Not sure how to order things: lint rules first, or implementation?

Chapter 1

Lint Rules in `parsley-garnish`

Catalogue of lint rules implemented.

Categorise these – but also somehow split into the ”simple” rules and the ”complex” rules. Simple rules can consist of a single heading, containing: * Explanation of the rule * Simple example to show a diagnostic, and a before and after if it's fixable * How it's implemented in the code * Proof (if applicable) * Limitations

Simple rule ideas: * Overly complex parser definitions * Manually calling implicitSymbol instead of using the implicit

Not sure how to lay out the complex rules yet – so far this is just the left-recursion removal rule. The other complex rule(s) will likely share implementation details with the Parser/Func representation, so work from there.

1.1 Avoid Redefining Existing Parsers

* Catch cases when user manually writes out a parser that is already defined in the library

1.2 Simplify Complex Parsers

* Apply parser laws, re-using Parser and Func representations to do cool things <- should this be a separate rule?

1.3 Ambiguous Implicit Conversions

> Heroin is just one letter away from heroine, and implicit conversions are the heroine we don't deserve.
>
> Jamie Willis, 2024

Implicit conversions are a powerful feature in Scala, allowing for automatic type coercion between types. However, they are often considered a double-edged sword, as they can be easily abused, leading to unexpected behaviour and making code harder to reason about.

* Implicit conversions are controversial, but less evil when it comes to dsls * from the man odersky himself: https://contributors.scala-lang.org/t/can-we-wean-scala-off-implicit-conversions/4388 * Cite Jamie's design pattern from the scala paper - pattern 0 and 2c * See his discussion in phd thesis (nothing has been elaborated on in the masters thesis)

* have to use syntacticrule because we want to be able to provide diagnostics even if compilation fails, due to the exact problem of clashing implicits * so actually this would've been better as lint-on-compile, to annotate the compiler error message at the exact location - we'd also crucially have more information at that stage * but because the only other option for scalafix is semanticrules that are completely after compilation time, we can't

gather even this partial information * wartremover actually has a wart for this ˆ, but it only targets the definition of implicit conversions, so it's not actually what we'd want out of a lint-on-compile rule

1.4 Remove Explicit Usage of Implicit Conversions

1.5 Refactor to use Parser Bridges

* This would be cool, idk if I have time though, but this should also piggyback off of Func * the pos bridges don't actually exist, so we can ignore that case and just say its too much code synthesis * shouldn't be too bad? idk * indicate limitations that this will only work if the ADT is defined in the same file, in order to extend it

1.6 Left Recursion Removal

Chapter 2

Implementation

Non-terminal detection. This may get reworked/renamed since it's pretty specialised for leftrec rn, and in reality it's just trying to grab all the parsers.

Other util things? ACTUALLY NEED TO DO: import combinators if they aren't already imported

2.1 Parser Representation

Representation of Parsley combinators in parsley-garnish. Compare with approach in Scala Parsley, take cues from the 2018 paper. * Approach to composites? Need to think about this. * For LeftRec: Parse ASTs into a small group of core combinators, but we also need to represent composite combinators as their own case classes – recombine/"simplify" after analysis is concluded, it doesn't really matter if we completely change what combinators are used as long as semantic meaning is preserved. * For others: probably need to parse directly into composite combinators, since we don't want to destructively modify what combinators have been used. * Optimisations: for us, the goal is human readability, so this is interesting to compare to the paper. Lots of similar stuff actually, like top-down peephole optimisations utilising parser laws (I think I do it this way? Need to double check). * For cleanliness to isolate boilerplate: https://blog.sumtypeofway.com/posts/introduction-to-recursion-schemes.html – we don't have a generic traversal, but we can decouple the recursive application of a given partial function from the actual pf itself (I've called it .transform for the Parser class)

2.2 Function Representation

Abstraction built over scalafix/meta ASTs to represent functions. Allows us to statically evaluate function composition/flipping etc, so it doesn't turn into one big mess – again, human readability of the transformed output is the goal. Abstraction is again an ADT as a lambda calculus, but with parameter lists so not everything is curried. îdk, this is still a work-in-progress. Seems that there might not be enough time to uncurry the leftrec analysis so this design decision might not be super important. Representation as a lambda calc has allocation overhead, but greatly simplifies function evaluation via beta reduction, instead of having to deal with high-level representations of compose/id (not too bad tbh) and flip (annoying). Also attempted to make it typed but that didn't go so well with Scala's limitations on type inference.

* Extracting method arguments (alongside their types) is very painful * Need to unify information from signature (within symbolinformation) and synthetics * synthetics exist in certain cases: .apply methods, showing the concrete type of a generic argument, implicit conversions * from https://scalacenter.github.io/scalafix/docs/developers/semantic-tree.html: SemanticTree is a sealed data structure that encodes tree nodes that are generated by the compiler from inferred type parameters, implicit arguments, implicit conversions, inferred .apply and for-comprehensions.

* map, lift (implicit and explicit), zipped, (.as perhaps?) – these should surely boil down into two cases: (x, y).xxx(f) and xxx(f, x, y) * named function literals (val) * named method literals (def) * anonymous functions i.e. lambdas * functions with placeholder syntax * apply methods of case classes - symbol will tell its a class signature

so we use this as a clue to look at synthetics??? * generic bridges – I reckon the information will probably show up in synthetics again

I think look at symbol signature first, then look at synthetics based on some heuristics (e.g. if no symbol sig – if this happens will there even be synthetics?, if class signature)