Tethys

A Toy Functional Programming Language
with a System Fω-based Core Calculus
https://github.com/ThePuzzlemaker/tethys

James [Redacted] (aka ThePuzzlemaker)

January 15, 2022

Contents

1 Introduction

This "paper" (which is really just a well-typeset, but somewhat informal write-up) introduces Tethys, a toy functional programming language based on a System Fω-based core calculus. Hence the title.

There are two parts of Tethys: the surface language, and the core calculus. The core calculus is the intermediate representation of Tethys which is used for type checking and inference, and for interpretation. The surface language is the higher-level interface that is eventually desugared by the compiler/interpreter to the core calculus.

The reference implementation in Rust will not use any particular "tricks" in terms of interpretation, instead just using a simple tree-walk interpreter or similar.

This language was created in order to conduct informal research (i.e., not actually discovering anything interesting, probably) on type systems, especially bidirectional typechecking and polymorphism. Tethys is named as such as it is the name of the co-orbital moon to Calypso; as my work on this language is "co-orbital", so to speak, to my work on Calypso.

Please note that I am not an expert in pretty much any subject this writeup covers. If you notice something you don't understand, or that you think is a mistake, don't fret to point it out.

1.1 Background

> Note: This section is taken from some slides I prepared for a lightning talk in the r/PL Discord server. If it seems a bit un-prose-y, that's why.

Originally, I thought I needed to use System Fω, an extension of the polymorphic typed lambda calculus (System F), with the extension of type lambdas. However, it has some problems:

- System F is already undecidable enough, with respect to inference. Adding *more* types of types does **not** help.
- Theoretically, bidirectional typechecking (my chosen strategy) doesn't *need* unification, but it sure makes it nice. And having type lambdas means you need higher order unfication, which is scary because anything including "higher-order" in its name is automatically scary. Also it's undecidable, so you have to restrict yourself to higher-order pattern unification, which is still quite scary.
- I just can't really find much good introductory literature on System Fω (at least, literature that I can understand).

As it turns out, System Fω is more expressive than F, but **way** much more of a pain. Not even Haskell, the paragon of kitchen sink type system features has it. (By default, because of course GHC's gonna have it as an extension).

Here are some of the key theoretical differences:

- Within the type system, type constructors are opaque. Defining a type just creates opaque term-level and type-level constants, which don't normalize to the type or term

they "really" mean. Of course, these are stripped out after typechecking, once they're no longer needed.

- Type constructors are injective, i.e. for some type constructor C and types X and Y, if $C\ X$ and $C\ Y$ are the same type, then X and Y are the same type.
- Differently named type constructors are never equal, even if they "mean" the same thing, i.e. for some type constructors $C1$ and $C2$, and type T, $C1\ T$ never equals $C2\ T$, no matter how they are defined.

Note that this is pretty much nominal typing! (If my understanding of nominal typing is correct.)

2 The Surface Language

This section has not been started yet.

3 The Declarative Core Calculus

This section is, unsurprisingly, work-in-progress.

4 References

4.1 Acknowledements

Thanks to Brendan Zabarauskas and András Kovács, who have helped with many type system details. Many thanks to Mark Barbone (aka MBones/mb64), who provided a code sample of their bidirectionally typed (with unification) algorithm for typechecking higher-rank System F (Barbone, 2021), and who also noted some issues they had with "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism" (Dunfield and Krishnaswami, 2020).

4.2 References

Barbone, M. (2021). Typechecking for higher-rank polymorphism. https://gist.github.com/mb64/f49ccb1bbf2349c8026d8ccf29bd158e

Cai, Y., Giarrusso, P. G., & Ostermann, K. (2016). System F-Omega with equirecursive types for datatype-generic programming. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 30–43. https://doi.org/10.1145/2837614.2837660

Dunfield, J., & Krishnaswami, N. R. (2020). Complete and easy bidirectional typechecking for higher-rank polymorphism. *Int'l Conf. Functional Programming*.

Foster, N. (2012). CS 4110 – Programming languages and logics: Lecture #27: Recursive types. https://www.cs.cornell.edu/courses/cs4110/2012fa/lectures/lecture27.pdf

Jones, M. P. (1995). A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, *5*(1), 1–35. https://doi.org/10.1017/S0956796800001210

Zimmerman, J. (2017, September 27). *System Fω and parameterization*. https://blog.jez.io/system-f-param/