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 arbitrary higher-order programs in System F from types and examples.

Additional Key Words and Phrases: Program Synthesis, Type Theory, Inductive Programming

1 Introduction

1.1 A tricky learning problem

Imagine we're teaching you a program. Your only data is the type $nat \rightarrow nat$. It takes a natural number, and returns a natural number. Any ideas? Perhaps a program which computes...

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

The good news is that $f(x) = x + 1$ is correct. The bad news is that the data let you learn a slew of other programs too. It doesn't constrain learning enough if we want to teach $f(x) = x + 1$. As teachers, we can provide better data.

Round 2. Imagine we're teaching you a program. But this time we give you an example of the program's behavior. Your data are the type $nat \rightarrow nat$ and an example $f(1) = 2$. It takes a natural number, and seems to return its successor. Any ideas? Perhaps a program which computes...

$$f(x) = x + 1, \qquad f(x) = x + 2 - 1, \qquad f(x) = x + 3 - 2, \qquad \cdots$$

The good news is that $f(x) = x + 1$ is correct. And so are all the other programs, as long as we're agnostic to some details. Types and examples impose useful constraints on learning. It's the data we use when learning in System F [Girard et al. 1989].

Existing work can learn successor from similar data [Osera 2015; Polikarpova et al. 2016]. But suppose nat is a church encoding. For some base type A, $nat := (A \rightarrow A) \rightarrow (A \rightarrow A)$. Natural numbers are then higher-order functions. They take and return functions. In this context, existing work can no longer learn successor.

1.2 A way forward

The difficulty is with how to handle functions in the return type. The type $nat \rightarrow nat$ returns a function, a program of type nat. To learn correct programs, you need to ensure candidates are the correct type or that they obey examples. Imagine we want to verify that our candidate program f obeys $f(1) = 2$. With the church encoding, $f(1)$ is a function, and so is 2. To check $f(1) = 2$ requires that we decide function equality—which is undecidable in a Turing-complete language [Sipser et al. 2006]. Functions in the return type create this issue. There are two ways out.

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

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. (Section 2 & 3)
- Prove the soundness and completeness of learning. (Section 2 & 3)
- Provide an implementation of learning, extending strong theoretical guarantees in practice. (Section 4 & 5)

2 System F

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

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.

5 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.

5.1 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.

5.2 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.

6 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

7 Appendix

7.1 Specification of System F

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

Fig. 2. Syntax in System F

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

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

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

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

Fig. 3. Typing in System F

Theorem 7.1 (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 $e \longrightarrow_\beta e'$.

Theorem 7.2 (PRESERVATION IN TYPING).
If $\Gamma \vdash e : \tau$ and $e \longrightarrow_\beta e'$, then $\Gamma \vdash e' : \tau$.

Theorem 7.3 (NORMALIZATION IN EVALUATION).
Well-typed programs in System F always evaluate to a value, to a normal form.

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

$$\frac{e_1 \rightarrow_\beta e_1'}{e_1 e_2 \rightarrow_\beta e_1' e_2}\ \text{(E-APP1)} \qquad \frac{e \rightarrow_\beta e'}{e\lceil\tau\rceil \rightarrow_\beta e'\lceil\tau\rceil}\ \text{(E-TAPP)}$$

$$\frac{e_2 \rightarrow_\beta e_2'}{e_1 e_2 \rightarrow_\beta e_1 e_2'}\ \text{(E-APP2)} \qquad (\Lambda\alpha.\lambda x{:}\alpha.e)\lceil\tau\rceil \rightarrow_\beta (\lambda x{:}\alpha.e)[\tau/\alpha]\ \text{(E-TSUB)}$$

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

Fig. 4. Evaluating in System F