Functions with generic parameter types

Asked 15 years, 10 months ago Modified 9 years, 7 months ago Viewed 21k times

48

I am trying to figure out how to define a function that works on multiple types of parameters (e.g. int and int64). As I understand it, function overloading is not possible in F# (certainly the compiler complains). Take for example the following function.

```
let sqrt_int = function
    | n:int   -> int (sqrt (float n))
    | n:int64 -> int64 (sqrt (float n))
```

The compiler of course complains that the syntax is invalid (type constraints in pattern matching are not supported it seems), though I think this illustrates what I would like to achieve: a function that operates on several parameter types and returns a value of the according type. I have a feeling that this is possible in F# using some combination of generic types/type inference/pattern matching, but the syntax has eluded me. I've also tried using the :? operator (dynamic type tests) and *when* clauses in the pattern matching block, but this still produces all sorts errors.

As I am rather new to the language, I may very well be trying to do something impossible here, so please let me know if there is alternative solution.

generics f# overloading type-inference typeclass

Share
Improve this question
Follow

edited Mar 19, 2014 at 4:15

abatishchev
100k ● 88 ● 301 ● 442

asked Feb 1, 2009 at 16:09

Noldorin
147k ● 56 ● 272 ● 307

5 Answers

Sorted by: Highest score (default)

65

Overloading is typically the bugaboo of type-inferenced languages (at least when, like F#, the type system isn't powerful enough to contain type-classes). There are a number of choices you have in F#:

- Use overloading on methods (members of a type), in which case overloading works much like as in other .Net languages (you can ad-hoc overload members, provided calls can be distinguished by the number/type of parameters)

- Use "inline", "^", and static member constraints for ad-hoc overloading on functions (this is what most of the various math operators that need to work on int/float/etc.; the syntax here is weird, this is little-used apart from the F# library)
- Simulate type classes by passing an extra dictionary-of-operations parameter (this is what INumeric does in one of the F# PowerPack libraries to generalize various Math algorithms for arbitrary user-defined types)
- Fall back to dynamic typing (pass in an 'obj' parameter, do a dynamic type test, throw a runtime exception for bad type)

For your particular example, I would probably just use method overloading:

```
type MathOps =
    static member sqrt_int(x:int) = x |> float |> sqrt |> int
    static member sqrt_int(x:int64) = x |> float |> sqrt |> int64

let x = MathOps.sqrt_int 9
let y = MathOps.sqrt_int 100L
```

Share

Improve this answer

Follow

edited Oct 25, 2009 at 23:32

answered Feb 1, 2009 at 18:50

Brian

119k ● 17 ● 243 ● 304

7 @Jon Harrop: Isn't this very question a compelling example? I expected that a modern language would have this feature. That function overloading should be on every language designers list. This is something that can be worked around but WHY!! It should just work. – user376591 Nov 30, 2010 at 10:38

8 @Gorgen: I don't believe it is, no. The problem is that is never "just works". Overloading in languages like C++ and C# grates against type inference. Overloading in languages like Haskell using features like type classes can massively degrade the performance of basic arithmetic operations when you least expect it. Standard ML chose ad-hoc polymorphism to cover a single special case (int vs float) but no others (e.g. scalar vs vector). OCaml has no overloading at all but recently adopted a solution known as "delimited" overloading. – J D Nov 30, 2010 at 12:47

8 Scala and C# 3 chose the compromise of "local" type inference which is just lame. F# chose the pragmatic compromise of some overloading (arithmetic operators) but in a framework that guarantees static resolution and no dispatch so it is always fast. These trade-offs are known but not solved. My personal preference is the F# solution because it solves the overloading where it matters most (arithmetic) without introducing awful performance characteristics where they often matter the most (arithmetic!). – J D Nov 30, 2010 at 12:47

2 @Gorgen: Exactly. Overloading screws with type inference so F# relegates it to the OO side where type inference is already screwed. – J D Dec 1, 2010 at 21:34

1 Multiple dispatch doesn't mean poor performance. It's a central design principle in Julia, so much so that one of its creators said they took it further than any other non-research language. Yet Julia is designed primarily for high-performance numerical computing and typically out-performs C. – GatesDA May 16, 2014 at 18:33

20

This works:

```
type T = T with
    static member ($) (T, n:int  ) = int   (sqrt (float n))
    static member ($) (T, n:int64) = int64 (sqrt (float n))

let inline sqrt_int (x:'t) :'t = T $ x
```

It uses static constraints and overloading, which makes a compile-time lookup on the type of the argument.

The static constraints are automatically generated in presence of an operator (operator `$` in this case) but it can always be written by hand:

```
type T = T with
    static member Sqr (T, n:int  ) = int   (sqrt (float n))
    static member Sqr (T, n:int64) = int64 (sqrt (float n))

let inline sqrt_int (x:'N) :'N = ((^T or ^N) : (static member Sqr: ^T * ^N ->
_) T, x)
```

More about this here.

Share

Improve this answer

Follow

edited May 6, 2015 at 21:59

answered Sep 28, 2013 at 20:40

Gus

26.2k ● 2 ● 53 ● 78

1 This is a very nice improvement over the solution provided by Brian and Mauricio, it has the benefit of a function without the dot-notation (compare Brian's) and it adds compile-time type checking (compare Mauricio's). Would you care to elaborate in your answer on how this works and whether the operator-definition is a requirement? – Abel May 6, 2015 at 19:36

2 Thanks @Abel, I did elaborate a bit more about the solution and included a link to a blog entry with more details. Mauricio's answer takes a very different approach which is also valid, it uses always the same code for all types by casting to-from `float` which is less code but if you want to work with big integers you may run into a limitation. – Gus May 6, 2015 at 21:59

15

Yes, this can be done. Take a look at this hubFS thread.

In this case, the solution would be:

```
let inline retype (x:'a) : 'b = (# "" x : 'b #)
let inline sqrt_int (n:'a) = retype (sqrt (float n)) : 'a
```

Caveat: no compile-time type checking. I.e. `sqrt_int "blabla"` compiles fine but you'll get a FormatException at runtime.

Share

Improve this answer

Follow

edited Nov 21, 2012 at 14:55

Marcus

6,097 ● 3 ● 29 ● 40

answered Feb 1, 2009 at 16:42

Mauricio Scheffer

99.6k ● 24 ● 195 ● 279

Thanks, that seems to be the solution (though it's not as straightforward as I might have hoped). Just to clarify, I would want something like this? let inline sqrt_int (n:^a) = retype (sqrt (float n)) : ^a – Noldorin Feb 1, 2009 at 17:14

1 Yup, that works. However, be aware that with this you lose compile-time type checking. I.e. sqrt_int "blabla" type-checks although you'll get a FormatException at runtime. – Mauricio Scheffer Feb 1, 2009 at 17:34

Ok, so there's really no point of using the hat operator in this case, right? If I happened to be using an arithmetic operator such as * in the function (before the cast), would that insure the compile-time check? – Noldorin Feb 1, 2009 at 18:47

Yes, in this case a normal type parameter works too: let inline sqrt_int (n:'a) = retype (sqrt (float n)) : 'a Not sure what you mean with the arithmetic operator. – Mauricio Scheffer Feb 1, 2009 at 20:50

2 What is that `(# "" x : 'b #)` beast? The compiler tells me, that construct is deprecated and should be only used in the F# library. And good luck googling for `(#` ... – mbx Dec 15, 2016 at 7:58

10

Here's another way using runtime type checks...

```
let sqrt_int<'a> (x:'a) : 'a = // '
    match box x with
    | :? int as i -> downcast (i |> float |> sqrt |> int |> box)
    | :? int64 as i -> downcast (i |> float |> sqrt |> int64 |> box)
    | _ -> failwith "boo"

let a = sqrt_int 9
let b = sqrt_int 100L
let c = sqrt_int "foo" // boom
```

Share Improve this answer Follow

answered Feb 1, 2009 at 21:21

Brian

119k ● 17 ● 243 ● 304

Interesting. Now I have too many options! One question: why you need the generic specifier <'a> when you specify a type constraint on x. I thought they were equivalent syntaxes. – Noldorin Feb 1, 2009 at 21:39

1 You can indeed omit the <'a> (try it). Note the potential perf difference; the method-overloading determines which version at compile-time, whereas this version does run-time

type-checking. (It might be that 'sqrt' overwhelms these considerations, though, I haven't measured.) – Brian Feb 1, 2009 at 21:46

1 And of course the other difference is that this version compiles (and throws at runtime) for non-ints, whereas the method overloading version will compile-time error for non-ints. – Brian Feb 1, 2009 at 21:47

3

Not to take away from the correct answers already provided, but you can in fact use type constraints in pattern matching. The syntax is:

```
| :? type ->
```

Or if you want to combine type checking and casting:

```
| :? type as foo ->
```

Share Improve this answer Follow

answered Feb 1, 2009 at 19:10

Joel Mueller

28.7k ● 9 ● 66 ● 89

That's what I initially thought I might be able to do. Unfortunately it gives a "runtime coercion" error (error FS0008). Together with the retype function provided in a link in mausch's post, it should however work as an alternative to the inline keyword if I understand it properly. – Noldorin Feb 1, 2009 at 20:36

the runtime coercion can be avoided by boxing the variable the type is matched on using `match box variable with` – Remko Jun 14, 2013 at 8:06