Learning in System F

Joey Velez-Ginorio

August 18, 2019

Abstract

We show that in addition to typing and generation, learning is natural in System F. That is, we can learn programs in System F without adding anything to the language. After briefly discussing System F and generation, we formalize learning as a relation and prove that certain programs can be learned in System F. We end on the conjecture that *any* program can be learned in System F (to be proved, hopefully soon).

1 System F

System F is an extension of the simply typed lambda calculus, discovered independently by both Jean-Yves Girard and John C. Reynolds in the early 1970's. It imbues the simply typed lambda calculus with polymorphism, the ability to abstract over types.

Polymorphism lets the same program work with several types. For instance, consider identity, which returns its input. Without polymorphism, we need a new identity for each type we deal with, like natural numbers or booleans. With polymorphism it's possible to build a generic identity which works over any type—a feat impossible in the simply typed lambda calculus.

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*	α	*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 1: Syntax of System F

Yet generic identity only hints at System F's expressive power. Polymorphism also lets you encode many useful types within System F. As opposed to introducing complex types explicitly, System F naturally encodes types like product, sum, and list. And with these types, we will see that System F has all it needs to learn programs from examples.

2 Generating in System F

You can't learn a program if you can't generate it. And if we're interested in learning any program in System F, we ought to ensure we can generate it. Thankfully, from System F's typing relation (Figure 2) we can derive an equivalent generating relation (Figure 3). That is, every well-typed program can be generated—and generated programs are sound with respect to the types they were generated from.

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

"Given a context Γ and type τ, I can generate program e."

From theorems 2.1 and 2.2, it follows that generating is equivalent to typing.

Theorem 2.1 (Completeness of Generating).
If $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash \tau \rightsquigarrow e$

Proof. Induction on the generating rules. □

Theorem 2.2 (Soundness of Generating).
If $\Gamma \vdash \tau \rightsquigarrow e$ *then* $\Gamma \vdash e : \tau$

Proof. Induction on the typing rules. □

Versions of the generating relation are the crux of recent type driven work in program synthesis and automated theorem proving, e.g. MYTH, Idris, Synquid. The types are typically more expressive, but the aim is the same. Use types to guide generation of programs, proofs.

Moreover, generating is natural in System F. It requires no additional machinery from System F. In some sense, this is unsurprising. Theorems 2.1 and 2.2 show that typing and generating are equivalent, and typing is certainly natural in System F by definition.

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

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

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

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

Figure 2: Typing in System F

3 Learning

However, what's surprising is that we can amend the generating relation to learn programs from examples. And without adding anything to System F. In other words,

Learning is natural in System F.

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

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

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

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

Figure 3: Generating in System F

This insight escapes recent work in program synthesis, which add machinery to languages in order to support learning. This can be convenient, and even necessary, depending on the class of programs you want to learn. But what we will see is that this machinery isn't necessary for a large class of programs, those expressible in System F.

The generating relation tells us that with a context and type, we can generate a program. But without more expressive types, or others ways of specifying what we want, the generating relation doesn't guarantee we generate a program we actually want. For example, suppose I want a program which takes a natural number and returns it, IDENTITY. Its type is $nat \to nat$. But so is a program which doubles a natural number. Hence we need better ways to guide generation to the program we want.

Enter examples. Suppose again I want to generate IDENTITY. But this time in addition to its type, I provide examples of its behavior: 1 returns 1 and 2 returns 2. Now clearly this program isn't doubling its input. The trick then is how to define a learning relation which guides generation using examples.

Informally, let's introduce the learning relation and walk through how we would use it.

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

"Given a context Γ, type τ, and examples χ, I can learn program e."

With this relation, we can ask whether IDENTITY is learnable given a context, type, and examples:

$$\cdot \vdash nat \to nat \rhd \langle\langle 1, 1\rangle, \langle 2, 2\rangle\rangle \rightsquigarrow \blacksquare$$

Examples are stored as tuples. They 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 there is one per example.

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

Let's refine these worlds, by applying them to their respective inputs. We extract the inputs from each example tuple.

$$1{:}nat, 2{:}nat \vdash list\,nat \rhd \langle\langle 1\rangle, \langle 2\rangle\rangle \rightsquigarrow [(\blacksquare)1, (\blacksquare)2]$$

Because $\blacksquare$ is applied to an argument, we know it must be an abstraction. Hence, we can also claim:

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

Now that we've ran out of inputs in our examples, the problem becomes how to generate a program which satisfy the outputs left in the example tuples:

$$1{:}nat, 2{:}nat \vdash list\, nat \rightsquigarrow [(\lambda x{:}nat.\blacksquare)1, (\lambda x{:}nat.\blacksquare)2]$$

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

Given the constraints on well-typed terms, it's easy to find x to fill the body of the abstraction. This will become clear in the formal proof to follow.

$$1{:}nat, 2{:}nat \vdash list\, nat \rightsquigarrow [(\lambda x{:}nat.x)1, (\lambda x{:}nat.x)2]$$

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

Having satisfied the outputs from our examples, we've informally shown IDENTITY $\equiv$ $\lambda x{:}nat.x$ is learnable in System F. And all the machinery comes from types and operators we can encode in System F: list and product types along with their constructors and deconstructors.

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

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

$$\frac{\Gamma, \bigcup_{i=1}^{n} \chi_i{:}\tau \vdash list\,\tau \rightsquigarrow [e_1, \ldots, e_n] \qquad \bigwedge_{i=1}^{n} e_i =_\beta \chi_i}{\Gamma \vdash list\,\tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow [e_1, \ldots, e_n]} \text{ (L-GEN)}$$

$$\frac{\Gamma, \bigcup_{i=1}^{n} \pi_1(\chi_i){:}\tau_a \vdash list\,\tau_b \rhd [\pi_2(\chi_1), \ldots, \pi_2(\chi_n)] \rightsquigarrow [e_1\pi_1(\chi_1), \ldots, e_n\pi_1(\chi_n)]}{\Gamma \vdash list\,\tau_a \to \tau_b \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow [e_1, \ldots, e_n]} \text{ (L-ABS)}$$

$$\frac{\Gamma, \alpha \vdash list\,\tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow [e_1\lceil\alpha\rceil, \ldots, e_n\lceil\alpha\rceil]}{\Gamma \vdash list\,\forall\alpha.\tau \rhd [\Lambda\alpha.\chi_1, \ldots, \Lambda\alpha.\chi_n] \rightsquigarrow [e_1, \ldots, e_n]} \text{ (L-TABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash list\,\tau_a \to \tau_c \rhd [\chi_1, \ldots, \chi_j] \rightsquigarrow [e_1, \ldots, e_j] \\ \Gamma \vdash list\,\tau_b \to \tau_c \rhd [\chi_1, \ldots, \chi_k] \rightsquigarrow [e_1, \ldots, e_k]\end{array}}{\Gamma \vdash list\,\tau_{a+b} \to \tau_c \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow [e_1, \ldots, e_n]} \text{ (L-SUM)}$$

Figure 4: Learning in System F

Before formally proving IDENTITY $\equiv \lambda x{:}nat.x$ is learnable in System F, let's discuss the learning relation in Figure 4. Note that we assume the typical encodings for lists and tuples in System F. Brackets, [], denote lists. π_1 and π_2 denote the first and second projections of a tuple.

(L-WRLD) creates n worlds from n examples at the start of learning. These n worlds allow us to keep track of constraints provided by the n examples.

(L-GEN) generates a program which satisfies all example outputs. This is done after passing the example inputs to their respective worlds.

(L-Abs) applies inputs to their respective worlds, by extracting the input from the example tuples.

(L-TAbs) applies polymorphic inputs to their respective worlds, by extracting the polymorphic input from the example tuples.

(L-Sum) distributes examples according to the types comprising a sum type. So if *bool* is a sum type, equivalent to $true + false$. Then all examples whose next input is type *true* will be sent off to a new sub-problem, the same with $false$. This helps to simplify the learning process, because each sub-problem only needs to worry about satisfying a smaller number of examples.

The next few proofs illustrate how we use these rules to learn programs. **Note that we assume typing (equivalently, generating) rules for product and list types as presented in TAPL (sections 11.6 and 11.2). Their derivations will be shown in the thesis).**

Theorem 3.1 (Learning identity).
$\cdot \vdash nat \to nat \rhd \langle\langle 1,1\rangle, \langle 2,2\rangle\rangle \leadsto \lambda x{:}nat.x$

Proof.

$$\dfrac{\dfrac{\dfrac{x{:}nat \in 1{:}nat, 2{:}nat, x{:}nat}{1{:}nat, 2{:}nat, x{:}nat \vdash nat \to nat \leadsto x}\,(\textsc{G-Var})}{1{:}nat, 2{:}nat \vdash nat \leadsto \lambda x{:}nat.x}\,(\textsc{G-Abs}) \qquad \dfrac{1{:}nat \in 1{:}nat, 2{:}nat}{1{:}nat, 2{:}nat \vdash nat \leadsto 1}\,(\textsc{G-Var})}{1{:}nat, 2{:}nat \vdash nat \leadsto (\lambda x{:}nat.x)1}\,(\textsc{G-App})$$

$$\dfrac{\dfrac{\dfrac{x{:}nat \in x{:}nat}{1{:}nat, 2{:}nat, x{:}nat \vdash nat \to nat \leadsto x}\,(\textsc{G-Var})}{1{:}nat, 2{:}nat \vdash nat \leadsto \lambda x{:}nat.x}\,(\textsc{G-Abs}) \qquad \dfrac{2{:}nat \in 2{:}nat}{1{:}nat, 2{:}nat \vdash nat \leadsto 2}\,(\textsc{G-Var})}{1{:}nat, 2{:}nat \vdash nat \leadsto (\lambda x{:}nat.x)2}\,(\textsc{G-App})$$

$$\dfrac{\dfrac{\cdots}{1{:}nat, 2{:}nat \vdash nat \leadsto (\lambda x{:}nat.x)2} \qquad 1{:}nat, 2{:}nat \vdash list\,nat \leadsto [\,]}{1{:}nat, 2{:}nat \vdash list\,nat \leadsto [(\lambda x{:}nat.x)2]}\,(\textsc{G-Cons})$$

$$\dfrac{\dfrac{\cdots}{1{:}nat, 2{:}nat \vdash nat \leadsto (\lambda x{:}nat.x)1} \qquad \dfrac{\cdots}{1{:}nat, 2{:}nat \vdash list\,nat \leadsto [(\lambda x{:}nat.x)2]}}{1{:}nat, 2{:}nat \vdash list\,nat \leadsto [(\lambda x{:}nat.x)1, (\lambda x{:}nat.x)2]}\,(\textsc{G-Cons})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{1{:}nat, 2{:}nat \vdash list\,nat \leadsto [(\lambda x{:}nat.x)1, (\lambda x{:}nat.x)2]} \qquad \begin{array}{c}(\lambda x{:}nat.x)1 =_\beta 1\\ (\lambda x{:}nat.x)2 =_\beta 2\end{array}}{1{:}nat, 2{:}nat \vdash list\,nat \rhd \langle\langle 1\rangle, \langle 2\rangle\rangle \leadsto [(\lambda x{:}nat.x)1, (\lambda x{:}nat.x)2]}\,(\textsc{L-Gen})}{\cdot \vdash list\,nat \to nat \rhd \langle\langle 1,1\rangle, \langle 2,2\rangle\rangle \leadsto [\lambda x{:}nat.x, \lambda x{:}nat.x]}\,(\textsc{L-Abs})}{\cdot \vdash nat \to nat \rhd \langle\langle 1,1\rangle, \langle 2,2\rangle\rangle \leadsto \lambda x{:}nat.x}\,(\textsc{L-Wrld})$$

□

Theorem 3.2 (Learning polymorphic identity).
$\cdot \vdash \forall\alpha.\alpha \to \alpha \rhd \langle\Lambda\alpha.\langle z, z\rangle\rangle \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x$

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{x{:}nat \in \alpha, z{:}\alpha, x{:}\alpha}{\alpha, z{:}\alpha, x{:}\alpha \vdash \alpha \rightsquigarrow x}\,(\textsc{G-Var})}{\alpha, z{:}\alpha \vdash \alpha \to \alpha \rightsquigarrow \lambda x{:}\alpha.x}\,(\textsc{G-Abs})}{\alpha, z{:}\alpha \vdash \forall\alpha.\alpha \to \alpha \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x}\,(\textsc{G-TAbs}) \qquad \alpha, z{:}\alpha \vdash \alpha \rightsquigarrow \alpha}{\alpha, z{:}\alpha \vdash \alpha \to \alpha \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil}\,(\textsc{G-App})$$

$$\dfrac{\dfrac{\dots}{\alpha, z{:}\alpha \vdash \alpha \to \alpha \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil} \qquad \dfrac{z{:}\alpha \in \alpha, z{:}\alpha}{\alpha, z{:}\alpha \vdash \alpha \rightsquigarrow z}\,(\textsc{G-Var})}{\alpha, z{:}\alpha \vdash \alpha \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z}\,(\textsc{G-App})$$

$$\dfrac{\dfrac{\dots}{\alpha, z{:}\alpha \vdash \alpha \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z} \qquad \alpha, z{:}\alpha \vdash \mathit{list}\,\alpha \rightsquigarrow [\,]}{\alpha, z{:}\alpha \vdash \mathit{list}\,\alpha \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z]}\,(\textsc{G-Cons})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\alpha, z{:}\alpha \vdash \mathit{list}\,\alpha \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z]} \qquad (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z =_\beta z}{\alpha, z{:}\alpha \vdash \mathit{list}\,\alpha \rhd \langle\langle z\rangle\rangle \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z]}\,(\textsc{L-Gen})}{\alpha \vdash \mathit{list}\,\alpha \to \alpha \rhd \langle\langle z, z\rangle\rangle \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil]}\,(\textsc{L-Abs})}{\cdot \vdash \mathit{list}\,\forall\alpha.\alpha \to \alpha \rhd \langle\Lambda\alpha.\langle z, z\rangle\rangle \rightsquigarrow [\Lambda\alpha.\lambda x{:}\alpha.x]}\,(\textsc{L-TAbs})}{\cdot \vdash \forall\alpha.\alpha \to \alpha \rhd \langle\Lambda\alpha.\langle z, z\rangle\rangle \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x}\,(\textsc{L-Wrld})$$

□

Theorem 3.3 (Learning not).
$\cdot \vdash bool \to bool \rhd \langle\langle true, false\rangle, \langle false, true\rangle\rangle \rightsquigarrow e$, where $e \equiv \lambda x{:}bool.case\ x\ of\ inl(true) \mapsto false \mid inr(false) \mapsto true$, $bool \equiv t + f$, $true : t$, and $false : f$.
Proof isn't complete, but shows how we distribute examples when encountering a sum type. The part omitted has been showcased in the other proofs.

Proof.

$$\dfrac{\dfrac{\dfrac{\dots \qquad \dots}{\begin{array}{c}\cdot \vdash \mathit{list}\,t \to bool \rhd \langle\langle true, false\rangle\rangle \rightsquigarrow [\lambda x{:}t.false] \\ \cdot \vdash \mathit{list}\,f \to bool \rhd \langle\langle false, true\rangle\rangle \rightsquigarrow [\lambda x{:}f.true]\end{array}}}{\cdot \vdash \mathit{list}\,bool \to bool \rhd \langle\langle true, false\rangle, \langle false, true\rangle\rangle \rightsquigarrow [e, e]}\,(\textsc{L-Sum})}{\cdot \vdash bool \to bool \rhd \langle\langle true, false\rangle, \langle false, true\rangle\rangle \rightsquigarrow e}\,(\textsc{L-Wrld})$$

□