Amy Language Server

Compiler Extension Final Report

Cédric Hölzl Matthieu De Beule
EPFL
matthieu.debeule@epfl.ch cedric.hoelzl@epfl.ch

1. Introduction

From labs 1 to 6, we implemented the various stages of an Amy compiler to WebAssembly. A working compiler could certainly be described as a big part of the ecosystem of a programming language, however the tooling to help the programmer be efficient should also be quite good to help make the adoption of the language an option for programmers.

Programmers expect certain features from their editor these days, like auto complete, go to definition, find references, inline compiler errors (with informative messages concerning type errors and such things). Traditionally, to provide this one would make a plugin specifically for the editor and the language (Python plugin for VSCode, a Python plugin for Sublime Text, a Python plugin for Vim, a Python plugin for Sourcegraph,... and this for every language). This presents a lot of work, most of which is not necessarily language-specific or editor-specific: a lot of it could be re-used. This traditional approach has m-times-n complexity, m editors and n languages.

A solution to this problem has become very popular lately, called the "Language Server Protocol". It consists in a protocol specifying how the editor (client) should communicate with a "language smartness provider" (server), where the server can be used by any client that conforms to the specification.

This reduces the complexity from $m \times n$ to $m + n$: implement the protocol in each editor once, and make a Language Server for each programming language once.

Our project consists of a Language Server for Amy. We did not implement all the functionality of a Language Server, but rather a proof of concept showing how one could adapt the existing compiler pipeline to provide "language smartness" for Amy to most editors.

2. Examples

For example, if a function gcd is defined as such:

```
def gcd(a : Int, b : Int) : Int = {
    ...
}
```

the editor will then be able to complete g to

```
gcd(a,b)
```

where a and b are the names given to the variables in the function declaration.

3. Implementation

We will quickly describe how we implemented the LSP and how this integrates with the existing compiler pipeline. We implemented function completion with naive arguments, which will complete a function with arguments. This uses the Lexer, Parser and NameAnalyzer stages of the pipeline.

3.1 Setup and testing of LSP infrastructure

We used the lsp4j library, as was suggested in the project description. We first wanted to get a working Language Server, that returned the same thing all the time. This posed more problems than we expected, since it was not easy to set up the editor. We had to do substantial reading in the specification and the documentation of the LSP plugins for the various editors. We finally managed to get SublimeText to show a hard-coded completion response (we still don't know how to set it up for Vim for example).

3.2 Using the existing compiler pipeline to complete functions

We run the pipeline until the NameAnalyzer, and then iterate on the definitions:

```
val pipeline = Lexer andThen Parser
                 andThen NameAnalyzer
pipeline.run(ctx)(files)._1.modules.foreach(_.defs.foreach ...
```

We then add the function definitions to the list of CompletionItems, and set the text to be inserted from the name of the function and the list of arguments.

```
completionItemPrintModule.setInsertText(name.name + "("
    +params.map(_.name.name)
        .reduceLeft((s,p) => s + ", "+ p) + ")")
```

4. Possible Extensions

We did not implement everything we wanted to implement at all, due to a lot of lost time setting up the editor and getting a basic server response.

We hoped to have time to work on more interesting completion (like completion dependent on scope), using types to suggest variables or functions, type errors, etc, so these would be a good way to extend what we have done.