# Polymorphism

Eduardo Bonelli

*"There may, indeed, be other applications of the system other than its use as a logic"*

Alonzo Church, 1932

March 14, 2019

# Topics

Polymorphism

Let-Polymorphism

Inference Algorithm

Extensions

# Polymorphism

What is the type of the following term?

$$\lambda x.x$$

- It clearly applies to arguments of any type
- To make sense of this, let us introduce type variables and type abstraction

$$\Lambda t.\lambda x : t.x$$

- This type of this function is typically written as

$$\Lambda t.\lambda x : t.x :: \Pi t.t \to t$$

# Polymorphism

$$\Lambda t.\lambda x : t.x :: \Pi t.t \to t$$

- In order to use it we instantiate it with different types
- For example, the following is the identity on $\mathbb{N}$ and has type $\mathbb{N} \to \mathbb{N}$ type

$$(\Lambda t.\lambda x : t.x)\,\mathbb{N}$$

- Hence, the following expression will evaluate to 3.

$$(\Lambda t.\lambda x : t.x)\,\mathbb{N}\,3$$

# Polymorphism

$$\Lambda t.\lambda x : t.x :: \Pi t.t \rightarrow t$$

- Another example, the identity on $\mathbb{B}$ and has type $\mathbb{B} \rightarrow \mathbb{B}$ type

$$(\Lambda t.\lambda x : t.x)\,\mathbb{B}$$

- Hence, the following expression will evaluate to true.

$$(\Lambda t.\lambda x : t.x)\,\mathbb{B}\ true$$

# Types of (Parametric) Polymorphism

$$\Lambda t.\lambda x : t.x :: \Pi t.t \to t$$

What is the set of types that we can apply it to?

1. Predicative Polymorphism
2. Impredicative Polymorphism
3. Type:Type Design

# Predicative Polymorphism

$$\Lambda t.\lambda x : t.x :: \Pi t.t \to t$$

What is the set of types that we can apply it to?

- Types of the simply typed lambda calculus (eg. $\mathbb{N}$ or $\mathbb{B} \to \mathbb{N}$), hence non-polymorphic types
- OCaml's let-polymorphism is this type of polymorphism (extended with a special rule for let).
- We will study let-polymorphism in detail today

# Impredicative Polymorphism

$$\Lambda t.\lambda x : t.x :: \Pi t.t \to t$$

What is the set of types that we can apply it to?

- Any type, including those constructed using $\Pi$ (such as $\Pi t.t \to t$ itself)
- Developed independently by Jean-Yves Girard (1971-72) and John Reynolds (1974)
- Called System F or $\lambda 2$
- We will study it in further detail later

# Type:Type Design

$$\Lambda t.\lambda x : t.x :: \Pi t.t \to t$$

What is the set of types that we can apply it to?

- We introduce a type of all the types $\star$
- We can apply the above function to any type, including those constructed using $\forall$ and the type $\star$
- This introduces much more that polymorphism:
  - Functions over types. Eg.
  $$(\Lambda t.\lambda x : t.x)\star \to \lambda x : \star.x$$
  - Dependent types. Eg.
  $$vec : \star \to \mathbb{N} \to \star$$
- We will study it when we introduce dependent types

# Monomorphism

```
# (fun f -> (f true, f 3)) (fun x -> 5);;
Error: This expression has type int but an expression was
expected of type bool

# let id = fun x -> x;;
val id : 'a -> 'a = <fun>
# (fun f -> (f true, f 3)) id;;
Error: This expression has type int but an expression was
expected of type bool
```

# Polymorphism

- In order to declare and use polymorphic functions we use let

```
# let f = fun x -> x in (f true, f 3);;
- : bool * int = (true, 3)
```

- Type inference is similar to the monomorphic case
- Details shall follow soon

# Type Inference in ML

We'll introduce two type systems: ML1 y ML2

- ML1
  - Abstract description
  - Convenient for introducing basic concepts of let-polymorphism
- ML2
  - Less abstract
  - The language in which we program in
  - We'll use it to perform type inference
- Both are equivalent in terms of typability

# ML1 – Type Expressions

$$\begin{array}{rcll} \sigma, \tau & ::= & s \mid Nat \mid Bool \mid \sigma \to \tau & \text{(monomorphic types)} \\ \pi & ::= & \sigma \mid \forall s.\pi & \text{(polymorphic type schemes)} \end{array}$$

Examples

- $\forall s.Nat \to s$
- $\forall s.s \to s$
- $\forall s.\forall t.(s \to t) \to s \to t$

Non-Examples

- $(\forall s.s \to s) \to Nat$

# ML1 – New Terms

$$
\begin{array}{rcll}
\sigma, \tau & ::= & s \mid Nat \mid Bool \mid \sigma \to \tau & \text{(monomorphic types)} \\
\pi & ::= & \sigma \mid \forall s.\pi & \text{(polymorphic type schemes)}
\end{array}
$$

$$
\begin{array}{rcl}
M & ::= & x \\
& \mid & true \mid false \mid if\ \ M\ \ then\ \ P\ \ else\ \ Q \\
& \mid & 0 \mid succ(M) \mid pred(M) \mid iszero(M) \\
& \mid & \lambda x : \sigma.M \mid M\,N \mid \\
& \mid & M\,\sigma \\
& \mid & \Lambda s.M \\
& \mid & \text{let } x : \pi = M \text{ in } M
\end{array}
$$

# ML1 – Type System (1/3)

$$\begin{array}{rcll} \sigma, \tau & ::= & s \mid Nat \mid Bool \mid \sigma \to \tau & \text{(monomorphic types)} \\ \pi & ::= & \sigma \mid \forall s.\pi & \text{(polymorphic type schemes)} \end{array}$$

- Typing Contexts

$$\Gamma = \{x_1 : \pi_1, \ldots, x_n : \pi_n\}$$

- Judgements

$$\Gamma \rhd M : \pi$$

- Note: third component of the judgement is now a polymorphic type scheme

# ML1 – Type System (2/3)

$$\frac{\Gamma(x) = \pi}{\Gamma \rhd x : \pi} \;(\text{T-Var})$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau} \;(\text{T-Abs})$$

$$\frac{\Gamma \rhd M : \sigma \to \tau \qquad \Gamma \rhd N : \sigma}{\Gamma \rhd M\,N : \tau} \;(\text{T-App})$$

Important: The types to the right of $\rhd$ in the typing judgements of (T-App) and (T-Abs) are monomorphic

# ML1 – Type System (3/3)

$$\frac{\Gamma \triangleright M : \pi \quad s \notin \Gamma}{\Gamma \triangleright \Lambda s.M : \forall s.\pi} \text{(T-TAbs)} \qquad \frac{\Gamma \triangleright M : \forall s.\pi}{\Gamma \triangleright M\,\sigma : \pi\{s := \sigma\}} \text{(T-TApp)}$$

$$\frac{\Gamma \triangleright M : \pi \qquad \Gamma, x : \pi \triangleright N : \sigma}{\Gamma \triangleright \text{let } x : \pi = M \text{ in } N : \sigma} \text{(T-Let)}$$

- Example: type

  let $g : \forall s.s \to Nat = \Lambda s.\lambda x : s.5$ in $\langle g\ Bool\ True, g\ Nat\ 3\rangle$
- Note: Removing the type annotations gives us the original system presented in [Milner, 1978, Damas and Milner, 1982]

# Comments on Rules

- In FP

  *"functions as first-class values"*

- In regards to let-polymorphism

  *"first-class values have monomorphic types"*

- Polymorphism is not "first-class"
  - Type variables are only quantified at the top-level; ej.

    $(\forall s.s \rightarrow s) \rightarrow Nat$ is not allowed
  - Type variables may only be instantiated with monomorphic types

    $(M\,\sigma)$

# Operational Semantics of ML1

$$V \quad ::= \quad \mathit{true} \mid \mathit{false} \mid \underline{n} \mid \lambda x : \sigma . M \mid \Lambda s . M$$

$$\frac{M \rightarrow M'}{M\,\sigma \rightarrow M'\ \sigma}\ (\text{E-TApp1})$$

$$\frac{}{(\Lambda s . M)\ \sigma \rightarrow M\{s := \sigma\}}\ (\text{E-TApp2})$$

$$\frac{M \rightarrow M'}{\text{let } x : \pi = M \text{ in } N \rightarrow \text{let } x : \pi = M' \text{ in } N}\ (\text{E-Let1})$$

$$\frac{}{\text{let } x : \pi = V \text{ in } N \rightarrow N\{x := V\}}\ (\text{E-Let2})$$

# Comment on Condition in (T-TAbs)

$$\frac{\Gamma \rhd M : \pi \quad s \notin \Gamma}{\Gamma \rhd \Lambda s.M : \forall s.\pi} \text{(T-TAbs)}$$

- Condition is necessary
- Example: $x : t \rhd (\Lambda t.x)\ Nat : Nat$
- Otherwise, preservation fails
- Parallel to what happens in logic

# Type Inference Problem – Recap

Given a term $U$ without type annotations, find a term $M$ (i.e. with type annotations), $\Gamma$ and $\sigma$ s.t.

1. $\Gamma \rhd M : \sigma$, for some $\Gamma$ and $\sigma$; and
2. $Erase(M) = U$

Note:

- We must define $Erase(\bullet)$
- We use $\sigma$ and not $\pi$ since

$$\Gamma \rhd M : \forall s_1. \dots . \forall s_n . \sigma \text{ iff } \Gamma \rhd M : \sigma\{\vec{s} := \vec{t}\}$$

  for fresh variables $\vec{t}$

# Type Inference Problem for ML1

$$
\begin{aligned}
\mathit{Erase}(\lambda x : \sigma.M) &\stackrel{\text{def}}{=} \lambda x.\mathit{Erase}(M) \\
\mathit{Erase}(\Lambda s.M) &\stackrel{\text{def}}{=} \mathit{Erase}(M) \\
\mathit{Erase}(M\,\sigma) &\stackrel{\text{def}}{=} \mathit{Erase}(M) \\
\mathit{Erase}(\text{let } x : \pi = M \text{ in } N) &\stackrel{\text{def}}{=} \text{let } x = \mathit{Erase}(M) \text{ in } \mathit{Erase}(N)
\end{aligned}
$$

- First clause $\Rightarrow$ from last class
- The others $\Rightarrow$ new

Given a term $U$ without type annotations, find a term $M$ (i.e. with type annotations), $\Gamma$ and $\sigma$ s.t.

1. $\Gamma \triangleright M : \sigma$, for some $\Gamma$ and $\sigma$; and
2. $\mathit{Erase}(M) = U$

- Left pending for now; will revisit when we introduce ML2

# ML2

- Less verbose formulation; easier on the programmer
- Same type expressions as ML1

$$
\begin{array}{rcl}
\sigma, \tau & ::= & s \mid Nat \mid Bool \mid \sigma \to \tau \\
\pi & ::= & \sigma \mid \forall s.\pi
\end{array}
$$

- Terms are simplified (hence also the type system)

$$
\begin{array}{rcl}
M & ::= & \ldots \\
& \mid & \cancel{M\,\sigma} \\
& \mid & \cancel{\Lambda s.M} \\
& \mid & \text{let } x : \pi = M \text{ in } M
\end{array}
$$

# ML2 – Type System

$$\begin{array}{rcl} \sigma, \tau & ::= & Nat \mid Bool \mid \sigma \to \tau \\ \pi & ::= & \sigma \mid \forall s.\pi \end{array}$$

- Sme typing contexts

$$\Gamma = \{x_1 : \pi_1, \ldots, x_n : \pi_n\}$$

- Judgements[1] only holding monomorphic types to the right of the $\rhd$ symbol

$$\Gamma \rhd_2 M : \sigma$$

- Notation: Let $S = s_1, \ldots, s_n$ be a sequence of type variables

$$\forall S.\sigma \text{ denotes } \forall s_1. \ldots .\forall s_n.\sigma$$

[1]Note use of subindex 2 to distinguish it from ML1

# ML2 – Type Rules (1/1)

$$\frac{\Gamma(x) = \forall \vec{s}.\tau}{\Gamma \rhd_2 x : \tau\{\vec{s} := \vec{\sigma}\}} \;(\textsc{T-Var})$$

$$\frac{\Gamma, x : \sigma \rhd_2 M : \tau}{\Gamma \rhd_2 \lambda x : \sigma.M : \sigma \to \tau} \;(\textsc{T-Abs}) \qquad \frac{\Gamma \rhd_2 M : \sigma \to \tau \qquad \Gamma \rhd_2 N : \sigma}{\Gamma \rhd_2 M\, N : \tau} \;(\text{T-A}$$

$$\frac{\Gamma \rhd_2 M : \tau \qquad \Gamma, x : \mathsf{gen}(\tau, \Gamma) \rhd_2 N : \sigma}{\Gamma \rhd_2 \mathsf{let}\ x : \mathsf{gen}(\tau, \Gamma) = M \ \mathsf{in}\ N : \sigma} \;(\textsc{T-Let})$$

- $\mathsf{gen}(\sigma, \Gamma) \stackrel{\mathrm{def}}{=} \forall S.\sigma$ where $S = FV(\sigma) \setminus FV(\Gamma)$
- $FV(\sigma)$ are the type variables in $\sigma$ (same with $FV(\Gamma)$)

# Examples

- let $g : \forall s.s \to Nat = \lambda x : s.5$ in $\langle g\ \ True, g\ \ 3\rangle$
- let $d : \forall s.(s \to s) \to s =$
  $\lambda f : s \to s.\lambda x : s.f\ \ (f\ \ x)$ in $\langle d\ \ (+1)\ 2, d\ \ not\ \ True\rangle$

# On Unicity of Types

- ¿$\Gamma \rhd_2 M : \tau \wedge \Gamma \rhd_2 M : \sigma \Rightarrow \tau = \sigma$?
  - No
  - Eg. let $x = \lambda z : s.z$ in $x$
- But one may prove that $M$ is typable with a unique principal type

Given $\Gamma$ and $M$, $\sigma$ is a principal type of $M$ under typing context $\Gamma$ if:

1. $\Gamma \rhd_2 M : \sigma$ is derivable and
2. $\Gamma \rhd_2 M : \tau$ derivable implies $\tau$ is an instance of $\sigma$

# Equivalence with ML1

## ML1 $\Rightarrow$ ML2

$$\Gamma \rhd M : \sigma \Rightarrow \Gamma \rhd_2 \textsc{EraseT}(M) : \sigma$$

where

$$\begin{aligned} \textsc{EraseT}(\Lambda s.M) &\stackrel{\text{def}}{=} \textsc{EraseT}(M) \\ \textsc{EraseT}(M\,\sigma) &\stackrel{\text{def}}{=} \textsc{EraseT}(M) \end{aligned}$$

## ML2 $\Rightarrow$ ML1

$$\Gamma \rhd_2 M : \sigma \Rightarrow \exists M'.\textsc{EraseT}(M') = M \wedge \Gamma \rhd M' : \sigma$$

- Inference for ML2 $\Rightarrow$ inference for ML1

# Inference Problem for ML2

$$Erase(\lambda x : \sigma . M) \stackrel{\text{def}}{=} \lambda x . Erase(M)$$
$$Erase(\text{let } x : \pi = M \text{ in } N) \stackrel{\text{def}}{=} \text{let } x = Erase(M) \text{ in } Erase(N)$$

- First clause $\Rightarrow$ from last class
- The other $\Rightarrow$ is new

Given a term $U$ without type annotations, find a term $M$ (i.e. with type annotations), $\Gamma$ and $\sigma$ such that

1. $\Gamma \rhd_2 M : \sigma$, for some $\Gamma$ and $\sigma$, and
2. $Erase(M) = U$

# Inference Problem for ML2

$$Erase(\lambda x : \sigma.M) \stackrel{\text{def}}{=} \lambda x.Erase(M)$$
$$Erase(\text{let } x : \pi = M \text{ in } N) \stackrel{\text{def}}{=} \text{let } x = Erase(M) \text{ in } Erase(N)$$

- First clause $\Rightarrow$ from last class
- The other $\Rightarrow$ is new

Given a term $U$ without type annotations, find ~~a term $M$ (i.e. with type annotations)~~, $\Gamma$ and $\sigma$ such that there exists some $M$ that verifies:

1. $\Gamma \triangleright_2 M : \sigma$, for some $\Gamma$ and $\sigma$, and
2. $Erase(M) = U$

- We will not be computing the decoration for $U$; this simplifies the algorithm

Polymorphism

Let-Polymorphism

Inference Algorithm

Extensions

# Inference Algorithm for ML2

$$\mathbb{W}(U, E)$$

- $U$ term without type annotations
- $E$ partial function that associates pairs $\langle typingcontext, type\rangle$ to type variables
  - Used for variables bound by `let`
  - $E = \emptyset$ in "top-level"
- First we present clauses defining $\mathbb{W}(U, E)$ on constants and variables, then we address the other constructors

# Inference Algorithm (constants and variables)

- First three clauses are the same as before

$$\begin{aligned}
\mathbb{W}(\mathit{0}, E) &\stackrel{\text{def}}{=} \emptyset \rhd \mathit{0} : \mathbb{N} \\
\mathbb{W}(\mathit{true}, E) &\stackrel{\text{def}}{=} \emptyset \rhd \mathit{true} : \mathbb{B} \\
\mathbb{W}(\mathit{false}, E) &\stackrel{\text{def}}{=} \emptyset \rhd \mathit{false} : \mathbb{B}
\end{aligned}$$

- This changes:

$$\mathbb{W}(x, E) \stackrel{\text{def}}{=} \begin{cases} \Gamma \rhd x : \sigma, & \text{if } E(x) = \langle \Gamma, \sigma \rangle \\ \{x : s\} \rhd x : s, & \text{otherw.}(s \text{ fresh variable}) \end{cases}$$

# Inference Algorithm

- The rest of the cases are the same as before except that we do not compute the type decoration
- For example
  - Let $\mathbb{W}(U, E) = \Gamma \rhd U : \tau$
  - Let $S = MGU(\{\tau \doteq \mathbb{N}\})$
  - Then

$$\mathbb{W}(succ(U), E) \stackrel{\text{def}}{=} S\Gamma \rhd U : \mathbb{N}$$

# Inference Algorithm (*let* case)

- Let $\mathbb{W}(U, E) = \Gamma_1 \triangleright U : \tau_1$
- Let $E' = E \cup \{x := \langle \Gamma_1, \tau_1 \rangle\}$
- Let $\mathbb{W}(V, E') = \Gamma_2 \triangleright V : \tau_2$
- Then

$$\mathbb{W}(\text{let } x = U \text{ in } V, E) \stackrel{\text{def}}{=} \Gamma_2 \triangleright \text{let } x = U \text{ in } V : \tau_2$$

# Examples

- let $i = \lambda x.x$ in $i\, i$
- let $g : \forall s.s \rightarrow Nat = \lambda x : s.5$ in $\langle g\ \ True, g\ \ 3\rangle$
- $\lambda x.$let $x = \lambda z.z\, x$ in $\langle x\, succ, x\, not\rangle$

# Properties of the Algorithm

1. Correctness
2. Completeness
3. Finds principal type

Items 2 and 3 are proved in [Damas and Milner, 1982]

# Complexity of Type Inference in ML

- Contrary to what was originally though, it is exponential [Mairson, 1990, Kfoury et al., 1990]
- Here is an example:

```
let x0=M (* M assumed to have some type sigma *)
in let x1=<x0,x0>
in let x2=<x1,x1>
in ...
in let xn=<xn-1,xn-1>
in xn
```

- Note that different uses of a let declaration have distinct occurrences of their type variables

# Extension 1 – Polymorphic Constants

```
data List a = Nil | Cons a (List a)
```

$$\begin{array}{rcl} \mathit{Nil} & :: & \forall s.\mathit{List}\ s \\ \mathit{Cons} & :: & \forall s.\mathit{List}\ s \to s \to \mathit{List}\ s \\ \mathit{caseL} & :: & \forall s.\forall t.\mathit{List}\ s \to t \to (s \to \mathit{List}\ s \to t) \to t \end{array}$$

We add observer (seen as a constant)

$$\frac{c : \forall \vec{s}.\tau}{\Gamma \rhd_2 c : \tau\{\vec{s} := \vec{\sigma}\}}\ (\textsc{Cst})$$

Example:

- Type $[1, 2, 3] :: \mathit{List\ Nat}$
- Type $\mathit{head} :: \forall s.\mathit{List}\ s \to s$ implemented using $\mathit{caseT}$

# Extensions of the Inference Algorithm

$$\mathbb{W}(c, E) \stackrel{\text{def}}{=} \emptyset \triangleright c : \tau \text{ if } c : \forall s_1. \ldots . \forall s_n . \tau$$

Example:

- $\mathbb{W}([1, 2, 3], E)$

# Let-Polymorphism and Expressions that Cause Side Effects

Let $P$ be the term

$$\text{let } r = \mathit{ref}(\lambda x.x) \text{ in } (r := (\lambda x : \mathit{Nat}.\mathit{succ}\ x); (!r)\, \mathit{true})$$

- $\mathbb{W}(P, \emptyset)$ returns ok
- What happens when we execute it?
- One possible solution: value restriction [Wright, 1995, Garrigue, 2004]
  - $x$ in let $x = M$ in $N$ can be treated polymorphically in $N$ only if $M$ is a value

$$\frac{\Gamma \rhd_2 V : \tau \qquad \Gamma, x : \mathrm{gen}(\tau, \Gamma) \rhd_2 N : \sigma}{\Gamma \rhd_2 \text{let } x : \mathrm{gen}(\tau, \Gamma) = V \text{ in } N : \sigma}\ (\text{T-Let})$$

- Not restrictive in practice

# Let-Polymorphism and Expressions that Cause Side Effects

$$\frac{\Gamma \rhd_2 V : \tau \qquad \Gamma, x : \mathsf{gen}(\tau, \Gamma) \rhd_2 N : \sigma}{\Gamma \rhd_2 \mathsf{let}\ x : \mathsf{gen}(\tau, \Gamma) = V \ \mathsf{in}\ N : \sigma} \ (\textsc{T-Let})$$

- Not restrictive in practice
- Typically taken care of by $\eta$-conversion

```
# let f = id id;;
val f : '_ a -> '_ a = <fun>
```

- After $\eta$-conversion

```
 # let g x = id id x;;
val g : 'a -> 'a = <fun>
```

# Extension 2 – Recursion

We replace:

$$\frac{\Gamma \rhd_2 M : \tau \qquad \Gamma, x : \mathsf{gen}(\tau, \Gamma) \rhd_2 N : \sigma}{\Gamma \rhd_2 \mathsf{let}\ x : \mathsf{gen}(\tau, \Gamma) = M\ \mathsf{in}\ N : \sigma}\ (\textsc{T-Let})$$

with:

$$\frac{\Gamma, x : \tau \rhd_2 M : \tau \qquad \Gamma, x : \mathsf{gen}(\tau, \Gamma) \rhd_2 N : \sigma}{\Gamma \rhd_2 \mathsf{letrec}\ x : \mathsf{gen}(\tau, \Gamma) = M\ \mathsf{in}\ N : \sigma}\ (\textsc{T-Letrec})$$

- Rule introduced in [Mycroft, 1984]

# Limitations of Monomorphic Recursion - Example 1

```
# let rec f = fun i x ->
    if i=0 then 7 else f (i-1) (x,x);;
Error: This expression has type 'a * 'b but an expression was expected
of type 'a
       The type variable 'a occurs inside 'a * 'b

# let rec f : 'a.  int -> 'a -> int = fun i x ->
    if i=0 then 7 else f (i-1) (x,x);;
val f : int -> 'a -> int = <fun>

# f 4 5;;
- : int = 7
```

# Limitations of Monomorphic Recursion - Example 1

$$
\begin{aligned}
\text{letrec } f \quad = \quad & \lambda i : Nat.\lambda x : s. \\
& \quad \mathit{if}\ \ \mathit{isZero}(i)\ \mathit{then}\ x \\
& \quad \mathit{else}\ f\ \ \mathit{pred}(i)\ \langle x, x\rangle \\
\text{in } (f\ 87)\ \mathit{true} &
\end{aligned}
$$

```
# let rec f : 'a. int -> 'a -> 'a = fun i x ->
    if i=0 then x else f (i-1) (x,x);;
Error: This expression has type 'a * 'a but an
    ↪ expression was expected
of type 'a
        The type variable 'a occurs inside 'a * 'a
```

- What does it evaluate to?
- Show that it is not typable (build a prospective derivation)
- The problem: $f$ is polymorphic only “outside” of $M$

# Limitations of Monomorphic Recursion - Example 2

```
# type 'a seq = Nil | Cons of 'a * ('a * 'a) seq;;

# Cons(1, Cons((2, 3),Cons(((4, 5), (6, 7)), Nil)));;
- : int seq = Cons (1, Cons ((2, 3), Cons (((4, 5), (6, 7)), Nil)))

# let rec size = function
  | Nil -> 0
  | Cons (_, xs) -> 1 + 2 * (size2 xs);;
Error: This expression has type ('a * 'a) seq
       but an expression was expected of type 'a seq
       The type variable 'a occurs inside 'a * 'a

# let rec size: 'a. 'a seq -> int = function
  | Nil -> 0
  | Cons (_, xs) -> 1 + 2 * (size xs);;
val size : 'a seq -> int = <fun>
```

- There are other ways to do this in OCaml (eg. fields of records and of object types can have polymorphic types)
- For more examples see [Hallett and Kfoury, 2005]

# Limitations of Monomorphic Recursion

$$\frac{\Gamma, \cancel{x : \tau}, x : \mathrm{gen}(\tau, \Gamma) \rhd_2 M : \tau \qquad \Gamma, x : \mathrm{gen}(\tau, \Gamma) \rhd_2 N : \sigma}{\Gamma \rhd_2 \mathsf{letrec}\ x : \mathrm{gen}(\tau, \Gamma) = M \ \mathsf{in}\ N : \sigma} \text{(T-LETREC)}$$

- Milner-Mycroft type system (1984)
- Inference undecidable for this system ([Henglein, 1993], and also [Kfoury et al., 1993])

Damas, L. and Milner, R. (1982).

Principal type-schemes for functional programs.

In DeMillo, R. A., editor, Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982, pages 207–212. ACM Press.

Garrigue, J. (2004).

Relaxing the value restriction.

In Kameyama, Y. and Stuckey, P. J., editors, Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings, volume 2998 of Lecture Notes in Computer Science, pages 196–213. Springer.

Hallett, J. J. and Kfoury, A. J. (2005).

Programming examples needing polymorphic recursion.

Electr. Notes Theor. Comput. Sci., 136:57–102.

Henglein, F. (1993).
Type inference with polymorphic recursion.
ACM Trans. Program. Lang. Syst., 15(2):253–289.

Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. (1990).
ML typability is dextime-complete.
In Arnold, A., editor, CAAP '90, 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings, volume 431 of Lecture Notes in Computer Science, pages 206–220. Springer.

Kfoury, A. J., Tiuryn, J., and Urzyczyn, P. (1993).
Type reconstruction in the presence of polymorphic recursion.
ACM Trans. Program. Lang. Syst., 15(2):290–311.

Mairson, H. G. (1990).
Deciding ML typability is complete for deterministic exponential time.
In Allen, F. E., editor, Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990, pages 382–401. ACM Press.

Milner, R. (1978).
A theory of type polymorphism in programming.
J. Comput. Syst. Sci., 17(3):348–375.

Mycroft, A. (1984).
Polymorphic type schemes and recursive definitions.
In Paul, M. and Robinet, B., editors, International Symposium on Programming, 6th Colloquium, Toulouse, April 17-19, 1984, Proceedings, volume 167 of Lecture Notes in Computer Science, pages 217–228. Springer.

Wright, A. K. (1995).

Simple imperative polymorphism.

Lisp and Symbolic Computation, 8(4):343–355.