Chapter 7: Functional Programming Languages

Aarne Ranta

Slides for the book ”Implementing Programming Languages. An Introduction to Compilers and Interpreters”, College Publications, 2012.

Fun: a language that is much simpler but in many ways more powerful than an imperative language.

A fragment of Haskell, with a grammar that has less than 15 rules.

But there are conceptual challenges

- recursion
- call by name
- closures
- polymorphic type inference

Concepts and tools needed for Assignment 5.

Programming paradigms

Imperative, a.k.a. **procedural**: a program is a series of **statements** that affect a **state**.

Functional: a program is just an expression.

- executing a program is just evaluation
- no state is needed

But also imperative programs evaluate expressions. Thus functional programming only uses a subset of what imperative programs use.

Example of a functional program

In Haskell, and also in Assignment 5:

```
doub x      = x + x ;
twice f x = f (f x) ;
quadruple = twice doub ;
main        = twice quadruple 2 ;
```

A program is a sequence of function definitions.

`main` prints an integer (in Haskell, it is more general).

The syntax of function applications

Haskell: just put the function and its arguments one after the other,

$$f\,x\,y\,z$$

C (and ordinary mathematics), use parentheses and commas,

$$f(x, y, z)$$

Walk through the computation of main

```
    main
  = twice quadruple 2
  = quadruple (quadruple 2)
  = twice doub (twice doub 2)
  = doub (doub (doub (doub 2)))
  = doub (doub (doub (2 + 2)))
  = doub (doub (doub 4))
  = doub (doub (4 + 4))
  = doub (doub 8)
  = doub (8 + 8)
  = doub 16
  = 16 + 16
  = 32
```

At each step, we replace some part of the expression by its definition, or variables by their actual arguments.

The replacement operation is called **substitution**.

First-order functions

The `doub` function can be defined in C and Java as well:

```
// doub x    = x + x

int doub (int x)
{
  return x + x ;
}
```

But this mechanism is restricted to what is called **first-order functions**: the arguments cannot themselves be functions.

Second-order functions

Possible in C++ (as in Haskell): take functions as arguments, as the `twice` function does

```
// twice f x = f (f x)

int twice(int f (int n), int x)
{
  return f(f(x)) ;
}
```

Functions as values

In a functional language, functions are **first-class citizens**, just like numbers:

- function expressions have values even without arguments
- functions can be arguments of functions
- functions can be return values

In C++, a function cannot be a return value:

```
// quadruple = twice doub

// not possible in V++:
(int f (int x)) quadruple()
{
  return twice(doub) ;
}
```

We must pass an additional argument, which enables `quadruple` to return an integer and not a function:

```
int quadruple(int x)
{
  return twice(doub, x) ;
}
```

This corresponds to another definition in Haskell:

```
quadruple x = twice doub x
```

This definition has the same meaning as the one without `x`.

Function types

We write a two-place integer function`max`

```
  max : Int -> Int -> Int
```

(Haskell uses a double colon `::` for typing, but we stick to a single `:`.)

The notation is right-associative, and hence equivalent to

```
  max : Int -> (Int -> Int)
```

The typing rule for function applications is:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B}$$

Partial application

The typing rule permits

```
  max 4 : Int -> Int
```

This is a function that returns the maximum of its argument and 4. Notice

Application is left-associative: `max 4 5` is the same as `(max 4) 5`.

The tuple type

One could also force total application by using a **tuple** of arguments:

```
maxt : (Int * Int) -> Int
```

Tuples are a type of its own, with the following typing rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A{*}B}$$

Currying

An equivalence between functions over tuples and two-place functions:

$$(A * B) \to C \Longleftrightarrow A \to B \to C$$

Converting the first to the second is called **currying**, with reference to Haskell B. Curry.

Currying simplifies the semantics and implementation of programming languages: it is enough to work with one-place functions!

Anonymous functions

In imperative languages, functions must be explicitly defined and given names.

In a functional language, any expression can be turned into an**anonymous functions**, by **lambda abstraction**:

```
  timesNine = twice (\x -> x + x + x)
```

Syntactically, a **lambda abstract** is an expression

$$\lambda x.e$$

(using λ instead of \\). The typing rule for lambda abstracts.

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}$$

Function definitions as syntactic sugar

It is enough to have definitions of constance, because a function definition

$$f\,x_1\ \ldots\ x_n = e$$

can be expressed as the definition of a constant as a lambda abstract,

$$f = \lambda x_1.\ \ldots\ \lambda x_n.e$$

Also this simplifies the implementation: the environment need only map identifiers to types (in the type checker) or values (in the interpreter).

As C++ has no anonymous functions, we cannot write `timesNine` directly, but have to define a named tripling function first:

```
// triple x = x + x + x
int triple(int x)
{
  return x + x + x ;
}

// timesNine = twice triple
int timesNine(int x)
{
  return twice(triple, x) ;
}
```

But there is an experimental **Lambda Library** in C++ permitting anonymous functions.

Our language

We can start with only four expression forms:

```
Exp ::=
    Ident                                 -- variables, constants
  | Integer                               -- integer literals
  | "(" "\" Ident "->" Exp ")"            -- abstractions
  | "(" Exp Exp ")"                       -- applications
```

Operational semantics

Judgements of the usual form,

$$\gamma \vdash e \Downarrow v$$

read, "in the environment γ, the expression e evaluates to the value v".

Notice that evaluation cannot change the environment!

Thus we have a **purely functional language**, a language without side effects.

The environment is a set of values assigned to variables,

$$x := v,\ y := w, \ldots$$

Values

We need to be more general now, because we need function as values.

The simplest view:

- *values are expressions*
- evaluation converts an expression to another expression
- evaluation stops when it cannot proceed further

The resulting expression is often simpler, as in

$$2 + 3 * 8 \Downarrow 26$$

But it can also be more complex, as in

```
    replicate 20 1 ⇓ [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

What is a value

`2 + 3 * 8`

is not a value, because the evaluation can proceed.

$2^{31} - 1$

can be more interesting a value than 2147483647 as an *answer*, because it is more informative.

`2 + 3 * x`

is a value, in the sense that we cannot take it any further, because known what `x` is.

In another sense, it isn't: we were expecting to get a number, but we don't have one yet. We must first give a value to `x` and then perform multiplication and addition.

Free variables

A value may not contain **free variables**, which could get any values.

Bound variables are different. They are the ones appearing in lambda bindings.

The precise definition of the set *free*, free variables of an expression:

$$
\begin{aligned}
&\textit{free}(x) = \{x\} \\
&\textit{free}(i) = \{\} \\
&\textit{free}(f\ a) = \textit{free}(f) \cup \textit{free}(a) \\
&\textit{free}(\lambda x.b) = \textit{free}(b) - \{x\}
\end{aligned}
$$

An expression that has no free variables (i.e. $\textit{free}(e) = \{\}$) is a **closed expression**.

An expression that does have free variables is an **open expression**.

Closures

Notice: the closed expressions are those where all variables are bound by lambdas.

Special case: expressions with no variables.

Approximation of values: closed expressions.

In addition to lambda, it makes sense to close an expression by just giving values of its free variables. For instance,

```
  (2 + 3 * x){x := 8}
```

The general form: a **closure** is an expression e with an environmen γ:

$$e\{\gamma\}$$

Values revisited

We will use two kinds of values:

- integers
- closures of lambda abstracts

Special case: closed lambda abstract (with an empty closure).

Operational semantics

Variables:

$$\frac{}{\gamma \vdash x \Downarrow v} \; x := v \text{ is in } \gamma$$

Integer literals:

$$\gamma \vdash i \Downarrow i$$

Lambda abstracts:

$$\gamma \vdash (\lambda x.e) \Downarrow (\lambda x.e)\{\gamma\}$$

Function applications:

$$\frac{\gamma \vdash f \Downarrow (\lambda x.e)\{\delta\} \quad \gamma \vdash a \Downarrow u \quad \delta, x := u \vdash e \Downarrow v}{\gamma \vdash (f\,a) \Downarrow v}$$

(That's all!)

Some explanations

Abstractions: the body of the lambda abstract may contain variables than the one bound by the lambda. These variables get their values in the environment.

Applications: this comes out as a modification of the imperative rule in Chapter 5,

$$\frac{\gamma \vdash a \Downarrow u \qquad x := u \vdash s_1 \ldots s_n \Downarrow v}{\gamma \vdash f(a) \Downarrow v} \text{ if } V\ f(T\,x)\{s_1 \ldots s_n\} \text{ in } \gamma$$

Modifications:

- now enough to consider functions with one argument
- evaluation has no side effects now
- the function body: in the imperative language, it is a sequence of statements, $s_1 \ldots s_n$; in the functional language, it is a lambda abstraction body e, which is an expression.

- the function f: in the imperative language, it is always an explicitly defined function symbol; in the functional language, it can be any expression (for instance, a lambda abstract or an application).

Thus the evaluation of f is not simply a look-up in the function table. But we can just replace the look-up by a step of evaluating the expression f. This evaluation results in a closure, with a lambda abstract $\lambda x.e$ and an environment δ. Then $(f\,a)$ is computed by evaluating e in an environment where the variable x is set to the value of the argument a.

Example

Assume the function definition

```
  doub x = x + x
```

which means the same as

```
  doub = \x -> x + x
```

Compute `doub 4` as follows:

$$\frac{\vdash \texttt{doub} \Downarrow (\lambda x.x + x)\{\} \quad \vdash 4 \Downarrow 4 \quad x := 4 \vdash x + x \Downarrow 8}{\vdash (\texttt{doub}\, 4) \Downarrow 8}$$

The applied function has no free variables. But this is just a limiting case.

Example showing the need of closures

A two-place function,

```
plus x y = x + y
```

Evaluation of the expression `plus 3 4` (i.e. `((plus 3) 4)`):

$$\dfrac{\dfrac{\vdash \texttt{plus} \Downarrow (\lambda x.\lambda y.x+y)\{\} \qquad \vdash 3 \Downarrow 3 \qquad x := 3 \vdash (\lambda y.x+y) \Downarrow (\lambda y.x+y)\{x := 3\}}{\vdash (\texttt{plus}\,3) \Downarrow (\lambda y.x+y)\{x := 3\}} \qquad \vdash 4 \Downarrow 4 \qquad x := 3, y := 4 \vdash x+y \Downarrow 7}{\vdash ((\texttt{plus}\,3)\,4) \Downarrow 7}$$

Call by value vs. call by name

Call by value: evaluate the argument first, then the body (as above):

$$\frac{\gamma \vdash f \Downarrow (\lambda x.e)\{\delta\} \quad \gamma \vdash a \Downarrow u \quad \delta, x := u \vdash e \Downarrow v}{\gamma \vdash (f\,a) \Downarrow v}$$

Call by name: the body with unevaluated argument:

$$\frac{\gamma \vdash f \Downarrow (\lambda x \to e)\{\delta\} \quad \delta, x := a\{\gamma\} \vdash e \Downarrow v}{\gamma \vdash (f\,a) \Downarrow v}$$

Notice that γ is the proper way to close the expression a, because it is the environment in which a would be evaluated if we were performing call by value.

Examples

The difference is enorbous! Consider

```
infinite = 1 + infinite
first x y = x
main = first 5 infinite
```

With call by value,

```
main
= first 5 infinite
= (\x -> \y -> x) 5 (1 + infinite)
= (\y -> 5) (1 + infinite)
= (\y -> 5) (2 + infinite)
...
```

which leads to non-termination. Even though the function `first` ignores its second argument, call-by-value requires this argument to be evaluated.

With call by name,

```
main
= first 5 infinite
= (\x -> \y -> x) 5 infinite
= (\y -> 5) infinite
= 5
```

There is no attempt to evaluate the second argument, because it is not needed by `first`.

Call-by-value and call-by-name are just two possible orders. But call-by-name has the property that it is "the most terminating" one: if there is *any* order that makes the evaluation of an expression terminate, then call-by-name is such an order.

Disadvantages of call by name

It may lead to some expressions getting evaluated many times: once for each time the argument is used.

Example:

```
doub x = x + x

doub (doub 8)
= doub 8 + doub 8  -- by name
= 8 + 8 + 8 + 8
= 32

doub (doub 8)
= doub 16          -- by value
= 16 + 16
= 32
```

Call by need

An intermediate strategy, used in Haskell.

As in call by name, the expression is not evaluated when it is put to the environment.

But when the value is needed for the first time, the result of evaluation is saved in the environment, and the next look-up of the variable will not need to compute it again.

To make this possible, evaluation changes the environment - just like in imperative languages.

Implementing an interpreter: the language

An extended language with two primitive forms of expressions: infix operations and if-then-else.

```
Exp3 ::= Ident
Exp3 ::= Integer
Exp2 ::= Exp2 Exp3
Exp1 ::= Exp1 "+" Exp2
Exp1 ::= Exp1 "-" Exp2
Exp1 ::= Exp1 "<" Exp2
Exp  ::= "if" Exp1 "then" Exp1 "else" Exp
Exp  ::= "\\" Ident "->" Exp
```

A program is a sequence of function definitions, each of which has the form

$$f\, x_1 \ldots x_n = e\,;$$

An example program

The function `pow` defines powers of 2 by recursion.

```
doub x = x + x ;
pow x = if (x < 1) then 1 else doub (pow (x-1)) ;
main = pow 30 ;
```

Execution of programs

Evaluate the expression `main`.

Environment: all functions mapped to their definitions.

Each definition looks like a closure, with an empty environment. For instance, the example program above creates the following environment:

```
doub := (\x -> x + x){}
pow  := (\x -> if (x < 1) then 1 else doub (pow (x-1))){}
main := (pow 30){}
```

Notice the empty environments {} in the values (closures) of each function.

(Putting the function definitions in these environments would be impossible: just try this with the recursive function `pow`!)

Operations on environments

$$\begin{array}{lll} \textit{Val} & \textit{lookup} & (\textit{Ident}\ x, \textit{Env}\ \gamma) \\ \textit{Env} & \textit{update} & (\textit{Env}\ \gamma, \textit{Ident}\ x, \textit{Val}\ v) \end{array}$$

The `lookup` function has to implement the **overshadowing** of identifiers:

- a variable overshadows a function symbol;
- an inner variable overshadows an outer variable.

The latter follows from the simple rule that, in $\lambda x \to e$, the *free occurrences* of x get bound in e.

Quiz: what is the value of

```
(\x -> \x -> x + x) 2 3
```

Answer: in the expression

```
  \x -> \x -> x + x
```

it is the second lambda that binds both variables in `x + x`. We get

```
(\x -> \x -> x + x) 2 3
  =  ((\x -> x + x){x := 2}) 3
  =   (\x -> x + x) 3
  =          3 + 3
  =            6
```

Syntax-directed interpreter code

Integer literal:

```
$\textit{eval}(\gamma, i)$ :
  return $i$
```

Alternatively, we can return $i\{\}$, if we uniformly want closures as values.

Variable expressions:

```
$\textit{eval}(\gamma, x)$ :
  $e\{\delta\} := \textit{lookup}(\gamma, x)$
  $\textit{eval}(\langle \textit{functions}(\gamma), \delta \rangle, e)$
```

We split the environment into a pair $\langle \textit{functions}, \textit{variables} \rangle$, with separate storage for functions and variables.

Arithmetic operations: reduced to integer operations in the implementation language:

```
$\textit{eval}(\gamma, a + b)$ :
  $u := \textit{eval}(\gamma, a)$
  $v := \textit{eval}(\gamma, b)$
  **return** $u + v$
```

$<$ has a similar rule, returning 1 if the comparison is true, 0 if it is false.

Conditionals: interpreted lazily, even if call by value is the general strategy:

```
$\mathit{eval}(\gamma, \texttt{if}\ c\ \texttt{then}\ a\ \texttt{else}\ b)$ :
 $u := \mathit{eval}(\gamma, c)$
 **if** $u = 1$
  $\mathit{eval}(\gamma, a)$
 **else**
  $\mathit{eval}(\gamma, b)$
```

Abstractions: return closures with the variables of the current environment,

```
*eval*$(\gamma, \lambda x.b)$ :
 **return** $(\lambda x.b)\{$*variables*$(\gamma)\}$
```

Notice that we take only the variables of the environment into the closure, not the function symbols.

Application is the most complex case.

A general rule for both call by value and call by name. The decision is made in just one point: when deciding what value to use for the bound variable when evaluating the body.

```
$\mathit{eval}(\gamma, (f\ a))$ :
 $(\lambda x.b)\{\delta\} := \mathit{eval}(\gamma, f)$
 **if** $\mathit{call\_by\_value}$
  $u := \mathit{eval}(\gamma, a)$
 **else**
  $u := a\{\mathit{variables}(\gamma)\}$
 $\mathit{eval}(\mathit{update}(\langle \mathit{functions}(\gamma), \delta\rangle, x, u), b)$
```

Implementing an interpreter

Less than 100 lines of Haskell code or a little more Java code.

The interpreter can be made parametrized on evaluation strategy, which is passed as a flag when the interpreter is called.

Type checking functional languages*

Our language has a type system known as the **simply typed lambda calculus**.

Two kinds of types:

- basic types, such as `int`;
- function types $A \to B$, where A and B are types.

The power comes from the unconstrained generation of function types from any other types:

```
int -> int
(int -> int) -> int
int -> (int -> int)
((int -> int) -> int) -> int
```

Type checking

Abstraction rule:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B}$$

Type checking is easy:

```
$\mathit{check}(\Gamma, \lambda x.b, A \to B)$ :
  $\mathit{check}(\mathit{extend}(\Gamma, x, A), b, B)$
```

Type inference

But what about type inference? Example:

```
\x -> x
```

has infinitely many types:

```
int                    -> int
(int -> int)           -> (int -> int)
(int -> int -> int) -> (int -> int -> int)
```

In fact, it has all types of the form

```
A -> A
```

Hence it is *impossible* to do type inference for all expressions - if we expect a unique type.

Typed abstraction

One way to solve the type inference problem is to include type information in syntax:

$$\lambda x : t.b$$

Polymorphism*

But a more common solution is a **polymorphic type system**: one and the same expression can have many types, usually depending on a **type variable**.

Introduced in ML in the 1970's, inherited by Haskell in the 1990's.

Inspired the **template system** C and the **generics** of Java.

Templates and generics

Simplest possible example: the identity function, in C++

```
// id : A -> A, id = \x -> x
template<class A> A id(A x)
{
  return x ;
}
```

and in Java

```
// id : A -> A, id = \x -> x
public static <A> A id(A x)
{
  return x ;
}
```

In both cases, `A` is a type variable. (C++ and Java use capital letters, Haskell uses small letters.)

The most general type

In C++ and Java, *calls* to polymorphic functions must indicate the actual types. This makes type inference easy.

In ML and Haskell, this is not required. But type inference works even then!

Hindley-Milner polymorphism has an algorithm that computes **the most general type**. Examples:

```
(\x -> x)             : a -> a
(\x -> \y -> x)       : a -> b -> a
(\f -> \x -> f (f x)) : (a -> a) -> a -> a
(\x -> x + x)         : int -> int
```

Notice: different variables mean more generality than the same variable. For example, `a -> b` is more general than `a -> a`.

How to infer the most general type

Start by introducing a variable `t` for the type of the expression:

```
(\f -> \x -> f (f x)) : t
```

Since the expression is a double abstraction, `t` must be a function type:

```
t = a -> b -> c
```

The body of the expression must of course obey this type:

```
f (f x) : c
```

Since `f` is used as a function here, it must have a function type:

```
f : d -> e
```

But since `f` is the variable bound by the first lambda, we also have

```
  f : a
```

and hence,

```
  a = d -> e
```

Thus the result of applying `f` must have type `e`. But it must also have type `c`, because `f (f x) : c`. What is more, it must also have type `d`, because `f` can be applied to its own result. Hence

```
  c = e = d
```

The type of `x` is on one hand `b` (as the second abstracted variable), on the other hand `d` (because `f` applies to `x`). Hence

```
  c = e = b = d
```

and, since `a = d -> e`,

```
  a = d -> d
```

We can now conclude with

```
  t = (d -> d) -> d -> d
```

as the most general type.

Mechanizing type inference

The procedure above was completely heuristic - a little bit like solving a Sudoku.

But there is a mechanical algorithm, using **unification** - a general method for solving a set of equations.

Unification takes two types (with variables) and returns a **substitution**, mapping type variables to types.

We will define the main type inference

$$\langle \textit{Subst}, \textit{Type} \rangle\ \textit{infer}(\textit{Exp}\ e)$$

and the auxiliary function finding **most general unifier**,

$$\textit{Subst}\ \textit{mgu}\ (\textit{Type}\ t, \textit{Type}\ u)$$

Substitution vs. context

Substitution: type variables to types.

Context: expression variables to types (as in Chapter 4).

We will keep the context implicit in the pseudocode, accessible by the functions

```
Type lookup (Ident $x$) // look up type of variable in context
Void extend (Ident $x$, Type $t$) // put new variable to context
Void free (Ident $x$) // remove variable from context
```

We also need, in the course of type inference

```
Ident fresh () // generate a new type variable
```

Applying a substitution to a type

We write

$$t\gamma$$

for applying γ to t, which means replacing the type variables in t with their values given in γ.

Example:

$$\texttt{(a -> c -> d)}\{\texttt{a:= d -> d, c:=d, b:=d}\} \Downarrow \texttt{(d -> d) -> d -> d}$$

(applied at one point in the type inference example above)

Code for type inference

Constants and variables are simple, and return the empty substitution $\{\}$:

```
$infer(i)$ :
  return $\langle \{\}, Int \rangle$

$infer(x)$ :
  $t := lookup(x)$
  return $\langle \{\}, t \rangle$
```

Lambda abstracts:

```
*infer*$(\lambda x.b)$ :
  $a$ := *fresh*() // variable for the type of x
  *extend*$(x, a)$
  $\langle \gamma, t \rangle$ := *infer*$(b)$ // infer the type of the body
  *free*$(x)$ // the new variable no more in scope
  **return** $\langle \gamma, a\gamma \to t \rangle$
```

Application: two slides later.

Example

We can now infer the type of the identity function mechanically:

```
$\mathit{infer}(\lambda x.x)$ :
  $a$ := $\mathit{fresh}()$
  $\mathit{extend}(x, a)$
  $\langle\{\}, a\rangle$ := $\mathit{infer}(x)$ // in context $x : a$
  **return** $\langle\{\}, a\{\} \to a\rangle$
```

By applying the empty substitution, we get the final type $a \to a$.

Type inference for function application

```
$\mathit{infer}(f\ a)$ :
  $\langle \gamma_1, t_1 \rangle := \mathit{infer}(f)$ // infer type of function
  $\langle \gamma_2, t_2 \rangle := \mathit{infer}(a)$ // infer type of argument
  $v := \mathit{fresh}()$ // variable for the return type
  $\gamma_3 := \mathit{mgu}(t_1\gamma_2,\ t_2 \to v)$ // combine information
  **return** $\langle \gamma_3 \circ \gamma_2 \circ \gamma_1,\ v\gamma_3 \rangle$
```

All information is finally gathered in the **composition of substitutions** $\gamma_3 \circ \gamma_2 \circ \gamma_1$.

Similar to the usual composition of functions:

$$t(\delta \circ \gamma) = (t\gamma)\delta$$

Defining unification

The function *mgu* takes two types and returns their **most general unifier**.

This is a substitution γ that gives the same result when applied to any of the two types:

$$t(\mathit{mgu}(t, u)) = u(\mathit{mgu}(t, u))$$

Of course, *mgu* can also fail, if the types are not unifiable.

Unification is defined by pattern matching on the type.

```
Subst mgu (Type $t$, Type $u$)

mgu($a_1 \to b_1,\ a_2 \to b_2$) : // both are function types
 $\gamma_1$ := mgu($a_1,\ a_2$)
 $\gamma_2$ := mgu($b_1\gamma_1,\ b_2\gamma_1$)
 return $\gamma_2 \circ \gamma_1$

mgu($v, t$) : // the first type is a type variable
 if $t = v$
   return $\{\}$
 else if occurs($v, t$)
   fail ("occurs check")
 else
   return $\{v := t\}$

mgu($t, v$) : // the second type is a type variable
 mgu($v, t$)

mgu($t, u$) : // other cases: succeeds only for equal types
 if $t = u$
   return $\{\}$
 else
   fail ("types not unifiable")
```

When does unification fail?

1. If the types are syntactically different, e.g.

- `Int` vs. `Double`
- `Int` vs. `a -> b`

2. By **occurs check**: if the type variable v occurs in the type t, then t and v are not unifiable.

- v vs. $v \rightarrow u$

Without occurs check, we would get

$$\{v := v \rightarrow u\}$$

which would need an "infinite type" $(\ldots(v \rightarrow u)\ldots \rightarrow u) \rightarrow u$

Another example

Type inference for function applications, with occurs check:

```
$\mathit{infer}(\lambda x.(x\ x))$ :
 $a := \mathit{fresh}()$
 $\langle \gamma, t \rangle := \mathit{infer}(x\ x)$ : // in context $x : a$
  $\langle \{\}, a \rangle := \mathit{infer}(x)$
  $\langle \{\}, a \rangle := \mathit{infer}(x)$
  $b := \mathit{fresh}()$
  $\gamma := \mathit{mgu}(a, a \to b)$ :
   $\mathit{fail}$ ("occurs check")
```

On the last line, *mgu* fails because of occurs check: a cannot unify with $a \to b$.

Self-application

A function cannot be applied to itself: occurs check blocks

```
\x -> (x x)
```

Quiz: however, a ”self-application” is completely legal in

```
(\x -> x)(\x -> x)
```

Can you explain why?