# Learning in System F

Joey Velez-Ginorio
Massachusetts Institute of Technology
Cambridge, Massachusetts, U.S.A.
joeyv@mit.edu

Nada Amin
Harvard University
Cambridge, Massachusetts, U.S.A.
namin@seas.harvard.edu

## Abstract

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.

***Keywords:*** 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 $nat \to nat$. It takes a natural number, and returns a natural number. Any ideas? Perhaps a program which computes...

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

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 seems to return its successor. Any ideas? Perhaps a program which computes...

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

Authors' addresses: Joey Velez-Ginorio, Massachusetts Institute of Technology, 43 Vassar St., Cambridge, Massachusetts, 02139, U.S.A., joeyv@mit.edu; Nada Amin, 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/

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 $A$, $nat := (A \to A) \to (A \to A)$. Natural numbers are then higher-order functions. They take and return functions. In this setting, existing work can no longer learn successor.

### 1.2 A way forward

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. 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)
- Implement 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. Additionally, we provide the specification and relevant theorems in the appendix.

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

*Why learn in System F?*

### 2.1 Syntax

System F's syntax is simple. 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 [Pierce 2002]. For example, consider this encoding of products:

$$\tau_1 \times \tau_2 \coloneqq \forall\alpha.(\tau_1 \to \tau_2 \to \alpha) \to \alpha$$
$$\langle e_1, e_2 \rangle \coloneqq \Lambda\alpha.\lambda f\!:\!(\tau_1 \to \tau_2 \to \alpha).f\,e_1\,e_2$$

### 2.2 Typing

System F is safe. Its typing ensures both progress and preservation, i.e. that well-typed programs do not get stuck and that they do not change type [Pierce 2002]. When we introduce learning, we lift this safety and extend it to programs we learn. Because we use this safety in later proofs, we state the progress and preservation theorems in the appendix.

### 2.3 Evaluation

System F is strongly normalizing. All its programs terminate. As a result, we can use a simple procedure for deciding equality of programs (including functions).

1. Run both programs until they terminate.
2. Check if they share the same normal form, up to alpha-equivalence (renaming of variables).
3. If they do, they are equal. Otherwise, unequal.

For example, this decision procedure renders these programs equal:

$$\lambda x\!:\!\tau.x \quad =_\beta \quad (\lambda y\!:\!(\tau \to \tau).y)\lambda z\!:\!\tau.z$$

The decision procedure checks that two programs exist in the transitive reflexive closure of the evaluation relation. This only works because programs always terminate, a property we formally state in the appendix.

## 3 Learning from Types

We present learning as a relation between contexts $\Gamma$, programs $e$, and types $\tau$.

$$\Gamma \vdash \tau \rightsquigarrow e$$

The relation asserts that given a context $\Gamma$ and type $\tau$, you can learn program $e$.

Like typing, we define the relation with a set of inference rules. These rules confer similar benefits to typing. We can prove useful properties of learning, and the rules guide implementation.

Unlike typing, we only consider programs $e$ in normal form. We discuss later how this pays dividends in the implementation. With reference to the syntax in the appendix, we define System F programs $e$ in normal form:

$$e \coloneqq \hat{e} \mid \lambda x\!:\!\tau.e \mid \Lambda\alpha.e$$
$$\hat{e} \coloneqq x \mid \hat{e}\,e \mid \hat{e}\,\lceil\tau\rceil$$

Learning $\boxed{\Gamma \vdash \tau \rightsquigarrow e}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash \tau \rightsquigarrow x}\ \text{(L-Var)} \qquad \frac{\Gamma, \alpha \vdash \tau \rightsquigarrow e}{\Gamma \vdash \forall\alpha.\tau \rightsquigarrow \Lambda\alpha.e}\ \text{(L-TAbs)}$$

$$\frac{\Gamma, x\!:\!\tau_1 \vdash \tau_2 \rightsquigarrow e_2}{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow \lambda x\!:\!\tau_1.e_2}\ \text{(L-Abs)} \qquad \frac{\Gamma \vdash \forall\alpha.\tau_1 \rightsquigarrow \hat{e}}{\Gamma \vdash \lceil\tau_2/\alpha\rceil\tau_1 \rightsquigarrow \hat{e}\,\lceil\tau_2\rceil}\ \text{(L-TApp)}$$

$$\frac{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow \hat{e} \quad \Gamma \vdash \tau_1 \rightsquigarrow e}{\Gamma \vdash \tau_2 \rightsquigarrow \hat{e}\,e}\ \text{(L-App)}$$

**Figure 1.** Learning from types in System F

### 3.1 Learning, a relation

If you squint, you may think that learning in Figure 1 looks a lot like typing in Figure 4. The semblance isn't superficial, but instead reflects an equivalence between learning and typing which we later state. Despite this, the learning relation isn't redundant. It forms the core of an extended learning relation in the next section, where we learn from both types and examples.

(L-Var) says if $x$ is bound to type $\tau$ in the context $\Gamma$, then you can learn the program $x$ of type $\tau$.

$$\frac{x{:}\tau \in x{:}\tau}{x{:}\tau \vdash \tau \rightsquigarrow x} \text{ (L-Var)}$$

(L-Abs) says if $x$ is bound to type $\tau_1$ in the context and you can learn a program $e_2$ from type $\tau_2$, then you can learn the program $\lambda x{:}\tau_1.e_2$ from type $\tau_1 \to \tau_2$ and $x$ is removed from the context.

$$\frac{\Gamma, x{:}\tau_1 \vdash \tau_2 \to \tau_2 \rightsquigarrow \lambda y{:}\tau_2.y}{\Gamma \vdash \tau_1 \to \tau_2 \to \tau_2 \rightsquigarrow \lambda x{:}\tau_1.\lambda y{:}\tau_2.y} \text{ (L-Abs)}$$

(L-App) says that if you can learn a program $\hat{e}$ from type $\tau_1 \to \tau_2$ and a program $e$ from type $\tau_1$, then you can learn $\hat{e}\ e$ from type $\tau_2$.

$$\frac{\Gamma \vdash \tau \to \tau \rightsquigarrow f \qquad \Gamma \vdash \tau \rightsquigarrow x}{\Gamma \vdash \tau \rightsquigarrow f\ x} \text{ (L-App)}$$

(L-TAbs) says that if $\alpha$ is in the context, and you can learn a program $e$ from type $\tau$, then you can learn a program $\Lambda\alpha.e$ from type $\forall\alpha.\tau$ and $\alpha$ is removed from the context.

$$\frac{\Gamma, \alpha \vdash \alpha \to \alpha \rightsquigarrow \lambda x{:}\alpha.x}{\Gamma \vdash \forall\alpha.\alpha \to \alpha \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x} \text{ (L-TAbs)}$$

(T-TApp) says that if you can learn a program $\hat{e}$ from type $\forall\alpha.\tau_1$, then you can learn the program $\hat{e}\ \lceil\tau_2\rceil$ from type $[\tau_2/\alpha]\tau_1$.

$$\frac{\Gamma \vdash \forall\alpha.\alpha \to \alpha \rightsquigarrow f}{\Gamma \vdash \tau \to \tau \rightsquigarrow f\lceil\tau\rceil} \text{ (T-TApp)}$$

### 3.2 Metatheory

Learning is a relation. Hence we can discuss its metatheory. We care most about two properties: soundness and completeness. Soundness ensures we learn correct programs. Completeness ensures we learn all programs.

We state the relevant theorems in this section but defer proofs to the appendix. Most of the heavy lifting is done by standard proofs of type systems, like progress and preservation. Learning exploits these properties of type systems to provide similar guarantees.

**Lemma 3.1** (Soundness of Learning).
*If* $\Gamma \vdash \tau \rightsquigarrow e$ *then* $\Gamma \vdash e : \tau$

**Lemma 3.2** (Completeness of Learning).
*If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash \tau \rightsquigarrow e$

Structural induction on the learning and typing rules proves these two lemmas. And together, they directly prove the equivalence of typing and learning.

**Theorem 3.3** (Equivalence of Typing and Learning).
*If and only if* $\Gamma \vdash \tau \rightsquigarrow e$ *then* $\Gamma \vdash e : \tau$

Because of the equivalence, we can extend strong metatheoretic guarantees to learning from examples.

## 4 Learning from Examples

To learn from examples, we extend our learning relation to include examples $[\chi]$.

$$\Gamma \vdash \tau \rhd [\chi] \rightsquigarrow e$$

The relation asserts that given a context $\Gamma$, a type $\tau$, and examples $[\chi]$, you can learn program $e$.

Examples are lists of tuples with the inputs and output to a program. For example, $[\langle 1, 1\rangle]$ describes a program whose input is 1 and output is 1. If we want more than one example, we can add to the list: $[\langle 1, 1\rangle, \langle 2, 2\rangle]$. And with System F, types are also valid inputs. So $[\langle Nat, 1, 1\rangle, \langle Nat, 2, 2\rangle]$ describes a polymorphic program instantiated at type $Nat$ whose input is 1 and output is 1. In general, an example takes the form

$$\chi ::= \langle e, \chi\rangle \mid \langle \tau, \chi\rangle \mid \langle e, Nil\rangle$$

Importantly, the syntax doesn't restrict what can be an input or output. Any program $e$ or type $\tau$ can be an input. Likewise, any program $e$ can be an output. We can describe any input-output relationship in the language. Note that we use the following short-hand notation for examples: $\langle \tau, \langle e_1, \langle e_2, Nil\rangle\rangle\rangle \equiv \langle \tau, e_1, e_2\rangle$.

### 4.1 Learning, a relation

Unlike the previous learning relation, Figure 2 looks a bit foreign. Nevertheless, the intuition is simple. We demonstrate with learning identity from examples. Without loss of generality, assume $Nat$ as a base type, and natural numbers.

$$\cdot \vdash Nat \to Nat \rhd [\langle 1, 1\rangle, \langle 2, 2\rangle] \rightsquigarrow \blacksquare$$

Examples describe possible worlds, one where our program's input is 1 and the other where our program's input is 2. Throughout learning we need a way to keep track of these distinct worlds. So our first step is always to duplicate $\blacksquare$, so that we have one per example.

$$\cdot \vdash list\ Nat \to Nat \rhd [\langle 1, 1\rangle, \langle 2, 2\rangle] \rightsquigarrow [\blacksquare, \blacksquare]$$

Now we can constrain each world, by applying them to their respective inputs. Because an example is a tuple, we extract components using the left and right projections $\pi_1$ and $\pi_2$.

$$\cdot \vdash list\ Nat \rhd [\langle 1\rangle, \langle 2\rangle] \rightsquigarrow [(\blacksquare)1, (\blacksquare)2]$$

LEARNING $\boxed{\Gamma \vdash \tau \rhd [\chi] \rightsquigarrow e}$

$$\frac{\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow [\texttt{let}\ (\cdot)\ \texttt{in}\ e_1, \ldots, \texttt{let}\ (\cdot)\ \texttt{in}\ e_n] \qquad \bigwedge_{i=1}^{n} e_i =_\beta e_n}{\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e} \text{ (L-WRLD)}$$

$$\frac{\Gamma \vdash \tau \rightsquigarrow e \qquad \bigwedge_{i=1}^{n}(\Gamma \vdash \texttt{let}\ (b_i)\ \texttt{in}\ e : \tau) \qquad \bigwedge_{i=1}^{n}(\texttt{let}\ (b_i)\ \texttt{in}\ e =_\beta \chi_i)}{\Gamma \vdash \tau \rhd [\langle \chi_1 \rangle, \ldots, \langle \chi_n \rangle] \rightsquigarrow [\texttt{let}\ (b_1)\ \texttt{in}\ e, \ldots, \texttt{let}\ (b_n)\ \texttt{in}\ e]} \text{ (L-BASE)}$$

$$\frac{\Gamma, x{:}\tau_a \vdash \tau_b \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow [\texttt{let}\ (b_1, x{:}\tau_a = e_1)\ \texttt{in}\ e_1, \ldots, \texttt{let}\ (b_n, x{:}\tau_a = e_n)\ \texttt{in}\ e_n]}{\Gamma \vdash \tau_a \rightarrow \tau_b \rhd [\langle e_1, \chi_1 \rangle, \ldots, \langle e_n, \chi_n \rangle] \rightsquigarrow [\texttt{let}\ (b_1)\ \texttt{in}\ \lambda x{:}\tau_a.e_1, \ldots, \texttt{let}\ (b_n)\ \texttt{in}\ \lambda x{:}\tau_a.e_n]} \text{ (L-EABS)}$$

$$\frac{\Gamma, \alpha \vdash \tau_a \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow [\texttt{let}\ (b_1, \alpha = \tau_b)\ \texttt{in}\ e_1, \ldots, \texttt{let}\ (b_n, \alpha = \tau_b)\ \texttt{in}\ e_n]}{\Gamma \vdash \forall \alpha.\tau_a \rhd [\langle \tau_b, \chi_1 \rangle, \ldots, \langle \tau_b, \chi_n \rangle] \rightsquigarrow [\texttt{let}\ (b_1)\ \texttt{in}\ \Lambda\alpha.e_1, \ldots, \texttt{let}\ (b_n)\ \texttt{in}\ \Lambda\alpha.e_n]} \text{ (L-ETABS)}$$

**Figure 2.** Learning from examples in System F

Since ■ is applied to an argument, we know it must be an abstraction. Hence we can rewrite each ■.

$$\cdot \vdash list\ Nat \rhd [\langle 1 \rangle, \langle 2 \rangle] \rightsquigarrow [(\lambda x{:}Nat.\blacksquare)1, (\lambda x{:}Nat.\blacksquare)2]$$

Now that we're out of inputs, the problem becomes how to generate a program for ■ such that each world satisfies their respective outputs.

$$\cdot \vdash Nat \rightsquigarrow (\lambda x{:}Nat.\blacksquare)1$$

$$(\lambda x{:}Nat.\blacksquare)1 =_\beta 1 \ \wedge \ (\lambda x{:}Nat.\blacksquare)2 =_\beta 2$$

# 5 Implementation

# 6 Experiments

# 7 Related Work

## 7.1 Type-driven synthesis

# 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.

# 9 Appendix

## 9.1 Specification of System F

**Theorem 9.1** (Progress in Typing).
*If $e$ is a closed, well-typed program, then either $e$ is a value or else there is some program $e'$ such that $e \to_\beta e'$.*

**Theorem 9.2** (Preservation in Typing).
*If $\Gamma \vdash e : \tau$ and $e \to_\beta e'$, then $\Gamma \vdash e' : \tau$.*

**Theorem 9.3** (Normalization in Evaluation).
*Well-typed programs in System F always evaluate to a value, to a normal form.*

Syntax

| | | | | | |
|---|---|---|---|---|---|
| $e ::=$ | | terms: | $\tau ::=$ | | types: |
| | $x$ | *variable* | | $\tau_1 \to \tau_2$ | *function type* |
| | $e_1 e_2$ | *application* | | $\forall \alpha.\tau$ | *polymorphic type* |
| | $\lambda x{:}\tau.e$ | *abstraction* | | $\alpha$ | *type variable* |
| | $e\lceil\tau\rceil$ | *type application* | | | |
| | $\Lambda \alpha.e$ | *type abstraction* | $\Gamma ::=$ | | contexts: |
| | | | | $\cdot$ | *empty* |
| $v ::=$ | | values: | | $x{:}\tau, \Gamma$ | *variable* |
| | $\lambda x{:}\tau.e$ | *abstraction* | | $\alpha, \Gamma$ | *type variable* |
| | $\Lambda \alpha.e$ | *type abstraction* | | | |

**Figure 3.** Syntax in System F

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

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

$$\frac{\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e_2 : \tau_1 \to \tau_2}\ \text{(T-Abs)} \qquad \frac{\Gamma \vdash e : \forall \alpha.\tau_1}{\Gamma \vdash e\lceil\tau_2\rceil : [\tau_2/\alpha]\tau_1}\ \text{(T-TApp)}$$

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

**Figure 4.** Typing in System F

EVALUATING $\boxed{e \rightarrow_\beta e'}$

$$\frac{e_1 \rightarrow_\beta e_1'}{e_1 e_2 \rightarrow_\beta e_1' e_2} \text{ (E-App1)} \qquad \frac{e \rightarrow_\beta e'}{e\lceil\tau\rceil \rightarrow_\beta e'\lceil\tau\rceil} \text{ (E-TApp)}$$

$$\frac{e_2 \rightarrow_\beta e_2'}{e_1 e_2 \rightarrow_\beta e_1 e_2'} \text{ (E-App2)} \qquad (\Lambda\alpha.\lambda x{:}\alpha.e)\lceil\tau\rceil \rightarrow_\beta (\lambda x{:}\alpha.e)[\tau/\alpha] \text{ (E-TSub)}$$

$$(\lambda x{:}\tau.e)v \rightarrow_\beta e[v/x] \text{ (E-Sub)}$$

**Figure 5.** Evaluating in System F