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 an approach for learning from types and examples. Beyond simplicity, System F's expressivity gives us the potential for two key ideas. (1) Learning as a first-class activity. That is, learning can happen anywhere in a program. (2) Bootstrapping its learning to higher-level languages—much the same way we bootstrap the capabilities of a functional core to modern, high-level functional languages. This work takes some first steps in that direction of the design space.

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 $nat \rightarrow nat$. It takes a natural number, and returns a natural number. Any ideas? Perhaps a program which computes...

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

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 $f(1) = 2$.

```
data Nat = Zero
         | S Nat

succ :: Nat -> Nat
succ n = [<Zero, S Zero>,<S Zero, S (S Zero)>]

main :: Nat -> Nat
main = succ
```

```
succ :: Nat -> Nat
succ n = case n[Nat] of
           Zero -> S Zero
           S n2 -> S (S n2)
```

```
lam x: (R.(Unit -> R) -> (R -> R) -> R).
  forall X.
    lam c0: (Unit -> R).
      (lam c1: (R -> R).
        (c1 (((x R) c0) c1))
```

Figure 1. An example learning problem, the learned solution in pretty print, and the learned solution in System F.

It takes a natural number, and seems to return its successor. Any ideas? Perhaps a program which computes...

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

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 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 $nat \rightarrow nat$ 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.

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/

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 (Figure 2). 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:

$$\begin{aligned} \tau_1 \times \tau_2 &:= \forall \alpha.(\tau_1 \to \tau_2 \to \alpha) \to \alpha \\ \langle e_1, e_2 \rangle &:= \Lambda \alpha. \lambda f : (\tau_1 \to \tau_2 \to \alpha). f e_1 e_2 \end{aligned}$$

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. We omit the presentation of typing, as it's similar equivalent to a learning relation presented in the next section.

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.

Syntax

$e ::=$		TERMS:
	x	*variable*
	$e_1 e_2$	*application*
	$\lambda x{:}\tau.e$	*abstraction*
	$e\lceil\tau\rceil$	*type application*
	$\Lambda\alpha.e$	*type abstraction*
$v ::=$		VALUES:
	$\lambda x{:}\tau.e$	*abstraction*
	$\Lambda\alpha.e$	*type abstraction*

$\tau ::=$		TYPES:
	$\tau_1 \to \tau_2$	*function type*
	$\forall\alpha.\tau$	*polymorphic type*
	α	*type variable*
$\Gamma ::=$		CONTEXTS:
	$\cdot$	*empty*
	$x{:}\tau, \Gamma$	*variable*
	α, Γ	*type variable*

Figure 2. Syntax in System F

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 in System F [Girard et al. 1989].

3 Learning from Types

We present learning as a relation between contexts Γ, programs e, and types τ.

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

The relation asserts that given a context Γ and type τ, 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:

$$\begin{aligned} e &::= \hat{e} \mid \lambda x\!:\!\tau.e \mid \Lambda\alpha.e \\ \hat{e} &::= x \mid \hat{e}\ e \mid \hat{e}\lceil\tau\rceil \end{aligned}$$

3.1 Learning, a relation

If you squint, you may think that learning in Figure 4 looks a lot like typing in Figure 3. 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 τ in the context Γ, then you can learn the program x of type τ.

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

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

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

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

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

Figure 3. Typing in System F

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

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

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

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

Figure 4. Learning from types in System F

(L-Abs) says if x is bound to type τ_1 in the context and you can learn a program e_2 from type τ_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 τ_1, then you can learn $\hat{e}\ e$ from type τ_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 α is in the context, and you can learn a program e from type τ, then you can learn a program $\Lambda\alpha.e$ from type $\forall\alpha.\tau$ and α 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. Most of the heavy lifting for proving these properties 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, for which learning from types is a key step.

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 Γ, a type τ, 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

$$\chi \coloneqq \langle e, \chi\rangle \mid \langle \tau, \chi\rangle \mid \langle e, Nil\rangle$$

Importantly, the syntax doesn't restrict what can be an input or output. Any program e or type τ 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 5 looks a bit foreign. Nevertheless, the intuition is simple. We demonstrate with learning polymorphic identity from examples. Without loss of generality, assume a base type Nat and natural numbers.

$$\cdot \vdash \forall\alpha.\alpha \to \alpha \rhd [\langle Nat, 1, 1\rangle] \rightsquigarrow \blacksquare$$

Examples describe possible worlds. $\langle Nat, 1, 1\rangle$ is a world where $\blacksquare$'s input is Nat and 1, with an output of 1. 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. We also introduce empty let bindings to record constraints on variables at later steps. (L-Wrld) in Figure 5 formalizes this step.

$$\cdot \vdash \alpha.\alpha \to \alpha \rhd [\langle Nat, 1, 1\rangle] \rightsquigarrow [\texttt{let}\ (\cdot)\ \mathit{in}\ \blacksquare]$$

Now, our target type is $\alpha.\alpha \to \alpha$. So we know $\blacksquare$ can bind a variable α to some type. And since we have inputs in $[\langle Nat, 1, 1\rangle]$, we know what that type is. (L-ETAbs) in Figure 5 formalizes this step.

$$\alpha \vdash \alpha \to \alpha \rhd [\langle 1, 1\rangle] \rightsquigarrow [\texttt{let}\ (\alpha = Nat)\ \mathit{in}\ \blacksquare]$$

Our target type is now $\alpha \to \alpha$. So we know $\blacksquare$ can bind a variable $x{:}\alpha$ to some program. And since we have inputs in $[\langle 1, 1\rangle]$, we know what that program is. (L-EAbs) in Figure 5 formalizes this step.

$$\alpha, x{:}\alpha \vdash \alpha \rhd [\langle 1\rangle] \rightsquigarrow [\texttt{let}\ (\alpha = Nat, x{:}\alpha = 1)\ \mathit{in}\ \blacksquare]$$

There aren't any inputs left to add bindings to our possible worlds. Therefore, we invoke the relation for learning from types (Figure 4) to generate a candidate for $\blacksquare$. Then we check

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 5. Learning from examples in System F

that this candidate evaluates to the correct output in each possible world.

$$\alpha, x{:}\alpha \vdash \alpha \rightsquigarrow x \qquad \texttt{let}\ (\alpha = Nat, x{:}\alpha = 1)\ in\ x =_\beta 1$$

With our examples satisfied, we can trivially extract the program $\Lambda\alpha.\lambda x{:}\alpha.x$ from the nested `let` expression. Note that these expressions are merely a syntactic sugaring of System F programs.

$$\begin{aligned}
\texttt{let}\ (\cdot)\ in\ t &\equiv t \\
\texttt{let}\ (x{:}\tau = s)\ in\ t &\equiv (\lambda x{:}\tau.t)s \\
\texttt{let}\ (\alpha = \tau)\ in\ t &\equiv (\Lambda\alpha.t)\lceil\tau\rceil \\
\texttt{let}\ (x{:}\tau = t_1, bs)\ in\ t_2 &\equiv \texttt{let}\ (x{:}\tau = t_1)\ in\ \texttt{let}\ (bs)\ in\ t_2 \\
\text{where } b &::= \ \cdot \ \mid\ x{:}\tau = e, b\ \mid\ \alpha = \tau, b
\end{aligned}$$

4.2 Metatheory

As before, we care most about two properties: soundness and completeness. Soundness ensures we learn correct programs from examples. Completeness ensures we learn all programs from examples.

We state the relevant theorems in this section. This time, there is no equivalence to typing. As a result, the proofs are more distinct. We omit the proofs here, but they are not controversial. Completeness and soundness proofs for similar relations exist in [Osera 2015].

Lemma 4.1 (Soundness of Learning).
If $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$ *then* $\Gamma \vdash e : \tau$

Lemma 4.2 (Completeness of Learning).
If $\Gamma \vdash e : \tau$ *and there exist some* $[\chi_1, \ldots, \chi_n]$ *which satisfies* e*, then* $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$*.*

Learning from examples maintains strong metatheoretic properties. As a result, it forms the basis for a powerful theory for learning programs from examples. However, the relation is non-deterministic, and does not directly translate to an algorithm. Nevertheless, it does provide a blueprint for what that algorithm looks like—as we saw when learning polymorphic identity. In the next sections we explore the strength of this blueprint by implementation.

5 Implementation

Throughout this section we discuss the transition from theory to practice. In doing so, we answer two questions:

- Does our learning relation yield an algorithm for learning programs from examples? The answer is yes.
- Can we lift learning in System F to learn in higher-level languages? The answer is also yes, but complicated.

5.1 Language: System F + Sugar

Our implementation is System F with sugared Haskell-like syntax for features of modern functional languages: datatype declarations, function declarations, case analysis, and let bindings. The only new syntactic form we introduce is for specifying learning problems. Otherwise, we desugar, typecheck, learn, and run programs in System F (and in that order).

We use standard methods for desugaring these features to System F. Algebraic data types and case analysis are the most involved. Algebraic data are encoded as folds, and case analysis is syntactic sugar for providing the requisite functions for the fold [Girard et al. 1989; Jansen 2013]. As a result, our case syntax deviates slightly from what you may be familiar with. We illustrate with example code from our implemented language.

```
data Bool = True
          | False

data Nat = Zero
         | S Nat

isZero :: Nat -> Bool
isZero n = case n[Bool] of
             Zero -> True
             S n2 -> False
```

There are two key distinctions from the typical case syntax. The type application on the case argument, which specifies the return type of the case analysis. And the lack of recursive function call in the case analysis. For inductive data like natural numbers, this recursive call is typical. Due to our encoding scheme, it's not necessary to recursively call the function. Instead, case analysis is syntactic sugar for defining the functions we use to fold over algebraic data. Under the hood, that case analysis looks like the following code snippet.

```
lam n:Nat.n[Bool]
  (lam _:Unit.True)
  (lam n2:Bool.False)
```

Learning specifications are first-class entities in the language, so they can appear anywhere a term can. If we encounter a learning specification during evaluation, we learn the program and then continue evaluation. In practice, we use a special declaration for learning specifications, since writing examples inside of other programs can quickly obfuscate code. The declaration is a type signature followed by a list of examples.

```
data Bool = True
          | False

data Nat = Zero
         | S Nat

isZero :: Nat -> Bool
isZero n = case n[Bool] of
             Zero -> True
             S n2 -> False

and :: Bool -> Bool -> Bool
and = [<True,True,True>,
       <True,False,False>,
       <False,True,False>,
       <False,False,False>]

bothZero :: Nat -> Nat -> Nat
bothZero = [<Zero,Zero,True>,
            <S Zero, S Zero, False>]
```

When using these declarations for learning, the code above the declaration sets the context for learning. For example, when learning and, the user-defined helper function isZero will be in context, and so will constructors for Bool and Nat. Interestingly, once we learn and it becomes part of the learning context for bothZero. This means we can set up a curriculum (a sequence) of learning problems. In practice, we find that setting up a curriculum leads to far quicker synthesis. For instance, the program for bothZero is much simpler if you already know and. Therefore it benefits to set up a curriculum, as we did, where you learn and first.

5.2 Learning from examples

Learning from examples has two stages. The first stage collects constraints from your examples. The second stage learns a term from the type which satisfies those constraints.

Our implementation is a faithful rendition of the first stage, captured by rules L-Wrld, L-EAbs, and L-ETAbs in Figure 5. The algorithm for learning only spends a marginal amount of time at this stage, and follows closely to the informal description given in section 4.1.

It's worth noting that the formal rules here are also fairly simple. Early iterations of this work sought out to find a minimal basis with which to express similar ideas in the type-directed synthesis literature [Osera 2015]. We pare down the incidental details in this work, and reformulate similar mechanisms in concise notation. For this stage in particular, this lends itself towards a near direct translation to our implementation.

Lastly, a key distinction in this work is that examples describe any input-output relationship in the language. Most related works impose restrictions on examples that break this property. As a result, learning isn't a first class activity in the language. Learning can't happen in arbitrary spots in a program, because you can't have functions in the output.

5.3 Learning from types

The second stage is learning from types. We have constraints to satify, and now need to search for a program to satisfy those constraints. The algorithm for learning spends almost all its time at this stage. This is unsurprising, given that search can be demanding. When learning non-trivial programs, it's always demanding.

The translation from the rules in Figure 4 to an algorithm is less direct than with learning from examples. So we list a set of design principles not inherent to the relation, but which are crucial in the design of its implicit algorithm.

- Enumerate programs by increasing size, where size is the number of nodes in the program's abstract syntax tree.

- Enumerate programs in normal form. An insightful analysis from [Osera 2015] describes how this dramatically reduces the space of programs.
- When instantiating a polymorphic type, only do so from types in the context.

These are the crux of the design commitments we make which deviate from the relation in Figure 4. For the first two points, [Osera 2015] is a treatise for implementing relations like Figure 4. For a more comprehensive tutorial on those design commitments we defer the interested there. Instead, we focus on the last point.

Unrestricted polymorphism is profoundly expressive [Girard et al. 1989]. And this makes learning hard. Because if you enrich your language with polymorphism, suddenly there are many more programs which inhabit a particular type. The last commitment is a way of taming the combinatorial explosion inherent to polymorphism. Similar to type inference, learning is only possible in System F if we make some concessions. And this is one possible concession that results in huge improvements in performance on programs which manipulate algebraic data.

The rules (L-App) and (L-TApp) are what trigger the combinatorial explosion. In (L-App), if I'm trying to learn a program for type τ_2 then one possibility is an application of two programs. To find that application, I have to learn a program of type $\tau_1 \rightarrow \tau_2$. But I don't have the type on-hand, and have to find all possible types to τ_2. With polymorphic types in the context, there can be many ways to arrive at τ_2. For example, with $\forall X.X \rightarrow \tau_2$ in the context there are many options: $Bool \rightarrow \tau_2$, $Nat \rightarrow \tau_2$, $(Bool \rightarrow Bool) \rightarrow \tau_2$, etc.

The inception of the heuristic came when looking at the System F encoding of case analyses over algebraic data. For many useful programs, the type application in the encoding is always at a type in the context. These type applications, like the ones in previous examples, specify the return type. And generally (but not always), the return type of a case analysis is a type that exists as a binding in the context.

5.4 Strengths/Weaknesses

The strengths of the implementation are in its ability to handle functions in the output, which enable the learning of a class of programs from examples not previously demonstrated. As testament, we show in the next section a slew of programs over algebraic types that we learn. In each case, the programs always return a function. Because each program works over higher-order encodings of algebraic data.

Another strength is the emphasis on a functional core for learning. The language we built on top of the functional core is incidental, and we merely lift the machinery for synthesis to the high-level language. But we could have built other high-level languages all while maintaining the same functional core for learning. This departs from typical design strategies for learning, which embed the learning machinery in the high-level language. This forces language designers who want to implement learning in their own languages to adopt idiosyncracies of other high-level languages to enable learning. System F has its own idiosyncracies for sure, but these are amenable to the development of a broad swath of languages—each where we can lift the learning machinery.

A weakness of the current implementation is that learning complex programs is difficult. Since all our programs desugar to System F, they are larger than their sugared representation. In practice, learning succeeds (generally, not just in this work) when there are means to tame the combinatorial explosion in programs. As programs grow larger, the more likely you are to feel the explosion.

While our design strategy in lifting synthesis is quite distinct from related works, we still wanted to be able to learn similar programs. And we do, but performance suffers. This is in part because our System F encodings are much larger, but also because we don't make all the same optimizations as related works. Because of this, we view the implementation as successfully demonstrating a proof of concept: that we can lift learning in System F to higher-level languages to learn non-trivial programs. It sets the stage for further research in this direction, helping to carve what functional core ought to be.

6 Experiments

The implementation comes with a benchmark suite, whose results we show in Figure 5. They are a variety of programs over algebraic data. Some are simple, and others are quite complex like Figure 6: a curriculum for learning an interpreter for expressions in the natural number semiring. We now analyze these results. Ending with some intuitions about when to expect learning to succeed and when to expect it to fail.

Generally, the language succeeds when algebraic data are simple and the folds required are easy to write in case syntax. If it's not easy to write the fold, then you can try and help by providing a curriculum. In many instances, this helps. Like with `nat_bothZero`, but not always. In instances where curriculums don't help, there's simply so much in context for learning that performance suffers. Or the shape of the algebraic data make it such that it's easy to generate well-typed but useless programs. Both of these are the case with `natexpr` in Figure 6. By the time learning happens at natexpr there is a lot in the learning context. Additionally, the shape of natural numbers is such that it's very easy to generate many terms of type Nat under any context. You can generate 1, 2, 3, etc. These candidates are generated despite the fact that constants aren't frequently used for the programs we write. Other algebraic data suffer from similar issues.

A path forward to resolving these bottlebecks are to extend the learning relation in Figure 5 to better exploit examples from algebraic data. In [Osera 2015], they develop clever

Name	Size	Time (s)	# Ex	# Help
bexpr_eval	139	2.50	15	7
bool_nand	58	1.08	4	2
bool_or	42	0.06	4	2
bool_and	42	0.03	4	2
bool_not	53	0.02	2	2
bool_xor	72	3.75	4	2
data_fun	37	21.21	13	8
nat_bothZero	192	0.28	10	6
nexpr_eval	230	53.93	10	7
fun_plus	60	0.01	7	6
nat_plus	68	0.05	3	4
church_like	43	9.10	3	2
not_fun2	77	0.02	4	3
not_fun	36	0.05	4	3
poly_id	4	0.002	3	2
poly_twice	12	0.004	3	3
twice	38	0.002	3	3

Figure 6. Performance on a benchmark suite. Breaks down size of learned System F program (includes size of provided components as well). Time to learn. Number of examples. Number of helper functions: including constructors from datatype declarations, user-defined functions, and learned functions.

ways to use examples to reduce the combinatorial explosion of program space. The trick would be to express the semantics of their learning formulation in terms of our functional core for learning, System F. Another path is by moving up the lambda cube [Barendregt 1992].

7 Related Work

7.1 Rosette

Rosette is a programming language which extends Racket with machinery for synthesis, or learning [Torlak and Bodik 2013]. It imbues the language with SMT solvers.[1] With them, it's possible to write programs with holes, called sketches [Solar-Lezama 2008], which an SMT solver can fill. If a sketch has holes amenable to reasoning via SMT, Rosette can be quite effective.

For instance, recent work shows Rosette can reverse-engineer parts of an ISA [Zorn et al. 2017]—the instruction set architecture of a CPU (central processing unit). From bit-level data, you can recover programs which govern the behavior of the CPU. This works because SMT solvers support reasoning about bit-vector arithmetic, the sort of arithmetic governing low-level behavior of CPUs [Kroening and Strichman 2016].

[1]Satisfiability modulo theories (SMT) is a spin on the satisfiability problem, where one checks whether a boolean formulae is satisfied by a certain set of assignments to its variables. [Barrett and Tinelli 2018]

7.2 Synquid

Synquid is a programming language which uses refinement types to learn programs [Polikarpova et al. 2016]. A refinement type lets you annotate programs with types containing logical predicates [Freeman 1994]. The logical predicates constrain the space of programs which inhabit that type, and are used to guide learning.

Consider learning with and without refinement types. I'm going to teach you a program. But all I'll say is that its of type $nat \to nat$. As we saw in the introduction, the type alone isn't enough to constrain learning. With synquid, instead of going the route of examples they go the route of enriching types. Enter refinement types, which let you learn as follows.

We want you to learn a program. But all we say is that its of type $x{:}nat \to \{y{:}nat \mid y = x+1\}$. It takes a natural number x, and returns a natural number y one greater than x. 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 I'm agnostic to implementation details. Refinement types impose useful constraints on learning, it's the crux of Synquid.

7.3 Myth

Myth is a programming language which uses types and examples to learn programs [Osera 2015]. Types, like in Synquid, let you annotate programs by their behavior. But Myth doesn't use refinement types, its types aren't as expressive.

```
data Nat = Zero
         | S Nat

data NExpr = Val Nat
           | NAdd NExpr NExpr
           | NMul NExpr NExpr

plus :: Nat -> Nat -> Nat
plus = [<Zero,Zero,Zero>,<Zero, S Zero, S Zero>, <S Zero, Zero, S Zero>]

mul :: Nat -> Nat -> Nat
mul a b = case a[Nat] of
          Zero -> Zero
          S c -> plus b c

neval :: NExpr -> Nat
neval = [<Val (S Zero), S Zero>,
         <NAdd (Val Zero) (Val Zero), Zero>,
         <NAdd (Val (S Zero)) (Val Zero), S Zero>,
         <NAdd (NAdd (Val Zero) (Val (S Zero))) (NAdd (Val Zero) (Val Zero)), S Zero>,
         <NMul (Val Zero) (Val Zero), Zero>,
         <NMul (Val (S Zero)) (Val Zero), Zero>,
         <NMul (NAdd (Val Zero) (Val (S Zero))) (NAdd (Val (S Zero)) (Val Zero)), (S Zero)>]

main :: Unit
main = neval
```

Figure 7. Learning an interpreter for arithmetic expressions in the natural number semiring. Multiplication provided by the user to speed up learning.

Instead of deferring to richer types, Myth offers examples to constrain learning. Examples are nice because they're often convenient to provide, and at times easier to provide than richer types. Think of the difference between teaching chess by explaining all its principles, versus teaching chess by playing. In practice, a mix of both tends to do the trick. Formally specifying the rules, but also letting example play guide learning as when my uncle taught me.

This work was most directly inspired by Myth, and we borrow and refine many ideas therein. Of special curiosity, that isn't of focus in Myth is the capability of learning as a first class activity in languages—which we further speculate about through the conclusion.

8 Conclusion

In the 60's, Peter Landin published *The next 700 programming languages* [Landin 1966]. He noticed a trend at the time, which was that everyone was designing different languages for solving different problems. Not because they had to, but because it was the only obvious design strategy. As a result, the idiosyncracies of the languages became interwined with the idiosyncracies of the problems they were trying to solve. So he formulated ISWIM, a functional core language with which you could express other languages. And the idea persists to this day. Modern functional languages typically sit on top a functional core. We don't need fundamentally new languages for every programming task.

It's a matter of empirical investigation, but the most important idea in this paper is whether we need fundamentally new languages for every learning task. As it stands, the proliferation of learning tools (Rosette, Synquid, Myth) means the proliferation of languages. But could there not be a functional core with which to express other learning mechanisms? We tackle this question with the following contributions:

- (On theory) A simple description of learning from examples as a relation.
- (On practice) An implementation of learning from examples in System F.
- (On practice) An implementation of a high-level language, which lifts learning from System F into itself.

This work steps in a different direction of the design space, and while there's clearly plenty left to be desired, we see plenty trail ahead.

References

Henk P Barendregt. 1992. Lambda calculi with types. (1992).

Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343.

Tim Freeman. 1994. *Refinement Types for ML*. Technical Report. Carnegie-Mellon University, Department of Computer Science, No. CMU-CS-94-110.

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

Jan Martin Jansen. 2013. Programming in the λ-calculus: From Church to Scott and back. In *The Beauty of Functional Code*. Springer, 168–180.

Daniel Kroening and Ofer Strichman. 2016. *Decision procedures*. Springer.

Peter J Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (1966), 157–166.

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.

Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Citeseer.

Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 135–152.

Bill Zorn, Dan Grossman, and Luis Ceze. 2017. Solver Aided Reverse Engineering of Architectural. *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017).