# Polymorphism

## Programming Language Design 2020

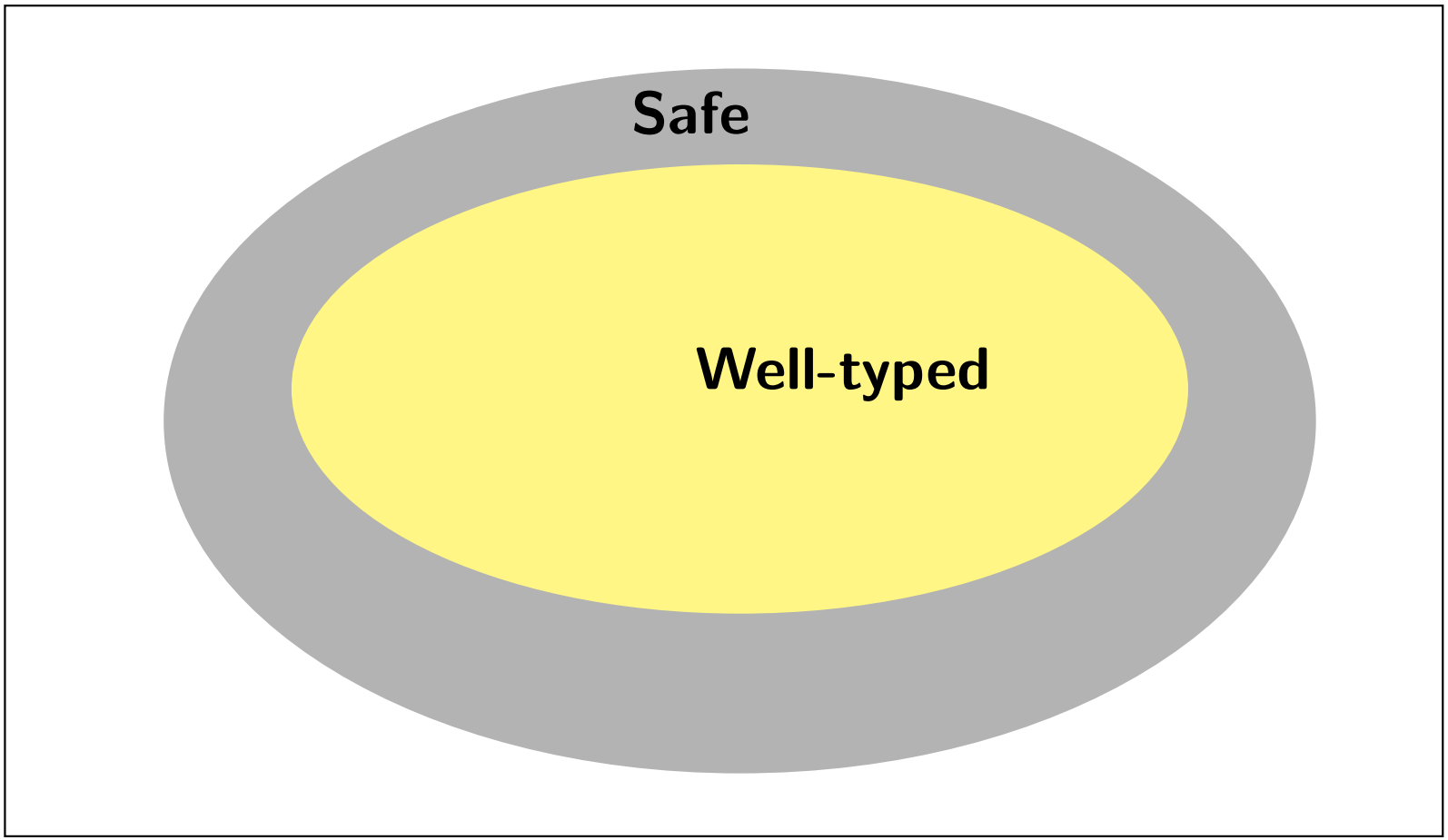
Hans Hüttel
hans.huttel@di.ku.dk

2 March 2021

# Part I

## Learning goals and motivation

# Every reasonable type system has slack




# Dealing with slack

```
id x = x

val myvalue = (id 7, id True)
```

In a simple type system, this tiny Haskell program would not be well-typed – but it ought to be.

Polymorphism turns out to be a good solution to the problem we just saw. If we can give the `id` function the type `t -> t` *for every type* `t`, that will be a way out.

# What is polymorphism?

The word *polymorphism* is Greek and means "many-shaped". In our setting it means that a piece of program can have multiple types depending on the context in which it occurs.

We are going to consider polymorphic functions, polymorphic values, and polymorphic value types – and different forms of polymorphism!

# Learning goals

- To be able to explain the differences between ad-hoc polymorphism and parametric polymorphism
- To be able to explain the notion of subtyping and argue why common type constructs are co- and contravariant
- To be able to identify examples of parametric polymorphism
- To be able to informally infer types in the setting of parametric polymorphism
- To be able to explain the central ideas in the Hindley-Milner algorithm for type inference
- To identify forms of polymorphism found in well-known programming languages such as Java and the languages of the ML family
- To understand the limitations of parametric polymorphism wrt. polymorphic recursion and mutable references

Part II

# Ad hoc polymorphism

# Ad hoc polymorphism

```
x = 1 + 2;
y = 2.718 + 3.141;
```

In this program fragment, + appears with two different interpretations:

- As addition of integers
- As addition of decimal (floating-point) numbers

This is an example of ad hoc polymorphism: The underlying algorithms for addition are very different in the two cases, but the symbol used is the same.

# Ad hoc polymorphism

Ad hoc polymorphism is also called *overloading*: The same function symbol can stand for many different functions that have different types and different implementations.

In FORTRAN, + can have at least the following types:

| | |
|---|---|
| $(int, int) \to int$ | integer addition, |
| $(int, real) \to real$ | convert 1st argument to real and use floating-point addition |
| $(real, int) \to real$ | convert 2nd argument to real and use floating-point addition |
| $(real, real) \to real$ | floating-point addition |

Some languages (C++, F#, ...) allow user-defined instances of overloaded operators.

# Overloading with dynamic typing

In dynamically typed languages, we see two different approaches to overloading.

**Values carry type description** There is a single shared operator implementation that gets selected at runtime for the instance that matches the argument types

**Type descriptors in values** Each value has a descriptor attached to it that points to the implementations of supported operators that we must use.

Advantage: We can add instances even at runtime.
Disadvantage: This is asymmetric!

# Interface polymorphism

A particular form of ad hoc polymorphism is the one that describes the family of operations that a type can support. That is called interface polymorphism.,

A type implements an interface by providing implementations of all operators in the interface specification.

We can use *explicit* or *implicit* interfaces.

# Explicit interfaces

Interfaces are explicitly declared and types are explicitly matched to interfaces at compile time. This can happen when the type is declared (Java, C#, ...) or, later, by specifying how an existing type implements an existing interface (Haskell).

An example from Haskell is the function

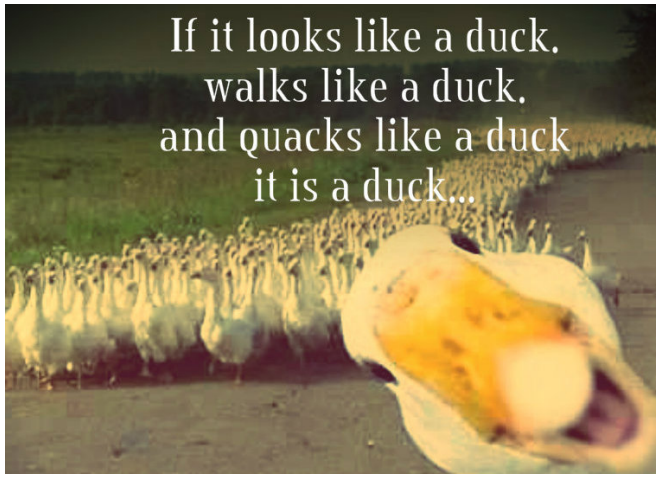
```
f (x,y) = if x < y then "lt" else "gt"
```

whose type is

```
f :: Ord a => (a, a) -> [Char]
```

This tells us that `f` is a function for every type `a` that has an implementation of the ordering predicate <. Types that allow for the ordering predicate live in the *type class* `Ord`.

# Implicit interfaces




Implicit interfaces are typically used in dynamically-typed language and are called "duck typing".

An interface is implicitly defined by how a function uses operators on its arguments. The interface is checked at runtime for individual operators, just as for simple overloading. All that matters is what an object can handle *right now*.

# Part III

# Subtype polymorphism

# Subtype polymorphism

Another approach to polymorphism is to define a specialization ordering on types.

The idea behind the subtype relation is that if $s \preceq t$, anywhere a *value* of type $t$ can be used, it is possible also to use a *value* of type $s$.

Think of $s \preceq t$ as saying that $s$ is a specialization of $t$.

# Subtype polymorphism

We require the subtype relation $s \preceq t$ to be a partial order; it must satisfy that

$$s \preceq s \qquad \preceq \text{ is reflexive}$$

$$s \preceq t \wedge t \preceq s \Rightarrow s = t \qquad \preceq \text{ is anti-symmetric}$$

$$s \preceq t \wedge t \preceq u \Rightarrow s \preceq u \qquad \preceq \text{ is transitive}$$

Sometimes, we need to require more of the ordering: That we can find the smallest common supertype or largest common subtype of two or more types. Sometimes, we also need a top or bottom type.

# Subtypes for types seen as sets of values

If a type is seen as a set of values, subtyping is the subset relation:

$$s \preceq t \Leftrightarrow \mathit{Values}(s) \subseteq \mathit{Values}(t)$$

Examples:

- `integer` $\preceq$ `real` $\preceq$ `complex`. This can require conversion of values.
- Subranges of integers: `1..9` $\preceq$ `0..10` $\preceq$ `integer` (Pascal) or `short` $\preceq$ `int` $\preceq$ `long` (C). This can require widening with sign extension.

# Subtypes for types seen as sets of capabilities

We could also think of subtyping as a capability ordering: Whatever you can do with a value of type $t$, you can also do with a value of a subtype $s$:

$$s \preceq t \Leftrightarrow \mathit{Capabilities}(s) \supseteq \mathit{Capabilities}(t)$$

Examples:

- Records (structs): $s \preceq t$ if $s$ has all the fields of $t$, and possibly more.
- Classes: $s \preceq t$ if $s$ has all the fields and methods of $t$, and possibly more. Inheritance is subtyping.
- Dually for sum (union or enumerated) types: $s \preceq t$ if $t$ has all the alternatives/values of $s$, and possibly more.

# Covariance and contravariance

An important notion, when subtyping is concerned, is that of variance for type constructs. Do we flip or preserve the subtype ordering, when we form a new type using a type construct?

If $O$ is a type construct, we say that $O$ is covariant if whenever $s \preceq t$, then $O(s) \preceq O(t)$.

If $O$ is a type construct, we say that $O$ is contravariant if whenever $s \preceq t$, then $O(t) \preceq O(s)$.

# Function types – covariance for result types

The function type construct is covariant for its result type:

If $s \preceq t$ and $t_A$ is any type, then $t_A \to s \preceq t_A \to t$.

For if we need a function of type $t_A \to t$, that is a function that will return a result known to be of type $t$. Since $s \preceq t$, we know that this also holds for a function of type $t_A \to s$. So such a function will serve our need,

# Function types – covariance for result types

The function type construct is covariant for its result type:

If $s \preceq t$ and $t_A$ is any type, then $t_A \to s \preceq t_A \to t$.

For if we need a function of type $t_A \to t$, that is a function that will return a result known to be of type $t$. Since $s \preceq t$, we know that this also holds for a function of type $t_A \to s$. So such a function will serve our need,

# Function types – contravariance for argument types

But the function type construct is contravariant for its argument type!

If $s \preceq t$ and $t_A$ is any type, then $t \to t_A \preceq s \to t_A$.

For if we need a function of type $s \to t_A$, that is a function that can be applied to any value of type $s$ and return a result known to be of type $t_A$.

But since $s \preceq t$, we know that a function of type $t \to t_A$ will also satisfy this – it can be applied to any argument of type $t$, including those that are also of the subtype $s$. So such a function will serve our need,

# Function types – contravariance for argument types

But the function type construct is contravariant for its argument type!

If $s \preceq t$ and $t_A$ is any type, then $t \to t_A \preceq s \to t_A$.

For if we need a function of type $s \to t_A$, that is a function that can be applied to any value of type $s$ and return a result known to be of type $t_A$.

But since $s \preceq t$, we know that a function of type $t \to t_A$ will also satisfy this – it can be applied to any argument of type $t$, including those that are also of the subtype $s$. So such a function will serve our need,

# Array types are invariant

Given a type `Animal` we can form the type `Animal[]`. Suppose `Cat` $\preceq$ `Animal`. What should hold for `Cat[]` and `Animal[]`?

Clearly, not every value of type `Animal[]` can be seen as a value of a type `Cat[]`. So the array construct cannot be contravariant, unless arrays are write-only.

But a value of type `Cat[]` also cannot be seen as a value of type `Animal[]`, for it should always be possible to update an array of type `Animal[]` with e.g. an entry of type `Dog`. So the array construct cannot be covariant either, unless arrays are read-only.

The only viable conclusion is that in general $s[] \preceq t[]$ if $s = t$. This is called invariance.

# More about variance of array and reference types

In Java and C#, the array type construct is covariant:

$s[] \preceq t[]$ if $s \preceq t$. However, as we have seen, this is not statically sound wrt. writes. We need to use downcasts (run-time checks) to ensure that this is safe when arrays are modified: we check if the type of the value to be stored is that of the run-time type of the array.

The mutable-variable type in C is also invariant!

We have that $\mathit{ref}\ s \preceq \mathit{ref}\ t$ only if $s = t$.

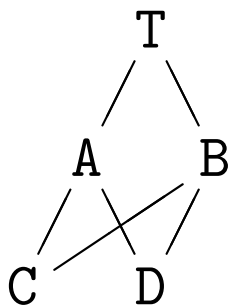
# Conditional expressions

Consider the conditional expression `A x = p ? c : d`, where `c` has type `C` and `d` has type `D`.

Then `A` must be a supertype of both `C` and `D` in order for this to be safe, since we must be able to use the value of the conditional expression no matter what it is.

There is an additional problem when multiple inheritance is allowed. What is then the type of `(p ? c : d)`?

`c` inherits from both `A` and `B`, while `d` inherits from both `A` and `B`.




# Part IV

# Parametric polymorphism

# Parametric polymorphism

Declaration with (implicit or explicit) type parameters.

- C++ templates (explicit)
- ML/F#/Haskell-style (Hindley-Milner) polymorphism (implicit)

# C++ templates

The identity function can be made polymorphic in C++:

```
template <typename T>
T identity(T x)
{
  return x;
}
```

Calls instantiate the type, e.g, `identity<int>(3)` or `identity<double>(3.1415)`.

The template declaration itself is not checked. Instead, an instance is created and checked at compile-time for each different use. This is essentially a form of macro expansion.

# Parametric polymorphism in ML, F# and Haskell

The functional languages ML, F# and Haskell allow for statically typed uniformly polymorphic definitions without explicit declarations of types.

In Standard ML we can define the length function on lists as

```
fun length [] = 0
  | length (x::xs) = 1 + length xs
```

The type of length is $\forall\alpha.\alpha\ \texttt{list} \to \texttt{int}$.

$\alpha$ is called a type variable. The type schema should be read as saying that `length` has type $\alpha\ \texttt{list} \to \texttt{int}$ *for every type* $\alpha$.

# Uniform parametric polymorphism

Parametric polymorphism as we see it in ML is uniform: Nothing in the definition of `length` needs to make assumptions about the element type of lists.

No ad-hoc overloaded operators are used, and there is no ability to detect the actual type when the function is called.

This also allows us to type check a polymorphic declaration *before* instantiation, essentially by checking a single instance.

When a *uniform representation* of values is used, instances can share a single implementation. Otherwise, compiler must generate one implementation per instance (or equivalence class).

# Hindley-Milner parametric polymorphism

The type system in the languages of the ML family is known as the Hindley-Milner type system.

It was discovered independently by Roger Hindley and Robin Milner in the 1970s.

# Types and type schemata

Types in the Hindley-Milner type system are built from:

- Type variables: `A`, `B`, ...
- Applications of type constructors (`*`, `->`, `int`, `list`, ...) to types.

A type schema is a $\forall$-quantified list of type parameters followed by a type description built from

- Type variables: `A`, `B`, ...
- Applications of type constructors (`*`, `->`, `int`, `list`, ...) to types or type parameters.
- Type parameters: $\alpha$, $\beta$, ...

# Instantiating type schemata

A type schema like `'a list -> int` is implicitly universally quantified over the free type variables, so it is formally written as

$$\forall\alpha(\texttt{list<}\alpha\texttt{>} \rightarrow \texttt{int})$$

We can instantiate a type schema to a type by instantiating its type variables. Here

$$\forall\alpha(\texttt{list<}\alpha\texttt{>} \rightarrow \texttt{int})\texttt{<string>} \;=\; \texttt{list<string>} \rightarrow \texttt{int}$$

where `list<string>` is the formal notation for `string list`.

# Polymorphic datatype declarations

In ML we can write polymorphic datatype declarations.

```
datatype 'a list = [] | 'a :: ('a list)
```

This tells us that a list of element type `'a` can be either the empty list `[]` or the composite list whose head is of type `'a` and whose tail is of type `'a list`.

`[]` and `::` are functions for building lists and we call them term constructors.

The empty list `[]` has the type schema $\forall\alpha(\texttt{list<}\alpha\texttt{>})$, and the cons term constructor `::` has the type schema $\forall\alpha(\alpha \times \texttt{list<}\alpha\texttt{>} \rightarrow \texttt{list<}\alpha\texttt{>})$.

# Hindley-Milner parametric polymorphism is good!

Its main advantages are that

- It is a simple but powerful type system for parametric polymorphism.
- The type system allows for an elegant algorithm for type inference.

The later type system for Haskell adds ad hoc polymorphism to Hindley-Milner in a nice way using type classes and retains full type inference (but we leave that story for some other occasion).

# Part V

# The Hindley-Milner algorithm for polymorphic type inference

# The underlying idea of type inference

Suppose we are given a program and must find the types of the entities in it. We do the following.

- Every entity in the program that has a type, is given a type variable.
- We build a collection of equations over types (aka type constraints) by analyzing the program. Here we use our knowledge of the types of pre-defined constants and functions.
- And then we solve the equations to find the values of the type variables. We use an algorithm called unification. If we cannot find a solution, this means that our program is not well-typed.

There are two equivalent approaches to this. Milner's $\mathcal{W}$ algorithm builds and solves the type constraints as it proceeds. MacQueen's algorithm first finds all the constraints and then uses unification.

# Type inference, informal example

We start with the untyped definition

```
fun length [] = 0
| length (x ::xs) = 1 + length xs
```

# Type inference, informal example

We then insert explicit type annotations in the definition of length, based on what we know about the types of constant and term constructors.

```
fun length ([] : list<A>) = (0 : int)
  | length (x (:: : B*list<B> -> list<B>) xs)
           = (1 :  int) (+ :  int*int -> int) (
               length xs)
```

# Type inference, informal example

After that, we propagate the type information so that all other subexpressions and pattern variables are also annotated. Here we use what we know from the first annotation.

For instance, in the second clause, we know from the type of `::` that `x` must have type `B` and `xs` must have type `list<B>`.

```
fun length ([] : list<A>) = 0 : int
  | length (((x : B) (:: : B*list<B> -> list<B>)
                     (xs :  list<B>)) :  list<B>)
           = ((1 :  int) (+ :  int*int -> int)
                ((length (xs : list<B>)) : int))
                     : int
```

# Type inference, informal example

```
fun length ([] : list<A>) = 0 : int
  | length (((x : B) (:: : B*list<B> -> list<B>)
                     (xs :  list<B>)) :  list<B>)
           = ((1 :  int) (+ :  int*int -> int)
                ((length (xs : list<B>)) : int))
                    : int
```

Each clause gives us information about the type of `length`:

| | |
|---|---|
| First rule: | `list<A> -> int` |
| Second rule: | `list<B> -> int` |
| Recursive call: | `list<B> -> int` |

These are *unified* to `list<A> -> int` by setting `A = B`.

The type is then *generalised* to the type schema $\forall\alpha(\texttt{list<}\alpha\texttt{>} \rightarrow \texttt{int})$.

# Type inference – the bindings used and updated

1. We use a *type environment* $\rho = [x_1 \mapsto t_1, f_1 \mapsto t_2, \ldots]$ to keep track of how names of functions, operators and constructors are bound to types or type schemata. Type environments are local to each point in the program, and are extended when new declarations are added.
2. We use a global set of *bindings* $\sigma = [A_1 \mapsto t_{A_1}, A_2 \mapsto t_{A_2}, \ldots]$ that keeps track of how type variables are bound to types. These bindings are updated (but never removed) by the inference algorithm. If the algorithm succeeds, the bindings will tell us the type of every entity in the program.

# Type inference – the functions needed

1. The *unification* function solves type constraints of the form $s = t$. It either updates the global bindings of type variables such that $s$ and $t$ are made equal (considering a bound type variable equal to the type to which it is bound), or *fails*, in which case the program has a type error.
2. The *instantiation* function instantiates a type schema to a new type by replacing type parameters with new type variables.
3. The *generalization* function introduces polymorphism by replacing unbound type variables with type parameters.

# Unification

When we unify $s$ and $t$ wrt. the bindings $\sigma$ we try to extend the bindings such that the equation $s = t$ can be satisfied. The rules are (ordered by priority)

1. If $s = t$, succeed with no further bindings and return $\sigma$.
2. If $s$ is a type variable $A$ *bound* in $\sigma$, find the type $u$ that $A$ is bound to and unify $u$ with $t$.
3. Symmetric rule for $t$.
4. If $s$ is an *unbound* type variable $A$, and $t$ does not contain $A$, extend $\sigma$ to $\sigma[t \mapsto A]$.
5. Symmetric rule for $t$.
6. If $s = c(s_1, \ldots, s_n)$ and $t = c(t_1, \ldots, t_n)$, where $c$ is a type constructor, unify $s_1$ with $t_1$ wrt. $\sigma$ and get $\sigma_1$. Then for $i = 2, \ldots, n$ unify $s_i$ with $t_i$ wrt. $\sigma_{i-1}$ and get $\sigma_i$. If any of these steps fails, **fail**. Otherwise succeed and return $\sigma_n$.
7. If none of these apply, **fail**.

# Occurs check in unification

Circular constraints are unsolvable, and we need to take care of that.

For instance, there is no type $A$ such that $\texttt{list} < A > = A$.

The "$t$ does not contain $A$" condition takes care of that.

1. If $t = A$, $t$ contains $A$.
2. If $t$ is an unbound type variable $B$ different from $A$, $t$ does not contain $A$.
3. If $t$ is a type variable $B$ bound to a type $u$, $t$ contains $A$ if $u$ contains $A$.
4. If $t$ is of the form $c(t_1, \ldots, t_n)$, $t$ contains $A$ if one of the $t_i$ contains $A$.

# Instantiation

Instantiation is needed when a (function) name bound to a type schema is *applied*.

For an example, suppose we are given the expression

```
1 :: []
```

We know that the cons constructor `::` has type $\forall\alpha.\alpha * \texttt{list}\,\alpha \to \texttt{list}\,\alpha$. But here this means that `::` is used with type $\texttt{int} * \texttt{list}\,\texttt{int} \to \texttt{list}\,\texttt{int}$.

A type schema $\forall\alpha_1, \ldots, \alpha_n(\tau)$ is *instantiated* to a type $t$ by, in $\tau$, replacing all occurrences of $\alpha_1, \ldots, \alpha_n$ by new, unbound type variables $\texttt{A}_1, \ldots, \texttt{A}_n$.

# Generalisation

Generalisation is done when a (function) *declaration* is made polymorphic.

Suppose we discover that a function `f` has type $A \to A$ and $A$ is not found in our type environment. Then `f` works for *every type* $A$, and `f` has the type schema $\forall\alpha.\alpha \to \alpha$.

A type $t$ is *generalised* to a type schema by first finding all unbound type variables $\mathtt{A}_1, \ldots, \mathtt{A}_n$ that occur in $t$, but *not* in the type environment. Then, a type schema $\forall\alpha_1, \ldots, \alpha_n(\tau)$ is constructed, where $\tau$ is a copy of $t$ where all occurrences of $\mathtt{A}_1, \ldots, \mathtt{A}_n$ are replaced by $\alpha_1, \ldots, \alpha_n$.

# Putting it all together

To perform type inference for a function declaration `fun f x = e` in a type environment $\rho$, we carry out the following recursive algorithm.

- Create two new type variables $A$ and $B$.
- Infer the type of $e$ in $\rho$ extended with bindings $x \mapsto A$ and $f \mapsto (A \to B)$.
  If this succeeds, we get a type $t$ for $e$. Otherwise, the entire algorithm fails.
- Unify $B$ with $t$.
- Generalize $A \to B$ (with respect to $\rho$) to a type schema $S$.
- Return $\rho$ extended with the binding $f \mapsto S$.

Note: In $e$, $f$ is bound to a type, but afterwards to a type schema.

# Part VI

# Going further

# Polymorphic recursion is difficult

Suppose we declared a datatype of nested lists in ML.

```
datatype Nested a = Cons a (Nested (List a)) |
    Empty
```

and tried to define a length function on nested lists as

```
fun nestlength Empty = 0
  | nestlength (Cons x nls) = 1 + nestlength nls
```

The Hindley-Milner algorithm will complain about this.

The reason is that the type of `nestlength nls` is `Nested (List a)`, not `Nested a`.

# Dealing with polymorphic recursion

When we perform type inference for a declaration `fun f x = e` in a type environment $\rho$, $f$ is first bound to an unknown type, but afterwards we know that this is a type schema.

We have to require whenever we mention `f` inside `e`, this occurrence of `f` (which is a recursive call) must have same type instance as the outer `f` in the definition.

If we allowed *polymorphic recursion*, type inference would become algorithmically undecidable.

If we really, really want polymorphic recursion, we have to add type annotations to our program – and then implicit typing is a thing of the past.

# Polymorphism and mutable values

Languages of the ML family are imperative; they allow for references and for assignment. We have a type constructor `ref t`. This becomes a challenge for polymorphism.

In `val x = []`, `x` is bound to the type schema $\forall\alpha(\texttt{list<}\alpha\texttt{>})$.

But now consider `y` in `val y = ref []`

If we allowed the type of `y` to be $\forall\alpha(\texttt{ref} < \texttt{list} < \alpha >>)$, the tiny program

```
y := 3 :: !y; "a":: !y
```

would give us a run-time error.

The solution is that we cannot generalise the type of variables of reference type.

# Polymorphism and mutable values

Languages of the ML family are imperative; they allow for references and for assignment. We have a type constructor `ref t`. This becomes a challenge for polymorphism.

In `val x = []`, `x` is bound to the type schema $\forall\alpha(\texttt{list<}\alpha\texttt{>})$.

But now consider `y` in `val y = ref []`

If we allowed the type of `y` to be $\forall\alpha(\texttt{ref} < \texttt{list} < \alpha >>)$, the tiny program

```
y := 3 :: !y; "a":: !y
```

would give us a run-time error.

The solution is that we cannot generalise the type of variables of reference type.

# Students and dogs

In the previous we wanted to build a system for registering students and their dogs. We defined the following types in C++.

```
struct student
    {
     char name
         [100];

     int age;
     float
         weight
         ;
         }
```

```
struct dog
     {
      char
          name
          [100];

      int age;
      float
          weight
          ;
           }
```

With name equivalence as our notion of type equivalence we can tell students and dogs apart.

# The case for name equivalence

But consider this value:

```
struct myFriend
    {
     "Brutus";
     21;
     45;
        }
```

What is its type?

# Structural equivalence and name equivalence

Type inference as we have described it assumes *structural equivalence*: Identical type expressions have the same type.

But most OO languages use *name equivalence*: Types are only identical if they have the same name.

Then type inference becomes difficult.

The constructor rule is omitted in unification, so if we want to deal with name equivalence, all variables holding composite values must be explicitly declared to have a named type.

# Constrained type variables

Earlier we saw the Haskell function

```
f (x,y) = if x < y then "lt" else "gt"
```

that has the type

```
f :: Ord a => (a, a) -> [Char]
```

The type of the function is a type scheme of the form $\forall\alpha \in \texttt{Ord}.(\alpha, \alpha) \rightarrow \texttt{[Char]}$. So we have a universal quantifier as in the type schemes we saw earlier, but we do not quantify over all types.

# Constrained type variables

Examples of this notion of constrained type variables that are bound to types that have specified properties are found in cases such as

- Type-class constraints in Haskell.
- Equality types in ML.
- Interface constraints in F#, Java and C#.

In type inference, we can deal with this by having constraints for type variables that we then also have to solve.

# Polymorphism in Standard ML

Ad-hoc polymorphism for arithmetic (defaults to `int`).

Core language uses polymorphic (Hindley-Milner) type inference with value restriction and equality-type constraints.

Additionally, the module system introduces a form of explicit *interface polymorphism*:

- A *signature* is an interface declaration.
- A *structure* is a collection of declarations that *implements* a signature.
- A *functor* is a function from (signature-constrained) structures to structures. This is a form of constrained, explicit parametric polymorphism over modules.

# Polymorphism in Java

- Ad-hoc polymorphism (arithmetic).
- Subtype polymorphism (subclassing with single inheritance).
- Interface polymorphism (with multiple inheritance).
- Constrained parametric polymorphism (generics).

A type in Java has the form

$$c \;\&\; i_1 \;\&\; \ldots \;\&\; i_n$$

where $c$ is a class name and the $i_j$ are interfaces. $c$ needs not implement the interfaces $i_1, \ldots, i_n$.

Even if `T1` $\preceq$ `T2`, `C<T1>` $\not\preceq$ `C<T2>`. Wildcards allow subtype constraints on type parameters.

# Polymorphism in Haskell

- Hindley-Milner type inference without value restriction (as no mutable variables) but with type-class constraints on type variables/parameters.
- Interface polymorphism (type classes) with multiple inheritance.
- Binding of types to type classes separated from type declaration (in instance declaration).
- Multi-parameter type classes.
- In Glasgow Haskell: Kind polymorphism (polymorphism over type constructors).

*No* subtype polymorphism.