_1$
=
a
b
c
d


a b c d
a
b
c
d
$\prod_{1,1}$
=
a b c d
a
b
c
d

6. Join

$$S_1 \bowtie S_2 = \{(a_1, \ldots, a_k, b_1, \ldots, b_m) \mid \begin{array}{l} (a_1, \ldots, a_{k+1}) \in S_1, \\ (b_1, \ldots, b_m) \in S_2, \\ \quad a_{k+1} = b_1 \end{array} \}$$

... in Datalog:

$$r(X_1, \ldots, X_k, Y_1, \ldots, Y_m) \quad :- \quad s_1(X_1, \ldots, X_k, Y_1), s_2(Y_1, \ldots, Y_m).$$

Discussion

Joins can be defined by means of the other operations ...

$$\begin{aligned} S_1 \bowtie S_2 \quad = \quad & \pi_{1,\ldots,k,k+2,\ldots,k+1+m}\,(\\ & S_1 \times S_2 \;\cap \\ & \mathcal{U}^k \times \pi_{1,1}(\mathcal{U}) \times \mathcal{U}^{m-1}) \end{aligned}$$

// For simplicity, we have assumed that $\mathcal{U}$ is the
// joint universe of all components.

Joins often allow to avoid expensive cartesian products.

The presented operations on relations form the basis of Relational Algebra ...

Background

Relational Algebra ...

$+$ is the basis underlying the query languages of Relational Databases $\Longrightarrow$ SQL

$+$ allows optimization of queries.
Idea: Replace expensive sub-expressions of the query with cheaper expressions of the same semantics !

$-$ is rather cryptic

$-$ does not support recursive definitions.

Example

The Datalog predicate


```
semester (X,Y) :- attends (Z,X), student (Z,_,Y)
```


... can be expressed in SQL by


```
SELECT attends.Title, Student.Semester
FROM   attends, Student
WHERE  attends.Matrikelnumber = Student.Matrikelnumber
```

Perspective

- Besides a query language, a realistic database language must also offer the possibility for insertion / modification / deletion.
- The implementation of a database must be able to handle not just toy applications like our examples, but to deal with gigantic mass data !!!
- It must be able to reliably execute multiple concurrent transactions without messing up individual tasks.
- A database also should be able to survive power supply failures.

$\Longrightarrow$ Database Lecture