Elements of Programming Languages
Lecture 8: Polymorphism and type inference

James Cheney
University of Edinburgh
October 23, 2023
Overview

- Last week we covered type definitions, records, datatypes, subtyping
- This week and next week, we will cover additional forms of abstraction
  - polymorphism, type inference
  - modules, interfaces
  - objects, classes
- Today:
  - polymorphism and type inference
Consider the humble identity function

- A function that returns its input:
  ```scala
def idInt(x: Int) = x
def idString(x: String) = x
def idPair(x: (Int, String)) = x
  ```

- Does the same thing no matter what the type is.
- But we cannot just write this:
  ```scala
def id(x) = x
  ```

(In Scala, every variable needs to have a type.)
Another example

- Consider a pair “swap” operation:

```scala
def swapInt(p: (Int, Int)) = (p._2, p._1)
def swapString(p: (String, String)) = (p._2, p._1)
def swapIntString(p: (Int, String)) = (p._2, p._1)
```

- Again, the code is the same in both cases; only the types differ.

- But we can’t write

```scala
def swap(p) = (p._2, p._1)
```

What type should `p` have?
Another example

Consider a higher-order function that calls its argument twice:

```scala
def twiceInt(f: Int => Int) = {x: Int => f(f(x))}
def twiceStr(f: String => String) =
  {x: String => f(f(x))}
```

Again, the code is the same in both cases; only the types differ.

But we can't write

```scala
def twice(f) = {x => f(f(x))}
```

What types should \( f \) and \( x \) have?
In Scala, function definitions can have *type parameters*

```scala
def id[A](x: A): A = x
```

This says: given a type `A`, the function `id[A]` takes an `A` and returns an `A`.

```scala
def swap[A,B](p: (A,B)): (B,A) = (p._2,p._1)
```

This says: given types `A,B`, the function `swap[A,B]` takes a pair `(A,B)` and returns a pair `(B,A)`.

```scala
def twice[A](f: A => A): A => A = {x:A => f(f(x))}
```

This says: given a type `A`, the function `twice[A]` takes a function `f: A => A` and returns a function of type `A => A`
Parametric Polymorphism

Scala’s type parameters are an example of a phenomenon called *polymorphism* (= “many shapes”)

More specifically, *parametric* polymorphism because the function is *parameterized* by the type.
- Its behavior cannot “depend on” what type replaces parameter A.
- The type parameter A is *abstract*

We also sometimes refer to A, B, C etc. as *type variables*
Polymorphism: More examples

- Polymorphism is even more useful in combination with higher-order functions.
- Recall `compose` from the lab:

```scala
def compose[A,B,C](f: A => B, g: B => C) =
{x:A => g(f(x))}
```

- Likewise, the `map` and `filter` functions:

```scala
def map[A,B](f: A => B, x: List[A]): List[B] = ...
def filter[A](f: A => Bool, x: List[A]): List[A] = ...
```

(though in Scala these are usually defined as methods of `List[A]` so the `A` type parameter and `x` variable are implicit)
Formalization

- We add *type variables* $A, B, C, \ldots$, *type abstractions*, *type applications*, and *polymorphic types*:

  
  $$
  e ::= \cdots | \Lambda A. \; e | e[\tau]
  $$
  
  $$
  \tau ::= \cdots | A | \forall A. \; \tau
  $$

- We also use (capture-avoiding) substitution of types for type variables in expressions and types.

- The type $\forall A. \; \tau$ is the type of expressions that can have type $\tau[\tau'/A]$ for any choice of $A$. ($A$ is bound in $\tau$.)

- The expression $\Lambda A. \; e$ introduces a type variable for use in $e$. (Thus, $A$ is bound in any type annotations in $e$.)

- The expression $e[\tau]$ instantiates a type abstraction

- Define $L_{\text{Poly}}$ to be the extension of $L_{\text{Data}}$ with these features
Formalization: Types and type variables

- Complication: Types now have variables. What is their scope? When is a type variable in scope in a type?
- The polymorphic type $\forall A. \tau$ binds $A$ in $\tau$.
- We write $FTV(\tau)$ for the free type variables of a type:

$$
FTV(A) = \{ A \} \\
FTV(\tau_1 \times \tau_2) = FTV(\tau_1) \cup FTV(\tau_2) \\
FTV(\tau_1 + \tau_2) = FTV(\tau_1) \cup FTV(\tau_2) \\
FTV(\forall A. \tau) = FTV(\tau) - \{ A \} \\
FTV(\tau) = \emptyset \text{ otherwise}
$$

$$
FTV(x_1: \tau_1, \ldots, x_n: \tau_n) = FTV(\tau_1) \cup \cdots \cup FTV(\tau_n)
$$

- Alpha-equivalence and type substitution are defined similarly to expressions.
Formalization: Typechecking polymorphic expressions

\[ \Gamma \vdash e : \tau \]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \Gamma \vdash e : \tau \quad A \notin FTV(\Gamma) ]</td>
<td>( \Gamma \vdash \Lambda A. e : \forall A. \tau )</td>
</tr>
<tr>
<td>[ \Gamma \vdash e : \forall A. \tau ]</td>
<td>( \Gamma \vdash e[\tau_0] : \tau[\tau_0/A] )</td>
</tr>
</tbody>
</table>

- Idea: \( \Lambda A. e \) must typecheck with parameter \( A \) not already used elsewhere in type context
- \( e[\tau_0] \) applies a polymorphic expression to a type. Result type obtained by substituting for \( A \).
- The other rules are unchanged
Formalization: Semantics of polymorphic expressions

- To model evaluation, we add type abstraction as a possible value form:

  \[ v ::= \cdots \mid \Lambda A. e \]

- with rules similar to those for \( \lambda \) and application:

  \[
  e \Downarrow v \quad \text{for } L_{\text{Poly}}
  \]

  \[
  \begin{align*}
  e \Downarrow \Lambda A. e_0 & \quad e_0[\tau/A] \Downarrow v \\
  e[\tau] \Downarrow v & \quad \Lambda A. e \Downarrow \Lambda A. e
  \end{align*}
  \]

- In \( L_{\text{Poly}} \), type information is irrelevant at run time.

- (Other languages, including Scala, do retain some run time type information.)
Convenient notation

- We can augment the syntactic sugar for function definitions to allow type parameters:
  
  \[
  \text{let fun } f[A](x : \tau) = e \text{ in } ...
  \]

- This is equivalent to:
  
  \[
  \text{let } f = \Lambda A. \lambda x : \tau. e \text{ in } ...
  \]

- In either case, a function call can be written as
  
  \[
  f[\tau](x)
  \]
Examples in L\textsubscript{Poly}

- **Identity function**
  
  \[ id = \Lambda A. \lambda x:A. \ x \]

- **Swap**
  
  \[ swap = \Lambda A. \Lambda B. \lambda x:A \times B. (\text{snd} \ x, \text{fst} \ x) \]

- **Twice**
  
  \[ twice = \Lambda A. \lambda f:A \rightarrow A. \lambda x:A. \ f(f(x)) \]

- For example:
  
  \[ \text{swap}[\text{int}][\text{str}](1, "a") \downarrow ("a", 1) \]

  \[ \text{twice}[\text{int}](\lambda x: 2 \times x)(2) \downarrow 8 \]
Examples, typechecked

\[ x:A \vdash x:A \]
\[ \vdash \lambda x:A. \ x : A \to A \]
\[ \vdash \forall A. \lambda x:A. x : \forall A.A \to A \]

\[ \vdash swap : \forall A.\forall B.A \times B \to B \times A \]
\[ \vdash swap[\text{int}] : \forall B.\text{int} \times B \to B \times \text{int} \]
\[ \vdash swap[\text{int}][\text{str}] : \text{int} \times \text{str} \to \text{str} \times \text{int} \]
Lists and parameterized types

- In Scala (and other languages such as Haskell and ML), type abbreviations and definitions can be parameterized.
- `List[_]` is an example: given a type `T`, it constructs another type `List[T]`

  deftype \( \text{List}[A] = [\text{Nil} : \text{unit}; \text{Cons} : A \times \text{List}[A]] \)

- Such types are sometimes called type constructors
- (See tutorial questions on lists)
- We will revisit parameterized types when we cover modules
Other forms of polymorphism

- Polymorphism refers to several related techniques for “code reuse” or “overloading”
  - Subtype polymorphism: reuse based on inclusion relations between types.
  - Parametric polymorphism: abstraction over type parameters.
  - Ad hoc polymorphism: Reuse of same name for multiple (potentially type-dependent) implementations (e.g. *overloading* + for addition on different numeric types, string concatenation etc.)

- These have some overlap
- We will discuss overloading, subtyping and polymorphism (and their interaction) in future lectures.
As seen in even small examples, specifying the type parameters of polymorphic functions quickly becomes tiresome:

\[
\text{swap[int][str]} \quad \text{map[int][str]} \quad \cdots
\]

Idea: Can we have the benefits of (polymorphic) typing, without the costs? (or at least: with fewer annotations)

*Type inference*: Given a program without full type information (or with some missing), infer type annotations so that the program can be typechecked.
A very influential approach was developed independently by J. Roger Hindley (in logic) and Robin Milner (in CS).

Idea: Typecheck an expression symbolically, collecting “constraints” on the unknown type variables.

If the constraints have a common solution then this solution is a most general way to type the expression.

- Constraints can be solved using unification, an equation solving technique from automated reasoning/logic programming.

If not, then the expression has a type error.
As an example, consider \textit{swap} defined as follows:

\[
\vdash \lambda x : A. (\text{snd } x, \text{fst } x) : B
\]

\(A, B\) are the as yet unknown types of \(x\) and \textit{swap}.
Hindley-Milner example [Non-examinable]

As an example, consider *swap* defined as follows:

\[ \vdash \lambda x : A. (\text{snd } x, \text{fst } x) : B \]

*A, B* are the as yet unknown types of *x* and *swap*.

A lambda abstraction creates a function: hence

\[ B = A \rightarrow A_1 \] for some *A_1* such that

\[ x : A \vdash (\text{snd } x, \text{fst } x) : A_1 \]
Hindley-Milner example [Non-examinable]

- As an example, consider \( \text{swap} \) defined as follows:

\[
\vdash \lambda x : A. (\text{snd } x, \text{fst } x) : B
\]

\( A, B \) are the as yet unknown types of \( x \) and \( \text{swap} \).

- A lambda abstraction creates a function: hence \( B = A \rightarrow A_1 \) for some \( A_1 \) such that

\[
x : A \vdash (\text{snd } x, \text{fst } x) : A_1
\]

- A pair constructs a pair type: hence \( A_1 = A_2 \times A_3 \) where

\[
x : A \vdash \text{snd } x : A_2 \text{ and } x : A \vdash \text{fst } x : A_3
\]
Hindley-Milner example [Non-examinable]

As an example, consider \texttt{swap} defined as follows:

\[
\vdash \lambda x : A. (\texttt{snd} x, \texttt{fst} x) : B
\]

\(A, B\) are the as yet unknown types of \(x\) and \texttt{swap}.

A lambda abstraction creates a function: hence \(B = A \rightarrow A_1\) for some \(A_1\) such that
\[
x : A \vdash (\texttt{snd} x, \texttt{fst} x) : A_1
\]

A pair constructs a pair type: hence \(A_1 = A_2 \times A_3\) where
\[
x : A \vdash \texttt{snd} x : A_2 \text{ and } x : A \vdash \texttt{fst} x : A_3
\]

This can only be the case if \(x : A_3 \times A_2\), i.e. \(A = A_3 \times A_2\).
Hindley-Milner example [Non-examinable]

As an example, consider \textit{swap} defined as follows:

\[
\vdash \lambda x : A. (\text{snd } x, \text{fst } x) : B
\]

\(A, B\) are the as yet unknown types of \(x\) and \textit{swap}.

A lambda abstraction creates a function: hence \(B = A \rightarrow A_1\) for some \(A_1\) such that

\(x : A \vdash (\text{snd } x, \text{fst } x) : A_1\)

A pair constructs a pair type: hence \(A_1 = A_2 \times A_3\) where

\(x : A \vdash \text{snd } x : A_2\) and \(x : A \vdash \text{fst } x : A_3\)

This can only be the case if \(x : A_3 \times A_2\), i.e. \(A = A_3 \times A_2\).

Solving the constraints: \(A = A_3 \times A_2\), \(A_1 = A_2 \times A_3\) and so \(B = A_3 \times A_2 \rightarrow A_2 \times A_3\).
Let-bound polymorphism [Non-examinable]

- An important additional idea was introduced in the ML programming language, to avoid the need to explicitly introduce type variables and apply polymorphic functions to type arguments.
- When a function is defined using `let fun` (or `let rec`), first infer a type:
  \[
  \text{swap} : A_3 \times A_2 \rightarrow A_2 \times A_3
  \]
- Then *abstract* over all of its free type parameters.
  \[
  \text{swap} : \forall A. \forall B. A \times B \rightarrow B \times A
  \]
- Finally, when a polymorphic function is *applied*, infer the missing types.
  \[
  \text{swap}(1, "a") \leadsto \text{swap[int][str]}(1, "a")
  \]
ML-style inference: strengths and weaknesses

- **Strengths**
  - Elegant and effective
  - Requires no type annotations at all

- **Weaknesses**
  - Can be difficult to explain errors
  - In theory, can have exponential time complexity (in practice, it runs efficiently on real programs)
  - Very sensitive to extension: subtyping and other extensions to the type system tend to require giving up some nice properties

(We are intentionally leaving out a lot of technical detail.)
Type inference in Scala

- Scala does not employ full HM type inference, but uses many of the same ideas.
- Type information in Scala flows from function arguments to their results
  
  ```scala
def f[A](x: List[A]): List[(A,A)] = ...  
f(List(1,2,3)) // A must be Int, don't need f[Int]
```

- and sequentially through statement blocks
  
  ```scala
  var l = List(1,2,3); // l: List[Int] inferred  
  var y = f(l); // y : List[(Int,Int)] inferred
  ```
Type inference in Scala

- Type information does **not** flow across arguments in the same argument list.

```scala
def map[A](f: A => B, l: List[A]): List[B] = ...
scala> map({x: Int => x + 1}, List(1,2,3))
res0: List[Int] = List(2, 3, 4)
scala> map({x => x + 1}, List(1,2,3))
<console>:25: error: missing parameter type
```

- But it **can** flow from earlier argument lists to later ones:

```scala
def map2[A](l: List[A])(f: A => B): List[B] = ...
scala> map2(List(1,2,3)) {x => x + 1}
res1: List[Int] = List(2, 3, 4)
```
Type inference in Scala: strengths and limitations

- Compared to Java, many **fewer** annotations needed
- Compared to ML, Haskell, etc. many **more** annotations needed
- The reason has to do with Scala’s integration of polymorphism and **subtyping**
  - needed for integration with Java-style object/class system
  - Combining subtyping and polymorphism is tricky (type inference can easily become undecidable)
  - Scala chooses to avoid global constraint-solving and instead propagate type information **locally**
Summary

- Today we covered:
  - The idea of thinking of the same code as having many different types
  - Parametric polymorphism: makes the type parameter explicit and abstract
  - Brief coverage of *type inference*.

- Next time:
  - Programs, modules, and interfaces