Learning in System F (Synthesis Pearl)

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 eventually 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 $nat \to nat$. 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 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, $nat := (A \to A) \to (A \to A)$. Natural numbers are then higher-order functions. They take and return functions. In this context, 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.

Author's address:

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

(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.
- Prove the soundness and completeness of learning.
- Provide an implementation of learning, extending strong theoretical guarantees in practice.

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:

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

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. 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

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

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

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

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

Fig. 1. Learning from types in System F

4 Learning from Examples, Declaratively

4.1 What are examples?

Examples $[\chi_1, \ldots, \chi_n]$ are lists of tuples, containing the inputs and output to a program. For instance $[\langle 1, 1\rangle]$ describes an example whose input is 1 and output is 1. If I want to specify more than one example, I can package examples χ into the list: $[\langle 1, 1\rangle, \langle 2, 2\rangle]$. Here I have two examples, each with one input and one output. In general, examples take the form

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

where e is an arbitrary program in System F and χ is an example. This syntax for examples lets us construct examples with arbitrary numbers of inputs, e.g. $\langle 10, 10, 20\rangle \equiv \langle 10, \langle 10, \langle 20, Nil\rangle\rangle\rangle$. Note that when examples have multiple inputs I use the short-hand notation for describing an $n-$tuple, $\langle 10, 10, 20\rangle$ in lieu of full notation $\langle 10, \langle 10, \langle 20, Nil\rangle\rangle\rangle$. Likewise when an example is merely $\langle e, Nil\rangle$, I use the short-hand $\langle e\rangle$—as Nil can be interpreted as the empty element.

For an example $\chi \equiv \langle e_1, \ldots, e_n\rangle$, an ordered list of inputs is given by $e_1, \ldots, e_{n-1}$. The last index always denotes an output. An example satisfies or describes a program, if when the ordered list of inputs $e_1, \ldots, e_{n-1}$ is applied to a program e, it is equivalent to the output e_n. That is,

$$(((e\ e_1)e_2)\ldots e_{n-1}) =_\beta e_n$$

For instance, $\chi \equiv \langle 1, 1\rangle$ satisfies the identity program $\lambda x{:}nat.x$ because

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

Similarly, a list of examples $[\chi_1, \ldots, \chi_n]$ satisfies some program if each example in the list satisfies the program. Note that with this notion of satisfaction, we can construct examples which satisfy

any program e, that is $\langle e \rangle$. It's an example with no input, and whose output is e. Because no inputs can be applied, and that $e =_\beta e$, $\langle e \rangle$ satisfies e.

With the learning relation, we can ask whether IDENTITY is learnable given a context, type, and examples. IDENTITY is a program which takes a natural number and returns it.

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

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.

The informal process of learning described can be made formal via the relation presented in Figure 2.

(L-WRLD) says that if you can learn a list of programs $[e_1, \ldots, e_n]$, where $e_1, \ldots, e_n$ are equivalent, then you can learn the program e_1. This rule is used to create n worlds for n examples at the start of learning.

$$\frac{\Gamma \vdash list\ nat \to nat \rhd [\langle 1, 1\rangle, \langle 2, 2\rangle] \rightsquigarrow [\lambda x{:}nat.x, \lambda x{:}nat.x]}{\Gamma \vdash nat \to nat \rhd [\langle 1, 1\rangle, \langle 2, 2\rangle] \rightsquigarrow \lambda x{:}nat.x}\ \text{(L-WRLD)}$$

(L-BASE) says that if you can learn a list of programs from its type and each e_i is equivalent to some χ_i for $0 \le i \le n$, then we can use each χ_i as an example output. For instance, $(\lambda x{:}nat.x)1 =_\beta 1$. This means we can use 1 as an example output for $(\lambda x{:}nat.x)$. When learning, this rule is the "base" case. After exhausting the example information, this rule turns the learning process into learning from types.

$$\frac{\Gamma, 1{:}nat \vdash list\ nat \rightsquigarrow [(\lambda x{:}nat.x)1] \qquad (\lambda x{:}nat.x)1 =_\beta 1}{\Gamma \vdash list\ nat \rhd [\langle 1\rangle] \rightsquigarrow [(\lambda x{:}nat.x)1]}\ \text{(L-BASE)}$$

(L-EABS) says that if you can learn a list of applications $[e_1\pi_1(\chi_1), \ldots, e_n\pi_1(\chi_n)]$, then you can learn a list of abstractions $[e_1, \ldots, e_n]$ from examples where each $\pi_1\chi_i$ are inputs for $0 \le i \le n$. Note that π_1 is the first projection of an example tuple.

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

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

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

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

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

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

Fig. 2. Learning from examples in System F

$$\frac{\Gamma, 1{:}\mathit{nat}, 2{:}\mathit{nat} \vdash \mathit{list}\ \mathit{nat} \rhd [\langle 1\rangle, \langle 2\rangle] \rightsquigarrow [(\lambda x{:}\mathit{nat}.x)1, (\lambda x{:}\mathit{nat}.x)2]}{\Gamma \vdash \mathit{list}\ \mathit{nat} \to \mathit{nat} \rhd [\langle 1, 1\rangle, \langle 2, 2\rangle] \rightsquigarrow [\lambda x{:}\mathit{nat}.x, \lambda x{:}\mathit{nat}.x]} \text{ (L-EABS)}$$

(L-ETABS) says that if you can learn a list of applications $[e_1\lceil\alpha\rceil, \ldots, e_n\lceil\alpha\rceil]$, then you can learn a list of polymorphic abstractions $[e_1, \ldots, e_n]$ from examples where each $\Lambda\alpha.\chi_i$ are inputs for $0 \leq i \leq n$.

$$\frac{\Gamma, \alpha \vdash \mathit{list}\ \alpha \to \alpha \rhd [\langle z, z\rangle] \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil]}{\Gamma \vdash \mathit{list}\ \forall\alpha.\alpha \to \alpha \rhd [\Lambda\alpha.\langle z, z\rangle] \rightsquigarrow [\Lambda\alpha.\lambda x{:}\alpha.x]} \text{ (L-TABS)}$$

(L-SUM) says that if you can learn a list of programs whose input is type τ_a and another list of programs whose input is type τ_b, then you can learn a list of program whose input is the sum type $\tau_a + \tau_b$ and whose examples contain inputs of both type τ_a and type τ_b. During learning, this rule is perhaps the most useful. It lets you distribute examples when encountering sum types as inputs, which ends up creating two sub-problems, each of which dealing with a smaller set of examples, which are easier to satisfy. In the example, let $e \equiv \mathit{case}(b)\ \mathit{of}\ \iota_1(\mathit{true}) \mapsto \mathit{false} \mid \iota_2(\mathit{false}) \mapsto \mathit{true}$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{list}\ \mathit{bool} \to \mathit{bool} \rhd [\langle \mathit{true}, \mathit{false}\rangle] \rightsquigarrow [\lambda x{:}\mathit{bool}.\mathit{false}] \\ \Gamma \vdash \mathit{list}\ \mathit{bool} \to \mathit{bool} \rhd [\langle \mathit{false}, \mathit{true}\rangle] \rightsquigarrow [\lambda y{:}\mathit{bool}.\mathit{true}]\end{array}}{\Gamma \vdash \mathit{list}\ (\mathit{bool} + \mathit{bool}) \to \mathit{bool} \rhd [\langle \mathit{true}, \mathit{false}\rangle, \langle \mathit{false}, \mathit{true}\rangle] \rightsquigarrow [e, e]} \text{ (L-SUM)}$$

This new relation extends learning from types. Now System F can learn from examples too. So far, our presentation of learning is both sound and complete. You can learn every program in System F from types. We now show that these results hold when learning from examples too.

5 Metatheory

Because our aim is to show every program in System F is learnable from examples, we want to show that learning is still equivalent to typing. As with learning from types, we show both completeness and soundness of learning with respect to typing—giving us the equivalence. These proofs entail a bit more work, but are far simpler than similar presentations of metatheory for languages which permit learning from examples, e.g. Myth [Osera 2015]. The mathematical convenience is afforded by not introducing any machinery into System F for learning.

5.1 Typing and Learning are still equivalent

To show completeness, we need to show that for any program in System F there exists a list of examples from which it can be learned. This turns out to be trivial. Hence, a stronger statement we want is that we can learn any program which is satisfied by a list of examples. It would be problematic if we could learn any program in System F, but only from a particular subset of the examples which describe that program. Showing both of these gives strong guarantees on learning. If a list of examples describes a program, you can learn that program—and this list exists for every program in System F.

Lemma 5.1 (Completeness$_a$ of Learning)**.**
If $\Gamma \vdash e : \tau$ *then there exist some* $[\chi_1, \ldots, \chi_n]$ *such that* $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$

Proof. For an arbitrary program e of type τ, let $[\langle e \rangle]$ constitute its example list. This example list has no input, only an ouput e. Hence the statemement $\Gamma \vdash \tau \rhd [\langle e \rangle] \rightsquigarrow e$ asks whether we can learn a program e of type τ whose output is e.

Now, note that for any program $e =_\beta e$ by definition of the reflexive, transitive, and symmetric evaluation relation. Additionally, for any program e of type τ, a list $[e]$ can be learned from type $list\ \tau$. Because learning from types is complete, this works for any program.

Knowing this, we can apply the following learning rules:

$$\frac{\Gamma, x{:}\tau \vdash list\ \tau \rightsquigarrow [e] \qquad e =_\beta e}{\Gamma \vdash list\ \tau \rhd [\langle e \rangle] \rightsquigarrow [e]}\ \text{(L-Base)}$$

$$\frac{\Gamma \vdash list\ \tau \rhd [\langle e \rangle] \rightsquigarrow [e] \qquad e =_\beta e}{\Gamma \vdash \tau \rhd [\langle e \rangle] \rightsquigarrow e}\ \text{(L-Wrld)}$$

Hence for any program e in System F, it can be learned from examples when the example is $[\langle e \rangle]$. □

This is nice, but the less interesting completeness result. We want to make sure that a list of examples which describes a program can be used to learn it. An example list satisfies (or describes) a program, where if the program is applied to the inputs, the corresponding outputs are equivalent.

Lemma 5.2 (Completeness$_b$ of Learning)**.**
If $\Gamma \vdash e : \tau$ *and there exist some* $[\chi_1, \ldots, \chi_n]$ *which satisfies* e*, then* $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$.

Proof. There are two general cases to prove, when e has inputs and when e has no inputs. For programs without input, see previous lemma. For programs with input let $\Gamma \vdash e : \tau$, where $[\chi_1, \ldots, \chi_n]$ are examples which satisfy e. Because of Theorem **??**, we also know $\Gamma \vdash \tau \rightsquigarrow e$. Because lists are learnable, we also know $\Gamma \vdash list\ \tau \rightsquigarrow [e_1, \ldots, e_n]$, where $\bigwedge_{i=1}^{n} e_i =_\beta e_n$.

Now let's deconstruct each χ_i into its input and output components: $\langle \chi_i^{in}, \chi_i^{out} \rangle$. Since each χ_i satisfies e, it must be that $\bigwedge_{i=1}^{n} e_i \chi_i^{in} =_\beta \chi_i^{out}$. Due to satisfaction, $e_i \chi_i^{in}$ is guaranteed to be well-typed. Hence $\Gamma, \bigcup_{i=1}^{n} \chi_i^{out}{:}\tau^{out}, \bigcup_{i=1}^{n} \chi_i^{in}{:}\tau^{in} \vdash list\ \tau^{out} \rightsquigarrow [e_1\chi_i^{in}, \ldots, e_n\chi_n^{in}]$. With this, we can apply the following rules:

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

Remembering that $\chi_i \equiv \langle\chi_i^{in}, \chi_i^{out}\rangle$ and that $\bigwedge_{i=1}^{n} e_i =_\beta e_n$, we finally prove the necessary result.

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

Note that if there are n inputs to the examples which satisfy e, then (L-EAbs) must be applied n times to fully reconstruct the examples. □

Lemma 5.3 (Soundness of Learning)**.**
If $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$ *then* $\Gamma \vdash e : \tau$

Proof. Case analysis on learning rules.

Case 1: (L-Wrld)

We know that we can learn from examples a list of length n where each entry is e.

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

After applying (L-EAbs) and (L-Base) it must be that $\Gamma \vdash list\ \tau' \rightsquigarrow [e_1\chi_1^{in}, \ldots, e_n\chi_n^{in}]$. The list is only learnable if each element is learnable, hence $\Gamma \vdash \tau' \rightsquigarrow e_i\chi_i^{in}$. And an application is only learnable if each side of the application is learnable, hence $\Gamma \vdash \tau \rightsquigarrow e$ (noting $e_i =_\beta e$). Because of the equivalence of learning from types and typing, we have $\Gamma \vdash e : \tau$.

Case 2: (L-Base)

We know we can learn learn from types a list of length n where each entry is e.

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

The list is only learnable if each element is learnable, hence $\Gamma \vdash \tau \rightsquigarrow e$. Because of the equivalence of learning from types and typing, we have $\Gamma \vdash e : \tau$.

Case 3: (L-EAbs)

We know we can learn learn from types a list of length n and type τ_b where each entry is $e\pi_1(\chi_i)$.

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

After applying (L-Base) it must be that $\Gamma, \bigcup_{i=1}^{n} \pi_1(\chi_i){:}\tau_a \vdash list\ \tau_b \rightsquigarrow [e_1\pi_1(\chi_1), \ldots, e_n\pi_1(\chi_n)]$. The list is only learnable if each element is learnable, hence $\Gamma, \bigcup_{i=1}^{n} \pi_1(\chi_i){:}\tau_a \vdash \tau_b \rightsquigarrow e_1\pi_1(\chi_1)$. And an application is only learnable if each side of the application is learnable, hence $\Gamma \vdash \tau_a \to \tau_b \rightsquigarrow e$ (noting $e_i =_\beta e$). Because of the equivalence of learning from types and typing, we have $\Gamma \vdash e : \tau_a \to \tau_b$.

Case 4: (L-ETAbs)

Same strategy as Case 3, except using type application.

Case 5: (L-Sum)

After assuming (L-Sum), reduces to proof of Case 3. □

Theorem 5.4 (Equivalence of Typing and Learning)**.**
If and only if $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$*, then* $\Gamma \vdash e : \tau$ *and* $[\chi_1, \ldots, \chi_n]$ *satisfies* e*.*

Proof. Directly from Lemmas 5.1, 5.2 and 5.3. □

Because we can only learn a program if and only if it is well typed, it follows that learned programs obey progress, preservation, and normalization. Each proof invokes the equivalence theorem between typing and learning, and then the respective progress, preservation, and normalization theorems for typing.

5.2 Learned programs still don't get stuck

Corollary 5.5 (Progress in Learning)**.**
If e *is a learned program, then either* e *is a value or else there is some program* e' *such that* $e \longrightarrow_\beta e'$*.*

Proof. Directly from Theorems 5.4 and 10.1. □

We shouldn't be able to learn programs which get stuck during evaluation, same as with typing. If I learn a program, either its a value or it can be evaluated to another program. When learning from examples, learning still obeys progress.

5.3 Learned programs still don't change type

Corollary 5.6 (Preservation in Learning)**.**
If $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$ *and* $e \longrightarrow_\beta e'$*, then* $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e'$*.*

Proof. Directly from Theorems 5.4 and 10.2. □

We shouldn't be able to learn programs of a different type than the one provided. If I learn a program, and it evaluates to another program, then I should be able to learn that new program from the same type. When learning from examples, learning still obeys preservation.

5.4 Learned programs still always halt

Corollary 5.7 (Normalization in Evaluation)**.**
Learned programs in System F always evaluate to a value, to a normal form.

Proof. Directly from Theorems 5.4 and 10.3. □

We shouldn't be able to learn programs which never finish computing. They must halt. As with learning from types, learning from examples only lets you learn halting programs.

6 Learning identity, not, and successor

For examples, any program in System F can serve as input or output. There are no restrictions. In similar works which allow learning from examples, like Myth [Osera 2015], there are restrictions on what examples can look like. Namely, functions cannot appear as output in an example. This makes it impossible to learn many higher-order programs. In fact, the motivation for this work started from observing this limitation in Myth. It made it impossible to learn compilers, higher-order programs which specify programming languages.

Here we show how to learn several programs from examples before discussing further the prospect of learning not only programs, but programming languages.

6.1 Learning polymorphic identity

Lemma 6.1 (Polymorphic identity is learnable)**.**
$\cdot \vdash \forall\alpha.\alpha \to \alpha \rhd [\Lambda\alpha.\langle z, z\rangle] \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x$

Proof.

(i) $\alpha, z{:}\alpha \vdash \alpha \to \alpha \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil$

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

(ii) $\alpha, z{:}\alpha \vdash \alpha \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z$

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

(iii) $\alpha, z{:}\alpha \vdash list\ \alpha \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z]$

$$\dfrac{\dfrac{(ii)}{\alpha, z{:}\alpha \vdash \alpha \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z} \qquad \alpha, z{:}\alpha \vdash list\ \alpha \rightsquigarrow [\,]}{\alpha, z{:}\alpha \vdash list\ \alpha \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z]}\,\text{(L-Cons)}$$

(iv) $\cdot \vdash \forall\alpha.\alpha \to \alpha \rhd [\Lambda\alpha.\langle z, z\rangle] \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x$

$$\dfrac{\dfrac{\dfrac{\dfrac{(iii)}{\alpha, z{:}\alpha \vdash list\ \alpha \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z]} \qquad (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z =_\beta z}{\alpha, z{:}\alpha \vdash list\ \alpha \rhd [\langle z\rangle] \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil z]}\,\text{(L-Base)}}{\alpha \vdash list\ \alpha \to \alpha \rhd [\langle z, z\rangle] \rightsquigarrow [(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil\alpha\rceil]}\,\text{(L-EAbs)}}{\cdot \vdash list\ \forall\alpha.\alpha \to \alpha \rhd [\Lambda\alpha.\langle z, z\rangle] \rightsquigarrow [\Lambda\alpha.\lambda x{:}\alpha.x]}\,\text{(L-ETAbs)}}{\cdot \vdash \forall\alpha.\alpha \to \alpha \rhd [\Lambda\alpha.\langle z, z\rangle] \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x}\,\text{(L-Wrld)}$$

□

6.2 Learning boolean not

Lemma 6.2 (Not is learnable)**.**
$\cdot \vdash (bool + bool) \to bool \rhd [\langle true, false\rangle, \langle false, true\rangle] \rightsquigarrow e$

Note: $e \equiv case(b)\ of\ \iota_1(true) \mapsto false \mid \iota_2(false) \mapsto true$. *Additionally, that* $true$ *and* $false$ *are learnable from any context.*

Proof.

(i) $true{:}bool \vdash bool \rightsquigarrow (\lambda x{:}bool.false)true$

$$\dfrac{\dfrac{true{:}bool, x{:}bool \vdash bool \rightsquigarrow false}{true{:}bool \vdash bool \rightsquigarrow \lambda x{:}bool.false}\,\text{(L-Abs)} \qquad \dfrac{true{:}bool \in true{:}bool}{true{:}bool \vdash bool \rightsquigarrow true}\,\text{(L-Var)}}{true{:}bool \vdash bool \rightsquigarrow (\lambda x{:}bool.false)true}\,\text{(L-App)}$$

(ii) $\cdot \vdash list\ bool \to bool \rhd [\langle true, false\rangle] \rightsquigarrow [\lambda x{:}bool.false]$

$$\dfrac{\dfrac{\dfrac{\textit{(i)}\quad true{:}bool \vdash list\ bool \rightsquigarrow []}{true{:}bool \vdash list\ bool \rightsquigarrow [(\lambda x{:}bool.false)true]}\ \text{(L-Cons)} \qquad (\lambda x{:}bool.false)true =_\beta false}{true{:}bool \vdash listbool \rhd [\langle false\rangle] \rightsquigarrow [(\lambda x{:}bool.false)true]}\ \text{(L-Sum)}}{\Gamma \vdash list\ bool \to bool \rhd [\langle true, false\rangle] \rightsquigarrow [\lambda x{:}bool.false]}\ \text{(L-EAbs)}$$

(iii) $false{:}bool \vdash list\ bool \rightsquigarrow (\lambda y{:}bool.true)false$

$$\dfrac{\dfrac{false{:}bool, y{:}bool \vdash bool \rightsquigarrow true}{false{:}bool \vdash bool \rightsquigarrow \lambda y{:}bool.true}\ \text{(L-Abs)} \qquad \dfrac{false{:}bool \in false{:}bool}{false{:}bool \vdash bool \rightsquigarrow false}\ \text{(L-Var)}}{false{:}bool \vdash bool \rightsquigarrow (\lambda y{:}bool.true)false}\ \text{(L-App)}$$

(iv) $\cdot \vdash list\ bool \to bool \rhd [\langle false, true\rangle] \rightsquigarrow [\lambda y{:}bool.true]$

$$\dfrac{\dfrac{\dfrac{\textit{(iii)}\quad false{:}bool \vdash list\ bool \rightsquigarrow []}{false{:}bool \vdash list\ bool \rightsquigarrow [(\lambda y{:}bool.true)false]}\ \text{(L-Cons)} \qquad (\lambda y{:}bool.true)false =_\beta true}{false{:}bool \vdash listbool \rhd [\langle true\rangle] \rightsquigarrow [(\lambda y{:}bool.true)false]}\ \text{(L-Sum)}}{\cdot \vdash list\ bool \to bool \rhd [\langle false, true\rangle] \rightsquigarrow [\lambda y{:}bool.true]}\ \text{(L-EAbs)}$$

(v) $\cdot \vdash (bool{+}bool) \to bool \rhd [\langle true, false\rangle, \langle false, true\rangle] \rightsquigarrow e$

$$\dfrac{\dfrac{\textit{(ii)}\qquad \textit{(iii)}}{\cdot \vdash list\ (bool{+}bool) \to bool \rhd [\langle true, false\rangle, \langle false, true\rangle] \rightsquigarrow [e, e]}\ \text{(L-Sum)}}{\cdot \vdash (bool{+}bool) \to bool \rhd [\langle true, false\rangle, \langle false, true\rangle] \rightsquigarrow e}\ \text{(L-Wrld)}$$

□

6.3 Learning church successor

Lemma 6.3 (Successor is learnable)**.**
Let $church \equiv \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$, $\bar{0} \equiv \Lambda\alpha.\lambda f{:}\alpha \to \alpha.\lambda x{:}\alpha.x$, $\bar{1} \equiv \Lambda\alpha.\lambda f{:}\alpha \to \alpha.\lambda x{:}\alpha.fx$, *and* $succ \equiv \lambda n{:}church.\Lambda\alpha.\lambda f{:}\alpha \to \alpha.\lambda x{:}\alpha.f(n\lceil\alpha\rceil fx)$. *And assume that* $\bar{0}$, $\bar{1}$, *and succ can be learned from any context.*

Show $\cdot \vdash church \to church \rhd [\langle \bar{0}, \bar{1}\rangle] \rightsquigarrow succ$.

Proof.

(i) $\bar{0}{:}church \vdash church \to church \rightsquigarrow succ$
Note: Can be learned from any context.
(ii) $\cdot \vdash church \to church \rhd [\langle \bar{0}, \bar{1}\rangle] \rightsquigarrow succ$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bar{0}{:}church \in \bar{0}{:}church}}{(i)\quad \bar{0}{:}church \vdash church \rightsquigarrow \bar{0}} \quad \bar{0}{:}church \vdash list\ church \rightsquigarrow []}{\bar{0}{:}church \vdash church \rightsquigarrow (succ)\bar{0}} \quad (succ)\bar{0} =_\beta \bar{1}}{\bar{0}{:}church \vdash list\ church \rightsquigarrow [(succ)\bar{0}]}\ \text{(L-Base)}}{\cdot \vdash list\ church \rhd [\langle \bar{1} \rangle] \rightsquigarrow [(succ)\bar{0}]}\ \text{(L-EAbs)}}{\cdot \vdash list\ church \to church \rhd [\langle \bar{0}, \bar{1} \rangle] \rightsquigarrow [succ]}\ \text{(L-Wrld)}}{\cdot \vdash church \to church \rhd [\langle \bar{0}, \bar{1} \rangle] \rightsquigarrow succ}$$

□

7 Learning from Examples, Practically

We have implemented a proof-of-concept prototype, with promising results. (Reviewers: see the artifact attached.)

The implementation of learning from types is in the function `genTerms` in `learning.hs`. All the code in this section is in Haskell and runs in `ghci`:

```
genTerms TyBool [] 5
```

That generates all terms of type `Bool` from the empty context, up to an AST size 5.

The implementation of learning from examples is in the function `lrnTerms` in `learning.hs`.

```
lrnTerms (TyAbs TyBool TyBool) [InTm TmTrue (Out TmTrue)] [] [] 3
```

That generates all terms of type `Bool->Bool` from the empty context, up to an AST size 3 *and* which satisfy the example $< tt, tt >$.

To generate polymorphic terms, our examples include types. These types are used to instantiate an example at a particular base type. For example, run the following in ghci to learn at type $(\forall X.X -> X)$ with examples $< Bool, tt, tt >$:

```
lrnTerms (TyTAbs "X" (TyAbs (TyVar "X") (TyVar "X")))
         [InTy TyBool (InTm TmTrue (Out TmTrue))]
         [] [] 4
```

This will produce the polymorphic identity function.

The implementation "works" minus a programs which require multiple type applications. There's also a bottleneck in performance that becomes apparent when synthesizing programs at around AST depth 20, because of the way the type application rule is currently implemented for learning.

8 Related Work

The literature on synthesis is both vast and rapidly expanding. Therefore, we inevitably cherry-pick among approaches that have inspired us while covering a varied landscape as well.

8.1 Type-driven synthesis

The seminal works on type-driven synthesis by Osera [2015]. demonstrates that you can have none-"magical" approaches to synthesis: everything is predictable, such as for instance, needing trace-complete examples. We take inspiration from this work, and suggest a way forward for examples with higher-order functions in the input/output exammples. We do not require trace completeness for System F is strongly normalizing.

Polikarpova et al. [2016] have extended the typed-driven approaach to refinement types. This is interesting because refinement types, such as Liquid Types [**?**] are rather expressive and the types can act as rich specification. The idea of liquid types has been researched in many languages beyond the initial ML-style setting: for example, Javascript [Chugh et al. 2012] and Haskell [Vazou et al. 2014]. Our work extends the type-driven approach in an orthogonal direction.

8.2 Other approaches to synthesis

Program Sketching [Solar-Lezama 2008] has been a promising approach to synthesis. It relies on specification and holes. We rely on types instead of full-blown specifications.

Microsoft Prose [Research [n.d.]] has enabled the program demonstration of Excel. The Microsoft team has packaged the lessons learned from FlashFill [Gulwani 2011] into a generic framework Flash Meta [Polozov and Gulwani 2015]. The key insight is that witness functions (also known as reverse semantics) are enough to speed up synthesis a for a domain-specific language. In Prose, the synthesizer author defines a grammar for the language, the semantics and the inverse semantics, and the framework automagically creates the synthesizer.

Rosette [Torlak and Bodik 2014] is a solver-aided language framework that similarly enables a separation of concern between domain and generic solving, based on satisfiability modulo theory underneath the hood. In Rosette, one can write a vanilla interpreter and get a synthesizer.

Another term for synthesis used in the learning community is program induction. Differentiable programming such as for Forth [Bosnjak et al. 2017] and Datalog [Raghothaman et al. 2019] use relaxation techniques to go from symbol to neural. Neural-Guided Search [Ellis et al. 2020; Zhang et al. 2018] uses neural networks *not* for creating programs but for guiding the search on the symbolic possibilities.

Inductive logic programming [Muggleton 1991] provides a classical avenue for synthesis in Prolog and related programming languages. An exciting recent development is the work of Cropper and Morel [2020] in which programs are learned through failures.

9 Conclusion

We offer a promising route to type-driven synthesis through learning in System F. Our main insight is that System F provides a fruitful ground for further synthesis work. In addition to a declarative theory, we developed a small implementation to testbed our ideas.

References

Matko Bosnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. 2017. Programming with a Differentiable Forth Interpreter. *ICML* (2017). https://arxiv.org/pdf/1605.06640.pdf

Ravi Chugh, Patrick M Rondon, and Ranjit Jhala. 2012. Nested refinements: a logic for duck typing. *ACM SIGPLAN Notices* 47, 1 (2012), 231–244.

Andrew Cropper and Rolf Morel. 2020. Learning programs by learning from failures. (2020). https://arxiv.org/pdf/2005.02259.pdf

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2020. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. (2020). https://arxiv.org/pdf/2006.08381.pdf

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

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA* (popl'11, january 26–28, 2011, austin, texas, usa ed.). https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/

Stephen Muggleton. 1991. Inductive logic programming. *New Generation Computing* (1991). http://www.doc.ic.ac.uk/~shm/Papers/ilp.pdf

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.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *OOPSLA 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (oopsla 2015 proceedings of the 2015 acm sigplan international conference on object-oriented programming, systems, languages, and applications ed.). 107–126. https://www.microsoft.com/en-us/research/publication/flashmeta-framework-inductive-program-synthesis/

Mukund Raghothaman, Kihong Heo, Xujie Si, and Mayur Naik. 2019. Synthesizing Datalog Programs using Numerical Relaxation. *IJCAI* (2019). https://arxiv.org/pdf/1906.00163.pdf

Microsoft Research. [n.d.]. Prose. https://microsoft.github.io/prose/team/. Accessed: 2020-07-09.

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. 2014. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices* 49, 6 (2014), 530–541.

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with Refinement Types in the Real World. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/2633357.2633366

Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural Guided Constraint Logic Programming for Program Synthesis. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 1737–1746. http://papers.nips.cc/paper/7445-neural-guided-constraint-logic-programming-for-program-synthesis.pdf

10 Appendix

10.1 Specification of System F

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*		α, Γ	*type variable*
	$\Lambda\alpha.e$	*type abstraction*			

Fig. 3. Syntax in System F

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

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

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

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

Fig. 4. Typing in System F

Theorem 10.1 (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 10.2 (Preservation in Typing)**.**
If $\Gamma \vdash e : \tau$ and $e \to_\beta e'$, then $\Gamma \vdash e' : \tau$.

Theorem 10.3 (Normalization in Evaluation)**.**
Well-typed programs in System F always evaluate to a value, to a normal form.

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

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

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

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

Fig. 5. Evaluating in System F