Background

Here's some code that turns `"Hello"` into `"Hippo"` in various mainstream languages:

```
"Hello".replaceAll("l", "p").replaceFirst("e", "i")
Java
"Hello".replaceAll("l", "p").replaceFirst("e", "i")
JS
"Hello".replace("l", "p").replace("e", "i", 1)
Python
"Hello".gsub("l", "p").sub("e", "i")
Ruby
"Hello".Replace("l", "p").Replace("e", "i")
C#
"Hello".replace("l", "p").replacen("e", "i", 1)
Rust
```

Although the function names differ—Java calls it `.replaceFirst` whereas Ruby calls it `.sub`—the basic structure of these chained calls looks identical across the board: `"Hello".foo("l", "p").bar("e", "i")`.

Here's the equivalent code in some functional languages:

```
"Hello".replace("l", "p").replace("e", "i")
Scala
"Hello".Replace("l", "p").Replace("e", "i")
F#

(-> "Hello"
Clojure
```

```
    (str/replace "l" "p")
    (str/replace-first "e" "i"))
```

```
"Hello"
Haskell
& replace "l" "p"
& replace "e" "i"
```

```
"Hello"
OCaml
|> Str.global_replace (Str.regexp "l") "p"
|> Str.replace_first (Str.regexp "e") "i"
```

```
"Hello"
Elixir
|> String.replace("l", "p")
|> String.replace("e", "i", global: false)
```

```
"Hello"
Gleam
|> string.replace("l", "p")
|> string.replace("e", "i")
```

```
"Hello"
Elm
|> String.replace "l" "p"
|> String.replace "e" "i"
```

```
"Hello"
Roc
|> Str.replaceEach "l" "p"
|> Str.replaceFirst "e" "i"
```

Among these functional languages, only F# and Scala write this in the mainstream way.

This makes life easier for F# and Scala beginners coming to either language from a mainstream one, because the structure of common expressions like this is familiar. It also lowers the barrier to people trying out those languages in the first place, because when they encounter code snippets before they've started learning the language, those code snippets look more comfortable to them. They don't look alien, they look familiar.

In contrast, the other examples from the functional languages not only diverge from the mainstream structure, their structures also have inconsistencies with one another.

- Clojure chains the calls with the `->` operator; Haskell uses `&` instead; Roc, Elm, Gleam, Elixir, and OCaml use the `|>` operator.
- Haskell, OCaml, Elm, and Roc separate function arguments with spaces and do not require parens for function calls, whereas Elixir and Gleam use the mainstream design of commas and touching parens in function calls. Clojure, being a lisp, requires parens—but on the outside—and no commas.

Besides familiarity and learning curve, there are some other observable differences between the `|>` examples and the `.` examples:

- The `|>` examples fully qualify their calls (e.g. `Str.replaceFirst`), so the source code tells you exactly where the function can be found. In the `.` examples, you have to infer that these are string operations. (In these specific examples, inferring that these are string operations is trivial, but other cases can be less so.)

- Because the `|>` examples are fully qualified, they are necessarily more verbose.
- The `.` examples are nicer to use with autocomplete.
    - The autocomplete menu appears when you tap the `.` key instead of having to hold shift (on most keyboards) and tap at least two keys (plus maybe a space first), namely: `|>`
    - In languages where `|>` is used, it's most common to call functions directly—without using the `|>` operator. When calling without the operator, autocomplete works differently and also less effectively (it can't start with a subject and autocomplete based on what operations work on that subject).
    - In contrast, in languages where `.` is used, it's the most common way to call functions, so autocomplete always appears after the `.` and most often gets to facilitate the "start with a subject and autocomplete operations that are supported on that subject" experience.

Looking again at the Scala and Roc examples side by side…

```
"Hello".replace("l", "p").replace("e", "i")
Scala
```

```
"Hello"
Roc
|> Str.replaceEach "l" "p"
|> Str.replaceFirst "e" "i"
```

Both of these are functional languages calling pure functions on immutable data.

The Scala one gets nicer autocomplete, a more familiar learning experience for beginners, more conciseness, and a bigger strangeness budget. The Roc version

is more self-documenting, but inlay hints in an editor can surface the same information without needing it to be written out in the source code.

The parens-and-commas calling style everywhere has a downside when it comes to certain DSLs. For example, roc-lang.org is built in part using a Roc HTML DSL that makes extensive use of calling via whitespace. Here's a snippet:

```
footer [] [
    div [id "footer"] [
        div [id "gh-link"] [
            a [id "gh-centered-link", href "…"] [
                ghLogo,
                span [id "gh-link-text"] [text
"…"],
            ],
        ],
        br [] [],
        text " powered by ",
        a [href "…"] [text "Netlify"],
    ],
],
```

With parens-and-commas calling, it would have the same shape, but more punctuation:

```
footer([], [
    div([id("footer")], [
        div([id("gh-link")], [
            a([id("gh-centered-link"), href("…")],
[
                ghLogo,
```

```
                span([id "gh-link-text"],
[text("…")]),
            ]),
        ]),
        br([], []),
        text(" powered by "),
        a([href("…")], [text("Netlify")]),
    ]),
]),
```

There's precedent for both of these DSLs in the wild. Elm has had success using the whitespace-calling DSL (we got this DSL directly from Elm), and Lustre—a Gleam framework which is also directly descended from Elm—uses the parens-and-commas style, and that style seems to be working out just fine for them too.

The reason Lustre uses parens-and-commas is that the Gleam language supports only that calling style. For a long time I didn't know this, but Gleam actually used to have a calling style just like Roc, Elm, and Haskell—with spaces and no parens! In an interview last year, Gleam's creator Louis explained why Gleam uses parens-and-commas today:

> "Rather than having an Elm syntax—although we did originally have a syntax like that, [now] we have a syntax that's much more familiar to JavaScript programmers, C programmers…and I originally thought, what's the point of that? It's just syntax, it doesn't matter. And then we switched syntax, and suddenly everyone was like, 'Oh, this language is great now!' And I was like…nothing's changed! So I was wrong, syntax really does matter."

I appreciate this point. And while I consider both of the above snippets readable, I do prefer reading the Elm-style DSL over the Lustre-style one—mostly because

the extra punctuation feels like unhelpful clutter in this case. So even considering Gleam's data point of people preferring the parens-and-commas style so much that it dramatically changed peoples' willingness to try out the language, there isn't a clear, unambigious winner here. Each style has pros and cons compared to the other.

That said, this proposal introduces some semantic features to Roc which essentially require a parens-and-commas calling style to be ergonomic. As such, this proposal also introduces a parens-and-commas calling style as an alternative to the current whitespace style. Whether or not the new style should become the "only way to do it" (with the associated DSL drawbacks) is left out of scope in this proposal, because:

- There's precedent for languages that do parens-and-commas calls having an optional whitespace-based style that coexists with the parens-and-commas style—for example, CoffeeScript and Ruby both have this.
    - Notably, Ruby allows whitespace calls but has a style guideline of "only use it in DSLs" (such as RSpec and ActiveRecord, both of which are widely used).
    - So there's mainstream precedent for having both calling styles coexist, and we can discuss whether we want that as a separate topic from this proposal.
- Regardless, we'd implement the new syntax as a nonbreaking change anyway, so we'll have an opportunity to try out having both supported at the same time no matter what.
- We can use that experience to inform whether we want to keep both or just one.

Paren Placement

Here's a call that comes up in the Roc port of the Bash script that builds roc-lang.org:

```
Dir.copyAll! "public" "build"
```

If I want to nest another function call inside this call, today the only way to write it is:

```
Dir.copyAll! (Str.append "pub" "lic") "build"
```

I've seen a lot of Elm beginners trip over this paren placement requirement in nested function calls. Back when I used to do a lot of in-person Intro to Elm workshops, it was extremely common to see beginners write it like this out of habit:

```
Dir.copyAll! Str.append("pub", "lic") "build"
```

This, of course, wouldn't compile. It was also common for beginners to add an extra comma that wasn't needed, and which would cause a different compiler error:

```
Dir.copyAll! Str.append("pub", "lic"), "build"
```

I saw mistakes like this happen so often that I added an explicit section on nested function calls to my Intro to Elm workshop, spelling out exactly where the parens go, and went out of my way to say, "This is a really easy beginner mistake to make, so don't feel bad if you do it! But if you get an error, remember…the parens go on the outside."

Even with that explicit section on where the parens go, I'd see a few beginners in every class make the mistake anyway. Muscle memory is powerful!

This proposal adds full support for a parens-and-commas calling style in Roc. So as a beginner, you can write the following and it will Just Work the way you expect.

```
Dir.copyAll!(Str.append("pub", "lic"), "build")
```

This should eliminate that common beginner stumbling block from the learning curve.

The same beginner concern about nested paren placement applies to tags as well, and this same style could be used for them. (Rust uses this syntax too.) For example:

```
answer = someFn(arg1, Ok(arg2), arg3)
```

This could be used in any tag application, nested or not, and in pattern matching too:

```
Ok(foo) -> …
```

Just like with function calls, for the scope of this proposal, using this style would be optional; we can later discuss whether we want to narrow our set of allowed styles.

Parsing Ambiguity

Over the years, a recurring common suggestion for Roc's function calling syntax has been to add commas between the arguments. This is because of the inconsistency between Roc's function types (which have commas separating the arguments), function declarations (that is, lambdas—which also have commas separating the arguments), and function calls (which today never have commas separating the arguments).

In the context of this proposal, you could think of this idea as "parens are optional" when doing the parens-and-commas calling style. So the idea is that either of these styles would Just Work, and would compile to the same thing:

```
Dir.copyAll!("public", "build")

Dir.copyAll! "public", "build"
```

Unfortunately, supporting commas-without-parens would create a parsing ambiguity in collection literals. Consider the following, from the earlier HTML DSL example:

```
[id "gh-centered-link", href "…"]
```

Today, this unambiguously parses as two function calls. It's equivalent to the following:

```
[id("gh-centered-link"), href("…")]
```

However, if Roc were to adopt the "commas always separate arguments, but parens are optional" calling style, then this could be parsed as a standalone expression:

```
id "gh-centered-link", href "…"
```

On its own, this expression would unambiguously parse to the equivalent of:

```
id("gh-centered-link", href("…"))
```

…which would mean that one might reasonably expect that adding square brackets around it would not change its meaning. That would have unfortunate consequences for our HTML DSL though, because it would mean that this:

```
[id "gh-centered-link", href "…"]
```

…would now parse as this:

```
[id("gh-centered-link", href("…"))]
```

That would break the DSL.

Alternatively, if we were to say that commas inside collection literals take precedence over function arguments, then it could parse the way the DSL wants it to—namely:

```
[id("gh-centered-link"), href("…")]
```

…but now, adding square brackets around a totally valid standalone function call like the earlier example of `id "gh-centered-link", href "…"` suddenly changes the meaning of that call, and introduces a compiler error—just because you put it in a list. You'd have to add parens to fix it, which means that now multi-arg function calls look different depending on whether they happen to be inside collection literals or not.

The fundamental problem here is that if functions can be called using commas but not parens, then when using that style inside collection literals (such as list literals) where elements are also separated by whitespace and commas, the meaning of a comma becomes ambiguous.

As such, this proposal does not propose a "commas but no parens" option.

Instead, this proposal would result in (perhaps temporarily) two supported calling syntaxes: the status quo (no parens, no commas), and parens-and-commas. There would be no in-between option of "commas but no parens," because that would introduce a parsing ambiguity.

In the future, we may decide that we only want to support one of these two styles, but as mentioned earlier, that question is left intentionally out of scope for this proposal. This is because we'll naturally want to implement the new style as a backwards-compatible change anyway, and our answer to that question can be strictly better-informed once we've had a chance to actually try it out with both styles supported at the same time.

Chaining with Dots

The `Dir.copyAll!` call we've been using is part of a larger expression in that script:

```
Dir.copyAll! "public" "build"
|> Result.mapErr CopyAllFailed
|> try
```

Here's how this expression could look in this proposal:

```
Dir.copyAll!("public",
"build").mapErr(CopyAllFailed)?
```

This could parse as valid Rust code, and if you removed the `?` at the end, it could parse as valid Ruby too. If you also removed the `!` in `copyAll!`, it could be syntactically valid in basically every mainstream programming language. This would make Roc feel more familiar and approachable to a lot of beginners!

Also notice that at each step of the way, autocomplete can now work consistently whenever you tap the `.` key. First, you have no subject to autocomplete on, because you want to create a new directory from scratch. So you type:

```
Dir.
```

Autocomplete could list all the exposed functions in the `Dir` module, including:

```
Dir.copyAll!(_, _)
```

So you select that one, fill in the blanks, and then add a dot after the close paren:

```
Dir.copyAll!("public", "build").
```

Again, after tapping the dot, you get a list of autocomplete options. This time, thanks to type inference, they are operations that work on a `Result` (we'll discuss how the compiler knows this in the next section). They might include:

```
Dir.copyAll!("public", "build").mapErr(_)
```

So you select that option, fill in the blank, and finally add the `?` at the end. Every step of the way, "press dot for autocomplete" was the consistent editor experience.

Also, this `?` design works in an obvious (and convenient) way between calls:

```
Foo.bar("a", "b")?.blah(Bar.baz)?.stuff(etc)?
```

Essentially, `?` works just like how it does in Rust—where it's used all over the place!

Static Dispatch

How did the compiler know this `.mapErr` refers to `Result.mapErr` specifically? Not just the autocomplete, but the Roc compiler itself—now the code no longer says `Result.mapErr`, so how does it know which exact `mapErr` function to call?

The answer is a form of static dispatch. The proposed algorithm works like this:

1. After type-checking, we know that `.mapErr` is being called on a `Result`.
2. `Result` is a nominal type which is defined in the `Result` module. (Well, it's not today…but let's pretend for a moment that it is and come back to that later.)
3. Therefore, we're going to look for this `mapErr` function in the top level of the `Result` module.
4. There it is! Pass the value in front of the dot to that function as its first argument, and we're done. (If the `Result` module did not define a top-level `mapErr`, or if we couldn't have accessed it in this module because it wasn't exposed, we'd get a compile-time error.)

In an earlier draft of this proposal, I was referring to calls in this document like `foo.mapErr(…)` as "static dispatch calls," because I felt that "method call" (which is the term I instinctively associate with that syntax) might suggest that `mapErr` is formally defined in some other way than just an an ordinary function declaration—after all, in most languages, it is! (Not just object-oriented languages, either…Haskell's typeclasses have "methods" and Rust's traits also have "methods"—and neither are defined as ordinary functions the way these are in Roc.)

However, after many instances of trying to correct myself from referring to these as "method calls," I realized it would be easier to embrace that terminology and just explain it as "in Roc, a method is an ordinary function whose first argument

happens to be defined in the same module." So if I refer to "the `mapErr` method," what I mean is:

- A normal function
- …that happens to be named named `mapErr`
- …which takes a nominal type as its first argument
- …and which is defined in the same module as that nominal type

Note that this "method-calling syntax" implies changing what the dot operator means:

- `x.0` is still a tuple access; just from looking at this, I can can tell that `x` is a tuple.
- `x.foo` is still a record field access; I can tell `x` is a record just from looking at this.
- `x.foo(y)` is now a statically-dispatched *method call,* which I can also tell just from looking at it. (This use of the dot operator is unrelated to records and tuples.)

Calling `x.foo(y)` is technically valid Roc syntax today, but it accesses a function from a record field and then calls it. This sees almost no use in practice, which is why it seems fine to repurpose the syntax. That said, if you really want to do that, this proposal still allows it: just write `(x.foo)(y)` (which you can also already do today if you like) so that the parens around `(x.foo)` isolate the record field access as its own expression.

Earlier, I noted that `Result` is not a nominal type today. This is true, and may or may not remain true. There's a proposal unrelated to this one which could make `Result` a nominal type, but this proposal can work either way. Just like how the compiler can infer Ability implementations for structural types, it can also infer method implementations for them too, meaning that `myResult.mapErr(foo)` can Just Work.

To be more specific: if you call `.mapErr()` on a type that's inferred to be a tag union with no tags other than `Ok` and `Err`, then the method call would resolve to the function `Result.mapErr`. The same could be done with all the relevant functions exposed by the `Result` module. (If `Result` were a nominal type, it wouldn't need special-casing, but the point is that `.mapErr()` can be made available on `Result` values either way.)

Static Dispatch can replace Abilities

In the expression `!foo.bar("hi")`, the type of `foo` would be inferred as:

```
a where a.bar(Str) -> Bool
```

You can read this as "some value (whose type has been named **a**) that has a method named `foo` which returns a `Bool` and takes a `Str` as its second argument." (The first argument in a method is always the value it's being dispatched on.)

The `where` in this syntax might look familiar: it's similar to the type syntax for Abilities.

As it turns out, static dispatch is more than a syntactic convenience; it can also replace Abilities altogether! Consider this static dispatch call:

```
foo.isEq(bar)
```

If `foo` is a nominal type, and its module exposes a function named `isEq`, this code will call that function passing `bar`. Now suppose all the builtin modules exposed functions which followed this convention:

```
Str.isEq : Str, Str -> Bool
```

```
Bool.isEq : Bool, Bool -> Bool

List.isEq : List a, List a -> Bool
    where a.isEq(a) -> Bool
```

I can call `myStr.isEq(other)`, `myBool.isEq(other)`, `myList.isEq(other)`, etc., and they all work. From there, we change the expression `a == b` to desugar to `a.isEq(b)` (instead of today's desugaring to `Bool.isEq a b`), and we end up with equality working the same way as today—just presented differently.

Notice that type for `List.isEq`, which includes the `where a.isEq(a) -> Bool` clause to express that you only have equality for a `List` if its elements also support equality. Today, expressing this requires special syntax—`implements Eq if a implements Eq`—which isn't very discoverable, and doesn't appear anywhere else in the language. Methods can express this constraint using an ordinary `where` clause!

The compiler could infer `.isEq()` method implementations for structural types just like it does for the Eq Ability today. It would follow the same rules; the compiler wouldn't infer `.isEq()` for functions, just like how it doesn't infer Eq for functions today, so calling `myFn.isEq(otherFn)` would still give a compile-time error.

And just like that, Abilities can be be completely removed from the language, and replaced by static dispatch. We've added one language feature but subtracted another. And it's better than that, because the replacement feature is not only simpler, it turns out to have more benefits than the ones we've discussed so far!

For example, replacing Abilities with methods simplifies module documentation, because what a module exposes becomes a plain list of types and functions. There's no special category of documentation for "Abilities defined on this type" because the equivalent in the dispatch design is just the plain functions the module already exposes.

In other words, instead of formally specifying a new relationship on the `Str` type…

```
Str implements [Eq, …]
```

…we add one plain function to the rest of the functions in the `Str` API:

```
Str.isEq : Str, Str -> Bool
```

It's looks and works exactly like all the others. There's no separate categorization for it!

We can also include a convenience syntax for auto-generating some of these functions, similarly to what we do today with Abilities—e.g. in the module header for Str.roc:

```
exposes [Str generating [toHash, encode, decode], …]
```

Just like today with abilities, there would be a hardcoded list of supported functions that could be auto-generated in this way. (The `generating` keyword could also be used in the definition of a nominal type itself, for the uncommon case where you want to generate the function but not expose it.)

Here are some implications of this design, all of which I think are major positives for Roc:

1. Modules expose functions and types and that's it. I *love* this. Elm has that property, Roc had that property before Abilities, and static dispatch allows Roc to have that property again while still addressing the use cases that Abilities address today.
2. In contrast to Abilities, there is no concern here for the "classification trap" where time gets wasted defining and implementing abilities for the sake of classifying types and formally naming what they "are," instead of focusing on their APIs. In this design, the API is literally all there is to consider—exactly as it should be!
3. Despite shifting the language even more strongly toward its philosophical foundation of "plain old functions and types and that's it," at the same time this change makes Roc—astonishingly—more familiar (at least syntactically) to beginners who have only used mainstream languages before.

Static Dispatch facilitating Functional Programming Techniques

The first time I heard about function composition and partial application (which happened to be in a Haskell tutorial), I was pretty confused. I don't remember the exact example anymore - it was too long ago - but I vaguely remember the tutorial example building up to a refactor of some expression like this:

```
map (\num -> abs num - 1) [-2, 0, 2]
```

If you put this into a Haskell repl, you get this answer:

```
[1, -1, 1]
```

So far, so good. Now for the refactor using partial application and function composition:

```
map (flip (-) 1 . abs) [-2, 0, 2]
```

This does the same thing, except it's expressed in a way where now, at a glance, I have no idea what it's doing. I have to mentally desugar it to understand it.

As I spent more time writing code in a functional style I learned a trick: I could write down the desugaring one step at a time instead of trying to keep it in my head. That way, it's harder to lose track of what I'm doing, and overall it gets me to the a correct understanding more quickly. Later I learned an even better trick: stop doing this.

If you're curious how this refactor works, the steps are:

```
map (\num -> abs num - 1) [-2, 0, 2]
original
map ((\num -> num - 1) . abs) [-2, 0, 2]
composition
map ((\num -> (-) num 1) . abs) [-2, 0, 2] prefix
application
map ((\num -> (flip (-)) 1 num) . abs) [-2, 0, 2]
flip (-)
map ((flip (-) 1) . abs) [-2, 0, 2]       partial
application
map (flip (-) 1 . abs) [-2, 0, 2]     drop unnecessary
parens
```

In comparison, how long does it take you to figure out what this code does?

```
[-2, 0, 2].map(.abs().sub(1))
```

Personally, I don't need to mentally desugar that to understand it. I'm not even sure I need to explain how it works, do I? If the document ended right here, with

no explanation of how the feature worked, do you think you could turn around and explain to a beginner what it desugars to? I bet you could!

Nevertheless, let's be clear about it: here's what the expression `.abs().sub(1)` would desugar to in this proposal.

```
\x -> x.abs().sub(1)
```

It's just like how `.foo` already desugars to `\x -> x.foo` and `.foo.bar` could reasonably desugar to `\x -> x.foo.bar` (but doesn't yet, as of this writing).

Here's an example of some real-world Roc code that could make use of this syntax:

```
reqHeader.value
|> Str.fromUtf8
|> Result.try \s -> s |> Str.split "=" |> List.get
1
|> Result.try Str.toI64
```

This could become:

```
Str.fromUtf8(reqHeader.value)
.try(.split("=").get(1))
.try(Str.toI64)
```

For comparison, here it is again but with the `.split("=").get(1)` desugared:

```
Str.fromUtf8(reqHeader.value)
.try(\s -> s.split("=").get(1))
```

```
    .try(Str.toI64)
```

(Note that the last line could also be `.try(.toI64())`, but that only saves one character…and the two characters in `()` are much less helpful than the three in `Str`.)

None of the concerns I wrote about in the FAQ apply to this syntax for partial application and pointfree function composition. It's so easy to understand, I feel weird labeling it as "partial application" and "pointfree function composition" even though that's what it technically is. I guess what feels weird is that it offers the benefits of both without the struggles that I've come to associate with them over the years in other languages!

This sugar is so straightforward, the formatter could even rewrite the desugared one to remove the `\s -> s` without needing any information besides its usual parse tree, just like it could shorten `\s -> s.foo` today. (Whether or not the formatter *should* shorten either of those is a separate discussion, and intentionally out of scope for this proposal.)

In fairness, Roc wouldn't be the first language to support opt-in partial application. One of the many uses of underscore in Scala is something similar; Scala would even let you write the `(\num -> abs num - 1)` example as `(abs _ - 1)`, and so would LiveScript. It's always been possible to add something that flexible to Roc, but it comes with a much deeper rabbit hole of tradeoffs than the much simpler `.abs().sub(1)`.

Operator Overloading

We've discussed operator overloading a lot in the past. One of the main concerns with adding it to the language is that it could be misused to create confusing code, which was one of the main reasons James Gosling decided to omit it from Java even though he was accustomed to having it from C++.

Still, there seem to be certain domains where it ranges from valuable to crucial. Game development, for example, wants a different in-memory representation of matrices and vectors from what other domains want (there's a more efficient representation which follows different conventions from the ones in normal mathematics), and all representations of vectors and matrices want the normal math operators to be available. There are also use cases for things like complex numbers, currency, and so on.

I think it's likely that at some point Roc will introduce some form of operator overloading, simply because Roc targets the long tail of domains, and we've already seen several domains where it's important enough to be a deal-breaker that Roc can't do it.

This is not a proposal about operator overloading, but this proposal does impact the design space for it. Today, if we want to implement operator overloading, we would need to change the type of common functions like Num.add to use abilities, e.g.

```
Num.add : Num a, Num a -> Num a
```

…becomes:

```
Num.add : a, a -> a
    where a implements Add
```

I certainly prefer the first type! Unfortunately, this change would need to be repeated for each operator; it would affect Num.sub too, and Num.mul, and so on.

In this proposal, we can have operator overloading while still maintaining the current (simpler) Num APIs. All we'd have to do is to desugar `a + b` as

`a.add(b)` and we're done. That's it, that's operator overloading. Want your type to work with the plus operator? Expose an `add` method.

On the one hand, this is an extremely simple design for operator overloading. On the other hand, it does become a bit easier to unintentionally opt into having operators work with your API.

For example, it's common in many languages to have a function like `Set.add`. (We happen to call it `Set.insert`, but `Set.add` was considered too.) If that's the best name for your function, is it okay that + now Just Works on your type, even if that usage doesn't make sense? It's a valid question. Also, the + operator in arithmetic is commutative, but nothing says `a.add(b)` has to be commutative; b doesn't even have to be the same type as a!

So, on the one hand, this design makes operator overloading extremely simple and straightforward. The learning curve is near zero. On the other hand, it does mean that we might want to revisit some of the Num names. For example, should it become `Num.plus` instead of `Num.add`? That seems less likely to cause collisions, and `a.plus(b)` is more clearly coupled to the semantics of + because it's literally the same word. `count.plus(1)` still reads fine to me.

This proposal wouldn't change how any of the existing operators desugar, and operator overloading in general is out of scope for this. However, since this proposal woul remove abilities in favor of methods, it seemed notable to mention the impact that would have on a (likely but not certain) future operator overloading addition to the language.

Chaining methods with non-methods

We've seen how this proposal's method calling syntax can be used as an alternative to the `|>` pipeline operator (at least, in cases where you'd be piping into a function from another module). However, sometimes you want to include

a locally-defined function in a chain of function calls. In other words, you don't want any kind of dispatch, you just want to call a function that's already in scope.

Doing that with the `|>` operator doesn't work great in method call chains. For example:

```
makeStr(a, b) |> transform(otherArg).append("!")
```

The precedence of the pipeline operator means this expression would desugar to:

```
transform(makeStr(a, b), otherArg).append("!")
```

In other words, the pipeline doesn't continue the previous chain of dot-calls, but rather it always separates two independent chains of dot-calls.

There's an additional ergonomics problem here: when we tap dot, we get autocomplete suggestions only for method calls on the subject. Local functions couldn't be included in that list, even if they would be perfectly valid calls to make on the given subject, because dot-syntax couldn't be used to add those calls to the chain.

This proposal fixes both of these problems by changing the `|>` syntax to something that looks more like our string interpolation syntax:

```
makeStr(a, b).(transform(otherArg)).append("!")
```

This `.(transform(otherArg))` part of the chain would instead be written today as `|> transform otherArg`. The differences are:

- Since this starts with a dot, it works seamlessly with autocomplete. When you press dot, you can see both dispatchable operations as well as local functions.
- Since it's both dot-based (instead of pipe-based) and enclosed in parens, you can continue a chain of dots normally afterwards, including any combination of dispatch calls and local function calls you like. There's no precedence problem!

Here's a concrete example of this in action, taken from the Rocci Bird source code:

Today

```
pipes =
    prev.pipes
    |> updatePipes
    |> List.appendIfOk pipe
```

Proposed

```
pipes = prev.pipes.(updatePipes).appendIfOk(pipe)
```

If desired, this could also be written in a multiline style like so:

```
pipes =
    prev
    .pipes
    .(updatePipes)
    .appendIfOk(pipe)
```

Amusingly, this is a real-world example of using all the dot syntax varieties except tuples:

- `.pipes` adds a record field access
- `.(updatePipes)` adds a local `updatePipes` call
- `.appendIfOk(pipe)` adds a `.appendIfOk(pipe)` method call

Importantly, although the syntax now looks more like a mainstream language, the semantics are exactly the same as before. It's still just a chain of pure function calls!

Inlay Type Hints

As mentioned earlier, one tradeoff here is that the dots version has removed module qualifiers from its calls. Specifically, it has `.appendIfOk(pipe)` in place of of the original's `List.appendIfOk pipe`. One way to compensate for this is an editor feature that's common today (but was not when Roc was first created)—inlay hints.

As of this writing, no Roc editor integration supports inlay type hints, but supposing we did have that, the proposed version could display in the editor like so:

```
pipes =
    prev                GameState
    .pipes              List Pipe
    .(updatePipes)      List Pipe
    .appendIfOk pipe    List Pipe
```

Here's the original `|>` version again, with the same inlay hints:

```
pipes =
    prev                        GameState
    .pipes                      List Pipe
    |> updatePipes              List Pipe
```

```
|> List.appendIfOk pipe     List Pipe
```

Here, the last inlay hint is redundant; assuming we already know that the builtin `List.appendIfOk` returns a `List`, we could infer that it was a `List Pipe` based on the other hints.

Of course, another way to look at this is that the inlay hint makes the qualified `List.` module name redundant. If we know the `.appendIfOk` call is taking a `List Pipe`, we can infer that it must be `List.appendIfOk` because that's where the `List` type is defined, and that's how static dispatch works.

Given that the information has become redundant, in this case the more concise dispatch version feels overall nicer to me (among the two versions with inlay hints), in that there are fewer tokens of source text on the screen, but it still shows all the same information. And the version with the inlay hints shows more info than the one without, because in that version I have to guess the types of the unqualified `.pipes` and `updatePipes`.

Granted, inlay hints are not always a complete replacement for the benefits of fully-qualified calls. There's a style difference, e.g. I can look at `|> List.appendIfOk pipe` and know that it's `List.appendIfOk`, whereas with `.appendIfOk pipe` I need to look at the previous line's type hint to be able to tell which function is being dispatched to.

There are also situations where inlay type hints are unavailable—for example, in the middle of single-line expressions. Although it's certainly possible for an editor to show inlay type hint in the middle of expressions, it isn't commonly done today. Similarly, code review tools (e.g. GitHub) also tend not to support inlay hints, at least not today.

Personally, even with inlay hints, I still have a minor aesthetic preference for the qualified call style. (Nostalgia probably plays some role in that.) But taken

together with all the other advantages of this style, to me it isn't even close. The sum of all the benefits of the dispatch style overwhelms this comparatively minor stylistic difference.

Type Aliases for `where` Constraints

The last thing to note about this proposal is that it would be desirable to have a way to make type aliases for `where` constraints, so that you wouldn't have to list out all the method names every time.

For example, consider a `List.sort` function (which we don't have today, but let's just think about how it would be defined today compared to in this proposal).

Today:

```
List.sort : List elem -> List elem
    where elem implements Sort
```

Proposed:

```
List.sort : List elem -> List elem
    where elem.order(elem, elem) -> [LT, EQ, GT]
```

Rather than writing it out like this every time—some Abilities like Hasher, Encoding, and Decoding, all have quite a lot of functions on them, so doing it like this could get very verbose and repetitive—in this proposal, we could use a type alias like so:

```
List.sort : List elem -> List elem
    where elem.Sort
```

```
Sort a : a
    where a.order(elem, elem) -> [LT, EQ, GT]
```

Basically the `where elem.Sort` constraint is saying that Sort is a type alias that only adds a `where` constraint to a value (if `Sort` is anything other than that, it would be a compile-time error to write `elem.Sort`), and all this code does is to propagate the constraints in the alias's `where`.

For an example of multiple constraints, we can look at `Dict.insert`:

Today:

```
Dict.insert : Dict k v, k, v -> Dict k v
    where k implements Hash & Eq
```

Proposed:

```
Dict.insert : Dict k v, k, v -> Dict k v
    where k.Hash, k.Eq

Hash a : a
    where
        a.hash(hasher, a -> hasher),
        hasher.Hasher,

Hasher a : a
    where
        ...etc
```

So everything ends up being about as concise as today's Abilities syntax.

Summary of Proposal

Here's what would change in this proposal:

Function Calling Syntax

- You can now call functions in the mainstream way, e.g. `foo(bar, baz)`
  - This style can also be used for tag applications and patterns.
- You can also continue call them in the current style, at least for now.
  - In the future, we may decide to reduce this down to only allowing one style or the other, but that's out of scope for this proposal.

Methods

- You can now call statically-dispatched "methods" on any value, e.g. `foo.bar(baz)`
  - These always use the mainstream calling style; there is no whitespace option
- Structural types get method implementations inferred like how Abilities do, e.g. they all have `.isEq()` (or don't) according to the same rules as today
- The way method dispatch gets resolved for nominal types is:
  - Look at the top-level declarations in the module where the type was defined
  - Look for a function with the same name as method name
  - Call it, passing the subject of the method as its first argument.
    - If it's inaccessible, e.g. the calling module can't import it, give an error.
- There is a "method accessor" syntax, e.g. `.foo()`
  - The methods can be pre-applied with arguments, e.g. `.foo(bar)`
  - It can be a chain of multiple method calls, e.g. `.foo(bar).baz(etc)`
- As an example of method types, the type of `.foo("x").bar(!baz)` would be:

```
a -> c
where
    a.foo(Str) -> b,
    b.bar(Bool) -> c,
```

- You can use type aliases to define `where` constraints, so that you don't have to keep writing out all the method names and types every time.
- Abilities become deprecated in favor of methods.

"Dot-style pipe" syntax

- You can now write `.(foo)` as an alternative to `|> foo`
    - Also `.(foo(bar, baz))` as an alternative to `|> foo bar baz`
- The `|>` operator becomes deprecated in favor of this.

The `?` Operator

- You can now use the postfix `?` operator. It works exactly the same way as `try`.
- The `try` keyword becomes deprecated in favor of this.

Examples

Here are some examples of existing Roc code refactored based on this proposal.

For example, here's a longer snippet from how the roc-lang.org build script would look in Roc today:

```
# Check jq version
try Cmd.exec! "jq --version"

# Create the build directory
```

```
    if try File.exists! "build" then
        try Dir.deleteAll! "build"

    try Dir.create! "build"

    # Copy public/ to build/
    Dir.copyAll! "public" "build"
    |> Result.mapErr CopyAllFailed
    |> try

    # Download the latest examples
    try Cmd.exec! "curl -fL -o examples-main.zip
https://…"
    try Cmd.exec! "unzip -o examples-main.zip"

    # Replace links in content/examples/index.md
    try replaceInFile!
        "content/examples/index.md"
        "](/"
        "](/examples/"
```

Here's the same code again, with parens-and-commas calling and the `?` suffix:

```
    # Check jq version
    Cmd.exec!("jq --version")?

    # Create the build directory
    if File.exists!("build")? then
        Dir.deleteAll!("build")?

    Dir.create!("build")?
```

```
    # Copy public/ to build/
    Dir.copyAll!("public",
"build").mapErr(CopyAllFailed)?

    # Download the latest examples
    Cmd.exec!("curl -fL -o examples-main.zip
https://…")?
    Cmd.exec!("unzip -o examples-main.zip")?

    # Replace links in content/examples/index.md
    replaceInFile!(
        "content/examples/index.md",
        "](/",
        "](/examples/",
    )?
```

Here's some HTTP request parsing logic today:

```
when headers |> List.keepIf \{ name } -> name ==
"cookie" is
    [reqHeader] ->
        reqHeader.value
        |> Str.fromUtf8
        |> Result.try \s -> s |> Str.split "=" |>
List.get 1
        |> Result.try Str.toI64
        |> Result.mapErr \_ -> BadCookie

    _ -> Err NoSessionCookie
```

Here's what it could be in this proposal:

```
when headers.keepIf(\{ name } -> name == "cookie") is
    [reqHeader] ->
        Str.fromUtf8(reqHeader.value)
        .try(.split("=").get(1))
        .try(Str.toI64)
        .mapErr(\_ -> BadCookie)

    _ -> Err(NoCookie)
```

Note that the `.try(Str.toI64)` could also be `.try(.toI64())`, but I think it's worth the extra two letters to make it more self-documenting.

With inlay hints, it could look like this:

```
when headers.keepIf(\{ name } -> name == "cookie") is
    [reqHeader] ->
        Str.fromUtf8(reqHeader.value)    Result Str
BadUtf8
        .try(.split("=").get(1))         Result Str [ …
]
        .try(Str.toI64)                  Result I64 [ …
]
        .mapErr(\_ -> BadCookie)         Result I64
[BadC…]

    _ -> Err(NoCookie)                   Result *
[NoCookie]
```

Some DSLs, such as parsers, can get more concise to use even without having to import things unqualified. For example, here's some code from a markdown parser:

Today:

```
    Parser.const (\str -> Heading One str)
    |> Parser.keep (
        Parser.chompWhile notEol
        |> Parser.map Parser.utf8
    )
    |> Parser.skip eol
    |> Parser.skip (Parser.strConst "==")
    |> Parser.skip (
        Parser.chompWhile \b -> notEndOfLine b && b ==
'='
    )
```

Proposed:

```
    Parser.const(\str -> Heading(One, str))
    .keep(Parser.chompWhile(notEol).map(Parser.utf8))
    .skip(eol)
    .skip(Parser.strConst("=="))
    .skip(Parser.chompWhile(\b -> notEol(b) && b ==
'='))
```

Writing `.skip` instead of `|> Parser.skip` makes the DSL more concise, without having to import the `keep` and `skip` functions unqualified—which is not only a convenience, it also means there's no chance of `keep` and `skip` shadowing anything (which could happen if they were imported unqualified).

Here's the HTML DSL snippet from the intro again:

```
    footer [] [
        div [id "footer"] [
```

```
            div [id "gh-link"] [
                a [id "gh-centered-link", href "…"] [
                    ghLogo,
                    span [id "gh-link-text"] [text
"…"],
                ],
            ],
            br [] [],
            text " powered by ",
            a [href "…"] [text "Netlify"],
        ],
    ],
```

Here it is again with parens-and-commas calling:

```
    footer([], [
        div([id("footer")], [
            div([id("gh-link")], [
                a([id("gh-centered-link"), href("…")],
[
                    ghLogo,
                    span([id "gh-link-text"],
[text("…")]),
                ]),
            ]),
            br([], []),
            text(" powered by "),
            a([href("…")], [text("Netlify")]),
        ]),
    ]),
```

Finally, for a much larger example, here's roc-ray-ball-physics today and in the proposal:

Today: https://gist.github.com/rtfeldman/2b9b64dcfd7f3fbed4b85111e8e02783
Proposed: https://gist.github.com/rtfeldman/ed83433a21c1ed28cabec71657003b06

Note that the proposed version in this case is using operator overloading, to give an additional sense of how that might look.