Tethys

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

The Puzzlemaker

September 7, 2021

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.

2 The Surface Language

This section has not been started yet.

3 The Declarative Core Calculus

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

3.1 Abstract Syntax

$\kappa ::=$	kinds:
$\quad *$	concrete types
$\quad \kappa \to \kappa$	type constructors

$\sigma, \tau ::=$	monotypes:
$\quad \alpha$	type variables
$\quad \texttt{bool}$	booleans
$\quad \texttt{int}$	64-bit signed integer
$\quad \{\overline{\tau_i}\}$	n-ary products
$\quad \langle\overline{\tau_i}\rangle$	n-ary sums
$\quad \sigma \to \tau$	arrows
$\quad \mu\alpha.\tau$	type-level least fixed-point
$\quad \lambda(\alpha :: \kappa).\tau$	type-level lambdas (type constructor)
$\quad \sigma\ \tau$	type-level application

$A, B, C ::=$	types:
σ, τ	monotypes
α	type variables
$\forall(\alpha :: \kappa).A$	universal quantification
$\tau\ A$	type-level application

$t ::=$	terms:
x	variables
c	constants
$e : A$	type annotation
$\{\overline{e_i}\}$	introduce product
$e.i$	product elimination (projection)
$\langle i = e \rangle$	variant introduction (injection)
$\textbf{case}\ e\ \textbf{of}\ e$	variant elimination
$\lambda x.e$	lambda abstraction
$e\ e$	lambda application
$e[\tau]$	type application
$\texttt{project}_{\mu\alpha.\tau} : \mu\alpha.\tau \to [\mu\alpha.\tau/\alpha]\tau$	isorecursive projection
$\texttt{embed}_{\mu\alpha.\tau} : [\mu\alpha.\tau/\alpha]\tau \to \mu\alpha.\tau$	isorecursive embedding

$c ::=$	constants and builtins:
$\texttt{fix} : \forall(\alpha :: *).(\alpha \to \alpha) \to \alpha$	least fixed-point combinator
$\texttt{true}, \texttt{false}$	boolean literals
$\ldots, -2, -1, 0, 1, 2, \ldots$	integer literals
$+, -, *, /, \texttt{mod} : \texttt{int} \to \texttt{int} \to \texttt{int}$	arithmetic operators
$<, >, \leqslant, \geqslant, \texttt{eq}, \texttt{neq} : \texttt{int} \to \texttt{int} \to \texttt{bool}$	arithmetic comparisons
$\texttt{and}, \texttt{or} : \texttt{bool} \to \texttt{bool} \to \texttt{bool}$	binary boolean operators
$\texttt{not} : \texttt{bool} \to \texttt{bool}$	negation
$\texttt{cond} : \forall\alpha.\texttt{bool} \to \alpha \to \alpha \to \alpha$	conditional

$\Gamma ::=$	contexts:

$\emptyset$	empty context
$\Gamma, x : A$	variable binding
$\Gamma, \alpha :: \kappa$	type variable binding

3.1.1 Kinding Rules

$$\frac{}{\Gamma \vdash \texttt{bool}, \texttt{int} :: *} \ (\textsf{DeclK-Builtin}) \qquad \frac{\Gamma \vdash \tau_i :: * \text{ (for all } i)}{\Gamma \vdash \{\overline{\tau_i}\} :: *} \ (\textsf{DeclK-Prod})$$

$$\frac{\Gamma \vdash \tau_i :: * \text{ (for all } i)}{\Gamma \vdash \langle\overline{\tau_i}\rangle :: *} \ (\textsf{DeclK-Sum}) \qquad \frac{\Gamma \vdash \sigma :: * \qquad \Gamma \vdash \tau :: *}{\Gamma \vdash \sigma \to \tau :: *} \ (\textsf{DeclK-Arr})$$

$$\frac{\Gamma, \alpha :: \kappa \vdash \tau :: \kappa}{\Gamma \vdash \mu\alpha.\tau :: \kappa} \ (\textsf{DeclK-Fix}) \qquad \frac{\alpha :: \kappa \in \Gamma}{\Gamma \vdash \alpha :: \kappa} \ (\textsf{DeclK-Var})$$

$$\frac{\Gamma, \alpha :: \kappa \vdash A :: *}{\Gamma \vdash \forall(\alpha :: \kappa).A :: *} \ (\textsf{DeclK-All}) \qquad \frac{\Gamma, \alpha :: \kappa \vdash A :: \kappa'}{\Gamma \vdash \lambda(\alpha :: \kappa).A :: \kappa \to \kappa'} \ (\textsf{DeclK-Lam})$$

$$\frac{\Gamma \vdash A :: \kappa \to \kappa' \qquad \Gamma \vdash B :: \kappa'}{\Gamma \vdash A\ B :: \kappa'} \ (\textsf{DeclK-App})$$

3.2 Normal Forms of Types

Normal forms of types are defined as a subset of the syntax of full types, where some terms can only be found with their "major" argument(s) in a "neutral" position, where they cannot be normalized further without more information.

$\sigma^{\mathsf{Nf}}, \tau^{\mathsf{Nf}} ::=$		normal monotypes:
	τ^{Ne}	neutral monotype
	$\texttt{bool}$	booleans
	$\texttt{int}$	64-bit signed integer
	$\{\overline{\tau_i^{\mathsf{Nf}}}\}$	n-ary products
	$\langle\overline{\tau_i^{\mathsf{Nf}}}\rangle$	n-ary sums
	$\sigma^{\mathsf{Nf}} \to \tau^{\mathsf{Nf}}$	arrows
	$\mu\alpha.\tau^{\mathsf{Nf}}$	type-level least fixed-point
	$\lambda(\alpha :: \kappa).\tau^{\mathsf{Nf}}$	type-level lambdas (type constructor)
$\sigma^{\mathsf{Ne}}, \tau^{\mathsf{Ne}} ::=$		neutral monotypes:
	α	type variables
	$\sigma^{\mathsf{Ne}}\ \tau^{\mathsf{Nf}}$	type-level application

$$
\begin{array}{llr}
A^{\mathsf{Nf}}, B^{\mathsf{Nf}}, C^{\mathsf{Nf}} ::= & & \text{normal types:} \\
& \sigma^{\mathsf{Nf}}, \tau^{\mathsf{Nf}} & \text{normal monotypes} \\
& A^{\mathsf{Ne}}, B^{\mathsf{Ne}}, C^{\mathsf{Ne}} & \text{neutral types} \\
& \forall(\alpha :: \kappa).A^{\mathsf{Nf}} & \text{universal quantification} \\
A^{\mathsf{Ne}}, B^{\mathsf{Ne}}, C^{\mathsf{Ne}} ::= & & \text{neutral types:} \\
& \alpha & \text{type variables} \\
& \tau^{\mathsf{Ne}}\ A^{\mathsf{Nf}} & \text{type-level application}
\end{array}
$$

3.3 Typing Rules

$\boxed{\Gamma \vdash e \Rightarrow A}$ Under context Γ, e synthesizes output type A

$\boxed{\Gamma \vdash e \Leftarrow A}$ Under context Γ, e checks against input type A

$\boxed{\Gamma \vdash A \bullet e \Rrightarrow C}$ Under context Γ, applying e to a function of type A synthesizes type C

$$
\frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ (\textsf{Syn-Var})
\qquad
\frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash e \Leftarrow A}\ (\textsf{Syn-Chk})
\qquad
\frac{}{\Gamma \vdash \texttt{true}, \texttt{false} \Rightarrow \texttt{bool}}\ (\textsf{Syn-Bool})
$$

$$
\frac{}{\Gamma \vdash \ldots, -2, -1, 0, 1, 2, \ldots \Rightarrow \texttt{int}}\ (\textsf{Syn-Int})
\qquad
\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}\ (\textsf{Syn-Ann})
$$

$$
\frac{\Gamma \vdash e_i \Rightarrow \tau_i \text{ (for all } i)}{\Gamma \vdash \{\overline{e_i}\} \Rightarrow \{\overline{\tau_i}\}}\ (\textsf{Syn-ProdIntro})
\qquad
\frac{\Gamma \vdash e \Rightarrow \{\overline{\tau_i}\}}{\Gamma \vdash e.i \Rightarrow \tau_i}\ (\textsf{Syn-ProdElim})
$$

$$
\frac{\Gamma \vdash e \Leftarrow \tau_i}{\Gamma \vdash \langle i = e \rangle \Leftarrow \langle \overline{\tau_i} \rangle}\ (\textsf{Chk-SumIntro})
\qquad
\frac{\Gamma \vdash e_1 \Rightarrow \langle \overline{\tau_i} \rangle \qquad \Gamma \vdash e_2 \Rightarrow \{\overline{\tau_i \to \sigma}\}}{\Gamma \vdash \textbf{case } e_1 \textbf{ of } e_2 \Rightarrow \sigma}\ (\textsf{Syn-SumElim})
$$

$$
\frac{\Gamma, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta}}{\Gamma \vdash \lambda x.e \Rightarrow \hat{\alpha} \to \hat{\beta}}\ (\textsf{Syn-MonoLam})
\qquad
\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.e \Leftarrow A \to B}\ (\textsf{Chk-PolyLam})
$$

$$
\frac{\Gamma, \hat{\alpha} \vdash [\hat{\alpha}/\alpha]A \bullet e \Rrightarrow C}{\Gamma \vdash \forall \alpha.A \bullet e \Rrightarrow C}\ (\textsf{Syn-PolyApp})
\qquad
\frac{\Gamma, \hat{\alpha}_1, \hat{\alpha}_2, \hat{\alpha} = \hat{\alpha}_1 \to \hat{\alpha}_2 \vdash e \Leftarrow \hat{\alpha}_1}{\Gamma, \hat{\alpha} \vdash \hat{\alpha} \bullet e \Rrightarrow \hat{\alpha}_2}\ (\textsf{Syn-ExstApp})
$$

$$
\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash A \bullet e \Rrightarrow C}{\Gamma \vdash e_1\ e_2 \Rightarrow C}\ (\textsf{Syn-MonoApp})
\qquad
\frac{\Gamma \vdash e \Rightarrow \forall \alpha.A}{\Gamma \vdash e[\tau] \Rightarrow [\tau/\alpha]A}\ (\textsf{Syn-TermTypeApp})
$$

Note that well-formedness of contexts, types, and kinds is implied in every typing rule.

4 References

[CGO16] Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. "System F-Omega with Equirecursive Types for Datatype-Generic Programming". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. Association for Computing Machinery, 2016, pp. 30–43. DOI: 10.1145/2837614.2837660.

[DK20] Jana Dunfield and Neelakantan R. Krishnaswami. "Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism". In: *Int'l Conf. Functional Programming*. Aug. 2020. arXiv: 1306.6032v2 [cs.PL].

[Fos12] Nate Foster. *CS 4110 – Programming Languages and Logics. Lecture #27: Recursive Types*. 2012. URL: https://www.cs.cornell.edu/courses/cs4110/2012fa/lectures/lecture27.pdf.

[Zim17] Jake Zimmerman. *System Fω and Parameterization*. Sept. 27, 2017. URL: https://blog.jez.io/system-f-param/.