

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
 - ⇒ formal process model (Software Engineering)
- proof of correctness
 - ⇒ verification

Tool: assertions

Example

```
1  public class GCD {
2      public static void main (String[] args) {
3          int x, y, a, b;
4          a = read(); x = a;
5          b = read(); y = b;
6          while (x != y)
7              if (x > y) x = x - y;
8              else     y = y - x;
9
10         assert(x == y);
11
12         write(x);
13     } // End of definition of main();
14 }    // 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 **!**

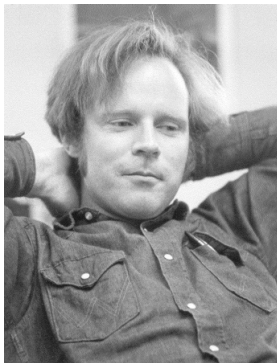
Problem

- 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.



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

1.1 Program Verification



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 ...

\implies **logic**

1.2 Background: Logic

Assertion: “All humans are mortal”,
“Socrates is a human”, “Socrates is mortal”

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

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

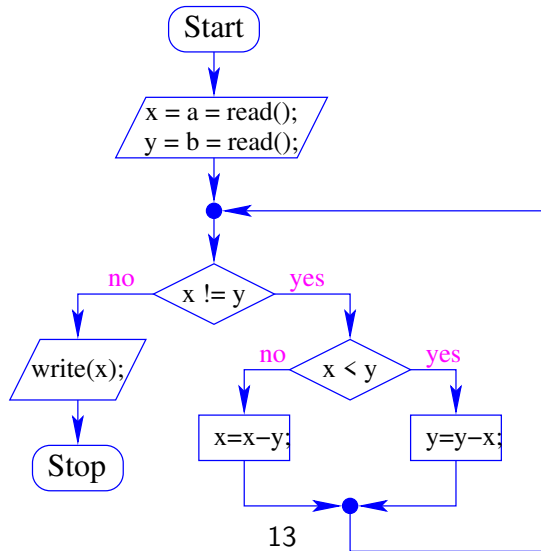
$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$A \vee (B \wedge A) \equiv A$$

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:

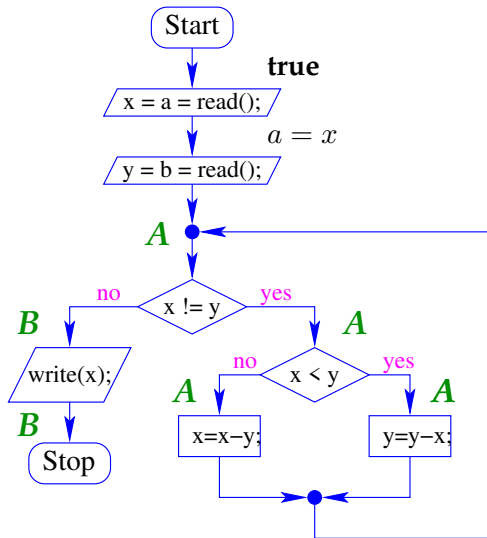
$$A \equiv \gcd(a, b) = \gcd(x, y)$$

- At program exit, we should have:

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

- These assertions should be **locally consistent** ...

Our Example



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

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

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:

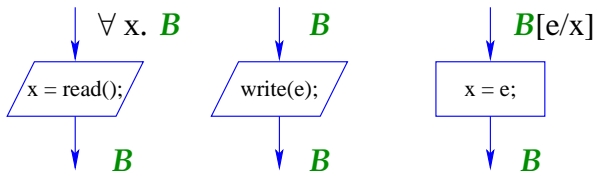
$$\begin{aligned} A[x - y/x] &\equiv \gcd(a, b) = \gcd(x - y, y) \\ &\equiv \gcd(a, b) = \gcd(x, y) \\ &\equiv A \end{aligned}$$

... 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

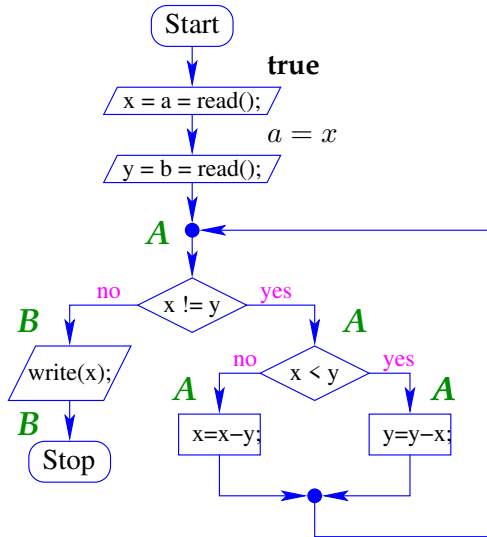


$$\begin{aligned}\mathbf{WP}[\text{;}] (B) &\equiv B \\ \mathbf{WP}[x = e;] (B) &\equiv B[e/x] \\ \mathbf{WP}[x = \text{read}();] (B) &\equiv \forall x. B \\ \mathbf{WP}[\text{write}(e);] (B) &\equiv B\end{aligned}$$

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

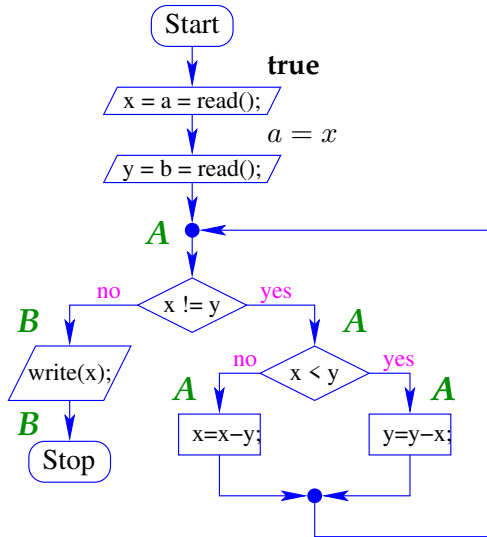


For the statements: `b = read(); y = b;` we calculate:

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

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

Orientation

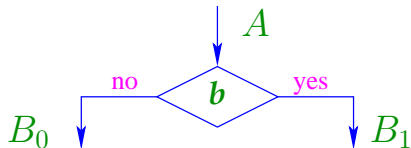


For the statements: `a = read(); x = a;` we calculate:

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

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

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

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

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

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

Then the weakest pre-condition is given by:

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

...for the GCD Example

$$b \equiv y > x$$

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

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

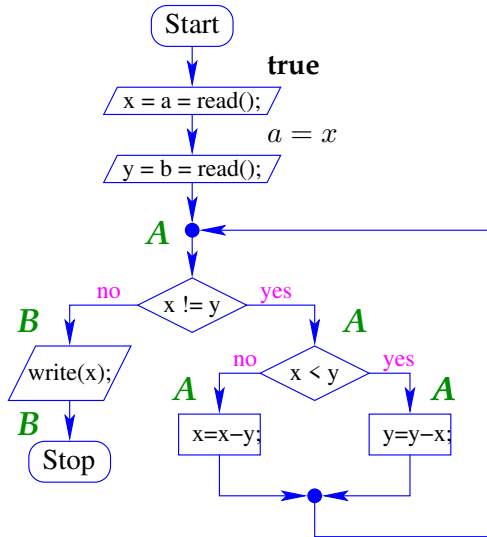


The weakest pre-condition is given by

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

... i.e., exactly A

Orientation



The argument for the assertion before the loop is analogous:

$$b \equiv y \neq x$$

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

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

$\implies A \equiv (A \wedge (x = y \vee x \neq y))$ 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

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

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

Trivial Properties

$\sigma \models \mathbf{true}$ for every σ
 $\sigma \models \mathbf{false}$ for no σ

$\sigma \models A_1$ 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)

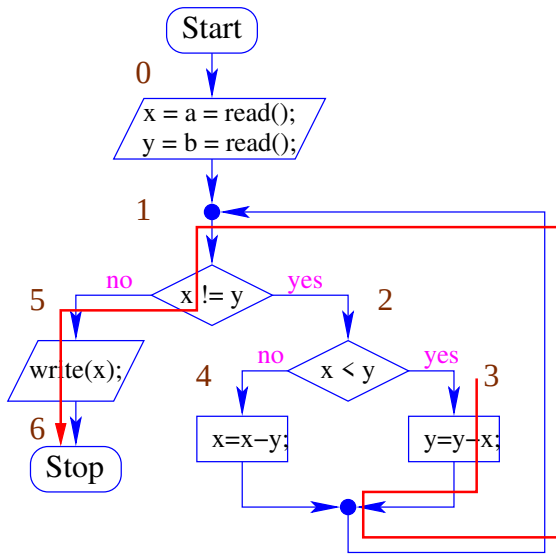
- An execution trace π traverses a path in the control-flow graph.
- It starts in a program point u_0 with an initial state σ_0 and leads to a program point u_m with a final state σ_m .
- Every step of the execution trace performs an action and (possibly) changes program point and state.

\implies The trace π can be represented as a sequence

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

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

Example



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

Then we obtain the following **execution trace**:

$$\begin{aligned} \pi = & (\mathbf{3}, \{x \mapsto 6, y \mapsto 12\}) \quad y = y - x; \\ & (\mathbf{1}, \{x \mapsto 6, y \mapsto 6\}) \quad \neg(x \neq y) \\ & (\mathbf{5}, \{x \mapsto 6, y \mapsto 6\}) \quad \text{write}(x); \\ & (\mathbf{6}, \{x \mapsto 6, y \mapsto 6\}) \end{aligned}$$

Important operation: Update of of state

$$\sigma \oplus \{x \mapsto d\} = \{z \mapsto \sigma z \mid z \neq x\} \cup \{x \mapsto d\}$$

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

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

Assumption: $\sigma_0 \models A$.

Proof obligation: $\sigma_m \models B$.

Idea

Induction on the length m of the execution trace.

Base $m = 0$:

The endpoint of the execution equals the startpoint.

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

\implies the claim holds.

Important Notion: Evaluation of Expressions

Program State

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

Arithmetic Expression

$$t \equiv 2 * z + y$$

Evaluation

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

Proposition

For (arithmetic) expressions t, e ,

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

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

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

Proposition

$$\sigma \oplus \{x \mapsto \llbracket e \rrbracket \sigma\} \models t_1 < t_2 \quad \text{iff} \quad \sigma \models t_1[e/x] < t_2[e/x]$$

Proof

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

Proposition

for every formula A ,

$$\sigma \oplus \{x \mapsto \llbracket e \rrbracket \sigma\} \models A \quad \text{iff} \quad \sigma \models A[e/\mathbf{x}]$$

Proof

Induction on the structure of formula A \square

Induction Proof of Correctness (cont.)

Step $m > 0$:

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

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

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

First, we deal with statements.

Case 1. $s_m \equiv ;$

Then

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

$$\implies B' \Rightarrow B$$

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

Induction Proof of Correctness (cont.)

Case 2. $s_m \equiv \text{write}(e);$

Then

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

$\implies B' \Rightarrow B$

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

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

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

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

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

$\implies \sigma_m \models B \quad \square$

Induction Proof of Correctness (cont.)

Case 4. $s_m \equiv x = \text{read}();$

Then \blacksquare $\sigma_m = \sigma_{m-1} \oplus \{x \mapsto c\}$ for some $c \in \mathbb{Z}$

\blacksquare $\mathbf{WP}[\![x = \text{read}();]\!](B) \equiv \forall x. B$

$\implies B' \Rightarrow \forall x. B \Rightarrow B[c/x]$

$\implies \sigma_m \models B \quad \square$

Induction Proof of Correctness (cont.)

Step $m > 0$:

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

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

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

Finally, consider tests $s_m \equiv b$.

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

Induction Proof of Correctness (cont.)

Case 1. $\sigma_m \models b$

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

$\implies \sigma_m \models b \wedge (b \Rightarrow B)$

$\implies \sigma_m \models B \quad \square$

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

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

$\implies \sigma_m \models \neg b \wedge (\neg b \Rightarrow B)$

$\implies \sigma_m \models B \quad \square$

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

⇒ Logic, automated theorem proving

1.4 Optimization

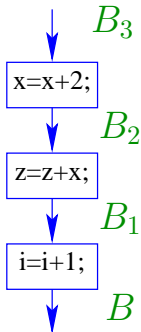
Goal: Reduction of the number of required assertions

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:

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

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

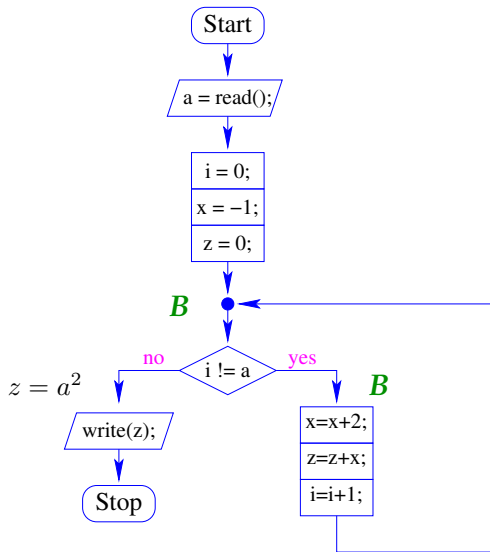
⇒ loop invariant

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

Example

```
1  int a, i, x, z;
2  a = read();
3  i = 0;
4  x = -1;
5  z = 0;
6
7  while (i != a) {
8      x = x + 2;
9      z = z + x;
10     i = i + 1;
11 }
12
13 assert(z == a * a);
14
15 write(z);
```

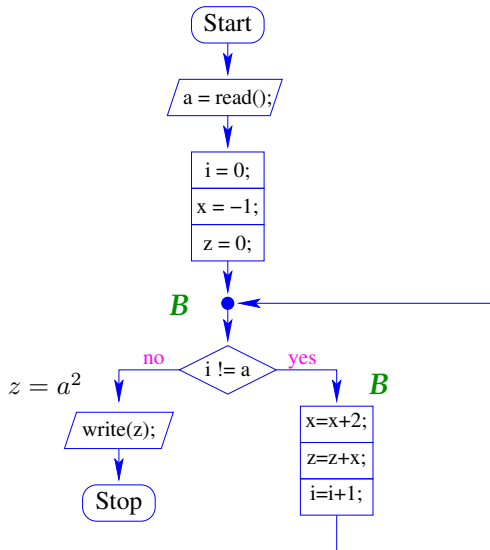
Example



We verify

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

Orientation



We verify:

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

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

```
1  int i, j, t;
2  t = 0;
3  i = read();
4  while (i > 0) {
5      j = read();
6      while (j > 0) { t = t + 1; j = j - 1; }
7      i = i - 1;
8  }
9  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

⇒ 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!

Example: Safe GCD Program

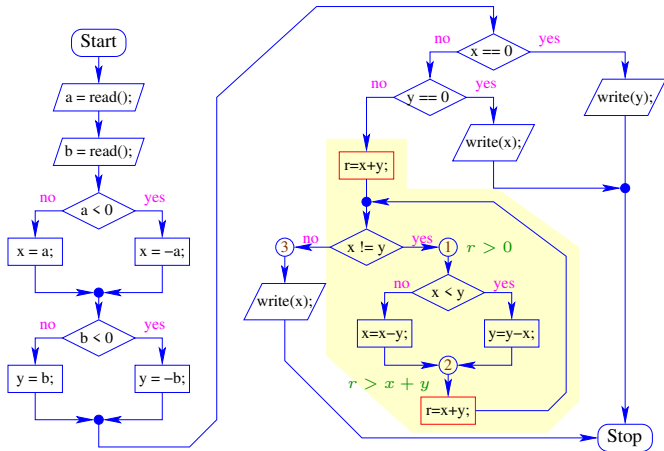
```
1  int a, b, x, y;
2  a = read(); b = read();
3  if (a < 0) x = -a; else x = a;
4  if (b < 0) y = -b; else y = b;
5  if (x == 0) write(y);
6  else if (y == 0) write(x);
7      else {
8          while (x != y)
9              if (y > x) y = y - x;
10             else      x = x - y;
11         write(x);
12     }
```

We choose: $r = x + y$

Transformation

```
1  int a, b, x, y, r;
2  a = read(); b = read();
3  if (a < 0) x = -a; else x = a;
4  if (b < 0) y = -b; else y = b;
5  if (x == 0) write(y);
6  else if (y == 0) write(x);
7      else { r = x + y;
8          while (x != y) {
9              if (y > x) y = y - x;
10             else      x = x - y;
11             r = x + y; }
12      write(x);
13 }
```

Orientation



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

- (1) $A \equiv x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$
- (2) $B \equiv x > 0 \wedge y > 0 \wedge r > x + y$
- (3) **true**

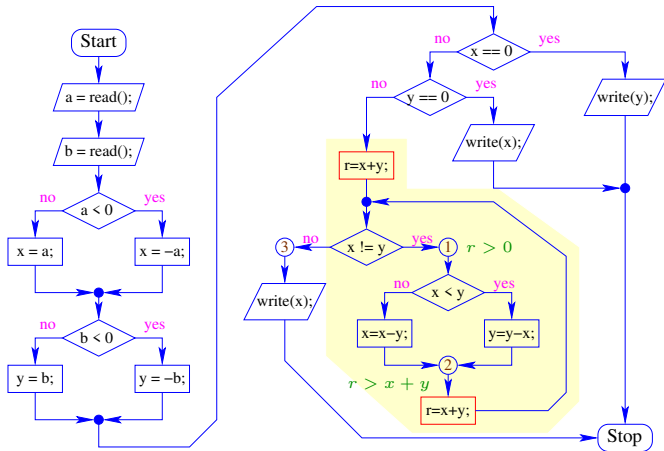
Then we have:

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

We verify:

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

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.
 \implies the program terminates!

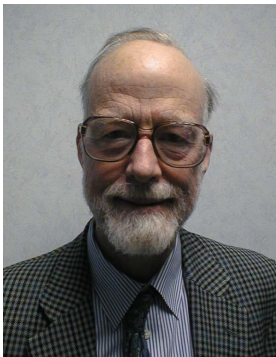
General Method

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

```
1  r = e0;  
2  while(b) {  
3      assert(r > 0);  
4      s;  
5      assert(r > e1);  
6      r = e1;  
7  }
```

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

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

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

$\{x > 7\}$ $\text{while } (x \neq 0) \ x = x - 1;$ $\{x = 0\}$

$\{\text{true}\}$ $\text{while } (\text{true});$ $\{\text{false}\}$

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

```
1  int a, b;  
2  int x, y;  
3  
4  void main () {  
5      a = read();  
6      b = read();  
7      mm();  
8      write (x - y);  
9  }
```

```
1  void mm() {  
2      if (a > b) {  
3          x = a;  
4          y = b;  
5      } else {  
6          y = a;  
7          x = b;  
8      }  
9  }
```

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:

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

Approach

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

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

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

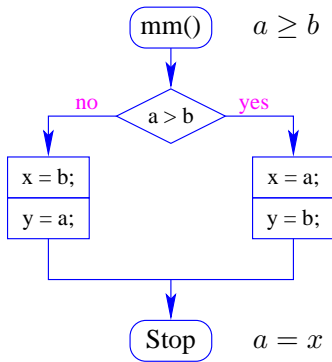
$$\{A\} \ \text{ss} \ \{B\}$$

holds.

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

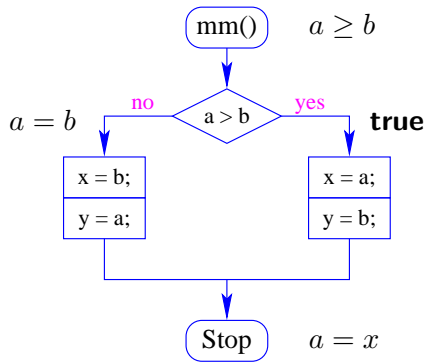
... in the Example

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 procedure is called in **multiple** places ...
- Even more complicated is the situation when a procedure is **recursive**: then it has possibly unboundedly many distinct calls **!?**

Example

```
1  int x, m0, m1, t;
2
3  void main () {
4      x = read();
5      m0 = 1; m1 = 1;
6      if (x > 1) f();
7      write (m1);
8  }
```

```
1  void f() {
2      x = x-1;
3      if (x > 1)
4          f();
5      t = m1;
6      m1 = m0 + m1;
7      m0 = t;
8  }
```


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 $\text{fib}(i - 1)$ and $\text{fib}(i)$, respectively ...

Problem

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

In the Example

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

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

General Approach

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

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

// both A and B may contain l_i

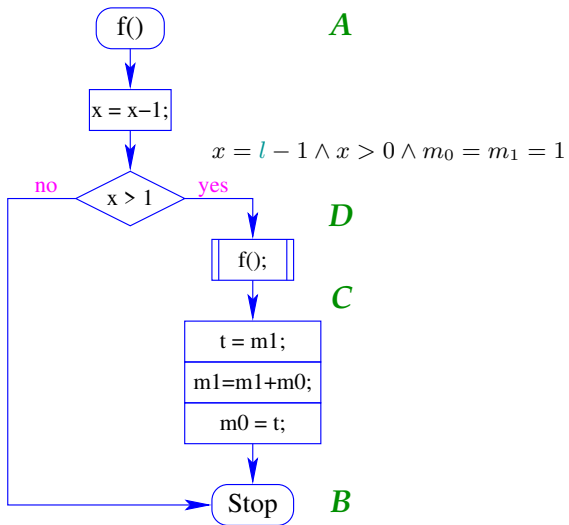
for each call of $\text{f}()$;

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

$$\{A\} \text{ ss } \{B\}$$

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 $\mathbf{WP}[\dots]$ and weakening ...

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

Question

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

Idea

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

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

// \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 $\{x = l\} \text{ double}(); \{x = 2l\}$

Table: $\forall l. (h = l) \Rightarrow (x = 2l)$
 $\equiv (x = 2h)$

For the Fibonacci function, we calculate:

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

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

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

Example: `double()`

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

We verify:

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

Remarks

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

- by substituting logical variables:

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

- by adding a condition C to the logical variables:

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

Remarks (cont.)

Valid pairs (A_1, B_1) are also obtained,

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

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

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

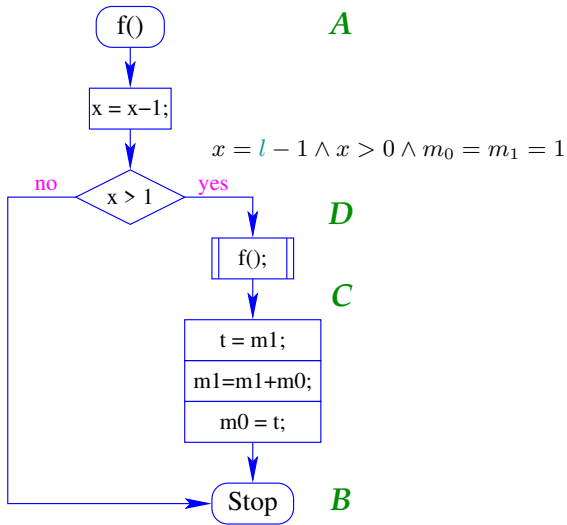
Application to Fibonacci

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

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

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

Orientation



For the conditional, we verify:

$$\mathbf{WP}[\![x>1]\!] (B, D) \equiv (x \leq 1 \wedge l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}) \vee \\ (x > 1 \wedge x = l - 1 \wedge m_1 = m_0 = 1)$$

$$\Leftarrow x > 0 \wedge x = l - 1 \wedge m_0 = m_1 = 1$$

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

```
1  {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 \underline{h} are only used for **global** variables !!
- As a new specific instance of adaptation, we obtain:

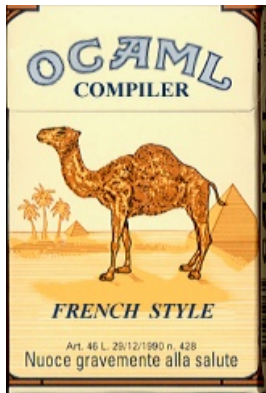
$$\frac{\{A\} \text{ f } (); \{B\}}{\{A \wedge C\} \text{ f } (); \{B \wedge C\}}$$

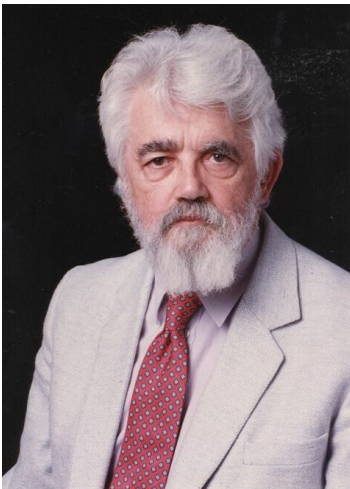
if C only speaks about logical variables or local variables of the caller.

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 \implies 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`.

```
1 seidl@linux:~> ocaml
2           OCaml version 4.14.0
3   ...
4   #
```

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

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

2.2 Expressions

```
1  # 3+4;;  
2  - : int = 7  
3  # 3+  
4    4;;  
5  - : int = 7  
6  #
```

- 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	= < > <= >= >

```

1  # -3.0 /. 4.0;;
2  - : float = -0.75
3  # "So" ^ " " ^ "it" ^ " " ^ "goes";;
4  - : string = "So it goes"
5  # 1 > 2 || not (2.0 < 1.0);;
6  - : 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!**

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

```
1  # let seven = 42;;  
2  val seven : int = 42  
3  # seven;;  
4  - : int = 42  
5  # let seven = "seven";;  
6  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

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

- Tuples

```
1  # (2, 3, 4, 5);;  
2  - : int * int * int * int = (2, 3, 4, 5)  
3  # ("hello", true, 3.14159);;  
4  -: string * bool * float = ("hello", true, 3.14159)
```

Simultaneous Definition of Variables

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

The latter use, though, will beforehand trigger the [warning](#):

this pattern-matching is not exhaustive.

Records:

Example

```
1  # type person = {given:string; sur:string; age:int};;
2  type person = { given : string; sur : string; age : int; }
3
4  # let paul = { given="Paul"; sur="Meier"; age=24 };;
5  val paul : person = {given = "Paul"; sur = "Meier"; age = 24}
6
7  # let hans = { sur="kohl"; age=23; given="hans"};;
8  val hans : person = {given = "hans"; sur = "kohl"; age = 23}
9
10 # let hans_i = {age=23; sur="kohl"; given="hans"};;
11 val hans_i : person = {given = "hans"; sur = "kohl"; age = 23}
12
13 # hans = hans_i;;
14 - : 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

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

... with pattern matching

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

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

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


Case Distinction: `match` and `if`

```
1  match n
2  with 0 -> "null"
3       | 1 -> "one"
4       | _ -> "uncountable!"
5
6  match e
7  with true  -> e1
8       | false -> e2
```

The second example can also be written as

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

Watch out for redundant and incomplete matches!

```
1  # let n = 7;;
2  val n : int = 7
3
4  # match n with 0 -> "zero";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
7  1
Exception: Match_failure ("", 5, -13).
9
10 # match n
11     with 0 -> "zero"
12         | 0 -> "one"
13         | _ -> "uncountable!";;
Warning: this match case is unused.
15 - : string = "uncountable!"
```

2.5 Lists

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

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

```
1  # let mt = [];;
2  val mt : 'a list = []
3
4  # let l1 = 1::mt;;
5  val l1 : int list = [1]
6
7  # let l = [1;2;3];;
8  val l : int list = [1; 2; 3]
9
10 # let l = 1::2::3::[];;
11 val l : int list = [1; 2; 3]
```

Caveat

All elements must have the **same** type:

```
1  # 1.0 :: 1 :: [];;  
2  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

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

2.6 Definition of Functions

```
1  # let double x = 2 * x;;  
2  val double : int -> int = <fun>  
3  
4  # (double 3, double (double 1));;  
5  - : 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.

```
1  # let double = fun x -> 2 * x;;  
2  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**:

```
1  # let factor = 2;;
2  val factor : int = 2
3
4  # let double x = factor * x;;
5  val double : int -> int = <fun>
6
7  # let factor = 4;;
8  val factor : int = 4
9
10 # double 3;;
11 - : int = 6
```


Caveat

A function is a value:

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

Recursive Functions

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

```
1  # let rec fac n = if n < 2 then 1 else n * fac (n - 1);;
2  val fac : int -> int = <fun>
3
4  # let rec fib = fun x -> if x <= 1 then 1
5                          else fib (x - 1) + fib (x - 2);;
6  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**.

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

We combine their definitions by means of the keyword **and**.

Definition by Case Distinction

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

... can be shorter written as

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

Case distinction for several arguments

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

... can also be written as

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

Local Definitions

Definitions introduced by `let` may occur locally:

```
1  # let x = 5
2      in let sq = x * x
3          in sq + sq;;
4  - : int = 50
5
6  # let facit n = let rec
7      iter m yet = if m > n then yet
8                    else iter (m + 1) (m * yet)
9      in iter 2 1;;
10 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.,

<code>("diamonds", 10)</code>	<code>≡</code>	<code>diamonds ten</code>
<code>("clubs", 11)</code>	<code>≡</code>	<code>clubs jack</code>
<code>("gras", 14)</code>	<code>≡</code>	<code>gras ace</code>

Disadvantages

- Testing of the color requires a comparison of strings
→ inefficient!
- Representation of Jack as 11 is not intuitive
→ 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

```
1  # type color = Diamonds | Hearts | Gras | Clubs;;
2  type color = Diamonds | Hearts | Gras | Clubs
3  # type value = Seven | Eight | Nine | Jack | Queen | King |
4                Ten | Ace;;
5  type value = Seven | Eight | Nine | Jack | Queen | King |
6                Ten | Ace
7  # Clubs;;
8  - : color = Clubs
9  # let gras_jack = (Gras, Jack);;
10 val gras_jack : color * value = (Gras, Jack)
```

Advantages

- The representation is intuitive.
- Typing errors are recognized:

```
1  # (Culbs,Nine);;  
2  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:

```
1  # Clubs < Diamonds;;  
2  - : bool = false;;  
3  # Clubs > Diamonds;;  
4  - : bool = true;;
```

Pattern Matching on constructors:

```
1  # let is_trump = function  
2    | (Hearts, _)    -> true  
3    | (_, Jack)     -> true  
4    | (_, Queen)    -> true  
5    | (_, _)        -> false  
6  
7  val is_trump : color * value -> bool = <fun>
```

By that, e.g.,

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

Another useful function:

```
1  # let string_of_color = function
2    | Diamonds -> "Diamonds"
3    | Hearts   -> "Hearts"
4    | Gras     -> "Gras"
5    | Clubs    -> "Clubs";;
6  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:

```
1  # let takes c1 c2 = match (c1,c2) with
2    | ((f1,Queen),(f2,Queen)) -> f1 > f2
3    | ((_,Queen),_)          -> true
4    | (_,(_,Queen))          -> false
5    | ((f1,Jack),(f2,Jack))  -> f1 > f2
6    | ((_,Jack),_)           -> true
7    | (_,(_,Jack))           -> false
8    | ((Hearts,w1),(Hearts,w2)) -> w1 > w2
9    | ((Hearts,_),_)         -> true
10   | (_,(Hearts,_))          -> false
11   | ((f1,w1),(f2,w2))       -> if f1=f2 then w1 > w2
12                               else false;;
```

```

1      ...
2      # let take card2 card1 =
3          if takes card2 card1 then card2 else card1;;
4
5      # let trick card1 card2 card3 card4 =
6          take card4 (take card3 (take card2 card1));;
7
8      # trick (Gras,Ace) (Gras,Nine) (Hearts,Ten) (Clubs,Jack);;
9      - : color * value = (Clubs,Jack)
10     # trick (Clubs,Eight) (Clubs,King) (Gras,Ten)
11         (Clubs,Nine);;
12     - : color * value = (Clubs,King)

```

Sum Types

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

Example: Optional Values

```
1  type 'a option = None | Some of 'a
2  let is_some x = match x with
3    | Some _ -> true
4    | None -> false
5  ...
```



```
1  ...
2  let get x = match x with
3    | Some y -> y
4  let value x a = match x with
5    | Some y -> y
6    | None -> a
7  let map f x = match x with
8    | Some y -> Some (f y)
9    | None -> None
10 let join a = match a with
11   | Some a' -> a'
12   | 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`

...

```
1  # Some;;
2  The constructor Some expects 1 argument(s),
3  but is here applied to 0 argument(s)
4
5  # None;;
6  - : 'a option = None
7
8  # Some 10;
9  - : int option = Some 10
10
11 # let a = Some "Hello!";;
12 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

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

Datatypes can be recursive:

```
1  type sequence = End | Next of (int * sequence)
2
3  # Next (1, Next (2, End));;
4  - : sequence = Next (1, Next (2, End))
```

Note the similarity to lists!

A corresponding polymorphic type could be

```
1  type 'a sequence = End | Next of ('a * 'a sequence)
2
3  # Next (1, Next (2, End));;
4  - : int sequence = Next (1, Next (2, End))
```

Recursive datatypes lead to recursive functions:

```
1  # let rec nth n s = match (n, s) with
2    | (_, End)           -> None
3    | (0, Next (x, _))    -> Some x
4    | (n, Next (_, rest)) -> nth (n-1) rest;;
5  val nth : int -> 'a sequence -> 'a option = <fun>
6
7  # nth 4 (Next (1, Next (2, End))));;
8  - : int = None
9
10 # nth 2 (Next (1, Next (2, Next (5, Next (17, End))))));;
11 - : int = Some 5
```

Another Example

```
1  # let rec down = function
2    | 0 -> End
3    | n -> Next (n, down (n-1));;
4  val down : int -> int sequence = <fun>
5
6  # down 3;;
7  - : int sequence = Next (3, Next (2, Next (1, End)));;
8
9  # down (-1);;
10 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 ...

```
1  let f x = x + 5
2
3  let g y = let z = 7
4            in if y > 5 then f (-y)
5            else z + f y
```

The first call is **tail**, the second is not.

⇒ From a tail call, we need not return to the calling function.

⇒ 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

```
1  let fac x = let rec facit n acc =
2      if n <= 1 then acc
3      else facit (n - 1) (n * acc)
4  in facit x 1
5
6  let rec loop x = if x < 2 then x
7      else if x mod 2 = 0 then loop (x / 2)
8      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

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

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

```
1  []
2  [n-1]
3  [n-1; n-2]
4  ...
5  [n-1; ...; 1]
```

as first argument \implies quadratic running-time!

Reversing a List – Version 2

```
1  let rev list =  
2    let rec r2 a l =  
3      match l  
4      with [] -> a  
5          | x::xs -> r2 (x::a) xs  
6  in r2 [] list
```

- ⇒ The local function r2 is tail-recursive !
- ⇒ it runs in linear running-time !!

3.2 Higher Order Functions

Consider the two functions

```
1  let f (a, b) = a + b + 1
2
3  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**:

```
1  # f;;
2  - : int * int -> int = <fun>
3
4  # g;;
5  - : 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` :

```
1  # f (3, 5);;
2  - : int = 9
3
4  # let g1 = g 3;;
5  val g1 : int -> int = <fun>
6
7  # g1 5;;
8  - : 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:

```
1  # let curry f a b = f (a,b);;  
2  val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>  
3  ...
```



```
1  ...
2
3  # let plus (x,y) = x+y;;
4  val plus : int * int -> int = <fun>
5
6  # curry plus;;
7  - : int -> int -> int = <fun>
8
9  # let plus2 = curry plus 2;;
10 val plus2 : int -> int = <fun>
11
12 # let plus3 = curry plus 3;;
13 val plus3 : int -> int = <fun>
14
15 # plus2 (plus3 4);;
16 - : int = 9
```

3.3 Some List Functions

```
1  let rec map f = function
2    [] -> []
3    | x::xs -> f x :: map f xs
4
5  let rec fold_left f a = function
6    [] -> a
7    | x::xs -> fold_left f (f a x) xs
8
9  let rec fold_right f = function
10   [] -> fun b -> b
11   | x::xs -> fun b -> f x (fold_right f xs b)
```

```
1  let rec find_opt f = function
2    [] -> None
3    | x::xs -> if f x then Some x
4    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:

```
1  map : ('a -> 'b) -> 'a list -> 'b list
2
3  fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
4
5  fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
6
7  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:

```
1  # string_of_int;;
2  val : int -> string = <fun>
3
4  # map string_of_int;;
5  - : int list -> string list = <fun>
6
7  # fold_left (+);;
8  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:

```
1  # let cons_r xs x = x::xs;;
2  val cons_r : 'a list -> 'a -> 'a list = <fun>
3
4  # let rev l = fold_left cons_r [] l;;
5  val rev : 'a list -> 'a list = <fun>
6
7  # rev [1; 2; 3];;
8  - : int list = [3; 2; 1]
9
10 # rev [true; false; false];;
11 - : bool list = [false; false; true]
```

Some of the Simplest Polymorphic Functions

```
1  let compose f g x = f (g x)
2  let twice f x      = f (f x)
3  let rec iter f g x = if g x then x else iter f g (f x);;
4
5  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
6  val twice   : ('a -> 'a) -> 'a -> 'a = <fun>
7  val iter    : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a = <fun>
8
9  # compose not not;;
10 - : bool -> bool = <fun>
11
12 # compose not not true;;
13 - : bool = true;;
14
15 # compose Char.chr plus2 65;;
16 - : char = 'C'
```

3.5 Polymorphic Datatypes

User-defined datatypes may be polymorphic as well:

```
1  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:


```
1  # Leaf 1;;
2  - : int tree = Leaf 1
3
4  # Node (Leaf ('a', true), Leaf ('b', false));;
5  - : (char * bool) tree = Node (Leaf ('a', true), Leaf ('b', false))
```

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

```
1  let rec size = function
2    | Leaf _      -> 1
3    | Node(t,t') -> size t + size t'
4
5  let rec flatten = function
6    | Leaf x      -> [x]
7    | Node(t,t') -> flatten t @ flatten t'
8
9  let flatten1 t = let rec doit t xs = match t with
10    | Leaf x      -> x :: xs
11    | Node(t,t') -> let xs = doit t' xs
12                      in doit t xs
13    in doit t []
14
15  ...
```

```
1  ...
2
3  val size : 'a tree -> int = <fun>
4  val flatten : 'a tree -> 'a list = <fun>
5  val flatten1 : 'a tree -> 'a list = <fun>
6
7  # let t = Node (Node (Leaf 1, Leaf 5), Leaf 3);;
8  val t : int tree = Node (Node (Leaf 1, Leaf 5), Leaf 3)
9
10 # size t;;
11 - : int = 3
12
13 # flatten t;;
14 val : int list = [1;5;3]
15
16 # flatten1 t;;
17 val : int list = [1;5;3]
```

3.6 Application: Queues

Wanted

Datastructure 'a queue which supports the operations

```
1  enqueue : 'a -> 'a queue -> 'a queue
2
3  dequeue : 'a queue -> 'a option * 'a queue
4
5  is_empty : 'a queue -> bool
6
7  queue_of_list : 'a list -> 'a queue
8
9  list_of_queue : 'a queue -> 'a list
```

First Idea

- Represent the queue by a list:

```
1  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:

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

- Insertion means append:

```
1  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 !!!

```
1  type 'a queue = Queue of 'a list * 'a list
2
3  let is_empty = function
4      Queue ([], []) -> true
5      | _             -> false
6
7  let queue_of_list list = Queue (list, [])
8
9  let list_of_queue = function
10     Queue (first, []) -> first
11     | Queue (first, last) ->
12         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:

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

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

```
1  let dequeue = function  
2      Queue ([],last) -> (match List.rev last  
3          with [] -> (None, Queue ([], []))  
4          | x::xs -> (Some x, Queue (xs, [])))  
5      | 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 !!!

⇒ 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:

```
1  # fun x y z -> x + y + z;;  
2  - : 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 ...

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

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

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

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

```
1  # let make_undefined () = fun x -> None;;  
2  val make_undefined : unit -> 'a -> 'b option = <fun>  
3  
4  # let def_one (x,y) = fun x' -> if x = x' then Some y else None;;  
5  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.

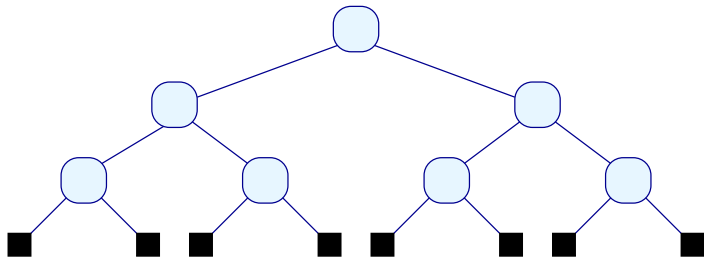
Wanted:

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

```
1  insert :      'a -> 'a d -> 'a d
2  delete :      'a -> 'a d -> 'a d
3  extract_min : 'a d -> 'a option * 'a d
4  extract_max : 'a d -> 'a option * 'a d
5  extract :      'a * 'a -> 'a d -> 'a list * 'a d
6  list_of_d :    'a d -> 'a list
7  d_of_list :    'a list -> 'a d
```

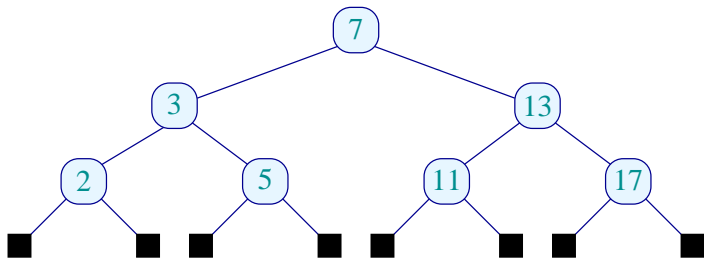

First Idea

Use **balanced** trees ...



First Idea

Use **balanced** trees ...



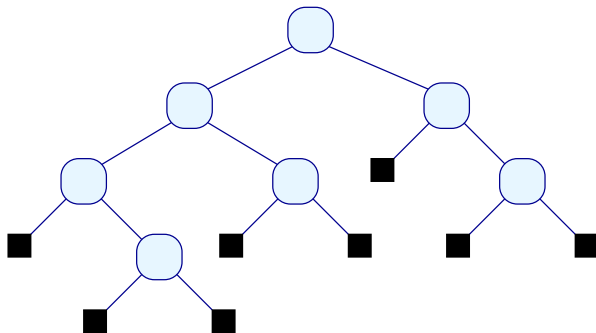
Discussion

- 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 ???

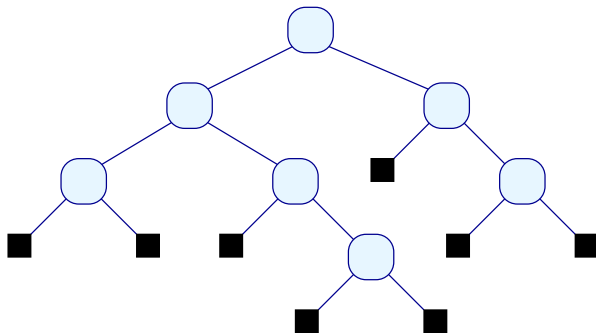
Second Idea

- 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 ...

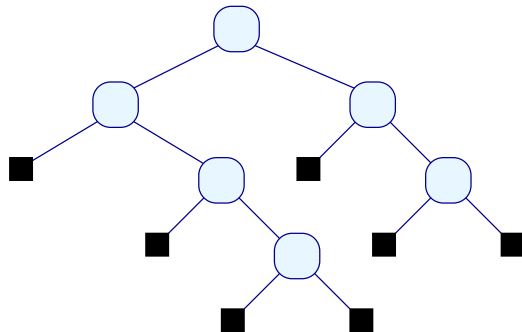
An AVL Tree



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 $k > 0$...

$$k = 1 :$$

$$N(1) = 1 = \text{fib}(1) = A^0$$

$$k = 2 :$$

$$N(2) = 2 = \text{fib}(2) \geq A^1$$

$$k > 2 :$$

Assume that the assertion holds for $k - 1$ and $k - 2 \dots$

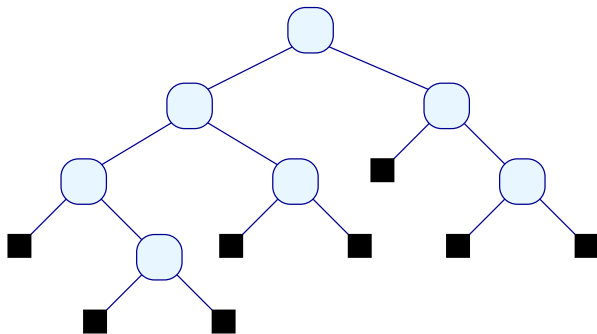
$$\begin{aligned} \implies N(k) &= N(k-1) + N(k-2) + 1 \\ &\geq \text{fib}(k-1) + \text{fib}(k-2) \\ &= \text{fib}(k) \end{aligned}$$

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

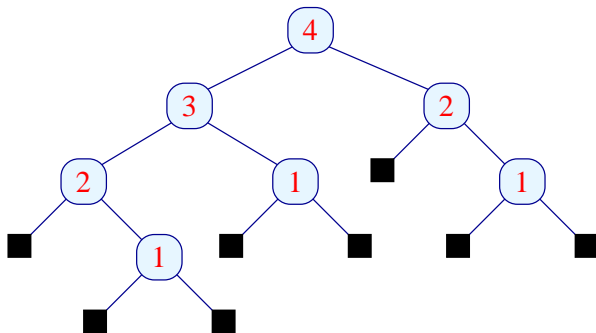
Second Idea (cont.)

- 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

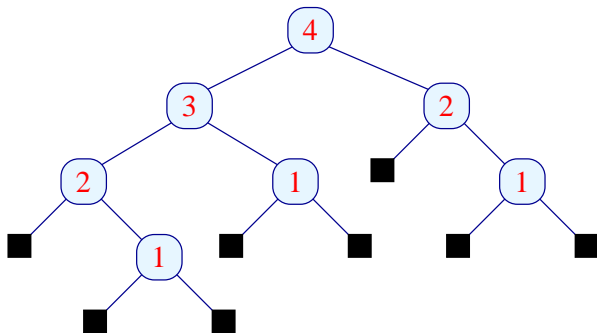


Third Idea

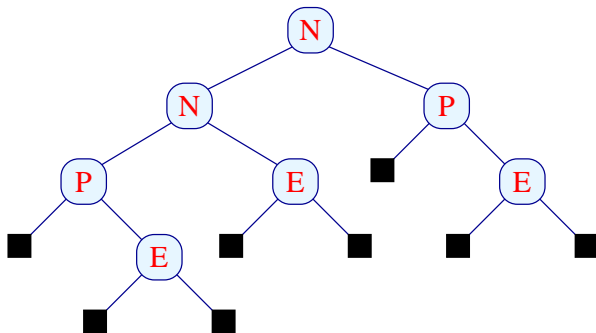
- Instead of the **absolute** depth, we store at each node only whether the difference in depth of the two subtrees is negative, positive or equal to zero !!!
- As datatype, we therefore define

```
1  type 'a avl = Null
2      | Neg of 'a avl * 'a * 'a avl
3      | Pos of 'a avl * 'a * 'a avl
4      | 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 is 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 ...

```

1  let rec insert x avl = match avl
2      with Null          -> (Eq (Null,x,Null), true)
3      | Eq (left,y,right) -> if x < y then
4          let (left,inc) = insert x left
5          in if inc then (Neg (left,y,right), true)
6          else          (Eq (left,y,right), false)
7      else let (right,inc) = insert x right
8          in if inc then (Pos (left,y,right), true)
9          else          (Eq (left,y,right), false)
10         ...

```

- 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.

```

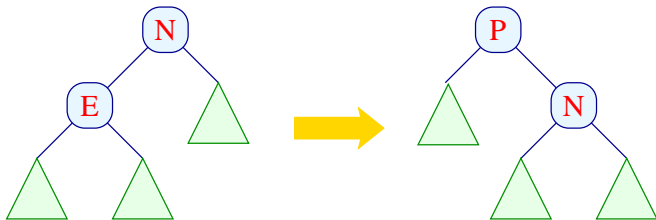
1  | Neg (left,y,right) -> if x < y then
2      let (left,inc) = insert x left
3      in if inc then let (avl,_) = rotateRight (left,y,right)
4          in (avl,false)
5      else          (Neg (left,y,right), false)
6  else let (right,inc) = insert x right
7      in if inc then (Eq (left,y,right), false)
8      else          (Neg (left,y,right), false)
9  | Pos (left,y,right) -> if x < y then
10      let (left,inc) = insert x left
11      in if inc then (Eq (left,y,right), false)
12      else          (Pos (left,y,right), false)
13  else let (right,inc) = insert x right
14      in if inc then let (avl,_) = rotateLeft (left,y,right)
15          in (avl,false)
16      else          (Pos (left,y,right), false);;

```

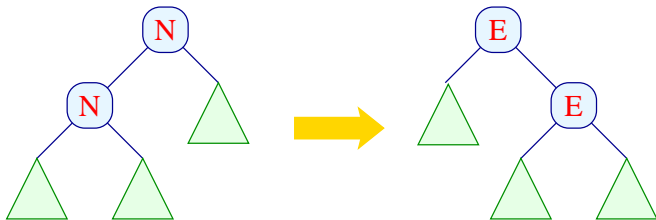
Comments

- 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 ...

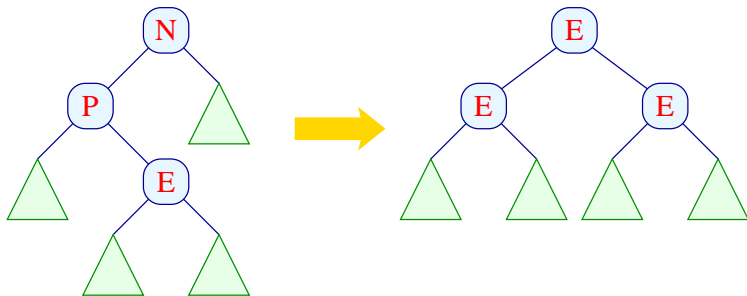
rotateRight



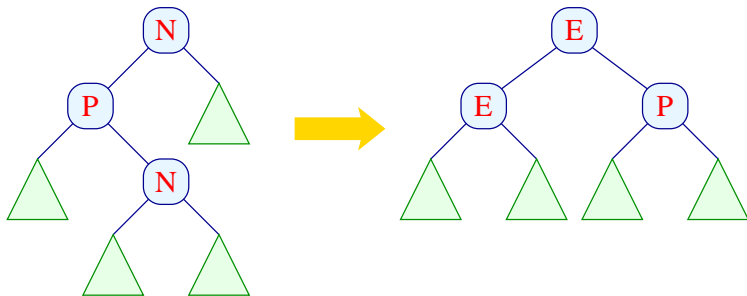
rotateRight



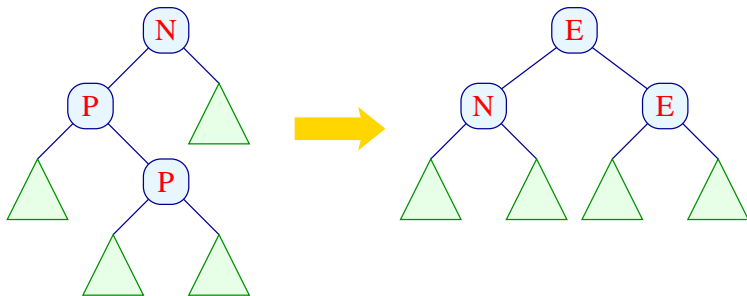
rotateRight



rotateRight



rotateRight



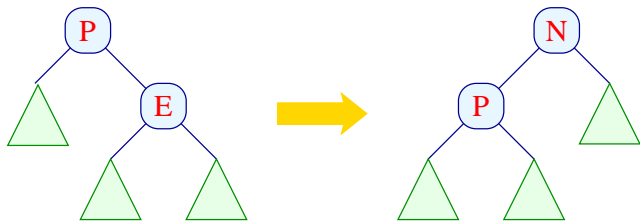
```

1  let rotateRight (left, y, right) = match left
2      with Eq (l1,y1,r1) -> (Pos (l1, y1, Neg (r1,y,right)), false)
3      |   Neg (l1,y1,r1) -> (Eq (l1, y1, Eq (r1,y,right)), true)
4      |   Pos (l1, y1, Eq (l2,y2,r2)) ->
5          (Eq (Eq (l1,y1,l2), y2, Eq (r2,y,right)), true)
6      |   Pos (l1, y1, Neg (l2,y2,r2)) ->
7          (Eq (Eq (l1,y1,l2), y2, Pos (r2,y,right)), true)
8      |   Pos (l1, y1, Pos (l2,y2,r2)) ->
9          (Eq (Neg (l1,y1,l2), y2, Eq (r2,y,right)), true)

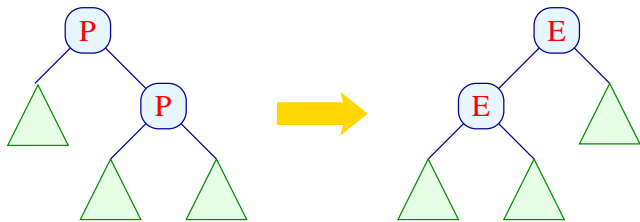
```

- 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.

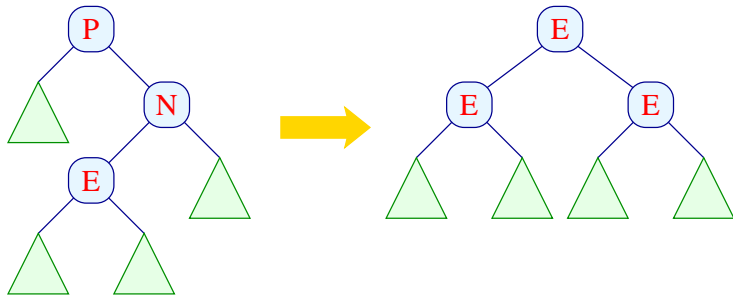
rotateLeft



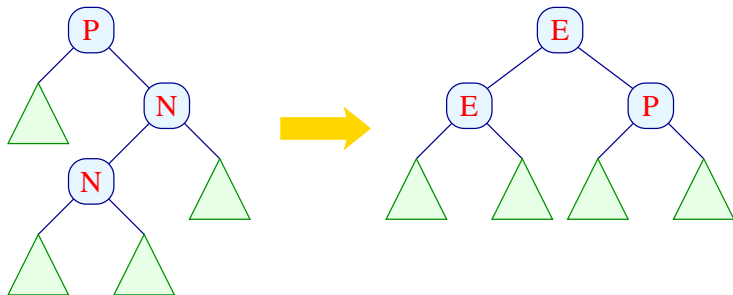
rotateLeft



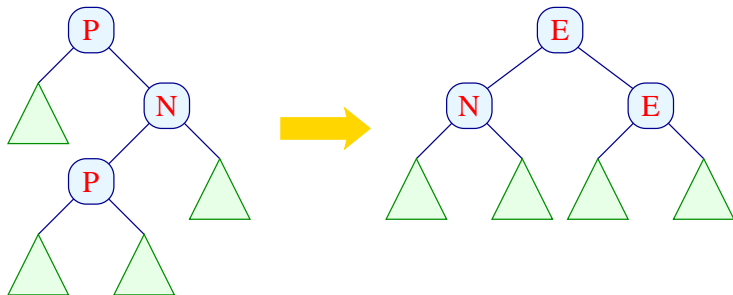
rotateLeft



rotateLeft



rotateLeft




```

1  let rotateLeft (left, y, right) = match right
2      with Eq (l1,y1,r1) -> (Neg (Pos (left,y,l1), y1, r1), false)
3      |   Pos (l1,y1,r1) -> (Eq  (Eq  (left,y,l1), y1, r1), true)
4      |   Neg (Eq  (l1,y1,r1), y2 ,r2) ->
5          (Eq (Eq (left,y,l1),y1,  Eq (r1,y2,r2)), true)
6      |   Neg (Neg (l1,y1,r1), y2 ,r2) ->
7          (Eq (Eq (left,y,l1),y1, Pos (r1,y2,r2)), true)
8      |   Neg (Pos (l1,y1,r1), y2 ,r2) ->
9          (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 constant multiple to $\log(n)$.
- In general, though, we are not interested in the extra bit at every call. Therefore, we define:

```
1  let insert x tree = let (tree,_) = insert x tree
2                        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 ...

```

1  let rec extract_min avl = match avl
2      with Null          -> (None, Null, false)
3      | Eq (Null,y,right) -> (Some y, right, true)
4      | Eq (left,y,right) -> let (first,left,dec) = extract_min left
5                              in if dec then (first, Pos (left,y,right), false)
6                              else          (first, Eq (left,y,right), false)
7      | Neg (left,y,right) -> let (first,left,dec) = extract_min left
8                              in if dec then (first, Eq (left,y,right), true)
9                              else          (first, Neg (left,y,right), false)
10     | Pos (Null,y,right) -> (Some y, right, true)
11     | Pos (left,y,right) -> let (first,left,dec) = extract_min left
12                             in if dec then let (avl,b) = rotateLeft (left,y,right)
13                                         in (first,avl,b)
14                             else          (first, Pos (left,y,right), false)

```

Discussion

- Rotation is only required when extracting from a tree of the form $\text{Pos}(\dots)$ 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  # 1 / 0;;  
2  Exception: Division_by_zero.  
3  
4  # List.tl (List.tl [1]);;  
5  Exception: Failure "tl".  
6  
7  # Char.chr 300;;  
8  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**:

```
1  # match 1+1 with 0 -> "null";;  
2  Warning: this pattern-matching is not exhaustive.  
3  Here is an example of a value that is not matched:  
4  1  
5  Exception: Match_failure ("", 2, -9).
```

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

Pre-defined Constructors for Exceptions

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

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

```
1  # Division_by_zero;;
2  - : exn = Division_by_zero
3
4  # Failure "complete nonsense!";;
5  - : exn = Failure "complete nonsense!"
```

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

```
1  # exception Hell;;
2  exception Hell
3
4  # Hell;;
5  - : exn = Hell
```

```
1  # Division_by_zero;;
2  - : exn = Division_by_zero
3
4  # Failure "complete nonsense!";;
5  - : exn = Failure "complete nonsense!"
```

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

```
1  # exception Hell of string;;
2  exception Hell of string
3
4  # Hell "damn!";;
5  - : exn = Hell "damn!"
```

Handling of Exceptions

As in **Java**, exceptions can be raised and handled:

```
1  # let divide (n,m) = try Some (n / m)
2      with Division_by_zero -> None;;
3
4  # divide (10,3);;
5  - : int option = Some 3
6  # divide (10,0);;
7  - : int option = None
```

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

```

1  let rec member x l = try if x = List.hd l then true
2                          else member x (List.tl l)
3      with Failure _ -> false
4
5  # member 2 [1;2;3];;
6  - : bool = true
7  # member 4 [1;2;3];;
8  - : bool = false

```

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

```

1  try <exn>
2  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  # 1 + (2 / 0);;  
2  Exception: Division_by_zero.  
3  
4  # 1 + raise Division_by_zero;;  
5  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:

```
1  # let f (x, y) = x / (y - 1);;
2  # let g (x, y) = try let n = try f (x,y)
3                      with Division_by_zero ->
4                      raise (Failure "Division by zero")
5                      in string_of_int (n * n)
6                      with Failure str -> "Error: " ^ str;;
7
8  # g (6, 1);;
9  - : string = "Error: Division by zero"
10
11 # g (6, 3);;
12 - : 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
⇒ now, the ordering of the evaluation matters !!!

- Naturally, OCaml allows to write to standard output:

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

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

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

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

```
1  # let infile = open_in "test";;
2  val infile : in_channel = <abstr>
3
4  # input_line infile;;
5  - : string = "The file's single line ...";;
6
7  # input_line infile;;
8  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** ...

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

Further Useful Values

```
1  stdin          : in_channel
2  input_char     : in_channel -> char
3  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 ...

```
1  # let outfile = open_out "test";;
2  val outfile : out_channel = <abstr>
3
4  # output_string outfile "Hello ";;
5  - : unit = ()
6
7  # output_string outfile "World!\n";;
8  - : unit = ()
9  ...
```

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

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

5.3 Sequences

In presence of side-effects, ordering matters!

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

```
1  # print_string "Hello";  
2  print_string " ";  
3  print_string "world!\n";;  
4  Hello world!  
5  - : unit = ()
```

Often, several strings must be output !

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

```
1  # let rec iter f = function
2      []      -> ()
3      | x::[] -> f x
4      | x::xs -> f x; iter f xs;;
5
6  val iter : ('a -> unit) -> 'a list -> unit = <fun>
7
8  # iter print_string ["Hello "; "world";"!\\n"];;
9  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**:

```
1  module Pairs =  
2      struct  
3          type 'a pair      = 'a * 'a  
4          let pair (a,b)    = (a,b)  
5          let first (a,b)   = a  
6          let second (a,b)  = b  
7      end
```


On this input, the compiler answers with the type of the module, its [signature](#):

```
1  module Pairs :  
2    sig  
3      type 'a pair = 'a * 'a  
4      val pair : 'a * 'b -> 'a * 'b  
5      val first : 'a * 'b -> 'a  
6      val second : 'a * 'b -> 'b  
7    end
```

The definitions inside the module are **not visible** outside:

```
1  # first;;  
2  Unbound value first
```

Access onto Components of a Module

Components of a module can be accessed via qualification:

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

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

```
1  # module Triples =  
2      struct  
3          type 'a triple = Triple of 'a * 'a * 'a  
4          let first  (Triple (a, _, _)) = a  
5          let second (Triple (_, b, _)) = b  
6          let third  (Triple (_, _, c)) = c  
7      end;;  
8  ...
```

```
1  ...
2  module Triples :
3      sig
4          type 'a triple = Triple of 'a * 'a * 'a
5          val first : 'a triple -> 'a
6          val second : 'a triple -> 'a
7          val third : 'a triple -> 'a
8      end
9
10 # Triples.first;;
11 - : 'a Triples.triple -> 'a = <fun>
```

... or several implementations of the same function:

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

Opening Modules

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

```
1  # open Pairs2;;  
2  
3  # pair;;  
4  - : 'a * 'a -> bool -> 'a = <fun>  
5  
6  # pair (4,3) true;;  
7  - : int = 4
```

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

```
1  # module A = struct let x = 1 end;;
2  module A : sig val x : int end
3
4  # module B =
5      struct
6          open A
7          let y = 2
8      end;;
9  module B : sig val y : int end
10
11 # module C =
12     struct
13         include A
14         include B
15     end;;
16 module C : sig val x : int val y : int end
```

Nested Modules

Modules may again contain modules:

```
1  module Quads = struct
2      module Pairs =
3          struct
4              type 'a pair      = 'a * 'a
5              let pair (a,b) = (a,b)
6              let first  (a,_) = a
7              let second (_,b) = b
8          end
9
10         type 'a quad = 'a Pairs.pair Pairs.pair
11         let quad (a,b,c,d) =
12             Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
13     ...
```

```

1      ...
2      let first  q = Pairs.first (Pairs.first q)
3      let second q = Pairs.second (Pairs.first q)
4      let third  q = Pairs.first (Pairs.second q)
5      let fourth q = Pairs.second (Pairs.second q)
6  end
7
8  # Quads.quad (1, 2, 3, 4);;
9  - : (int * int) * (int * int) = ((1, 2), (3, 4))
10
11 # Quads.Pairs.first;;
12 - : '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

```

1  module Sort = struct
2      let single lst = map (fun x -> [x]) lst
3
4      let rec merge l1 l2 = match (l1, l2)
5          with ([], _) -> l2
6              | (_, []) -> l1
7              | (x::xs, y::ys) -> if x < y then x :: merge xs l2
8                                   else y :: merge l1 ys
9
10     let rec merge_lists = function
11         [] -> [] | [l] -> [l]
12         | l1::l2::l3 -> merge l1 l2 :: merge_lists l3
13
14     let sort lst = let lst = single lst
15                     in let rec doit = function
16                         [] -> [] | [l] -> l
17                         | l -> doit (merge_lists l)
18                     in doit lst
19 end

```

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

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

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

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

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

```
1  # module MySort : Sort = Sort;;  
2  module MySort : Sort  
3  
4  # MySort.single;;  
5  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 specialized:

```
1  module type A1 = sig
2    val f : 'a -> 'b -> 'b
3  end
4
5  module type A2 = sig
6    val f : int -> char -> int
7  end
8
9  module A = struct
10    let f x y = x
11  end
```

```

1  # module A1 : A1 = A;;
2  Signature mismatch:
3  Modules do not match: sig val f : 'a -> 'b -> 'a end
4                           is not included in A1
5  Values do not match:
6      val f : 'a -> 'b -> 'a
7  is not included in
8      val f : 'a -> 'b -> 'b
9
10 # module A2 : A2 = A;;
11 module A2 : A2
12
13 # A2.f;;
14 - : 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

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

A signature allows to hide the implementation of a queue:

```
1  module type Queue = sig
2      type 'a queue
3
4      val empty_queue : unit -> 'a queue
5
6      val is_empty : 'a queue -> bool
7
8      val enqueue : 'a queue -> 'a -> 'a queue
9
10     val dequeue : 'a queue -> 'a * 'a queue
11 end
```



```
1  # module Queue : Queue = ListQueue;;
2  module Queue : Queue
3
4  # open Queue;;
5
6  # is_empty [];;
7  This expression has type 'a list but is here used with type
8  'b queue = 'b Queue.queue
```



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:

```
1  module type Queue = sig
2    type 'a queue = Queue of ('a list * 'a list)
3
4    val empty_queue : unit -> 'a queue
5
6    val is_empty : 'a queue -> bool
7
8    val enqueue : 'a -> 'a queue -> 'a queue
9
10   val dequeue : 'a queue -> 'a option * 'a queue
11 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:

```
1  module type Decons = sig
2      type 'a t
3      val decons : 'a t -> ('a * 'a t) option
4  end
5
6  module type GenFold = functor (X : Decons) -> sig
7      val fold_left : ('b -> 'a -> 'b) -> 'b -> 'a X.t -> 'b
8      val fold_right : ('a -> 'b -> 'b) -> 'a X.t -> 'b -> 'b
9      val size : 'a X.t -> int
10     val list_of : 'a X.t -> 'a list
11     val iter : ('a -> unit) -> 'a X.t -> unit
12 end
13 ...
```

```

1  ...
2  module Fold : GenFold = functor (X : Decons) ->
3      struct
4          let rec fold_left f b t =
5              match X.decons t with None -> b |
6                  Some (x, t) -> fold_left f (f b x) t
7
8          let rec fold_right f t b =
9              match X.decons t with None -> b |
10                  Some (x, t) -> f x (fold_right f t b)
11
12         let size t = fold_left (fun a x -> a + 1) 0 t
13
14         let list_of t = fold_right (fun x xs -> x :: xs) t []
15
16         let iter f t = fold_left (fun () x -> f x) () t
17     end

```

Now, we can [apply](#) the functor to the module to obtain a new module ...

```

1  module MyQueue = struct open Queue
2      type 'a t = 'a queue
3
4      let decons = function
5          | Queue ([], xs)      -> (
6              match rev xs with [] -> None
7              | x :: xs -> Some (x, Queue (xs, [])))
8          | Queue (x :: xs, t) -> Some (x, Queue (xs, t))
9  end
10
11 module MyAVL = struct open AVL
12     type 'a t = 'a avl
13
14     let decons avl =
15         match extract_min avl with
16         | None, avl      -> None
17         | Some (a, avl) -> Some (a, avl)
18 end

```

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

By that, we may define

```
1  let sort list = FoldAVL.list_of (
2                                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

<code>Test.cmo</code>	bytecode for the module
<code>Test.cmi</code>	bytecode for the signature
<code>a.out</code>	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

$$\begin{aligned} E \quad ::= & \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid \\ & (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \\ & \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \\ & \text{fun name} \rightarrow E \mid E E_1 \end{aligned}$$
$$P \quad ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

Short-cut

$$\text{fun } x_1 \rightarrow \dots \text{fun } x_k \rightarrow e \quad \equiv \quad \text{fun } x_1 \dots x_k \rightarrow e$$

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 mutually recursive global definitions of variables f_1, \dots, f_m :

```
let rec   $f_1$   =   $E_1$   
      and   $f_2$   =   $E_2$   
      ...  
      and   $f_m$   =   $E_m$ 
```

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

$$\begin{aligned} V ::= & \text{const} \mid \text{fun name}_1 \dots \text{name}_k \rightarrow E \mid \\ & (V_1, \dots, V_k) \mid [] \mid V_1 :: V_2 \end{aligned}$$

A MiniOCaml Program ...

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

Examples of Values ...

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

$$(TU) \quad \frac{e_1 \Rightarrow v_1 \quad \dots \quad e_k \Rightarrow v_k}{(e_1, \dots, e_k) \Rightarrow (v_1, \dots, v_k)}$$

Lists

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

Global definitions

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

Local definitions

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

Function calls

$$(\text{APP}) \quad \frac{e \Rightarrow \text{fun } x \rightarrow e_0 \quad e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{e \ e_1 \Rightarrow v_0}$$

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

$$(\text{APP}') \quad \frac{e_0 \Rightarrow \text{fun } x_1 \dots x_k \rightarrow e \quad e_1 \Rightarrow v_1 \dots e_k \Rightarrow v_k \quad e[v_1/x_1, \dots, v_k/x_k] \Rightarrow v}{e_0 \ e_1 \ \dots \ e_k \Rightarrow v}$$

This derived rule makes proofs somewhat simpler.

Pattern Matching

$$(PM) \quad \frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \dots, v_k/x_k] \quad e_i[v_1/x_1, \dots, v_k/x_k] \Rightarrow v}{\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m \Rightarrow v}$$

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

Built-in operators

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

Unary operators are treated analogously.

The built-in equality operator

$$v = v \Rightarrow \text{true}$$

$$v_1 = v_2 \Rightarrow \text{false}$$

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

Example 1

$$\begin{array}{c} \text{(OP)} \frac{17 \Rightarrow 17 \quad 4 \Rightarrow 4 \quad 17 + 4 \Rightarrow 21}{17 + 4 \Rightarrow 21} \quad 21 \Rightarrow 21 \quad 21 = 21 \Rightarrow \text{true} \\ \text{(OP)} \frac{\quad}{17 + 4 = 21 \Rightarrow \text{true}} \end{array}$$

The built-in equality operator

$$v = v \Rightarrow \text{true}$$

$$v_1 = v_2 \Rightarrow \text{false}$$

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 $v \Rightarrow v$

$$\begin{array}{c} \text{(OP)} \frac{17 + 4 \Rightarrow 21}{17 + 4 \Rightarrow 21} \quad 21 = 21 \Rightarrow \text{true} \\ \text{(OP)} \frac{\quad}{17 + 4 = 21 \Rightarrow \text{true}} \end{array}$$

Example 2

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

$$\begin{array}{c} \text{(GD)} \frac{f = \text{fun } x \rightarrow x+1}{f \Rightarrow \text{fun } x \rightarrow x+1} \quad \text{(OP)} \frac{16+1 \Rightarrow 17}{16+1 \Rightarrow 17} \quad \text{(GD)} \frac{s = \text{fun } y \rightarrow y*y}{s \Rightarrow \text{fun } y \rightarrow y*y} \quad \text{(OP)} \frac{2*2 \Rightarrow 4}{2*2 \Rightarrow 4} \\ \text{(APP)} \frac{\quad}{f \ 16 \Rightarrow 17} \quad \text{(APP)} \frac{\quad}{s \ 2 \Rightarrow 4} \quad 17+4 \Rightarrow 21 \\ \text{(OP)} \frac{\quad}{f \ 16 + s \ 2 \Rightarrow 21} \end{array}$$

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

Example 3

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

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

$$\begin{array}{c}
\text{(GD)} \frac{\text{app} = \text{fun } x \ y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \ y \rightarrow \dots} \quad \text{(APP')} \quad \text{(PM)} \frac{\frac{[] \Rightarrow [] \quad 2::[] \Rightarrow 2::[]}{\text{match } [] \ \dots \Rightarrow 2::[]}}{\text{app } [] \ (2::[]) \Rightarrow 2::[]} \\
\text{(LI)} \frac{\text{app } [] \ (2::[]) \Rightarrow 2::[]}{1 :: \text{app } [] \ (2::[]) \Rightarrow 1::2::[]} \quad \text{(PM)} \frac{\text{match } 1::[] \ \dots \Rightarrow 1::2::[]}{\text{app } (1::[]) \ (2::[]) \Rightarrow 1::2::[]}
\end{array}$$

// 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

$\text{app } l_1 \ l_2$ terminates for all list values l_1, l_2 .

Proof

Induction on the length n of the list l_1 .

$n = 0$ i.e., $l_1 = []$. Then

$$\text{(APP')} \frac{\text{(GD)} \frac{\text{app} = \text{fun } x \ y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \ y \rightarrow \dots}}{\text{match } [] \text{ with } [] \rightarrow l_2 \mid \dots \Rightarrow l_2} \text{(PM)} \frac{}{\text{app } [] \ l_2 \Rightarrow l_2}$$

$n > 0 :$ i.e., $l_1 = h :: t$.

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

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

for some l . We deduce

$$\begin{array}{c}
 \text{(GD)} \frac{\text{app} = \text{fun } x \ y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \ y \rightarrow \dots} \quad \text{(PM)} \frac{\text{(LI)} \frac{\text{I.H.}}{\text{app } t \ l_2 \Rightarrow l}}{h :: \text{app } t \ l_2 \Rightarrow h :: l}}{\text{match } h :: t \text{ with } \dots \Rightarrow h :: l} \\
 \text{(APP')} \frac{}{\text{app } (h :: t) \ l_2 \Rightarrow h :: l}
 \end{array}$$

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

$$C ::= \text{const} \mid (C_1, \dots, C_k) \mid [] \mid C_1 :: C_2$$

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

$$c ::= \text{bool} \mid \text{int} \mid \text{unit} \mid c_1 * \dots * c_k \mid c \text{ list}$$

Discussion

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

```
1  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.



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

$$\frac{v_1 = v'_1 \ \dots \ v_k = v'_k}{(v_1, \dots, v_k) = (v'_1, \dots, v'_k)}$$

$$\frac{v_1 = v'_1 \quad v_2 = v'_2}{v_1 :: v_2 = v'_1 :: v'_2}$$

Functions

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

\implies extensional equality

We have:

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

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

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

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 :

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

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 repertoire of rewrite rules for reducing the equality of expressions to the equality of, possibly simpler expressions ...

Simplification of local definitions

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

Simplification of function calls

$$\frac{e_0 = \text{fun } x \rightarrow e \quad e_1 \text{ terminates}}{e_0 \ e_1 = e[e_1/x]}$$

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:

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

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

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

Accordingly,

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

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

$$\frac{e_0 = \text{fun } x_1 \dots x_k \rightarrow e \quad e_1, \dots, e_k \text{ terminate}}{e_0 \ e_1 \dots e_k = e[e_1/x_1, \dots, e_k/x_k]}$$

This derived rule allows to shorten some proofs considerably.

Rule for pattern matching

$$\frac{e_0 = []}{\text{match } e_0 \text{ with } [] \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m = e_1}$$

$$\frac{e_0 \text{ terminates} \quad e_0 = e'_1 :: e'_2}{\text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x :: xs \rightarrow e_2 = e_2[e'_1/x, e'_2/xs]}$$

We are now going to apply these rules ...

7.3 Equational Proofs for MiniOCaml

Example 1

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

We want to verify that

- (1) $\text{app } x \ [] = x$ for all lists x .
- (2) $\text{app } x \ (\text{app } y \ z) = \text{app } (\text{app } x \ y) \ z$
for all lists x, y, z .

Idea: Induction on the length n of x

$n = 0$ Then $x = []$ holds.

We deduce:

$$\begin{aligned} \text{app } x \ [] & \stackrel{\text{def } x}{=} \text{app } [] \ [] \\ & \stackrel{\text{app}}{=} \text{match } [] \text{ with } [] \rightarrow [] \mid h::t \rightarrow h :: \text{app } t \ [] \\ & \stackrel{\text{match}}{=} [] \\ & \stackrel{\text{def } x}{=} x \end{aligned}$$

$n > 0$

Then: $x = h :: t$ where t has length $n - 1$.

We deduce:

```
app x []  $\stackrel{\text{def } x}{=}$  app (h::t) []  
           $\stackrel{\text{app}}{=}$  match h::t with [] -> [] | h::t -> h :: app t []  
           $\stackrel{\text{match}}{=}$  h :: app t []  
           $\stackrel{\text{I.H.}}{=}$  h :: t  
           $\stackrel{\text{def } x}{=}$  x
```

Analogously we proceed for assertion (2) ...

$n = 0$

Then: $x = []$

We deduce:

$$\begin{aligned} \text{app } x \text{ (app } y \text{ } z) &\stackrel{\text{def } x}{=} \text{app } [] \text{ (app } y \text{ } z) \\ &\stackrel{\text{app}}{=} \text{match } [] \text{ with } [] \rightarrow \text{app } y \text{ } z \mid h::t \rightarrow \dots \\ &\stackrel{\text{match}}{=} \text{app } y \text{ } z \\ &\stackrel{\text{match}}{=} \text{app (match } [] \text{ with } [] \rightarrow y \mid \dots) \text{ } z \\ &\stackrel{\text{app}}{=} \text{app (app } [] \text{ } y) \text{ } z \\ &\stackrel{\text{def } x}{=} \text{app (app } x \text{ } y) \text{ } z \end{aligned}$$

$n > 0$

Then $x = h :: t$ where t has length $n - 1$.

We deduce:

$$\begin{aligned} \text{app } x \text{ (app } y \text{ } z) &\stackrel{\text{def } x}{=} \text{app } (h :: t) \text{ (app } y \text{ } z) \\ &\stackrel{\text{app}}{=} \text{match } h :: t \text{ with } [] \rightarrow \text{app } y \text{ } z \\ &\quad | h :: t \rightarrow h :: \text{app } t \text{ (app } y \text{ } z) \\ &\stackrel{\text{match}}{=} h :: \text{app } t \text{ (app } y \text{ } z) \\ &\stackrel{\text{I.H.}}{=} h :: \text{app } (\text{app } t \text{ } y) \text{ } z \\ &\stackrel{\text{match, app}}{=} \text{app } (h :: \text{app } t \text{ } y) \text{ } z \\ &\stackrel{\text{match}}{=} \text{app } (\text{match } h :: t \text{ with } [] \rightarrow [] \\ &\quad | h :: t \rightarrow h :: \text{app } t \text{ } y) \text{ } z \\ &\stackrel{\text{app}}{=} \text{app } (\text{app } (h :: t) \text{ } y) \text{ } z \\ &\stackrel{\text{def } x}{=} \text{app } (\text{app } x \text{ } y) \text{ } z \end{aligned}$$

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:

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

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

Example 2

```
1  let rec rev = fun x -> match x
2      with []      -> []
3      | h::t       -> app (rev t) [h]
4
5  let rec rev1 = fun x -> fun y -> match x
6      with []      -> y
7      | h::t       -> rev1 t (h::y)
```

Claim

$\text{rev } x = \text{rev1 } x []$ for all lists x .

More generally,

$\text{app } (\text{rev } x) \ y = \text{rev1 } x \ y$ for all lists x, y .

Proof: Induction on the length n of x

$n = 0$

Then: $x = []$. We deduce:

$$\begin{aligned} \text{app } (\text{rev } x) \ y &\stackrel{\text{def } x}{=} \text{app } (\text{rev } []) \ y \\ &\stackrel{\text{rev}}{=} \text{app } (\text{match } [] \text{ with } [] \rightarrow [] \mid \dots) \ y \\ &\stackrel{\text{match}}{=} \text{app } [] \ y \\ &\stackrel{\text{app, match}}{=} y \\ &\stackrel{\text{match}}{=} \text{match } [] \text{ with } [] \rightarrow y \mid \dots \\ &\stackrel{\text{rev1}}{=} \text{rev1 } [] \ y \\ &\stackrel{\text{def } x}{=} \text{rev1 } x \ y \end{aligned}$$

$n > 0$

Then $x = h::t$ where t has length $n - 1$.

We deduce (ommitting simple intermediate steps):

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

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:

$$\text{rev } x = \text{rev1 } x \ []$$

follows from:

$$\text{app } (\text{rev } x) \ y = \text{rev1 } x \ y$$

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

Example 3

```
1  let rec sorted = fun x -> match x
2      with h1::h2::t -> (
3          match h1 <= h2
4          with true  -> sorted (h2::t)
5              | false -> false)
6      | _           -> true
7
8  and merge = fun x -> fun y -> match (x,y)
9      with ([],y) -> y
10     | (x,[]) -> x
11     | (x1::xs, y1::ys) -> (
12         match x1 <= y1
13         with true  -> x1 :: merge xs y
14             | false -> y1 :: merge x ys)
```

Claim

$\text{sorted } x \wedge \text{sorted } y \rightarrow \text{sorted } (\text{merge } x \ y)$
for all lists x, y .

Proof: Induction on the **sum** n of lengths of x, y .

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

$n = 0$ Then: $x = [] = y$

We deduce:

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

$$n > 0$$

Case 1: $x = []$.

We deduce:

$$\begin{aligned} \text{sorted (merge } x \text{ } y) &\stackrel{\text{def } x}{=} \text{sorted (merge } [] \text{ } y) \\ &\stackrel{\text{merge}}{=} \text{sorted } y \\ &\text{by assumption} \\ &= \text{true} \end{aligned}$$

Case 2: $y = []$ analogous.

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

We deduce:

$$\begin{aligned} \text{sorted } (\text{merge } x \ y) &\stackrel{\text{def } x, y}{=} \text{sorted } (\text{merge } (x1 :: xs) \ (y1 :: ys)) \\ &\stackrel{x1 \leq y1}{=} \text{sorted } (x1 :: \text{merge } xs \ (y1 :: ys)) \\ &\stackrel{\text{def } y}{=} \text{sorted } (x1 :: \text{merge } xs \ y) \\ &= \dots \end{aligned}$$

Case 3.1: $xs = []$

We deduce:

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

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

In particular: $x1 \leq x2 \wedge \text{sorted } xs$.

We deduce:

```
...  $\stackrel{\text{def } xs}{=}$  sorted (x1 :: merge (x2::xs') y)
     $\stackrel{x2 \leq y1}{=}$  sorted (x1 :: x2 :: merge xs' y)
     $\stackrel{x1 \leq x2}{=}$  sorted (x2 :: merge xs' y)
     $\stackrel{\text{merge}}{=}$  sorted (merge xs y)
     $\stackrel{\text{I.H.}}{=}$  true
```

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

In particular: $x1 \leq y1 < x2 \wedge \text{sorted } xs$.

We deduce:

```
... def xs/y = sorted (x1 :: merge (x2::xs') (y1::ys))
    y1 < x2 = sorted (x1 :: y1 :: merge (x2::xs') ys)
    def xs = sorted (x1 :: y1 :: merge xs ys)
    x1 ≤ y1 = sorted (y1 :: merge xs ys)
    merge = sorted (merge xs y)
    I.H. = true
```

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

We deduce:

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

Case 4.1: $ys = []$

We deduce:

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

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

In particular: $y1 \leq y2 \wedge \text{sorted } ys$.

We deduce:

```
... def ys = sorted (y1 :: merge x (y2::ys'))  
    y2 < x1 = sorted (y1 :: y2 :: merge x ys')  
    y1 <= y2 = sorted (y2 :: merge x ys')  
    merge = sorted (merge x ys)  
    I.H. = true
```

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

In particular: $y1 < x1 \leq y2 \wedge \text{sorted } ys$.

We deduce:

```
... def x,ys = sorted (y1 :: merge (x1::xs) (y2::ys'))
x1 ≤ y2 = sorted (y1 :: x1 :: merge xs (y2::ys'))
def ys = sorted (y1 :: x1 :: merge xs ys)
y1 < x1 = sorted (x1 :: merge xs ys)
merge = sorted (merge x ys)
I.H. = true by induction hypothesis
```

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

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

When you want to play with it within [utop](#), use the following sequence of commands:

```
1  #thread;;
2  #directory "+threads";;
3  #load "unix.cma";;
4  #load "threads.cma";;
```

Or load directly via:

```
1  utop -I +threads
```


Example

```
1  module Echo = struct
2    open Thread
3
4    let echo () = print_string (read_line () ^ "\n")
5
6    let main =
7      let t1 = create echo () in
8      join t1;
9      print_int (id (self ()));
10     print_string "\n"
11  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:

```
1  > ./a.out
2  Hello Echo!
3  Hello Echo!
4  0
5  >
```

- 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:

```
1  type 'a channel
2  type 'a event
3
4  new_channel : unit      -> 'a channel
5  always      : 'a        -> 'a event
6  sync        : 'a event  -> 'a
7  receive     : 'a channel -> 'a event
8  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 events returns their values.

```
1  module Exchange = struct open Thread open Event
2      let thread ch = let x = sync (receive ch) in
3                      print_string (x ^ "\n");
4                      sync (send ch "got it!")
5
6      let main = let ch = new_channel () in
7                 let _ = create_thread ch in
8                 print_string "main is running ... \n";
9                 sync (send ch "Greetings!");
10                 print_string ("He " ^ sync (receive ch) ^ "\n")
11  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 \implies **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:

```
1  > ./a.out
2  main is running ...
3  Greetings!
4  He got it!
5  >
```


Example: A global memory cell

A global memory cell, in particular in presence of multiple threads, can be realized by implementing the signature `Cell`:

```
1  module type Cell = sig
2    type 'a cell
3
4    val new_cell : 'a      -> 'a cell
5    val get      : 'a cell -> 'a
6    val put      : 'a cell -> 'a      -> unit
7  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**:

```
1  type 'a req = Get of 'a channel | Put of 'a  
2  type 'a cell = 'a req channel
```

The channel transports requests to the memory cell, which either provide the new value or the back channel ...

```
1  let get cell = let reply = new_channel () in
2      sync (send cell (Get reply));
3      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.

```
1  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:

```
1  let new_cell x =  
2    let cell = new_channel () in  
3    let rec serve x =  
4      match sync (receive cell) with  
5      | Get reply ->  
6        sync (send reply x);  
7        serve x  
8      | Put y -> serve y  
9    in  
10   let _ = create serve x in  
11   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

...

```
1  let main =  
2    let cell = new_cell 1 in  
3    print_int (get cell);  
4    print_string "\n";  
5    put cell 2;  
6    print_int (get cell);  
7    print_string "\n"
```

Now, the execution yields

```
1  > ./a.out  
2  1  
3  2  
4  >
```

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:

```
1  module type Lock = sig
2      type lock
3      type ack
4
5      val new_lock : unit -> lock
6      val acquire  : lock -> ack
7      val release  : ack  -> unit
8  end
```

Execution of the operation `acquire` returns an element of type `ack` which is used to return the lock:

```
1  type ack = unit channel
2  type lock = ack channel
```

For simplicity, `ack` is chosen itself as the channel by which the lock is returned.

```
1  let acquire lock = let ack = new_channel () in
2                          sync (send lock ack);
3                          ack
```


The unlock channel is created by `acquire` itself

```
1  let release ack = sync (send ack ())
```

... and used by the operation `release`.

```
1  let new_lock () =  
2    let lock = new_channel () in  
3    let rec acq_server () = rel_server (sync (receive lock))  
4    and rel_server ack =  
5      sync (receive ack);  
6      acq_server ()  
7    in  
8    let _ = create acq_server () in  
9    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`...

```
1  let dead =
2    let l1 = new_lock () in
3    let l2 = new_lock () in
4    let th (l1, l2) =
5      let a1 = acquire l1 in
6      let _ = delay 1.0 in
7      let a2 = acquire l2 in
8      release a2; release a1;
9      print_int (id (self (()))); print_string " finished\n"
10   in
11   let t1 = create th (l1, l2) in
12   let t2 = create th (l2, l1) in
13   join t1
```

The result is

```
1      > ./a.out
```

OCaml waits forever ...

Example: Semaphores

Occasionally, there is more than one copy of a resource. Then **semaphores** are the method of choice ...

```
1  module type Sema = sig
2      type sema
3
4      new_sema : int -> sema
5      up       : sema -> unit
6      down     : sema -> unit
7  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 ...

```
1  module Sema = struct
2    open Thread
3    open Event
4
5    type sema = unit channel option channel
6
7    let up sema = sync (send sema None)
8
9    let down sema =
10      let ack = (new_channel () : unit channel) in
11      sync (send sema (Some ack));
12      sync (receive ack)
13  ...
```

```

1    ...
2    let new_sema n = let sema = new_channel () in
3        let rec serve (n, q) =
4            match sync (receive sema) with
5            | None -> ( match dequeue q with
6                | None, q -> serve (n + 1, q)
7                | Some ack, q -> sync (send ack ());
8                            serve (n, q))
9            | Some ack ->
10                if n > 0 then (
11                    sync (send ack ());
12                    serve (n - 1, q))
13                else serve (n, enqueue ack q) in
14    let _ = create serve (n, new_queue ()) in
15    sema
16 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

```
1  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

```
1  let forever f init =  
2    let rec loop x = loop (f x) in  
3    let _ = create loop init in  
4    ()  
5  
6  let add1 (in1, in2, out) =  
7    forever  
8      (fun () -> sync (send out (sync (receive in1) +  
9                      sync (receive in2))))  
10   ()
```

Disadvantage

If a value arrives at the second input channel first, the thread nonetheless must wait.

Second Attempt

```
1  let add (in1, in2, out) =  
2    forever  
3      (fun () ->  
4        let a, b = select [  
5          wrap (receive in1) (fun a -> (a, sync (receive in2)));  
6          wrap (receive in2) (fun b -> (sync (receive in1), b))  
7        ]  
8        in  
9          sync (send out (a + b)))  
10   ()
```

This program must be digested slowly ...

Idea

- Initiating input or output operations, generates **events**.
- Events are data objects of type `'a event`.
- The function

```
1  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

```
1  choose : 'a event list -> 'a event
2  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:

```
1  let select = comp sync choose
```

Typically, that event occurs that finds its communication partner first.

Further Examples

The function

```
1  copy : 'a channel * 'a channel * 'a channel -> unit
```

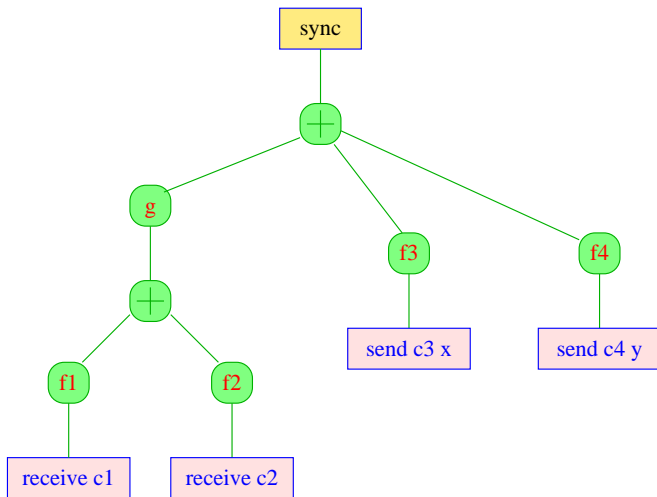
is meant to copy a read element into two channels:

```
1  let copy (read, out1, out2) =  
2      forever  
3          (fun () ->  
4              let x = sync (receive read) in  
5                  select  
6                      [  
7                          wrap (send out1 x) (fun () -> sync (send out2 x));  
8                          wrap (send out2 x) (fun () -> sync (send out1 x));  
9                      ])  
10     ()
```

Apparently, the event list may also consist of send events — or contain both kinds.

```
1  type 'a cell = 'a channel * 'a channel
2
3  let get (get_chan, _) = sync (receive get_chan)
4  let put (_, put_chan) x = sync (send put_chan x)
5
6  let new_cell x =
7      let get_chan = new_channel () in
8      let put_chan = new_channel () in
9      let rec serve x = select [
10          wrap (send get_chan x) (fun () -> serve x);
11          wrap (receive put_chan) serve;
12      ]
13  in
14  let _ = create serve x in
15  (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

```
1  module type Swap = sig
2      type 'a swap
3
4      val new_swap : unit      -> 'a swap
5      val swap      : 'a swap -> 'a      -> 'a event
6  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:

```

1  module Swap = struct
2      open Thread
3      open Event
4
5      type 'a swap = ('a * 'a channel) channel
6
7      let new_swap () = new_channel ()
8
9      let swap ch x =
10         let c = new_channel () in
11         choose [
12             wrap (receive ch) (fun (y, c) -> sync (send c x); y);
13             wrap (send ch (x, c)) (fun () -> sync (receive c));
14         ]
15     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 ...

```
1  module type Timer = sig
2      set_timer      : float          -> unit event
3      timed_receive  : 'a channel -> float          -> 'a option event
4      timed_send     : 'a channel -> 'a -> float -> unit option event
5  end
```

```

1  module Timer = struct open Thread open Event
2
3      let set_timer t = let ack = new_channel () in
4          let serve () = delay t; sync (receive ack) in
5          let _ = create serve () in
6          send ack ()
7
8      let timed_receive ch time = choose [
9          wrap (receive ch) (fun a -> Some a);
10         wrap (set_timer time) (fun () -> None);
11     ]
12
13     let timed_send ch x time = choose [
14         wrap (send ch x) (fun a -> Some ());
15         wrap (set_timer time) (fun () -> None);
16     ]
17 end

```

8.3 Threads and Exceptions

An exception must be handled within the thread where it has been raised.

```
1  module Explode = struct
2    open Thread
3
4    let thread x =
5      x / 0;
6      print_string "thread terminated regularly ...\n"
7
8    let main =
9      let _ = create thread 0 in
10     delay 1.0;
11     print_string "main terminated regularly ...\n"
12 end
```

... yields

```
1  > /.a.out
2  Thread 1 killed on uncaught exception Division_by_zero
3  main terminated regularly ...
```

The thread was killed, the OCaml program terminated nonetheless.

Also, uncaught exceptions within the wrapper function terminate the running thread:

```
1  module ExplodeWrap = struct open Thread open Event open Timer
2      let main =
3          try sync (wrap (set_timer 1.0) (fun () -> 1 / 0))
4              with _ ->
5                  0;
6                  print_string "... this is the end!\n"
7  end
```

Then we have

```
1  > ./a.out
2  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**:

```
1  module type Mailbox = sig
2      type 'a mbox
3
4      val new_mailbox : unit      -> 'a mbox
5      val receive     : 'a mbox -> 'a event
6      val send        : 'a mbox -> 'a      -> unit
7  end
```

For the implementation, we rely on a server which maintains a queue of sent but not yet received messages.

Then we implement:

```
1  module Mailbox = struct
2      open Thread
3      open Queue
4      open Event
5
6      type 'a mbox = 'a channel * 'a channel
7
8      let send (in_chan, _) x = sync (send in_chan x)
9
10     let receive (_, out_chan) = receive out_chan
11     ...
```

```

1  ...
2  let new_mailbox () = let in_chan = new_channel ()
3    and out_chan = new_channel () in
4    let rec serve q =
5      if is_empty q then serve (enqueue
6        (sync (Event.receive in_chan)) q)
7    else select [
8      wrap (Event.receive in_chan)
9        (fun y -> serve (enqueue y q));
10     wrap (Event.send out_chan (first q))
11       (fun () -> let _, q = dequeue q in
12         serve q);
13   ]
14  in
15  let _ = create serve (new_queue ()) in
16  (in_chan, out_chan)
17  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`:

```
1  module type Multicast = sig
2      type 'a mchannel and 'a port
3
4      val new_mchannel : unit          -> 'a mchannel
5      val new_port     : 'a mchannel -> 'a port
6      val receive      : 'a port     -> 'a event
7      val multicast    : 'a mchannel -> 'a          -> unit
8  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.

```
1  module Multicast = struct open Thread open Event
2    module M = Mailbox
3
4    type 'a port = 'a M.mbox
5
6    type 'a mchannel = 'a channel * 'a port channel
7
8    let new_port (_, req) = let m = M.new_mailbox () in
9                          sync (send req m); m
10
11    let multicast (send_ch, _) x = sync (send send_ch x)
12
13    let receive port = M.receive port
14    ...
```

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:

```
1  ...
2  let new_mchannel () = let send_ch = new_channel () in
3    let req = new_channel () in
4    let send_port x port = M.send port x in
5    let rec serve ports = select [
6      wrap (Event.receive req) (fun p -> serve (p :: ports));
7      wrap (Event.receive send_ch) (fun x ->
8        let _ = create (List.iter (send_port x)) ports in
9        serve ports);
10   ]
11  in
12  let _ = create serve [] in
13  (send_ch, req)
14  ...
```

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`:

```
1  ...
2  let main = let mc = new_mchannel () in
3    let thread i = let p = new_port mc in
4      while true do
5        let x = sync (receive p) in
6        print_int i; print_string ": ";
7        print_string (x ^ "\n")
8      done
9    in
10   let _ = create_thread 1 in
11   let _ = create_thread 2 in
12   let _ = create_thread 3 in
13   delay 1.0;
14   multicast mc "Hello!";
15   multicast mc "World!";
16   multicast mc "... the end.";
17   delay 10.0
18 end
```

We obtain

```
1  -  ./a.out
2  3: Hello!
3  2: Hello!
4  1: Hello!
5  3: World!
6  2: World!
7  1: World!
8  3: ... the end.
9  2: ... the end.
10 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

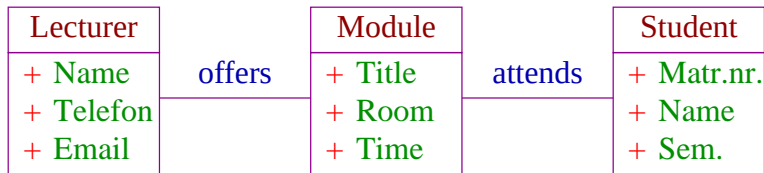
- Beyond the language concepts discussed in the lecture, OCaml has diverse further concepts, which also enable object oriented programming.
- Moreover, OCaml has elegant means to access functionality of the operating system, to employ graphical libraries and to communicate with other computers

...

⇒ OCaml is an interesting alternative to Java.

9 Datalog: Computing with Relations

Example 1: The Study Program of a TU



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 described 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 Schluri	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
 \implies **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” ?

⇒ Datalog

Idea: Table \iff Relation

A relation R is a set of tuples, i.e.,

$$R \subseteq \mathcal{U}_1 \times \dots \times \mathcal{U}_n$$

where \mathcal{U}_i is the set of all possible values for the i th 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

```
1  offers ("Esparza", "Discrete Structures").
2  offers ("Nipkow", "Pearls of Informatics III").
3  offers ("Seidl", "Funct. Programming and Verification").
4  offers ("Seidl", "Optimization").
5
6  attends (123456, "Optimization").
7  attends (123456, "Funct. Programming and Verification").
8  attends (123456, "Discrete Structures").
9  attends (543345, "Funct. Programming and Verification").
10 attends (543345, "Discrete Structures").
11 attends (131175, "Optimization").
```

Rules can be used to deduce further facts ...

... in the Example

```
1  has_attendant (X,Y) :- offers (X,Z), attends (M,Z),  
2                           student (M,Y,_).  
3  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

```
1  ?- 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:

```
1  Z = "Hans Dampf"  
2  Z = "Anna Blume"  
3  Z = "Effi Briest"
```

Further Queries

```
1  ?- semester ("Discrete Structures", X).  
2      X = 2  
3      X = 4  
4  
5  ?- attends (X, "Funct. Programming and Verification"),  
6      attends (X, "Discrete Structures").  
7      X = 123456  
8      X = 543345
```

Caveat

A query may contain none, one or several variables.

An Example Proof

The rule

```
1  has_attendant (X,Y) :- offers (X,Z), attends (M,Z),  
2                               student (M,Y,_).
```

holds for all X, M, Y, Z .

By means of the substitution

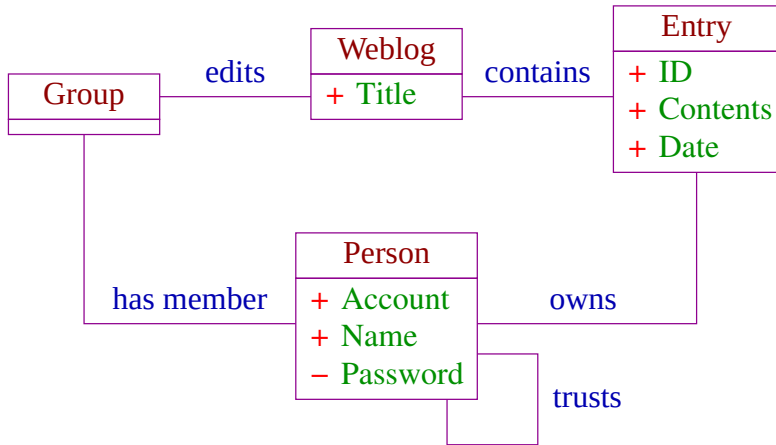
```
1  "Seidl"/X    "Funct. Programming ..."/Z  
2  543345/M    "Anna Blume"/Y
```

we prove

$$\frac{\begin{array}{l} \text{offers ("Seidl", "Funct. Programming ...")} \\ \text{attends (543345, "Funct. Programming ...")} \\ \text{student (543345, "Anna Blume", 3)} \end{array}}{\text{has_attendant ("Seidl", "Anna Blume")}}$$

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

```
1  may_add (X,W) :- edits (Z,W),
2                        has_member (Z,X).
3  may_delete (X,E) :- owns (X,E).
4  may_modify (X,E) :- owns (X,E).
5  may_modify (X,E) :- owns (Y,E),
6                        trusts (Y,X).
7  may_read (X,E) :- contains (W,E),
8                      may_add (X,W).
9  may_read (X,E) :- may_read (Y,E),
10                     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

```
1 equals (X,X).
```

\Rightarrow 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 $p(a_1, \dots, a_k)$, it is shown that each constant a_i 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

$$\mathcal{F}(M) = \{h[\underline{a}/\underline{X}] \mid \exists h :- l_1, \dots, l_k. \in W : \\ l_1[\underline{a}/\underline{X}], \dots, l_k[\underline{a}/\underline{X}] \in M\}$$

// $[\underline{a}/\underline{X}]$ 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

```
1  edge (a,b).  
2  edge (a,c).  
3  edge (b,d).  
4  edge (d,a).  
5  t (X,Y) :- edge (X,Y).  
6  t (X,Y) :- edge (X,Z), t (Z,Y).
```

Relation **edge** :

	a	b	c	d
a				
b				
c				
d				

$t^{(0)}$

	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				

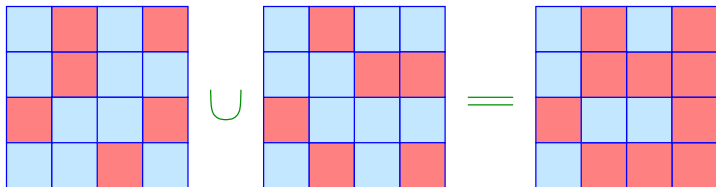
Discussion

- 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 \implies **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



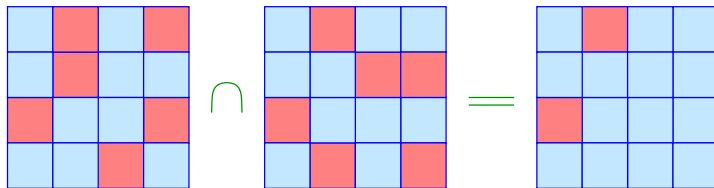
... in Datalog:

$$\begin{aligned}r(X_1, \dots, X_k) &:- s_1(X_1, \dots, X_k). \\r(X_1, \dots, X_k) &:- s_2(X_1, \dots, X_k).\end{aligned}$$

Example

```
1 attends_Esparza_or_Seidl (X) :- has_attendant ("Esparza", X).
2 attends_Esparza_or_Seidl (X) :- has_attendant ("Seidl", X).
```

2. Intersection



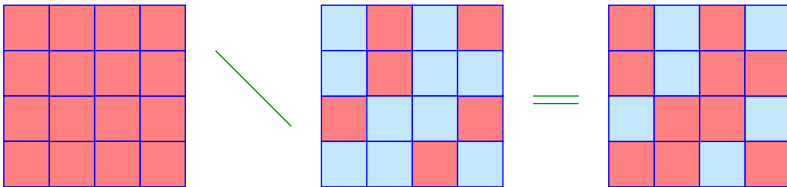
... in Datalog:

$$r(X_1, \dots, X_k) \quad :- \quad s_1(X_1, \dots, X_k), \\ s_2(X_1, \dots, X_k).$$

Example

```
1 attends_Esparza_and_Seidl (X) :- has_attendant ("Esparza", X),  
2                               has_attendant ("Seidl", X).
```

3. Relative Complement



... in Datalog:

$$r(X_1, \dots, X_k) \quad :- \quad s_1(X_1, \dots, X_k), \text{ not}(s_2(X_1, \dots, X_k)).$$

i.e., $r(a_1, \dots, a_k)$ follows when $s_1(a_1, \dots, a_k)$ holds but $s_2(a_1, \dots, a_k)$ is not **provable**.

Example

```
1  does_not_attend_Seidl (X) :- student (_,X,_),  
2                                not (has_attendant ("Seidl", X)).
```

Caveat

The query

```
1  p("Hello!").  
2  ?- not (p(X)).
```

results in infinitely many answers.

⇒ we allow negated literals only if all occurring variables have already occurred to the left in non-negated literals.

```
1  p("Hello!").  
2  q("Damn ...").  
3  ?- q(X), not (p(X)).  
4      X = "Damn ..."
```

Caveat (cont.)

Negation is only **meaningful** when s does not recursively depend on $r \dots$

```
1  p(X) :- not (p(X)).
```

\dots is **not easy** to interpret.

\Rightarrow We allow $\text{not}(s(\dots))$ only in rules for predicates r of which s is independent

\Rightarrow **stratified negation**

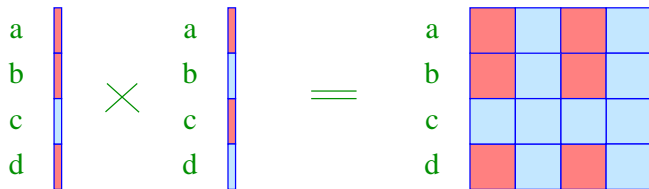
// Without recursive predicates, every negation is stratified.

4. Cartesian Product

$$S_1 \times S_2 = \{(a_1, \dots, a_k, b_1, \dots, b_m) \mid \begin{array}{l} (a_1, \dots, a_k) \in S_1, \\ (b_1, \dots, b_m) \in S_2 \end{array}\}$$

... in Datalog:

$$r(X_1, \dots, X_k, Y_1, \dots, Y_m) \quad :- \quad s_1(X_1, \dots, X_k), s_2(Y_1, \dots, Y_m).$$



Example

```
1  lecturer_student (X,Y) :- lecturer (X,_,_),  
2                               student (_,Y,_).
```

Comments

- The product of independent relations is very **expensive**.
- It should be **avoided** whenever possible **;-)**

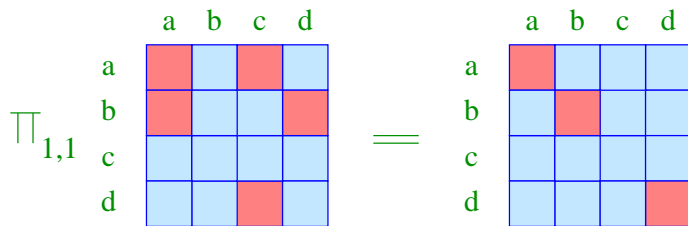
5. Projection

$$\pi_{i_1, \dots, i_k}(S) = \{(a_{i_1}, \dots, a_{i_k}) \mid (a_1, \dots, a_m) \in S\}$$

... in Datalog:

$$r(X_{i_1}, \dots, X_{i_k}) \quad :- \quad s(X_1, \dots, X_m).$$

$$\prod_1 \begin{array}{c} \begin{array}{ccccc} & a & b & c & d \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{array}{|c|c|c|c|} \hline \text{red} & \text{blue} & \text{red} & \text{blue} \\ \hline \text{red} & \text{blue} & \text{blue} & \text{red} \\ \hline \text{blue} & \text{blue} & \text{blue} & \text{blue} \\ \hline \text{blue} & \text{blue} & \text{red} & \text{blue} \\ \hline \end{array} & = & \begin{array}{|c|} \hline \text{red} \\ \hline \text{red} \\ \hline \text{blue} \\ \hline \text{red} \\ \hline \end{array} \end{array} \end{array}$$



6. Join

$$S_1 \bowtie S_2 = \{(a_1, \dots, a_k, b_1, \dots, b_m) \mid \begin{array}{l} (a_1, \dots, a_{k+1}) \in S_1, \\ (b_1, \dots, b_m) \in S_2, \\ a_{k+1} = b_1 \end{array} \}$$

... in Datalog:

$$r(X_1, \dots, X_k, Y_1, \dots, Y_m) \quad :- \quad s_1(X_1, \dots, X_k, Y_1), s_2(Y_1, \dots, Y_m).$$

Discussion

Joins can be defined by means of the other operations ...

$$S_1 \bowtie S_2 = \pi_{1,\dots,k,k+2,\dots,k+1+m} \left(S_1 \times S_2 \cap \mathcal{U}^k \times \pi_{1,1}(\mathcal{U}) \times \mathcal{U}^{m-1} \right)$$

// 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
⇒ 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

```
1 semester (X,Y) :- attends (Z,X), student (Z,_,Y)
```

... can be expressed in SQL by

```
1 SELECT attends.Title, Student.Semester  
2 FROM attends, Student  
3 WHERE attends.Matrikelnummer = Student.Matrikelnummer
```

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 failure

⇒ Database Lecture