Contextual Abstraction

Principles of Functional Programming

Con-text

what comes with the text,
but is not in the text

Context Takes Many Forms

- the current configuration
- the current scope
- the meaning of “<” on this type
- the user on behalf of which the operation is performed
- the security level in effect
- ...

Code becomes more modular if it can *abstract* over context.

That is, functions and classes can be written without knowing in detail the context in which they will be called or instantiated.

How Is Context Represented?

So far:

- global values
- global mutable variables
- “Monkey Patching”
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables
- "Monkey Patching"
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables *what if different modules need different settings? interference can be dangerous!*
- “Monkey Patching”
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables *what if different modules need different settings? interference can be dangerous!*
- “Monkey Patching” *- more powerful ways to shoot yourself in the foot...*
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables *what if different modules need different settings? interference can be dangerous!*
- “Monkey Patching” *- more powerful ways to shoot yourself in the foot...*
- dependency injection frameworks (e.g. Spring, Guice) *- outside the language, rely on bytecode rewriting $\rightarrow$ harder to understand and debug.*

Functional Context Representation

In functional programming, the natural way to abstract over context is with function parameters.

+ flexible
+ types are checked
+ not relying on side effects

But sometimes this is too much of a good thing! It can lead to

- many function arguments
- which hardly ever change
- repetitive, errors are hard to spot

Example: Sorting

We have seen `sort` functions. For instance, here's an outline of a method `sort` that takes as parameter a `List[Int]` and returns another `List[Int]` containing the same elements, but sorted:

```
def sort(xs: List[Int]): List[Int] =
  ...
  ... if x < y then ...
  ...
```

At some point, this method has to compare two elements `x` and `y` of the given list.

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `T` for the type of elements:

```
def sort[T](xs: List[T]): List[T] = ...
```

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `T` for the type of elements:

```
def sort[T](xs: List[T]): List[T] = ...
```

But this does not work, because there's not a single comparison method < that works for all types.

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `T` for the type of elements:

```
def sort[T](xs: List[T]): List[T] = ...
```

But this does not work, because there's not a single comparison method < that works for all types.

In other words, we need to ask the question: What is the meaning of < on type `T` *at the call site*?

This means querying the call-site context.

Parameterization of `sort`

The most flexible design is to pass the comparison operation as an additional parameter:

```
def sort[T](xs: List[T])(lessThan: (T, T) => Boolean): List[T] =
  ...
  ... if lessThan(x, y) then ...
  ...
```

Calling Parameterized sort

We can now call `sort` as follows:

```
val ints = List(-5, 6, 3, 2, 7)
val strings = List("apple", "pear", "orange", "pineapple")

sort(ints)((x, y) => x < y)
sort(strings)((s1, s2) => s1.compareTo(s2) < 0)
```

Parameterization with Ordering

There is already a class in the standard library that represents orderings:

```
scala.math.Ordering[A]
```

Provides ways to compare elements of type `A`. So, instead of parameterizing with the `lessThan` function, we could parameterize with `Ordering` instead:

```
def sort[T](xs: List[T])(ord: Ordering[T]): List[T] =
  ...
  ... if ord.lt(x, y) then ...
  ...
```

Ordering Instances

Calling the new `sort` can be done like this:

```
import scala.math.Ordering

sort(ints)(Ordering.Int)
sort(strings)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

```
object Ordering:
  val Int = new Ordering[Int]:
    def compare(x: Int, y: Int) =
      if x < y then -1 else if x > y then 1 else 0
```

Reducing Boilerplate

Problem: Passing around `Ordering` arguments is cumbersome.

```
sort(ints)(Ordering.Int)
sort(strings)(Ordering.String)
```

Sorting a `List[Int]` value always uses the same `Ordering.Int` argument, sorting a `List[String]` value always uses the same `Ordering.String` argument, and so on…

Implicit Parameters

We can reduce the boilerplate by making `ord` an *implicit parameter*.

```
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] = ...
```

Then, calls to `sort` can omit the `ord` parameter:

```
sort(ints)
sort(strings)
```

The compiler infers the argument value based on its expected type.

Type Inference

We have seen that the compiler is able to *infer types* from *values*.

That is, the previous calls to `sort` are augmented as follows:

```
sort(ints)
sort(strings)
```

Type Inference

We have seen that the compiler is able to *infer types* from *values*.

That is, the previous calls to `sort` are augmented as follows:

```
sort[Int](ints)
sort[String](strings)
```

Term Inference

The Scala compiler is also able to do the opposite, namely to *infer expressions* (*aka* terms) from *types*.

When there is exactly one "obvious" value for a type in a `using` clause, the compiler can provide that value to us.

```
sort[Int](ints)
sort[String](strings)
```

Term Inference

The Scala compiler is also able to do the opposite, namely to *infer expressions* (*aka* terms) from *types*.

When there is exactly one “obvious” value for a type, the compiler can provide that value to us.

```
sort[Int](ints)(using Ordering.Int)
sort[String](strings)(using Ordering.String)
```