Chapter 1: Correctness of Programs:

⇒ Programmers make mistakes but some systems must not have errors.
(planes, airbags, rocket systems)
↳ careful engineering, systematic testing, proof of correctness through verification.

1.0 Assertions:

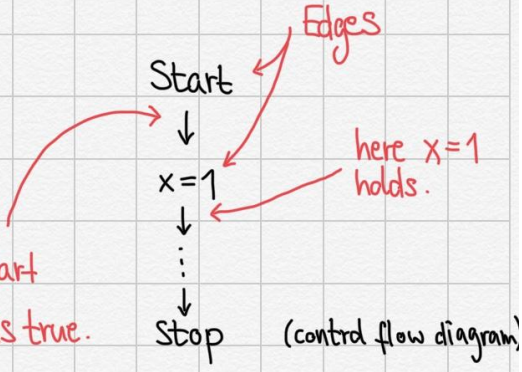
⇒ In order to prove that the program is correct, we must know what to expect from the program execution. We write that down with logic using assertions.
The static method assert() expects a boolean argument.
The calls of assert are only evaluated if Java is launched with the options -ea.

If the argument is true → execution continues, else AssertionError is thrown.
Attention! The check must not change the program state.
Use distinct inspector classes to inspect the properties of complicated data structures without interference.

1.1 Program Verification:

⇒ General method to guarantee a given assertion is valid.
Each program point is annotated with a logic formula and proven to be valid.

A program point corresponds to an edge of the control-flow diagram.



(control flow diagram)

After the start nothing holds → strongest assertion is true.

We need to prove that the assertions are locally consistent.

Assignments:

→ Every assignment transforms a post-condition B into a minimal assumption that must be valid before the execution.
That minimal assumption is called the weakest precondition.

weakest precondition: $WP[\![\text{assignment}]\!](\text{postcondition}) \equiv \text{postcondition}[\text{assignment-substitution}]$

(A given precondition A is valid, if $A \Rightarrow WP[\![\cdots]\!](B)$)

Some special operations in MiniJava have defined weakest preconditions.

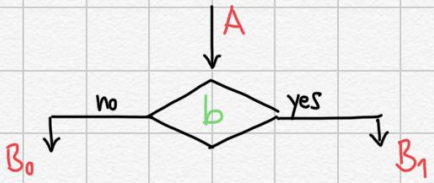
no operation: $WP[\![;]\!](B) \equiv B$

assignment: $WP[\![x=e;]\!](B) \equiv B[e/x]$ ← substitute e for x.

read: $WP[\![x=\text{read}();]\!](B) \equiv \forall x.\, B$ ← B must be valid no matter what x is.

write: $WP[\![\text{write}(e);]\!](B) \equiv B$

Conditionals:



$\Rightarrow A \wedge \neg b \rightarrow B_0$

$\Rightarrow A \wedge b \rightarrow B_1$

must hold.

weakest precondition: $WP[\![b]\!](B_0, B_1) \equiv ((\neg b) \Rightarrow B_0) \wedge (b \Rightarrow B_1) \equiv (\neg b \wedge B_0) \vee (b \wedge B_1)$

Summary of systematic local consistency:

1. Annotate each program point with an assertion.
2. The program start recieves the annotation true.
3. If statement s between A and B then $A \Rightarrow WP[\![s]\!](B)$.
4. If the conditional branch with condition b between A and B_0, B_1 then $A \Rightarrow WP[\![b]\!](B_0, B_1)$

⇒ locally consistent.

1.2 Correctness:

- The program state σ in MiniJava consists only of variable assignments.
  $\sigma = \{x \mapsto \cdots, y \mapsto \cdots, \cdots\}$
  A state σ satisfies an assertion A if every variable in A substituted by its value in σ is equivalent to true. ($\sigma \models A$)

Trivial Properties:

$\sigma \models \text{true}$ For every σ $\qquad$ $\sigma \models A_1$ and $\sigma \models A_2 \equiv \sigma \models A_1 \wedge A_2$

$\sigma \models \text{false}$ For no σ $\qquad$ $\sigma \models A_1$ or $\sigma \models A_2 \equiv \sigma \models A_1 \vee A_2$

- 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 ends in a program point u_m with a final state σ_m.
  Every step in the execution trace can change the program point and state.
  $\pi = (u_0, \sigma_0)\, s_1\, (u_1, \sigma_1) \ldots (u_m, \sigma_m)$ (s_i are elements of the control flow diagram)

If p is a MiniJava program and π is the execution trace starting at point u and ending at point v then:

⇒ All program point in p are annotated with locally consistent assertions.

⇒ The starting point u is annotated with A
⇒ The ending point v is annotated with B
} If the initial state of π satisfies A then the final state of π must satisfy B.

Must ensure, that:
- The assertion at the starting point is true
- Every further program point has an assertion
- Proof that all assertions are locally consistent.

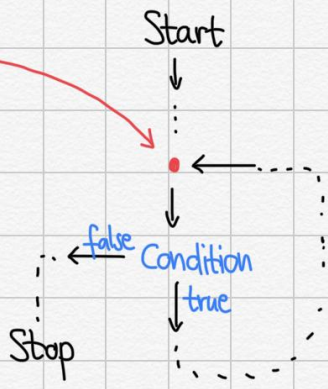
1.3 Optimization:

⇒ In general, if a program has no loops, a weakest precondition can be calculated for each program point. (↗ Assignments, Conditionals)

Loops:

→ For loops, you have to find a loop invariant, which holds:
- before the condition
- at the entry of the loop and the exit of the loop

The assertions for all other program points can be obtained by means of $WP[\![\ldots]\!]()\ldots$



1.4 Termination:

⇒ The approaches above can only prove an assertion to be valid whenever that program point is reached. It is needed to guarantee termination of a program.

- Programs without loops terminate always!
- Programs with "well behaving loops" also terminate.
  ↳ That means there are only loops of the form

```
for(i=x; i>0; i--)
{
  ...
}
```

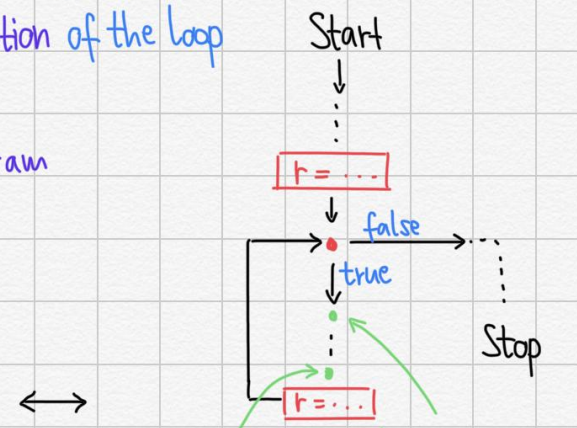
i is always positive
i get decremented in every iteration
i is not modified in other matters

⇒ You can make sure a loop is executed finitely often and therefor well behaving by doing the following:

For each loop
- introduce an auxiliary variable r that is always >0 whenever the loop is entered
- decrease r during every iteration of the loop

If you can verify those two properties, the program will always terminate.

```
r=e0
while(...) {
 assert(r>0)
 :
 assert(r>e1)
 r=e1
}
```



Chapter 2: The Basics in OCaml

⇒ The basic Syntax and style of programming in the functional programming language OCaml.

2.1 Interpreter Environment:

⇒ You can call the ocaml interpreter with ocaml or using utop
Variables, functions, etc... can be inserted an evaluated immediatly

`#use "file.ml";;` lets you read from a file.
`#` means the interpreter is waiting for input
`;;` causes evaluation of the given input → The result gets returned with its type.

2.2 Expressions:

⇒ In OCaml everything has a type that is inferred automatically.

primitive datatypes:

int: +, -, *, /, mod
float: +., -., *., /.
bool: not, ||, && ← OCaml uses "short-term evaluation"
string: ^ (Stringconcatenation)
char: ...

comparision operators on the other hand are polymorphic: =, <> (not equal), <, <=, >=, >

2.3 Definition of values:

A variable can be assigned a value by means of let.
⇒ `#let x = 3+4;;`.
`(val x : int = 7)`

Later calls to that variable return the assigned value. ⇒ `#x;;`
`(- : int = 7)`

Variable names MUST start with small letters.

Variables in OCaml are immutable. Another "new definition" of an already existing variable is allowed, but will create a new variable with the same name that from there on hides the old variable.

2.4 Complex Datatypes:

⇒ In addition to the primitive datatypes there are also Tupels and Records predefined in OCaml.

Tupels:

⇒ (a, b, c, ..., ..) Elements in the tupel can each have different types. (int, float, etc...)

Tupel can have any length ≥ 2. (There is also a tupel of length 0: () → unit)

Records:

Attributes can be from any type.

⇒ type newtype = {attriba: string ; attribb: int ; ...}

The order in which the attributes are listed doesn't matter.

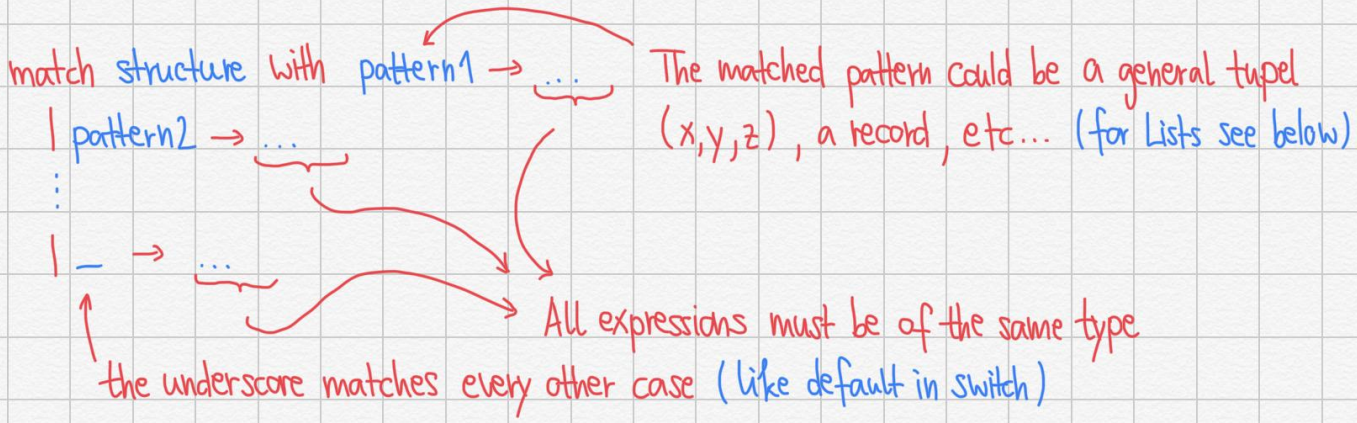
Records are tuples with named components whose ordering is irrelevant.

type names and variables start with a small letter.

You can access record components via recordname.componentname

2.5 Pattern matching and case destinction:

⇒ To get information about the structure and its components of a complex datatype, you can do pattern matching.



Make sure every match is not redundant nor has incomplete matches.

2.6 Lists:

⇒ All elements in a list must have the same type.
You can construct Lists by means of [] and ::

> The type of the elements in a list can be arbitrary

[1; 2; 3] = 1::(2::(3::[]))
↖ right association of the :: operator

Pattern matching with a list is normally of the following structure:

```
match mylist with [] → ...
| x::xs → ...
```

x: first element; xs: rest of the list

2.7 Functions in OCaml:

⇒ Function in general are defined in the following pattern:

```
let rec func_name para1 para2 ... = func_body
```

The function name is just a variable whose value is a function.
(val function_name : int → int)

You can alternativly also introduce a variable whose value is a function.
let func_name = fun para1 para2... → func_body

Mind that the values of the variables used in a function are evaluated at the point of the function definition! (static binding)

A function is recursive, if it calls itself (directly or indirectly which is called mutually recursive)

⇒ OCaml offers the keyword rec.

Case destinction with function definitions:
⇒ example...

let rec len = fun l → match l
with [] → ... | ...

syntactic sugar →

let rec len = function [] → ... | ...

Local Definitions in functions:
⇒ Definitions by let may occur locally combined with the keyword in.
⇒ example...

let x=5 in
let func_name para1 para2... = ...

(x is only visible in the function func_name)

2.8 User-defined Datatypes:

Enumeration Types: type type_name = Enum1 | Enum2 | Enum3 | ...

A constructor usually start with a capital letter

⇒ This representation is intuitive and internally efficient

(example → playing cards)

Sum Types: type type_name = Enum of 'a_type | ...

⇒ Sum types generalize of enums in that constructors now may have arguments.
Those datatypes can be defined recursivly!
example: type nat = Zero | Succ(nat)

Option Datatype: OCaml provides the built in datatype option with constructors None and Some.

⇒ This datatype of choice is used when a function is only partially defined.

Chapter 3: A closer Look at Functions in OCaml

⇒ Since OCaml is a functional programming language, it is important to have a closer look at functions.

3.1 Last Calls:

« A last call in the body of a function is a call whose value provides the whole value for the body »

⇒ You don't have to return to the calling function from the last call. Therefor the stack space can immediatly be recycled.

A recursive function is called tail recursive if all calls to it are last calls.
(→ Tail-recursive functions can be executed efficiently as loops)

3.2 Higher Order Functions:

⇒ OCaml follows the principle of Currying.
Consider the fdlowing two functions:

Tupel

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

Sequence

f has one single argument
⇒ f: int*int → int

g has the argument a. The result of this is again a function that is then applied to the argument b.
⇒ g: int → int → int

So technically g is a higher order function because its result is again a function.
The application of a higher order function to one argument is called partial.

3.3 List functions:

⇒ The List module implemented in OCaml provides some useful methods for handeling Lists. (→ called list functionals)

List.hd mylist , List.tl mylist : Returns the first or last element of the list.
List.length mylist : Returns the amount of elements in the list.
List.rev mylist : Returns the reversed list.
List.map func mylist : Applies the function func to every element of the list.
List.fold_left func acc mylist : Traverses the list from left to right and applies func acc element to every element of the list. The accumulator is defined at the start, then updated to the result of func in every iteration and at the end returned.
List.fold_right func acc mylist : Does the same thing as fold_left but traverses the list from right to left.
List.find_opt func mylist : Returns the first element of the list that satisfies the predicate func. None if there isn't any in the list.

3.4 Polymorphic functions:

« Functions which operate on equally structured data of various types are called polymorphic »

OCaml shows the types 'a, 'b, ... Those are type variables which can be instantiated by any type. But each occurence then must have the same type!

example: Every list functional above is polymorphic since it does matter what kind of list you are dealing with...

3.5 Polymorphic Datatypes:

« User-defined datatypes may be polymorphic, too. Those are called type constructors because they allow to create a new type from another type 'a »

example: 'a tree = Leaf of 'a | Node of ('a tree * 'a tree)
→ The function on that tree are again polymorphic.

3.6 Queues:

Representation by a list:

type 'a queue = 'a list

- is_empty: If list is empty
- queue_of_list and list_of_queue are trivial...
- dequeue: access top element
- enqueue: concatenate the old list with new element at the end.

⇒ Insertion requires list length many calls...

—

Representation by two lists:

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

- is_empty: If both lists are empty

Idea:

⇒ Insertion is in the second list

- enqueue: Add to the front of the second list

⇒ Extraction is normally from the first list unless it's empty

- dequeue: Returns tupel of (dequeued element, new queue)
  ↑ option datatype (could not exist)

⇒ Amortized cost analysis shows only constant costs for insertion and extraction.

3.7 Anonymous functions:

⇒ Since functions are only data, they can be used without naming them.

λ-calculus for functions: fun para1 para2... → func_body

Anonymous functions are convenient if just used once in a program.

Recursive functions cannot be defined that way.

Chapter 5 : Practical Features in OCaml

⇒ Even though OCaml is a functional programming language, Side Effects, etc.. can be useful in some szenarios...

5.1 Exceptions:

⇒ In case of a runtime error, the OCaml system generates an exception.

Predefined Constructors for Exceptions:

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

Theese constructors for the first class citizen datatype exn are extensible...
exception exc_name can add the new exception exc_name...

Exception-Handeling: Exceptions can be handled and raised through the keywords:

```
try
  ...
with Exception → ...
```

following the keyword with, the exception value can be inspected by means of pattern matching on the exn datatype (→ several exceptions can be caught at the same time)

The programmer can trigger exceptions on his/her own through the keyword raise

```
let div x y = if y=0 then raise Division_by_zero
              else x/y
```

← the exception can replace any expression!

5.2 Textual Input and Output:

⇒ Since reading from the input and writing to the output violates the paradigm of functional programming ⇒ Those operations are realized by means of side-effects. We use functions whose return value is as irrelevant as unit...

Console output → print_string "..." (Returns unit)
input → read_line () (Takes unit and returns string)

Reading from a file →
```
let file = open_in "filename" in
try
  let line = input_line file in
with End_of_file → ...
```
file has type in_channel.

if there is no further line available, End_of_file is raised.

close_in file ← You MUST always make sure, that every channel no longer needed gets closed!

⇒ stdin is the console as text input
input_char returns the next char of the input and
input_line returns the next line of the input as a string.
in_channel_length returns the length of the channel.

Writing to a file →
```
let file = open_out "filename" in
```
file has type out_channel.

Each function writes a string into the file...
```
let variant1 = output_string file "message..." in
let variant2 = Printf.fprintf file "message..." in
```

close_out file ← You MUST always make sure, that every channel no longer needed gets closed!
(The written words may only appear inside the file once it has been closed)

5.3 Sequences:

⇒ Since side-effects can now occur, ordering matters!

Actions can be sequenced by the sequence operator ;

example:
```
print_string "Test01";
print_string "Test02";
print_string "Test03"
```

⇒ List.iter func mylist
can be used to do something with every element of a list...

Chapter 6 : The Module System of OCaml

⇒ To structure bigger projects, OCaml offers the concept of modules.

6.1 Modules:

Creating and Accessing Modules:

```
module Mod_name =
  struct
    type ...
    let func_name ...
    let ab ...
  end
```

Definitions inside a module can only be accessed via qualification: Mod_name.func_name

⇒ Several function with the same name can be defined in different modules.

⇒ The compiler returns the signature of the module.

Opening and Including Modules:

`open Mod_name` ⟶ To avoid explicit qualification, you can open a module to make all definitions directly accessible (just like import in Java)

`include Mod_name` ⟶ Includes the definitions of another module into the present one.

⇒ Only include can change the signature of a module.

Nested Modules:

```
module Outer = struct
  module Inner = struct
                 :
```

⟶ Module can again contain other modules.
Access through: Outer.Inner.func_name

6.2 Types and Signatures:

« The signiture of a module is inferred automatically.
An explicit indication of the signiture allows to restrict what a module exports. »

```
module Mod1 = struct
  :
  let func = ...
  let help_func = ...
end
```

⇒ The automatically inferred signature allows access to for example help_func...

to hide functions ⟹ we introduce a new signiture

```
module type Mod1_sig = sig
  val func = ...
  :
end
module Mod2 : Mod1_sig = Mod1
```

⇒ The function help_func is no longer exported/visible from the outside.

⇒ ↯ The types in the signiture must fit to the module (→ raises Signiture mismatch)

6.3 Information Hiding:

⇒ You can use signitures to hide the structure/implementation of a datatype from the user. (obfuscate the true nature)

example: Define the module Queue with type 'a queue = 'a list.
To hide, that we used a list as a queue we define the signiture

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

← now the methods can only be applied to queues...

6.4 Functors:

« Modules of higher order are called Functors.
They receive a sequence of modules as parameters which can be used in the module's body »

example:

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

This module offers the function to split a sequence of something into one element and the rest.

```
module type Folds = functor (X : Deconstruct) -> sig
  val fold_left : ...
  val fold_right : ...
end
```

This module is a functor, which can you functions of the module Deconstruct. (ex. X.decons)

We can then apply this functor to models to obtain new ones...

```
module FoldXY = Folds (XY)
```

← By that we define FoldXY.fold_left ...

↑ The function XY.decons must exist in the module.

6.5 Separate Compilation:

⇒ OCaml programs can be compiled by calling ocamlc test.ml
Here test.ml is treated as a sequence of definitions for a new module test.
The compiler will create bytecode for the module, bytecode for the signature and an executable.

You can compile a module test.ml with a specific signiture test.mli by calling ocamlc test.mli test.ml

You can also combine modules by calling ocamlc B.mli B.ml A.mli A.ml

Chapter 7 : Formal Verification for OCaml

⇒ To make sure an OCaml Program behaves as it should, we require formal semantics to prove assertions about programs.

7.1 Introducing MiniOCaml:

⇒ We only consider a fraction of OCaml. That means only base type int, bool, tupels and lists, recursive functions at top level but no input, output or modifiable datatypes.

Expressions := const | name | $op_1 E$ | $E_1 op E_2$ | $(E_1, \dots, E_k)$ | let name = E_1 in E_0 |
match E with $P_1 \to E_1 | \dots | P_k \to E_k$ | fun name → E

Patterns := const | name | $(P_1, \dots, P_k)$ | $P_1 :: P_2$

Values := const | fun $name_1 \dots name_k \to E$ | $(V_1, \dots, V_k)$ | $V_1 :: V_2$ | $[]$

7.2 Big-step operational semantics:

⇒ A value is returned for every expression. It is an expression that cannot be further evaluated.
We define a relation $e \Rightarrow v$ between every expression and its value by means of axioms and rules. (Big-step)

Tupel: $T \dfrac{e_1 \Rightarrow v_1 \quad \dots \quad e_k \Rightarrow v_k}{(e_1, \dots, e_k) \Rightarrow (v_1, \dots, v_k)}$

Lists: $L \dfrac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow V_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$

Global Definitions: $GD \dfrac{f = e \qquad e \Rightarrow v}{f \Rightarrow v}$ (functionname; don't forget "="; often predefine as π_f)

Local Definitions: $LD \dfrac{e_1 \Rightarrow v_1 \qquad e[v_1/x] \Rightarrow v_0}{\text{let } x = e_1 \text{ in } e_0 \Rightarrow v_0}$ (every x is substituted by v_1)

Function Calls:

$$\text{App}'\ \frac{e_0 \Rightarrow \text{fun } x_1 \dots x_k \to e \quad e_1 \Rightarrow v_1 \ \cdots\ e_k \Rightarrow v_k \quad e[v_1/x_1 \dots v_k/x_k] \Rightarrow v}{e_0\, e_1 \dots e_k \Rightarrow v_0}$$

use π_f here

substitute every variable by its value

e_0: function name; $e_1 \dots e_k$: function arguments

Pattern Matching:

$$\text{PM}\ \frac{e_0 \Rightarrow v_0 \qquad e_i[v_1/x_1, \dots, v_k/x_k] \Rightarrow v}{\text{match } e_0 \text{ with } p_1 \to e_1 \mid \cdots \mid p_m \to e_m \Rightarrow v}$$

only the pattern that matches v_0 (— if no patterns match)

Operators:

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

Here you use the "real" Operator to add/mul/div/...etc. the values of the expressions.

Equality: $v_1 = v_2 \to$ false and $v = v \to$ true

Only if v_1, v_2, v do not contain functions.

Big-step semantics is well suited for proving that the evaluation of a function terminates for a particular argument.
You can use induction for that purpose.

- Base case: $n = 0$ — n is often an int that can also represent the length of a list.
- Induction step: We assume that ... terminates with ... for all $n \geq 0$
  Now, we show that it also terminates with ... for input $n+1$.

always do induction of the argument that is changed in function.

by I.H.
⋮
...

□

Equality can only be proven for values that do not contain function (= comparables)
The Big-step semantics allow to verify that optimizing transformations are correct.

7.3 Extended notation of equality:

⇒ If you want to prove that $f\ a\ b = g\ a\ b$ holds for the functions f, g you have to extend the notion of equality:

We assume:

Termination:

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

both terminating

Non-termination:

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

Tupel:

$$\frac{e_1 = e_1' \ \dots \ e_k = e_k'}{(e_1, \dots, e_k) = (e_1', \dots, e_k')}$$

Lists:

$$\frac{e_1 = e_1' \qquad e_2 = e_2'}{e_1 :: e_2 \ = \ e_1' :: e_2'}$$

Functions:

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

evaluate functions for all arguments (theoretically, maybe by induction)

Substitution Lemma:

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

if attributes e_1 and e_2 are equal, they can be interchanged everywhere

Local definitions:

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

e_1 and v_1 represent the same value

1. $e[v_1/x]$ terminates $\Rightarrow e[e_1/x] = e[v_1/x] = v$
2. $e[v_1/x]$ does not terminate $\Rightarrow$ both sides do not terminate, equality is true.

Function calls:

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

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

(multiple arguments)

Pattern matching:

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

7.4 Proofs for MiniOCaml Programs:

⇒ For a given definition of f arg1... g arg1... you can verify that f arg1... = g arg1... for all possible arguments.

⇒ Induction over the changing argument ↯ List, int, etc... require a different base case.
⇒ Almost always generalize the claim by adding accumulators
⇒ Sometimes case destinction is needed to prove a claim.
⇒ It is always assumed the functions terminate.

Example: We prove the more general statement:

acc + f ... = g acc ...

We do so by induction over on n ← maybe length of a list

- Base case: n = 0

  acc + f ... [] ...
  = ...
  ⋮
  = g acc [] ...

- Inductive step: We assume the statement holds for n ≥ 0
  Now, we prove it for n+1

  acc + f x::xs ...
  = ...
  ⋮ by induction hypothesis
  = g acc x::xs

□

Chapter 8 : Parallel Programming

⇒ OCaml provides support for creating multiple threads and communicating between them.

8.0 Threads:

The module Thread collects all basic functionality for creation of concurrency.

- create task function arg → creates a new thread
- self () → returns thread executing the function
- id t → returns id of the thread as an int (Use: id(self()))

- exit () → terminates current thread
- join t → waits for thread t to terminate
- delay seconds → delay current thread for some seconds

To compile a OCaml program using threads, you use:
» ocamlc -thread unix.cma threads.cma file.ml
The program can be tested by ./a.out

8.1 Channels:

⇒ OCaml offers a module Event for creating polymorphic channels on which the threads communicate on.

- new_channel () → creates a new polymorphic channel
- send channel data → sends data on a channel
- receive channel → returns the data received on a channel
- sync event → returns the value of events and blocks current thread until event has "happend"
  ↳ rendez-vous for synchronous communication.

⚠ Ordering of send and receive must be designed carefully, since threads can get blocked (Deadlock possible)

General structure:

```
module X = struct
  type 'a req = A of .. | B of ...
  type 'a t = 'a req channel
  :
```

```
let create arg1 arg2... =
  let c = new_channel () in
  let task () =
    let result = ... in
    sync (send c result) in
  let _ = create task () in c
```

always create a new channel

this is the "server task"

dummy variable to create the thread

return the channel

8.2 Examples of using threads:

Global Memory Cell:

```
module Cell = struct
  type 'a req = Get of 'a channel | Put of 'a
  type 'a cell = 'a req channel

  let get cell = let reply = new_channel ()
                 in sync (send cell (Get reply));
                    sync (receive reply)
  let put cell x = sync (send cell (Put x))
  let new_cell x =
      let cell = new_channel () in
      let rec serve x = match sync (receive cell) with
        | Get(reply) -> sync (send reply x);
                        serve x;
        | Put(y) -> serve y; in
      create serve x; cell
```

request for server to send data on that channel

tail recursive and always calls itself to keep the server running...

Locks:

⚡ Two threads aquiring two seperate locks in different order may lead to a deadlock!

```
module Lock = struct
  type ack = unit channel          <- unit channel since you just send acquire/release
  type lock = ack channel
  let aquire lock =
      let ack = new_channel () in
      sync (send lock ack); ack
  let release ack = sync (send ack ())
  let new_lock () =                <- wait for another thread to aquire the lock
      let lock = new_channel () in
      let rec acq_server () = rel_server (sync (receive lock))
  and let rel_server ack =         <- wait for release of the lock
      sync (receive ack); acq_server () in
  create acq_server (); lock
```

Semaphores:

↳ When you have more than one copy of a resource...

```
module Sema = struct
    type sema = unit channel option channel
    let up sema = ...            <- release a resource
    let down sema = ...          <- aquire a resource
    let new_sema n = ...
```

accumulating a parameter, maintaining the number of free resources

8.3 Selective Communication:

⇒ Since a thread does not know which communication rendez-vous will occur first, you need a non-deterministic choice between several actions.

- `select [ wrap (receive c1) (fun ... → ...);`
  `wrap (receive c2) (fun ... → ...)`
  `]`

  List of event waiting to

  once the event is received, the function is computed...

  → First event that appears will be used...

- `let select = sync (choose)`

⚡ An exception can only be handled within the thread where it has been raised.

8.4 Buffered Communication:

⇒ A channel for sending without blocking can be realized by the module Mailbox.

```
module mailbox = struct
  type 'a mbox = 'a channel * 'a channel
  let send (in,_) x = sync (send in x)
  let receive (_,out) = receive out
  let new_mailbox() =
    let in = new_channel () and out = new_channel () in
    let rec server q =
      if (is_empty q) then server (enqueue (sync (receive in)) q)
      else select [
          wrap (receive in) (fun y → server (enqueue y q));
          wrap (send out) (first q)) (fun () → let (_,q) = dequeue q in server q)
        ]
    in create server (new_queue ()); (in,out)
end
```

one channel for sending and one for receiving

the server will answer immediatly

Queue

⇒ Channels with synchronous communication allows to simulate asynchronous communication.