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

1.1 A tricky learning problem

Imagine we're teaching you a program. Your only data is the type $nat \rightarrow nat$. 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 $nat \rightarrow nat$ and an example $f(1) = 2$. It takes a natural number, and seems to return its 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, $nat := (A \rightarrow A) \rightarrow (A \rightarrow A)$. Natural numbers are then higher-order functions. They take and return functions. In this context, existing work can no longer learn successor.

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/

1.2 A way forward

The difficulty is with how to handle functions in the return type. The type $nat \rightarrow nat$ 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. To check $f(1) = 2$ requires that we decide function equality—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.

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

Route 1 is the approach of existing work. They don't allow functions in the return type, but keep an expressive Turing-complete language for learning. This can be a productive move, as many interesting programs don't return functions.

Route 2 is the approach we take. We don't impose restrictions on the types or examples we learn from. We instead sacrifice Turing-completeness. We choose a language where function equality is decidable, but still expressive enough to learn interesting programs. Our work shows that this too is a productive move, as many interesting programs return functions. This route leads us to several contributions:

- Detail how to learn arbitrary higher-order programs in System F. (Section 2 & 3)
- Prove the soundness and completeness of learning. (Section 2 & 3)
- Provide an implementation of learning, extending strong theoretical guarantees in practice. (Section 4 & 5)

2 System F

We assume you are familiar with System F, the polymorphic lambda calculus. You should know its syntax, typing, and evaluation. If you don't, we co-opt its specification in [Pierce 2002]. For a comprehensive introduction we defer the confused or rusty there.

Our focus in this section is to state the specification of System F: its syntax, typing, and evaluation. Along the way, we motivate why properties of each are advantageous for learning. Treat this section as an answer to the following question:

Why are we learning in System F?

2.1 Syntax

System F's syntax is simple. Just look at Figure 1. There aren't many syntactic forms. Whenever we state, prove, or implement things in System F we often use structural recursion on the syntax. A minimal syntax means we are succint when we state, prove, or implement those things.

While simple, the syntax is still expressive. We can encode many staples of typed functional programming: algebraic data types, inductive types, and more [Girard et al. 1989]. For example, consider this encoding of the product type:

erberberbrberb

SYNTAX

$e ::=$	TERMS:	$\tau ::=$	TYPES:
x	*variable*	$\tau_1 \rightarrow \tau_2$	*function type*
$e_1 e_2$	*application*	$\forall \alpha . \tau$	*polymorphic type*
$\lambda x{:}\tau . e$	*abstraction*	α	*type variable*
$e[\tau]$	*type application*	$\Gamma ::=$	CONTEXTS:
$\Lambda \alpha . e$	*type abstraction*	$\cdot$	*empty*
$v ::=$	VALUES:	$x{:}\tau, \Gamma$	*variable*
$\lambda x{:}\tau . e$	*abstraction*	α, Γ	*type variable*
$\Lambda \alpha . e$	*type abstraction*		

Fig. 1. Syntax in System F

TYPING $\boxed{\Gamma \vdash e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{T-VAR}) \qquad \frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda \alpha . e : \forall \alpha . \tau}\ (\text{T-TABS})$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 . e_2 : \tau_1 \rightarrow \tau_2}\ (\text{T-ABS}) \qquad \frac{\Gamma \vdash e : \forall \alpha . \tau_1}{\Gamma \vdash e[\tau_2] : [\tau_2/\alpha]\tau_1}\ (\text{T-TAPP})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ (\text{T-APP})$$

Fig. 2. Typing in System F

2.2 Typing

2.3 Evaluation

3 Learning from Types

4 Learning from Examples

5 Implementation

6 Experiments

7 Related Work

8 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).

Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

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.