Named and default arguments for polymorphic object-oriented languages

A discussion on the design implemented in the Scala language

Lukas Rytz
Programming Methods Laboratory
École Polytechnique Fédérale de Lausanne
lukas.rytz@epfl.ch

Martin Odersky
Programming Methods Laboratory
École Polytechnique Fédérale de Lausanne
martin.odersky@epfl.ch

ABSTRACT

This article describes the design and implementation of named and default arguments in the Scala programming language. While these features are available in many other languages there are significant differences in the actual implementations. We present a design that unifies the most reasonable properties for an object-oriented language and provides new possibilities by allowing default arguments on generic and implicit parameters. We also present a solution for the problem of writing a lightweight generic update function for algebraic datatypes.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Procedures, functions, and subroutines*

General Terms

Languages, Design

Keywords

Named arguments, default arguments, Scala

1. INTRODUCTION

When applying a method to a set of arguments, every argument expression has to be mapped to one of the method's parameters. In most programming languages this mapping is defined by the position of the argument expressions in the argument list (i.e. the n^{th} argument defines the n^{th} parameter of the method), in which case the arguments are called *positional*. An alternative is using *named* arguments, where the programmer explicitly mentions the parameter name that an argument expression defines.

The additional verbosity of named arguments is worthwhile in applications of methods with a large number of parameters or with multiple parameters of the same type.

The use of named arguments improves code readability by making the role of an argument explicit, and it eliminates the risk of confusing the positions of parameters of the same type, an error which cannot be caught by a type system otherwise. Compare the following method applications:

```
frame.setSize(200, 300) // which one is the height?
frame.setSize(height = 200, width = 300)
```

Another important application for named arguments comes up when using them in conjunction with *default* arguments, the second concept discussed in this article. A language with default arguments allows specifying default values for the parameters of a method. These defaults are used whenever a method application leaves some of the parameters undefined.

With positional arguments, the only possibility to use defaults is to leave some parameters at the end undefined. With named arguments it is possible to provide a value for an arbitrary subset of the parameters and use the defaults for the others.

Named and default arguments are very common features and can be found in a large number of programming languages (see section 4). We investigate how they are implemented in other languages and analyze how the different design choices affect the interaction with other language features such as method overriding, virtual method calls and overloading resolution. We present two new use cases for default arguments by allowing to use them on parameters of a generic type and on implicit parameters.

The rest of this paper is structured as follows. Section 2 discusses the design choices for implementing named and default arguments. In section 3 we explain the compilation technique used in Scala. Section 4 compares our implementation with other languages, and section 5 concludes.

2. DESIGN CHOICES

A central constraint when extending an existing language with named and default arguments is source compatibility: the new feature should not invalidate any existing code. But even with this restriction there are a number of design decision to be made.

While this section provides a qualitative discussion on Scala's design of named and default arguments, a more precise description of the semantics, defined by a translation into equivalent code without parameter names or defaults, can be found in the Scala language specification [11].

2.1 Flexibility

Our implementation of named and default arguments is designed to provide a high degree of flexibility for the programmer. A first fundamental decision is whether the use of named arguments should be optional or not. We decided to make them optional because this gives the programmer more flexibility when calling a method, and it does not force him to chose which parameters should be named when defining a method. It also allows using named arguments when calling a method that was written before named arguments were available in the language, because there is no change in the way parameters are defined.

A disadvantage of enabling named arguments on all parameters is that parameter names immediately become part of the public interface of a method, and changing them can invalidate client code. This is also true for source code which was written before named arguments were available in the language. Presumably, programmers were less careful in choosing parameter names in the past because they expected them to be private.

When using named arguments, Scala allows to specify the parameters in a different order than in the method definition. We also allow mixing positional and named arguments, in this case all positional arguments have to appear left of the named arguments. Unlike for instance C# 4.0 [4] where only the last parameters of a method are allowed to have defaults, any parameter can have a default argument in Scala.

2.2 Evaluation order

Unlike OCaml [10] or ADA95 [8], Scala has a definite evaluation order for method arguments. Thus, we need to define an evaluation order in the presence of named and default arguments. Given that positional arguments are evaluated left-to-right, the same is done for named arguments. So in the following example, `a()` is always evaluated before `b()`, no matter what the order of parameters in the method definition is.

```
m(y = a(), x = b())
```

When an argument list of an application is completed with default arguments, the default expressions are evaluated after the specified arguments of that parameter list. The default arguments which are used in a particular application behave just like specified arguments: they are evaluated exactly once, except if the corresponding parameter is a by-name parameter, in which case they are evaluated every time the method body refers to the parameter. Default expressions which are not used in a method application are not evaluated.

If an application has multiple argument lists (see 2.3.1), these are always evaluated left-to-right. This implies that default expressions used in the first argument list are evaluated before any arguments of the second argument list.

The following, slightly tricky example illustrates evaluation of default arguments. First, the default expression `{i += 1; i}` is evaluated, then the specified argument `i`, and finally the default expression `{i += 2; i}`.

```
var i = 1
def f(x: Int = {i += 1; i})
     (y: Int = {i += 2; i}, z: Int) = (x, y, z)
f()(z = i) // returns (2, 4, 2)
```

2.3 Interaction with other features

This section describes how an elaborate integration of named and default arguments with other language features enables a number of useful programming patterns.

2.3.1 *Curried method definitions*

In Scala, methods can have multiple parameter lists. Method definitions with more than one parameter list are called *curried* because they allow partial application.

When writing a curried method definition, default arguments can depend on earlier parameters of the same method. In other words, the parameters are visible not only in the method body, but also in all subsequent parameter lists. This enables the programmer to specify meaningful default arguments in some additional situations:

```
def resizeImage(i: Image)(h: Int = i.height,
                          w: Int = i.width) = { .. }
```

To solve the ambiguity between a partially applied method and a method application using default arguments, Scala uses default arguments only for completing underspecified arguments lists. In order to use the default arguments of the above definition, an empty argument list has to be provided, i.e. `resizeImage(myImage)()`.

2.3.2 *Overriding and virtual method calls*

When a method is being overridden in a subclass, the parameter names of the overriding method do not need to be the same as in the superclass. For type-checking an application which uses named arguments, the static type of the method determines which names have to be used. However the actual method which is called is still resolved dynamically: even if an application uses the parameter names of a superclass, an implementation of a subclass with different parameter names might be called. The following example illustrates this behavior:

```
trait A {
  def f(a: Int, b: Int): Int = a + b
}
class B extends A {
  override def f(x: Int, y: Int) = x - y
}
val a: A = new B
a.f(b = 2, a = 3) // returns 1
```

With the ability to choose parameter names freely it is possible to re-use a parameter name from the superclass for a different parameter in the subclass. Doing so should be avoided because it can can lead to unexpected errors when changing a type to a more specific one:

```
trait A           { def f(a: Int, b: Int): Int }
class B extends A { def f(b: Int, a: Int) = b - a }

val b = new B
(b: A).f(a = 3, b = 2) // returns 1
     b.f(a = 3, b = 2) // returns -1
```

Default arguments behave very similarly to methods themselves with respect to subclassing and overriding.

When a method with default arguments is overridden or implemented in a subclass, the parameters of the overriding method inherit the defaults from their counterparts in the superclass. It is also possible to override default arguments and to add new ones to parameters which do not have a default in the superclass. For type-checking an application, the static type of the method determines which parameters have a default value.

Since all method calls in Scala are virtual, the most systematic approach is to apply dynamic dispatch to default arguments as well. In other words, just like the actual method code which is executed, the default value which is used at run-time is determined by the dynamic type of the receiver object. The following example shows how this design is more intuitive:

```
class Office {
  def call(to: Number, encrypt = false) { .. }
}
class CIA extends Office {
  override def call(to: Number, encrypt = true) { .. }
}
val office: Office = new CIA
office.call(president) // will be encrypted
```

2.3.3 Overloading resolution

In Scala methods can be *overloaded*, which means that multiple methods with the same name but with differing argument types can be defined in a class. When the method name in an application references several members of a class, the process of *overloading resolution* is applied to identify a unique member. The integration of named and default arguments affects this process and brings up additional design choices.

Overloading resolution is performed in two phases: first, the set of *applicable* members is determined, then the *most specific* alternative among them is selected. A member m is applicable to a given argument list if the application of m to these arguments can be successfully type-checked. Clearly, the presence of named and default arguments affects the definition of applicability. For instance, the method

```
def f(a: String, b: Int = 1) = { .. }
```

is applicable in the expressions `f("str")` and `f(b = 2, a = "str")`, but not in `f(c = "str")`.

The selection of a most specific alternative on the other hand is solely based on the method signature, named or default arguments have no influence on this process. Assume there are two applicable methods A and B with parameter types (Ts) and (Us) respectively. Method B is more specific than A if A can be successfully applied to arguments (us) of type (Us), but not vice versa, i.e. B is not applicable to arguments (ps) of type (Ts). In the following example the second method is more specific than the first.

```
def f(a: Int, b: Object) = { .. }
def f(a: Int, b: String) = { .. }
```

The two methods in the next example are not related, none is more specific than the other.

```
def g(a: Int, b: Object) = { .. }
def g(b: String, a: Int) = { .. }
```

One could argue that named arguments *should* affect the definition of a method being more specific than another. Given the expression

```
g(a = 1, b = "text")
```

both methods are applicable and the second one could be seen as being more specific in the context of that particular application. This design would make the definition of *more specific* depend on the context of the application, which complicates both the specification an implementation of the language. We think that this additional complication is not worth its limited benefit because it affects only method invocations which use named arguments to actually re-order the parameters. So in Scala, the application in the above example is ambiguous because none of the methods is more specific than the other.

Note that this restriction is correct: if a method is more specific than another under the restriction, the same is true in the unrestricted case. In other words, it can only make less application expressions applicable, never more.

A more formal and complete definition of overloading resolution in Scala can be found in [11].

2.3.4 Generic parameters

In existing systems it is not possible to specify a default argument on parameters with a generic type. We solve this problem in Scala by type-checking default arguments with a special expected type, which is obtained by replacing all occurrences of type parameters in the parameter type with the undefined type. In the following example, the default argument `1` is type-checked with an undefined expected type. When the default argument is used in an application, its type has to conform to the actual instantiation of the type parameter T.

```
def f[T](x: T = 1) = x
f()    // returns 1: Int
f("2") // returns "s": String
f[String]() // error: type mismatch.
            // found: Int, required: String
```

2.3.5 Functional update for algebraic datatypes

Programming languages with algebraic datatypes often provide a language construct for cloning instances of datatypes and optionally updating specific fields in the clone. For instance in Haskell [9] datatypes with field labels can be non-destructively updated using a special syntax.

It turns out that with the ability to specify default arguments on generic parameters, no further language extension is required to write a functional update method for datatypes. For example in a datatype for tuples, such an update method called `copy` would look like this:

```
case class Tuple[A, B](a: A, b: B) {
  def copy[A1, B1](a: A1 = this.a, b: B1 = this.b) =
    Tuple[A1, B1](a, b)
}

val t1: Tuple[Int, String] = Tuple(1, "a")
val t2: Tuple[Int, Int] = t1.copy(b = 2)
```

In its current version, the Scala compiler automatically generates these `copy` methods for every case class, so the above example also works without the manually defined method in class `Tuple`.

2.3.6 Implicit parameters

Methods in Scala can have parameters marked as 'implicit'. When an implicit parameter is omitted in a method application an argument is automatically provided: the compiler searches in the environment of the method call for a value marked as 'implicit' with a matching type. It is a compile-time error if no matching value can be found.

We generalized this design by allowing default arguments on implicit parameters. If a method invocation does not specify a value for an implicit parameter with a default, the compiler will first try to find a matching implicit value. If no such value can be found, the default argument is used instead. This enables a convenient new programming pattern, as illustrated by the following example from the Scala standard library.

```
package scala.io
object Source {
  def fromURL(url: URL)
             (implicit codec: Codec = Codec.default) =
    fromInputStream(url.openStream())(codec)
}
```

Users can define an implicit `Codec` value once in their code and from then on omit the second parameter of the method. The default argument is used only as a fallback value if no implicit is available.

```
class User {
  implicit val userCodec = [...]
  val s = Source.fromURL("file:///path/to/file")
}
```

3. SCALA'S COMPILATION TECHNIQUE

The compilation of method applications with named arguments is very simple, it just has to make sure that the arguments are evaluated in the order they appear in the source code. This is achieved by creating a local value for every argument and then calling the method with a permutation of these local values. A method call $f(e_1, \ldots, e_m, l_{m+1} = e_{m+1}, \ldots, l_n = e_n)$ with m positional and $n-m$ named arguments is transformed to a block of the form

```
{
  val x_1 = e_1
  ...
  val x_n = e_n
  f(x_1, ..., x_m, σ(x_{m+1},...,x_n))
}
```

where σ is a permutation mapping the named arguments to the corresponding parameter positions.

The compilation of default arguments is a bit more involved. For every default argument expression the compiler generates a method computing that expression. In order to support defaults depending on other parameters, these methods are parametrized by the type parameters of the original method and by the value parameter sections preceding the corresponding parameter. The curried method definition `def f[T](a: Int = 1)(b: T = a+1)(c: T = b)` generates the following three methods:

```
def f$default$1[T]: Int =  1
def f$default$2[T](a: Int): Int = a + 1
def f$default$3[T](a: Int)(b: T): T = b
```

For constructor defaults, these methods are added to the companion object of the class (which makes them similar to static methods in Java). For other methods, the default methods are generated at the same location as the original method definition.

Method applications using default arguments are transformed in a similar fashion as named arguments. A local value is generated for every specified argument and for every default argument which is used. These local values are used not only for calling the actual method, but also for calling the methods computing the default arguments. For example, the application `f[String]()("str")()` is transformed to the following block:

```
{
  val x$1 = f$default$1[String]
  val x$2 = "str"
  val x$3 = f$default$3[String](x$1)(x$2)
  f[String](x$1)(x$2)(x$3)
}
```

When a default argument is overridden in a subclass, the method computing the default will simply override the one from the superclass. This compilation technique makes sure that default arguments are resolved dynamically, as described in the previous section.

3.1 Limitations

In order to make the result of the compilation stable across multiple execution, we decided to choose a deterministic naming scheme for the compiler-generated methods computing the default argument expressions. The name of these methods is composed of the original method name, the string `$default$` and an number indicating the parameter position.

When defining overloaded methods, this naming scheme does not allow to use default arguments in more than one of the overloaded alternatives. We consider this limitation as acceptable because default arguments eliminate the need for arity-based overloading and type-based overloading is a non-central and sometimes even risky [2] feature which should not be used too extensively.

4. NAMED AND DEFAULT ARGUMENTS IN OTHER LANGUAGES

Most of the features of named and default arguments we implemented in Scala can also be found in other languages. However, every language implements a slightly different combination of them, which results in differing designs. This section reviews the implementations of named and / or default arguments in the most important languages in this area and compares them with our design.

4.1 OCaml, F# and Common Lisp

In OCaml [5] the parameter types of a function type can either be annotated with a *label* or not. When applying a function, all labeled parameters have to be specified using a labeled (named) argument[1]. This restriction allows

[1]OCaml allows omitting argument labels in total applications

mixing positional and named arguments freely, but in turn it requires the programmer to decide for every parameter whether it should be labeled or not when writing a function.

In F# [15] parameters also have to be named explicitly, but it is allowed to specify a named parameter using a positional argument when applying the method. Accordingly the positional arguments have to be provided first in an argument list, just as in Scala.

Default arguments in OCaml support the same features as in Scala, they can be arbitrary expressions which can also depend on earlier parameters of the function. However, since arguments with defaults are represented with a different type (T `option`) an overriding definition cannot inherit or add defaults to a function.

F# only supports optional arguments without syntactic support for defaults. However, defaults can be implemented using simple pattern matches on optional arguments, which provides exactly the same functionality as in OCaml.

Even though Lisp [13] is not a statically typed language, named and default arguments in Lisp are to some extent similar to OCaml. Lisp supports *keyword* parameters which optionally can have a default value, an arbitrary expression which can depend on other parameters. Keyword parameters have to be specified at the end of a parameter list, and they can only be applied using a named arguments, not with positional ones.

4.2 C# 4.0 and VB.NET

Starting with version 4.0, C# supports named and default arguments [4]. The implementation of named arguments is similar to ours: using named arguments is optional, overriding methods can chose different parameter names and the static type of the method defines which names have to be used. Default arguments on the other hand are limited to compile-time constants such as numbers, strings or enumeration values. When a virtual method call uses a default argument, the static type of the method determines which default value is used, so the phone call in the example of section 2.3.2 would not be encrypted.

The implementation of named and default arguments in Visual Basic is similar to C#.

4.3 Smalltalk and Objective-C

In Smalltalk [6] and Objective-C [1] all parameter names are part of the method name and therefore need to be specified at call-site in the same order. When overriding a method in a subclass the parameter names cannot be changed.

Neither language supports default arguments, even though they can be emulated in Smalltalk using call patterns.

4.4 ADA and C++

Except for the argument evaluation order which is left unspecified, named arguments in ADA [8] are the same as in our implementation or in C#. C++ [14] does not support named arguments.

ADA and C++ allow specifying default arguments on any method parameter. Defaults are arbitrary expressions, however they cannot depend on other parameters. Default arguments are not fully integrated with subclassing and overriding: defaults can be overridden but not inherited. Similar to C#, the static type of the method defines which default argument expression is evaluated, even if the method call is virtual. So the phone call of the example in 2.3.2 would again not be encrypted.

4.5 Python, Ruby, Groovy and Clojure

While Python [12] supports built-in named arguments, they can be emulated using an additional hash map parameter in Ruby, Groovy and Clojure [16, 3, 7] which simply maps names to values. This latter technique forces the use of named arguments for these parameters, it is not possible to specify them using a positional argument.

Python, Ruby and Groovy support method overriding, and they allow to change the parameter names in a subclass. However, doing so can lead to unexpected errors as the following Python example shows:

```
class A:
  def f(self, a): return a

# B is a subclass of A
class B(A):
  # override 'f', choose a different parameter name
  def f(self, b): return b

def withA(a): print a.f(a = 100)

withA(B()) # TypeError: f() got an unexpected
           # keyword argument 'a'
```

The method `withA` expects an instance of `A` as argument, but it will fail to execute when passing it an instance of `B`, even though `B` is a subclass of `A`.

Regarding the implementation of default arguments, there are some differences between the four languages. In Python all default argument expressions are evaluated exactly once, namely when evaluating the method definition. Ruby is closer to our design, defaults are re-evaluated every time they are being used, and they can depend on other parameters of the method. In Groovy, default arguments are just syntactic sugar for creating overloaded methods with different arities. This technique allows only using the rightmost n default arguments of a method, leaving arbitrary parameters unspecified is not possible. In Clojure finally, defaults can be emulated using so-called *map-bindings* and the `:or` operator[2]. All default argument expressions are evaluated on every function application, even if no defaults are being used.

5. SUMMARY

When adding named and default arguments to an existing object-oriented programming language there is a number of possibilities for the new features to cooperate with existing parts of the language. We tried to optimize this integration with respect to currying, subclassing, method overriding and overloading resolution. Allowing named arguments on generic and implicit parameters enables new programming patterns which are useful in everyday-programming.

The design presented in this article is fully implemented in the current development version of the Scala language, including the compiler-generated `copy` methods for case classes. It will be included in the 2.8.0 release, nightly builds can be downloaded from the web-site[3].

[2]see `http://clojure.org/special_forms`

[3]`http://www.scala-lang.org/node/212`

6. REFERENCES

[1] Apple Inc. The Objective-C 2.0 Programming Language – Objects, Classes, and Messaging. http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocObjectsClasses.html, 2009.

[2] G. Bracha. Systemic Overload. http://gbracha.blogspot.com/2009/09/systemic-overload.html, 2009.

[3] Codehaus Foundation. Groovy User Guide. http://groovy.codehaus.org/User+Guide.

[4] M. Corporation. C# Version 4.0 Specificaiton Draft. http://msdn.microsoft.com/en-us/vcsharp/dd819407.aspx, 2009.

[5] J. Garrigue. Labeled and optional arguments for Objective Caml. In *JSSST Workshop on Programming and Programming Languages*, 2001.

[6] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[7] R. Hickey. Clojure Reference. http://clojure.org/Reference.

[8] Intermetrics, Inc. Ada 95 Reference Manual. http://www.adahome.com/rm95, 1994.

[9] S. P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[10] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.11, Documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml, 2008.

[11] M. Odersky. The Scala Language Specification. http://www.scala-lang.org/node/198, 2009.

[12] Python Software Foundation. The Python Language Reference. http://docs.python.org/3.1/reference/index.html.

[13] G. L. Steele. *Common LISP: The Language*. Digital Press, 1984.

[14] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[15] D. Syme. The F# draft Language Specification. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.pdf, 2009.

[16] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, 2004.