

Interactive Tactic-Based Program Synthesis

Author: Carlos Fernando Amador Martínez Quintero

Supervisor: Tomáš Petříček, Ph.D.

INTRODUCTION

- In functional programming languages with strong type systems, the types constrain the structure of programs.
- In dependently-typed proof assistants such as Lean and Rocq, *tactics* (small commands) guide proof construction.
- How feasible is the usage of tactics for functional programming in general?

WHAT IS A TYPED HOLE?

$$\lambda(x : \mathbb{Z}) \mapsto (- : \mathbb{B})$$

A part of the program whose structure is still unknown but its type is (partially) known. We can use a tactic to fill this hole!

TWO THEOREMS: SOUNDNESS AND COMPLETENESS

The Soundness Theorem. We can only obtain well-typed programs by synthesizing expressions from typed holes:

Theorem (soundness of transformation). *If τ is a type such that $\text{TypeVar}(\tau) \subseteq \text{TypeVar}(\Gamma)$ and the statements $\Sigma, \Gamma \vdash e : \tau$, $\Sigma, \Gamma \vdash e \Rightarrow^* e', \theta_U$ both hold, then $\Sigma, \theta_U \Gamma \vdash e' : \theta_U \tau$.*

The Completeness Theorem. By synthesizing expressions from typed holes, we can construct any well-typed program:

Theorem (completeness of transformation). *Suppose that Σ, Γ are contexts so that $\text{UVar}(\Sigma) = \text{UVar}(\Gamma) = \emptyset$. If e^+ is a well-typed expression without any typed holes (meaning that $\Sigma, \Gamma \vdash e^+ : \tau$ for some type τ) and $\text{UVar}(\tau) = \text{UVar}(e^+) = \emptyset$, then we have that $\Sigma, \Gamma \vdash (- : \tau) \Rightarrow^* e^+, \theta_U$ for some substitution θ_U .*

RULES FOR SYNTHESIS

From a **function type**, we can synthesize a function:

$$\frac{(x : \tau_1) \notin \text{dom}(\Sigma) \cup \text{dom}(\Gamma)}{\Sigma, \Gamma \vdash (\tau_1 \rightarrow \tau_2) \rightsquigarrow \lambda(x : \tau_1) \mapsto (- : \tau_2), \emptyset} \text{(INTRO)}$$

From **any type**, we can synthesize an application of an unknown function and unknown input:

$$\frac{u \in \text{Fresh}(\Sigma, \Gamma) \setminus \text{UVar}(\tau)}{\Sigma, \Gamma \vdash \tau \rightsquigarrow (- : u \rightarrow \tau)(- : u), \emptyset} \text{(APPLY)}$$

...and many other possibilities!

GOALS

- A **mathematical formalism to describe the synthesis of programs through tactics that fill in *typed holes***.
- Prove two desirable properties of our system: *soundness* and *completeness*.
- Implement a Haskell prototype of this system.

THE LANGUAGE

Types, Expressions and Programs

$$\begin{aligned} &\text{let } (\text{add} : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}) := \\ &\lambda(x : \mathbb{Z}) \mapsto \lambda(y : \mathbb{Z}) \mapsto x + y \end{aligned}$$

Typing Judgments

$$\Sigma, \{c : \mathbb{B}, x : \mathbb{Z}\} \vdash (\text{if } c \text{ then } x \text{ else } 0) : \mathbb{Z}$$

Synthesis $\rightsquigarrow$ and Transformation $\Rightarrow$

$$\begin{aligned} \Sigma, \Gamma \vdash (a \times b \rightarrow a) \rightsquigarrow \lambda(p : a \times b) \mapsto (- : a), \emptyset \\ \Sigma, \Gamma \vdash (- : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}) \ x \ y \Rightarrow \text{mult } x \ y \end{aligned}$$

REPL IMPLEMENTED IN HASKELL

Lets you construct programs through tactics...

```
> letrec length [t] -> Int
Current hole: 0
```

```
letrec (length : [t] → ℤ) =
  (-#0 : [t] → ℤ)
```

Suggestions:

1. `length [ varLocal length ]`
2. `λ(x1 : [t]) ↦ (-#1 : ℤ) [ intro x1 ]`
3. `(snd (-#1 : ?1 × [t] → ℤ)) [ globalApply snd ]...`

...and even ranks and proposes possible tactics!



GitHub Repository for the Haskell Prototype

CONCLUSION

We formally defined a toy functional language with polymorphic types, and used it to provide a mathematical formalism to describe tactic-based program synthesis. The Haskell implementation, under the constraints of a purely functional programming language, required the usage of zippers for efficient program navigation, and occasional usage of monads.