Functional Languages; Control Structures

Programming Language Design 2021

Hans Hüttel

Based on slides by Torben Mogensen and Fritz Henglein

February 23rd, 2022

Learning goals – Functional languages

- To be able to give simple examples of higher-order functions and explain why they are higher-order
- To be able to explain the notion of closures and why they are important for static scope rules
- To be able to explain the principles of the program transformations that make it possible to implement higher-order functions when we assume static scope rules: λ-lifting, closure conversion and defunctionalization.
- To be able to apply these program transformations.

Learning goals – Control structures

- To be able to classify control structures in imperative and functional programming languages.
- To be able to explain the various notions of jumps and their pros and cons.
- To be able to explain the notion of list homomorphism and why it is useful.
- To be able to explain why jumps can be implemented in a language without jumps.
- To be able to explain the characteristics of the various forms of multithreading.

Part I

Programming with functions

Why is it called functional programming?

Languages such as Haskell, F#, Standard ML, OCaml, Elm and Lisp are functional programming languages. In these languages, functions are first-class citizens. Functions are values, and like any other value

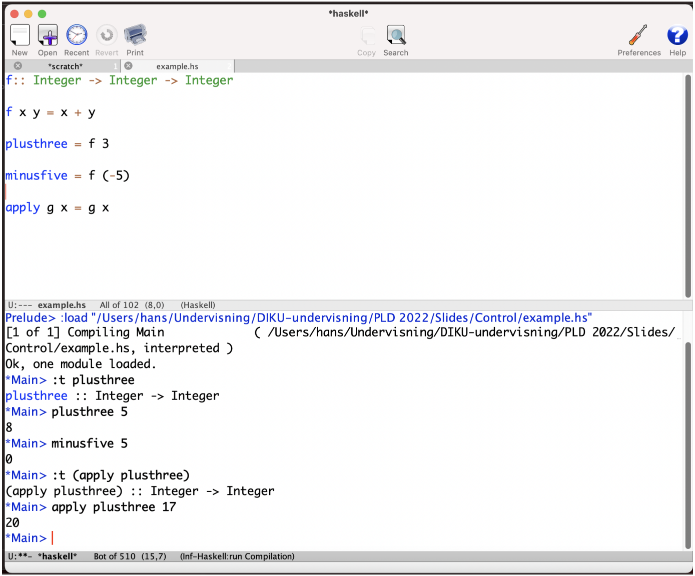
- There is a notation for function values, namely λ-notation
- Functions can be passed to functions as arguments
- Functions can be returned from functions as return values

Functional programming

Here is an example from Haskell.

```
f :: Integer -> Integer -> Integer

f x y = x + y

plusthree = f 3

minusfive = f (-5)

apply g x = g x
```

What happens here?



```
f:: Integer -> Integer -> Integer

f x y = x + y

plusthree = f 3

minusfive = f (-5)

apply g x = g x
```

U:--- example.hs All of 102 (8,0) (Haskell)

```
Prelude> :load "/Users/hans/Undervisning/DIKU-undervisning/PLD 2022/Slides/Control/example.hs"
[1 of 1] Compiling Main             ( /Users/hans/Undervisning/DIKU-undervisning/PLD 2022/Slides/
Control/example.hs, interpreted )
Ok, one module loaded.
*Main> :t plusthree
plusthree :: Integer -> Integer
*Main> plusthree 5
8
*Main> minusfive 5
0
*Main> :t (apply plusthree)
(apply plusthree) :: Integer -> Integer
*Main> apply plusthree 17
20
*Main>
```

U:**- *haskell* Bot of 510 (15,7) (Inf-Haskell:run Compilation)

Higher-order and first-class functions

In programming languages, a function can be a

First-order function: A first-order function neither accepts functions as arguments nor returns them.

Higher-order function: A higher-order function accepts functions as arguments or returns a function as a value. Functional programming languages have higher-order functions, as do many other languages.

Some functions have no free variable occurrences in their body. If we want to pass or return a function of this form, it is enough to use a *function pointer* (the address of code of its body).

But how do we implement higher-order functions that allow us to take *non-closed* functions as argument or to return such functions?

Closures

When we assume static scope rules, we need to know the following about a function when we make use of it:

- The code of the function
- The bindings that existed when the function was declared (that is, an environment)

Such a pair

$$\langle \text{code}, \mathit{env} \rangle$$

is called a closure. This is a central notion.

The funarg problem

The downwards funarg problem is that of how to pass a function as a parameter – and how to implement this in a first-order setting.

The problem is not particular to functional programming languages but is more general.

In ALGOL 60, PASCAL we only have the downwards funarg situation, so a stack is enough. We use a static link: Pointer to activation record with bindings for free variables.

The funarg problem

The upwards funarg problem is that of returning (or otherwise transmitting ”upwards”) a function from a function call – and how to implement this in a first-order setting. Here a stack is not enough!

A simple example is the Haskell function

```
compose f g = \x -> g ( f (x))
```

If we call compose f1 f2, the result should be that a function is returned. In the call, activation records for f1 and f2 will be placed on the stack. But after the call finishes, they are gone! But we need to know f1 and f2 to make use of the function that was returned.

Part II

λ-lifting

Implementing nested (local) functions

In the following, we deal with the problem of free identifiers in a setting where we assume static scope rules.

The approach (which we also mentioned in the podcast about scope rules and parameter passing) is to turn free identifiers in a function body into additional, implicit parameters of the function.

If f can modify free identifiers (some functional languages are also imperative), they should be passed as references.

Implementing nested (local) functions

We study three program transformations that are used to transform programs such that we can more easily implement local functions.

- λ-lifting: How to move nested function definitions to top-level.
- Closure conversion: How to implement higher-order functions in a first-order setting
- Defunctionalisation: How to remove function pointers of λ-lifted code.

Nested (local) functions: Example

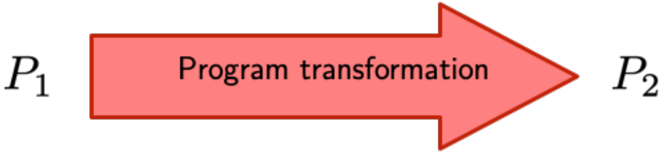
```
fun f x =
  let val a = x + 1
      fun g y =
        let val b = x − 1
            fun h z = if z = 0 then b + x else
                f b + a
        in h y + h b end
  in g a + 3 end
```

- f is closed (has no free variables).
- g also depends on the values of x, a.
- h also depends on the values of x, a, b.

so the free variables are really implicit parameters of g and h.

λ-lifting

The technique of λ-lifting is what one calls a program transformation.



P_1

Program transformation

P_2

A program transformation is a mapping that takes a program P_1 and generates another program P_2 that has the same behaviour as P_1 but also satisfies certain properties that P_1 may not have.

λ-lifting transforms a program P_1 with nested scopes in which some functions may be open into a program P_2 that does not have nested scopes and where all functions are closed.

λ-lifting

We transform open functions into closed functions by adding the implicit parameters to their free variables and moving all declarations to the same, global scope.

```
fun f x =
  let val a = x + 1
      fun g y =
        let val b = x − 1
            fun h z = if z = 0 then b + x else
                  f b + a
        in h y + h b end
  in g a + 3 end
```

The λ-lifted version

```
fun f'        x = let
                  val a = x + 1
                  in
                     g' x a a + 3
                  end

fun g' x a    y = let
                  val b = x - 1
                  in
                       h' x a b y + h' x a b b
                  end

fun h' x a b z = if z = 0 then b + x else f' b + a
```

The implicit parameters must of course now also be mentioned in the calls.

λ-lifting

We can group parameters into two tuples – the implicit (λ-lifted) and the explicit (given):

```
fun f'  ()     x = let
                 val a = x + 1
                 in
                    g' (x,a) a + 3
                 end

fun g' (x,a)   y = let
                 val b = x - 1
                 in
                      h' (x,a,b) y + h' (x,a,b) b
                 end

fun h' (x,a,b) z = if z = 0 then b + x else f' b + a
```

An overview of the algorithm

We assume that every function in the program has a unique name. If not, rename the functions such that this is the case.

We repeat the following until all free variables and local functions have been eliminated.

1. For every function f, replace every free variable in the definition of f with an additional argument to f. Pass that argument as an additional argument to every call of f.
2. Replace every local function definition with free variables with a global function that has the same definition.

Why is this useful?

With λ-lifting we eliminate the need for closures.

All functions become closed and all of them have exactly two parameters (that can be tuples).

We can implement nested scopes using techniques for global scopes and by using parameter passing (we return to this later!).

Part III

Higher-order to first-order

Closure conversion

Closure conversion is a program transformation that takes a program P_1 that contains functions that may be open and use functions as first-class values.

The transformation produces a program P_2 that has the same behaviour as P_1 but where only closed functions (without free variables) are used as first-class values.

The idea behind closure conversion

Every function receives its own closure $\langle \mathrm{code}, \mathit{env} \rangle$ as an extra argument.

When we need the value of a free variable inside the code of a function, we look it up in *env*. In this way, functions are closed (which is closures are called closures!).

We represent a function closure as a pair (closed function, values of free variables).

An example of closure conversion

Here is an anonymous function which takes an integer x and applies a function that mentions x to each element of a list lst. The map function is a higher-order function. The example is due to Xavier Leroy.

```
fun x lst ->
  let rec map f lst =
    match lst with [] -> []
                 | hd :: tl -> f hd :: map f
                   tl
  in
    map (fun y -> x + y) lst
```

Closure conversion

For a closure k =(code,env), we can retrieve its code with field_0 (k) and the variable x_i in env with field_i (k).

We get

```
fun x lst ->
  let rec map f lst =
        match lst with [] -> []
                     | hd :: tl -> field_0(f)(f,
                        hd) :: map f tl
        h (c,y) = let x = field_1(c) in (x + y,x
            )
  in
        map h
        lst
```

All data passed are now first-order values or function closures.

Defunctionalisation

The goal is to implement higher-order functions without function pointers, using first-order values only.

Instead of passing a function pointer we pass the *name* of the function.

Replace function pointer `f` in a function closure by a unique constructor `F`

Define evaluation function eval such that

$$\mathrm{eval}\ (F\ x)\ y = f\ x\ y$$

Defunctionalisation: Example

Consider the program

```
f' () (x, p) = p x + p 1
g' () a      = (f'() (a, h' a)
                + f'() (17, g'()), h' a)
h' a c       = a + c
```

For each function we define a term constructor.

Next we define a function eval that maps constructors to the top-level functions such that

```
eval (F ()) (x, p) = f' () (x, p)
eval (G ()) a      = g' () a
eval (H a) c       = h' a c
```

Defunctionalisation: Example

Here is the definition of the evaluation function.

```
eval (F ()) (x, p) = p x + p 1
eval (G ()) a   = (f'() (a, h' a) +
                   f'() (17, g'()), h' a)
eval (H a) c       = a + c
```

Next, we replace function calls by the corresponding term constructors.

```
eval (F ()) (x, p) = p x + p 1
eval (G ()) a   = (F () (a, H a)
                + F () (17, G ()), H a)
eval (H a) c       = a + c
```

Defunctionalisation: Example

Finally, we must remember to use the eval function at all of these new "function calls".

```
eval (F ()) (x, p) = eval p x + eval p 1
eval (G ()) a      = (eval (F ()) (a, H a) +
                      eval (F ()) (17, G ()), H a)
eval (H a)  c      = a + c
```

We now have a program that uses *only one function*, namely eval. This function is called dispatch in the lecture notes.

Pros and cons: Defunctionalization

- There are no closures or function pointers.
- Defunctionalization requires an analysis of the entire program: We have to find all top-level functions that can be passed to another function.
- eval can be statically typed with generalized algebraic data types, but not with ordinary data types.

Part IV

Control structures

Data-dependent execution

A program rarely performs the same *sequence* of computation steps for every input.

- You may want to *choose* between different possible computations depending on data.
- You may want to *repeat* some computations a number of times that depend on data.
- You may want to *interleave* two or more sub-computations that depend on each other.

To this end, programming languages have *control structures*.

Forms of control

- Unstructured (jumps)
- Structured:
  - Conditionals
  - Loops
- Exceptions and continuations
- Function calls
- Concurrency

Some of these only make sense for imperative, non-functional languages.

Part V

Conditionals, loops and breaks

Structured control structures

Structured control structures are explicitly delimited by scopes (nested blocks of code).

The only kind of overlap allowed is nesting.

In the following we consider

- Conditionals: Data-dependent choice between different paths (branches).
- Loops: Repetition.
- Breaks: Exit from structured control.

Biconditionals

Biconditionals are control structures that provide a two-way choice based on a truth value.

- In statements, the else-part is optional in some languages. This is then equivalent to having *skip* as the else-part.
- In expressions, there has to be an else-part (why?). Some languages use biconditionals in their definition of Boolean connectives: An expression b_1 **and** b_2 is equivalent to the biconditional expression **if** b_1 **then** b_1 **else** b_2 – so b_2 is only evaluated if we have to. This is known as short-circuiting and was introduced by John McCarthy.
- Sometimes the syntax will allow multiple then-parts (`elsf`, LISP COND). This is the same as **if-then-else-(if-then-else-(if-then-else...))**

Multiway-conditionals

Multiway-conditionals are based on matching patterns to a value. We know them from **case...of**, **switch** and **match-with** constructs.

The behaviour of a multiway-conditional depends on what patterns we allow and how they are matched. Some design choices here are

- Should patterns be disjoint or may they overlap? Which branch do we execute if more than one pattern matches?
- Patterns can allow variables – are they instantiated? Do we allow non-linear patterns, in which variables can appear multiple times?

Fall-through

Multiway conditionals in many C-like languages allow for fall-through: A **switch** statements allow multiple branches to be executed in the same match. This will happen if we do not add a break statement to a branch. Then the flow of control simply proceeds to the next branch!

Fall-through

The fragment

```
{int n = 2;
     switch (n) {
     case 1: {cout << "Eins␣\n";}
     case 2: {cout << "Zwei␣\n";}
     case 3: {cout << "Drei␣\n";}
     default: {cout << "Etwas␣anders␣\n";
     }}
}
```

will output

```
Zwei
Drei
Etwas anders
```

Loops

Type of loop	Example
Type 1: Number of iterations known at compile-time.	**do/for** loop with static bounds
Type 2: Number of iterations known when starting loop.	**do/for** loops with dynamic bounds, but no modification of loop counter variable inside body. Loop over collection (map, fold, foreach).
Type 3: Number of iterations known only when loop finishes (if it ever finishes!)	**repeat until** and **while** loops. **do/for** loop with modification of loop counter variable.

Properties of the three kinds of loops

Type 1: Can be fully unrolled by compiler.
Equivalent to straight-line code (data-independent sequence of computation steps).

Type 2: Known to terminate (if body does).
Can be vectorized if iterations do not interfere, or if the loop is equivalent to a fold/reduce with an associative operator.

Type 3: Required for Turing completeness.

Bulk operations

Bulk operations (aka iterators) appear in functional languages where loops usually do not exist, but also in e.g. Java, Python, Kotlin and other imperative languages.

They are implicit (type-2) loops over all elements in a collection: **map, fold, filter, scan, zip, list comprehensions, . . .**

This is a concise notation and abstracts away the order of evaluation. Using iterators can be faster than using loops – the interpretation overhead can be eliminated, and in a setting without side effects one can exploit parallelism.

Well-behaved list operations

Some iterators satisfy useful properties that we can make use of.

Let sum be a function that computes the sum of elements in a list and let $++$ be list concatenation.

Then we have that

$$\mathsf{sum}(\mathsf{l1} \mathbin{++} \mathsf{l2}) = \mathsf{sum}(\mathsf{l1}) + \mathsf{sum}(\mathsf{l2})$$

Or let max be a function that finds the maximal elements in a list. Herewe have that

$$\mathsf{max}(\mathsf{l1} \mathbin{++} \mathsf{l2}) = \mathsf{max}(\mathsf{max}(\mathsf{l1}), \mathsf{max}(\mathsf{l2}))$$

List homomorphisms

sum and max are examples of list homomorphisms. In the textbook we simply call them homomorphic functions.

A function f is a list homomorphism if there exists a $\oplus$ such that $f(x{+}{+}y) = f(x) \oplus f(y)$. For sum, $\oplus$ is $+$, and for max, $\oplus$ is max applied to pairs of numbers.

Other examples are find and minimum.

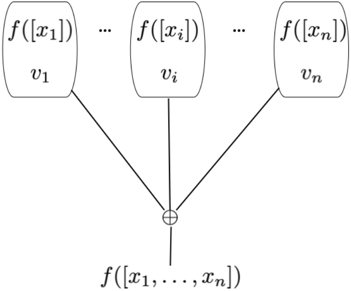
Since $++$ is associative, we know that $\oplus$ must also be associative on the range of f.

Map homomorphisms

A function f is a *map homomorphism* if it is a list homomorphism and $\oplus = ++$.

Examples are map and filter .

Homomorphisms are useful!



Homomorphisms allow for efficient parallelisation: We can compute the value of f for sublists in parallel and combine the values with $\oplus$. This idea is used in languages such as Map-Reduce, Futhark, NESL, ...

Breaks

There are different kinds of breaks that can appear in structured control structures:

- Short-circuit evaluation of Boolean expressions can also be though of as a kind of break.
- `break` is of course a break. If we execute it, we exit a loop or conditional before it finishes.
- `continue` skipsto the end of the loop body, but will not exit the loop.

Breaks can also be thought of as jumps.

Part VI

Jumps

Jumps

A jump transfers control to an indicated position in the program.

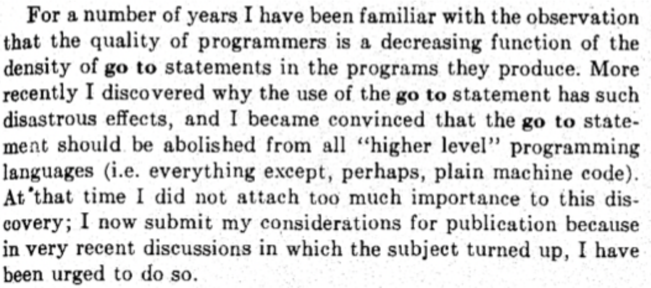
Jumps are often seen as low-level constructs – they are found in assembly language and appear in many early higher-level programming languages

Jumps and scopes

- In FORTRAN, COBOL and BASIC jumps are global.
  You can jump to anywhere from anywhere.
- In ALGOL 60 and PASCAL, jumps are scoped. It is possible to jump out of a block, but not into a block.
  Pascal allows jumps out of a procedure body!
- In C: We can jump into or out of blocks.
  The GCC extension has assigned jumps.

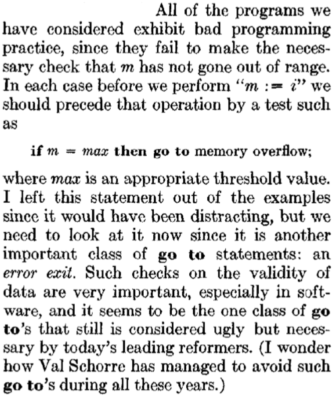
To goto or not to goto?

Jumps make programs harder to read, some have argued. In 1978, Edsger Dijkstra published his short paper "Goto Statement Considered Harmful".



For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

Structured programming with gotos



All of the programs we have considered exhibit bad programming practice, since they fail to make the necessary check that *m* has not gone out of range. In each case before we perform "*m* := *i*" we should precede that operation by a test such as

if *m* = *max* **then go to** memory overflow;

where *max* is an appropriate threshold value. I left this statement out of the examples since it would have been distracting, but we need to look at it now since it is another important class of **go to** statements: an *error exit*. Such checks on the validity of data are very important, especially in software, and it seems to be the one class of **go to**'s that still is considered ugly but necessary by today's leading reformers. (I wonder how Val Schorre has managed to avoid such **go to**'s during all these years.)

In 1974, Donald E. Knuth published a paper in which he argued that jumps are not necessarily always bad. Exceptions (considered later) are meant as a nicer approach to having error exits.

Eiichi Goto

The Japanese computer scientist Eiichi Goto is known for inventing one of the first general-purpose computers in Japan. He once said to Donald Knuth that everybody wants to eliminate him!

Error exits

When an error occurs, we can return an error or exception value, such as

- Null pointer.
- Empty list.
- Negative number.
- `NONE` / `None` / `Nothing` from option type.

All but the last option are usually bad ideas!

Exception value handling easily gets overlooked. A better idea is to have a notion of exceptions and exception handling.

Exceptions

An exception is a *dynamically scoped* jump (local in or out of procedure).

Exceptions

- Can be raised/thrown explicitly by statement or implicitly by a run-time error.
- Can carry information to an *exception handler*.
- Can be used to signal errors or to implement backtracking.
- Makes evaluation order observable: We see which exception is raised first.

Continuations

Continuations allow us to understand what jumps do.

A continuation is a copy/part of a run-time stack.

A continuation represents “the remainder of the current program”.

- Call continuation: causes the program to abort the current computation and continue from the continuation.
- Replace current run-time stack with stack in continuation and jump to code pointer of topmost activation frame.

What we can describe with continuations

Exceptions. A try-catch block stores a continuation that starts at the handler. Activated by raising exception.

`setjmp()` in C. Activated by `longjmp()`.

`call-with-current-continuation (call/cc)`. (Scheme, SML/NJ, Racket) Continuation stored as function that can be called multiple times.

Continuations

How much of the current continuation is copied?

Shallow copying: We copy only the code pointer/return address. Used for exceptions.

Top frame copied: Used by `setjmp()`. Allows coroutines (see later) to be implemented.

Deep copying: We copy the entire return stack! Used by call/cc. Slow unless stack is heap-allocated (as in Scheme, SML/NJ, Racket).

Function calls as control

Tail-calls to functions without parameters are basically jumps to named labels (and can be compiled as such), so mutually recursive functions can be as unstructured (and as efficient) as using `GOTO`.

Calls are only more structured than jumps if

- You use arguments/results instead of side-effects.
- Functions return rather than tail-calling other functions.
- Functions don't use exceptions or continuations.
- You use maps, folds and scans instead of explicit recursion over data structure.

No parameters involved

Backus' FP language avoids named parameters and uses maps and fold instead of recursion (similar to APL):

$$\textbf{Def}\ \texttt{IP} \equiv (/+) \circ (\alpha\times) \circ \texttt{trans}$$

Part VII

Expressiveness

Which control structures do we need?

A programming language without control structures has *limited expressive power*. It is not Turing-complete – there are algorithms that we cannot implement.

The Böhm-Jacopini theorem

In 1966, Corrado Böhm and Giuseppe Jacopini proved that one can express every algorithm using only the following control structures:

- Sequential composition
- If-then-else statements
- While loops
- Assignments
- Expressions with natural numbers

In other words, a programming language with statements that follow the syntax

$$S ::= x := e \mid \texttt{if } e \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } e \texttt{ do } S \mid S_1; S_2$$

is Turing-complete and therefore all we need.

No jumps needed!

A consequence of the Böhm-Jacopini theorem is that *we do not need jumps*.

We can get rid of jumps using a simple translation.

The main idea is to think of labels as values that we test.

A language with jumps

Here is a language with jumps. We allow goto-statements as well as subroutines. We can enter a subroutine with `gosub` ℓ_1 `then` ℓ_2 where ℓ_2 is the return label. We return to this label from the subrouting using `return`.

Jump statements J have the form

$$\begin{aligned} J ::= {} & \texttt{goto } l \mid \text{if } C \text{ then } \ell_1 \texttt{ else } \ell_2 \\ & \mid \texttt{gosub } \ell_1 \texttt{ then } \ell_2 \mid \texttt{return} \end{aligned}$$

C ranges over a set of conditions. And $\ell_1, \ell_2 \ldots$ ranges over a set of *labels* and l can be a label or a variable.

A language with jumps

Basic blocks B are labelled - they contain a simple statement S without jumps or labels (we do not specify them here) and a statement J with jumps.

$$\begin{aligned} P &::= B_1, \ldots, B_n \quad (n \geq 1) \\ B &::= \ell : \; S; J \\ S &::= \ldots \end{aligned}$$

Translating unstructured to structured control

A program starts at label 1 and stops when control is passed to label 0. We use a stack (in the form of an array) to keep track of return addresses. We introduce a variable c that gives us the value of the current label (think of c as a program counter) and a stackpointer variable p.

We translate a program to a main repeat loop containing a large case statement that switches according to the value of c:

```
$p := 0;$
$c := 1;$
repeat
case $c$ of
    translated basic blocks
end
until $c = 0$
```

Translation of basic blocks

A basic block $\ell : S; J$ is translated to $\ell : S, T(J)$, where ℓ is now a case label.

The translation T of jump statements is defined by

$$T(\texttt{goto}\ l) = c := l$$

$$T(\texttt{if}\ C\ \texttt{then}\ l_1\ \texttt{else}\ l_2) = \texttt{if}\ C\ \texttt{then}\ c := l_1\ \texttt{else}\ c := l_2$$

$$T(\texttt{gosub}\ l_1\ \texttt{then}\ l_2) = \begin{cases} \texttt{stack[}p\texttt{]} := l_2 \\ p := p - 1 \\ c := l_1 \end{cases}$$

$$T(\texttt{return}) = \begin{cases} p := p - 1 \\ c := \texttt{stack[}p\texttt{]} \end{cases}$$

Note: This is not very efficient. We can normally do better by analyzing the structure of the program.

Part VIII

Multithreading

Multithreading

A program consists of multiple *threads* that take turns executing in such a way that, when control passes back to a thread, it continues from where it was.

This is concurrency (interleaved execution), not parallelism (simultaneous execution).

The following forms of multithreading are well-known:

- Coroutines
- Generators
- Threads with
    - Shared memory
    - Thread-local memory

Coroutines

Control is passed to another coroutine with `resume`.

```
coroutine A
{
  print "Twas brillig, and the slithy toves";
  resume B;
  print "All mimsy were the borogoves,";
  resume B;
}
coroutine B
{
  print "Did gyre and gimble in the wabe:";
  resume A;
  print "And the mome raths outgrabe.";
  resume A;
}

resume A;
```

(Apologies to Torben and transitively to Lewis Caroll.)

Generators

A generator is a mix of coroutine and function: Called with `next` instead of `resume` and uses `yield` instead of `return` to pass control and return value back to caller.

```
 generator random
{
int seed = 1111; while true do {
  seed := (seed * 8121 + 28411) mod 134456;
yield seed;
} }
      print next random;
      print next random;
      print next random;
```

This gives the output:

```
42290
64877
95920
```

Threads

Threads do not explicitly yield control (except when terminating), but can be interrupted and resumed at (almost) any time by an outside agent (usually the OS).

Two types of communication between threads:

- Through shared mutable state.
- By message passing.

Shared mutable state can give race conditions, so synchronisation of *mutually exclusive access* is needed.

Race conditions

Processes can share information by sharing variables but that can lead to race conditions.

Here are two threads running in parallel:

```
(x := 2;              (x := 7;
 x := x + 3)           x := x + 1)
```

After the thread have finished, x can have the value 5 but in another run it could instead have the value 11. It depends on the order in which the four assignments are executed! This kind of behaviour is something we would like to avoid.

Synchronization mechanisms for shared state

Semaphore: A flag that signals if a resource is in use. Requires atomic test-and-set operation. To avoid *busy waiting*, a queue of waiting threads can be used.

Monitor: Encapsulates a resource in class-like object that can be accessed only through methods. This prevents access without checking semaphore, and forces threads to release resources when a method returns. The methods must still use a semaphore to get access, but semaphore use is more localised.

Neither guarantee consistent states, so you still need to verify correctness, but with monitors the code that must be verified is smaller.

Message passing

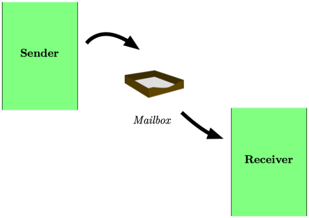
Threads (processes) do not share resources/data, so information is passed explicitly through messages.

A message is sent to a *mailbox* where another thread can read it. This is usually implemented as a (priority) queue.

Mailboxes/threads relation can be 1:1 or many:many.

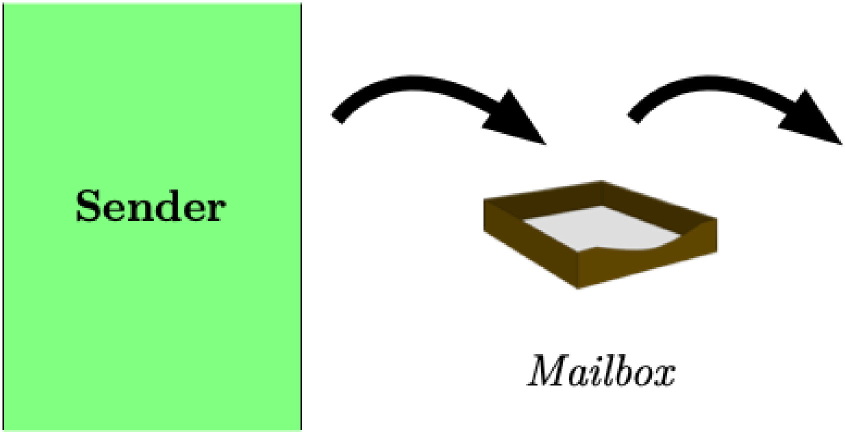
A message can be simple (non-pointer) data or a thread identifier.

Asynchronous message passing



Message passing can be synchronous: The sender proceeds before the message is read by the recipient.

Synchronous message passing



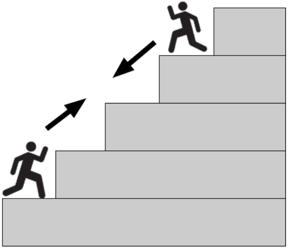
Message passing can be synchronous: The sender waits until the message is read by the recipient.

At the receiving end

The same design choices apply to the receiver:

Does the receiving thread wait until the message arrives when it tries to receive a message? Or will it proceed anyhow?

Deadlocks



Deadlock can occur if two threads wait for messages from each other. Here two people are waiting for the other person to say "I will go back so you can pass".

Language design is not enough here. We need program analysis techniques to be able to prevent parallel programs from deadlocking.