Learning in System F

JOEY VELEZ-GINORIO, Massachusetts Institute of Technology, U.S.A.
NADA AMIN, Harvard University, U.S.A.

Program synthesis, type inhabitance, inductive programming, and theorem proving. Different names for the same problem: learning programs from data. Sometimes the programs are proofs, sometimes they're terms. Sometimes data are examples, and sometimes they're types. Yet the aim is the same. We want to construct a program which satisfies some data. We want to learn a program.

What might a programming language look like, if its programs could also be learned? We give it data, and it learns a program from it. This work shows that System F yields a simple approach for learning from types and examples. Beyond simplicity, System F gives us a guarantee on the soundness and completeness of learning. We learn correct programs, and can learn all observationally distinct programs in System F. Unlike previous works, we don't restrict what examples can be. As a result, we show how to learn arbitrary higher-order programs in System F from types and examples.

Additional Key Words and Phrases: Program Synthesis, Type Theory, Inductive Programming

1 INTRODUCTION

Imagine we're teaching you a program. Your only data is the type $nat \to nat$. It takes a natural number, and returns a natural number. Any ideas? Perhaps a program which computes...

$$f(x) = x, \qquad f(x) = x + 1, \qquad f(x) = x + 2, \qquad f(x) = x + \cdots$$

The good news is that $f(x) = x + 1$ is correct. The bad news is that the data let you learn a slew of other programs too. It doesn't constrain learning enough if we want to teach $f(x) = x + 1$. As teachers, we can provide better data.

Round 2. Imagine we're teaching you a program. But this time we give you an example of the program's behavior. Your data are the type $nat \to nat$ and an example $f(1) = 2$. It takes a natural number, and returns the successor. Any ideas? Perhaps a program which computes...

$$f(x) = x + 1, \qquad f(x) = x + 2 - 1, \qquad f(x) = x + 3 - 2, \qquad \cdots$$

The good news is that $f(x) = x + 1$ is correct. And so are all the other programs, as long as we're agnostic to some details. Types and examples impose useful constraints on learning. It's the data we use when learning in System F [Girard et al. 1989].

Existing work can learn successor from similar data [Osera 2015; Polikarpova et al. 2016]. But suppose nat is a church encoding. For some base type A, $nat \coloneqq (A \to A) \to (A \to A)$. Natural numbers are then higher-order functions. They take and return functions. Suddenly, existing work can no longer learn successor.

The difficulty is with how to handle functions in the return type. The type $nat \to nat$ returns a function, a program of type nat. To learn correct programs, you need to ensure candidates are the correct type or that they obey examples. Imagine we want to verify that our candidate program f obeys $f(1) = 2$. With the church encoding, $f(1)$ is a function, and so is 2. In other words, $f(1) = 2$ requires that we decide the equivalence of functions—which is undecidable in a Turing-complete language [Sipser et al. 2006]. Functions in the return type create this issue. There are two ways out:

Authors' addresses: Joey Velez-Ginorio, Brain & Cognitive Sciences, Massachusetts Institute of Technology, 43 Vassar St., Cambridge, Massachusetts, 02139, U.S.A., joeyv@mit.edu; Nada Amin, John A. Paulson School of Engineering and Applied Sciences, Harvard University, 29 Oxford St., Cambridge, Massachusetts, 02138, U.S.A., namin@seas.harvard.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

(1) Don't allow functions in the return type, keep Turing-completeness.
(2) Allow functions in the return type, leave Turing-completeness.

2 LEARNING FROM TYPES

3 LEARNING FROM EXAMPLES

4 IMPLEMENTATION

5 EXPERIMENTS

6 RELATED WORK

7 CONCLUSION

REFERENCES

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7.

Peter-Michael Santos Osera. 2015. Program synthesis with types. *University of Pennsylvania, Department of Computer Science. PhD Thesis.* (2015).

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 522–538.

Michael Sipser et al. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.