The following is copied, lightly edited, from a message on the review of SE-0111: Remove type system significance of function argument labels. You can read the original message on the Swift forums.

Language history, a.k.a. story time!

We started out with the "perfect" model of a function type being a map from a tuple to a tuple. Different argument labels were just overloads. It really was quite a simple model, other than not having 1-tuples. Well, and variadics and default values and trailing closures didn't make sense anywhere but in functions, but still. Very simple.

(And inout. And autoclosure. And maybe a few more.)

Then we hit a snag: naming guidelines. We wanted argument labels to be something people felt comfortable using, something that would be encouraged over a sea of unlabeled arguments. But even before the Swift 3 naming conventions were hammered out, the natural names for argument labels didn't seem to match the names you'd want to use in the function. So we split the names of parameters off from the names of tuple elements.

(This was precipitated by wanting to import Objective-C methods, but I think it would have come up regardless.)

As seen earlier in the thread, argument labels don't make for good tuple element labels. Especially with the Swift 3 guidelines, argument labels usually don't make sense without the context provided by the base name, and two methods that happen to share argument labels might not actually be very similar, while two methods that are duals of each other might have different argument labels due to, well, English (e.g. `add(to:)` vs. `remove(from:)`).

The real blow, however, came with that very first idea: that we could treat methods with different argument labels as simple overloads in type. This led to poor diagnostics where the compiler couldn't decide whether to believe the types or the argument labels, and might tell you you have the wrong argument labels rather than a type mismatch. For pretty much every Apple API, this was the wrong decision. On top of all that, it was really hard to refer to a method when you didn't want to call it. (Most methods with the same base name still have unique labels, so you don't need the types to disambiguate.)

So we introduced the notion of "full names", which are the things you see written as `move(from:to:)` (and which are represented by DeclName in the compiler). Almost immediately diagnostics got better, testing optional protocol requirements got shorter, and a lot of compiler implementation got simpler.

And then we kind of got stuck here [at the time]. We [had] full names used throughout the compiler, but tuple labels still appear[ed] in types. They're still used in mangling. We got rid of the "tuple splat" feature, but still modele[ed] out-of-order arguments as "tuple shuffles". And we allow a number of conversions that look like they should be invalid, but aren't.

(And it's important that we continue allowing them, or at least some of them, because we want to be able to pass existing functions to things like `map` and `reduce` without worrying about conflicting labels.)

So we've given up the perfect ideal of tuple-to-tuple. But we did it because we value other things more than that ideal: variadics, default values, trailing closures, inout, autoclosure, distinct argument labels and parameter names, referencing a function by full name, and diagnostics that better match the user's likely intent (particularly given the naming guidelines and existing libraries). I think that's a worthwhile trade.

Jordan

P.S. Anyone is allowed to think this is not a worthwhile trade! But part of the purpose of this story is to show that [at the time of SE-0111] we [were] already 90% of the way towards making tuples and function arguments completely separate, even if they have similar syntax. This proposal gets us to maybe 95%.