Learning in System F

Joey Velez-Ginorio

University of Oxford

A thesis submitted for the degree of

Master of Science in Mathematics and Foundations of Computer Science

Trinity 2019

Acknowledgements

A year at Oxford felt like an impossibility not too long ago. Yet here I am in the closing days of my degree—about to finish the impossible. These efforts are a product of the time and patience others have had with me, who have nurtured my thoughts, and inspired me to seek opportunity to learn. These opportunities, occuring over the past few years and culminating in this work, have brought me to the strangest corners of human thought. I am content to have made it so far, and still so eager to see more of what's out there.

My friends made this journey far easier. To venture, somewhat alone, into these odd corners of human thought can make one feel alone in their thoughts. Yet the ear of a friend goes a long way. And at times, they renew interest in lost thoughts or open new avenues to explore. Their stories and perspectives have helped my view of the world grow more complete, both in and outside of research. I'm grateful.

My advisers made this journey far easier. They too have spent time in odd places, far more time than me. But it's this familiarity with the terrain, the heuristics of thought they've developed, which have made traversing my research more comforting. They've been an ear to some of my wild ideas, and have supported me through a balance of guidance and intellectual freedom. They've nudged me into better directions when I've been lost, and at the same time let me run with whatever's going on in my head. To conceive and to execute on such odd ideas with them has been an experience I cherish.

Abstract

Programming languages let us communicate with machines. Yet to date, nearly every language makes this a laborious process. Every instruction, with fine detail, must be specified to the machine. We aren't doomed to program this way. Instead, an upgrowth of works in programming languages aim to design programming languages which allow for more informal specifications of programs. Programs can be specified by providing examples of expected behavior—or by providing an informative type. Think of how one might communicate the rules of game to a friend, both by formal specification and by examples of play. I describe these languages as having machinery for "learning". You specify the program you want, then the language uses its machinery to learn the program you specified.

In this work I show that System F, a classic and storied programming language, can "learn" programs. Unlike previous works which introduce machinery explicitly for learning, I show System F needs nothing new. Moreover, System F can learn programs no previous language has been shown to learn—higher-order programs from examples. These strengths give System F the capacity to learn not only programs, but programming languages.

In Chapter 1, I introduce the problem of learning. These include musings over learning as discussed in cognitive science, followed by learning in programming languages.

In Chapter 2, I introduce System F. Here I tersely cover the language and state important theorems which hold in System F and which make learning possible.

In Chapter 3, I begin the technical treatment of learning, when data are types. A key theorem is proved. Namely, that any program in System F can be learned from types.

In Chapter 4, I extend learning from types, when data are examples. Another key theorem is proved. Namely, that any program in System F can be learned from examples.

In Chapter 5, I muse over the significance of learning in System F—discussing the prospect of learning programming languages in System F. Considerations for future work, including barriers to implementation and extending to new type systems, are discussed.

With the exception of Chapter 2, all content and interpretatons are original contributions.

Contents

Chapter 1

To learn a program

The term "learning", like a lot of other everyday terms, is used broadly and vaguely in the English language, and we carry those broad and vague usages over to technical fields, where they often cause confusion.

Herbert Simon
Why should machines learn? (1983)

1.1 The verb, learn

My uncle taught me chess. First, he laid down the facts. Bishops do this. Knights do that. You want to checkmate. From there, the rules were laid out. Ideally, I should have learned chess then. But I didn't. My attention lapsed and sometimes I forgot which piece did what. These gaps however, were filled by playing. Whatever sense I had of chess quickly revised itself. Examples of play helped me learn chess.

To learn chess, I don't mean anything mysterious. To learn chess is to know its rules. These tell you what moves you can and can't make. And what conditions must be met to win or lose. If I think bishops only move forward, I haven't learned chess. If I know bishops move diagonally, I'm learning chess.

To learn a program, I don't mean anything mysterious. To learn a program is to know its rules, its instructions. These tell you what computation it does and does not do. If I think helloWorld only prints "hello", I have not learned helloWorld[1]:

```
main( ){
    printf("hello, world\n");
}
```

[1] helloWorld is a famous test program. It prints out the words "hello, world". Its adoption traces back to a 1974 memo from Bell Labs by Brian Kerrighan. [12]

1.2 Humans learn programs

Uncontroversially, humans learn programs. Hordes of programmers learn programs everyday. Programs like helloWorld are often the first they learn. It was my first.

But we can push the idea further. Maybe programmers aren't the only ones learning programs. What if we all do? If the brain really is a computer, then to learn a concept is to learn a program.

1.2.1 A recurrent observation

In 1666, Leibniz said human reasoning is computation [16]. He envisioned the *characteristica universalis*[2]: a formal language for expressing all thoughts, all concepts. Paired with his calculus ratiocinator[3], Leibniz mused over machines which could think like us. His works anticipate the development of symbolic logic and the formal foundations of computation we enjoy today.

In 1843, Lovelace translates a lecture on Babbage's analytical engine, the precursor to modern programmable computers. She has a fundamental realization. In the notes of her translation, she remarks [18]:

> *Again, it might act upon other things besides number.....Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.*

A machine designed to compute with numbers is not restricted to reason over numbers. Numbers can denote more, like the "relations of pitched sounds." From which to reason about numbers is to reason about music, to create music.

In 1958, Von Neumann's unfinished notes for Yale's Silliman lectures are published posthumously. They're titled "The Computer and the Brain" [30]. He muses on the correspondence between computation and human thought, except now in reference to the brain.

1.2.2 A current controversy

The recurrent observation, a correspondence between human thought and computation, led to the inception of cognitive science in the 1970s. The observation lives on with tenuous status. That is, it's an open question as to how strict to interpret the correspondence, or even if it holds. Are brains literally computers, something like a computer, or something else entirely?

Opposition to the correspondence emerges in the dynamical systems approach. It claims that counter to orthodoxy, the brain is not a computer but instead a dynamical system [28]. A growing body of empirical work substantiates the approach [19].

Yet empirical work also substantiates the correspondence. Many works, including some of my own, show that you can predict what people learn when you treat learning as learning programs. [10, 29, 14]. It leaves cognitive science in a thorny situation. Robust empirical work support both, opposing views.[4]

Despite this, I'm inspired by the controversy. My alliances are with the pioneers of computing, that there is a real correspondence between human thought and computers.

[2] Universal characteristic

[3] A framework for computation, predating mathematical logic. Leibniz built a machine called the Stepped Reckoner, using the calculus ratiocinator as its model of computation.

[4] It's possible that these are complentary, not competing views. This is argued in [5]. However, to date the views are often framed in opposition.

The current work aims to explore the correspondence. If the brain is a computer, then it's programs are written in a peculiar programming language. One which lets us both write and learn the programs which govern our behavior. What might that language look like?

1.3 Computers learn programs

Program synthesis, type inhabitance, inductive programming, and theorem proving [11]. Different names for the same problem: learning programs from data. Sometimes the programs are proofs, sometimes they're terms.[5] Sometimes data are examples, sometimes they're types. Yet the aim is the same.

What might a programming language look like, if its programs could also be learned by machines? To moment, the literature consists of languages which explicitly introduce machinery to enable learning. They are useful to discuss before beginning on the greater exposition—learning in System F, a presentation which avoids explicitly introducing machinery.

1.3.1 Rosette

Rosette is a programming language which extends Racket with machinery for synthesis, or learning [27]. It imbues the language with SMT solvers.[6] With them, it's possible to write programs with holes, called sketches [26], 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 [32]—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 [13].

1.3.2 Synquid

Synquid is a programming language which uses refinement types to learn programs [24]. A refinement type lets you annotate programs with types containing logical predicates [8]. 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 $nat \to nat$. 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 (type) let you learn a slew of other programs too. It doesn't constrain learning enough if I want to teach $f(x) = x + 1$. Enter refinement types, which let data be specific.

Round 2. I'm going to teach you a program. But all I'll 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…

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

[5] The equivalence between programs, terms, and proofs is established by the Curry-Howard correspondence. [31]

[6] 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. [3]

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.

1.3.3 Myth

Myth is a programming language which uses types and examples to learn programs [20]. Types, like in Synquid, let you annotate programs by their behavior. But Myth doesn't use refinement types, its types aren't as expressive. Let's see how Myth constrains learning without refinement types.

I'm going to teach you a program. But all I'll say is that its of type $nat \to nat$. Frustrated, you point out that you have several programs in mind, and can't discern which is correct from my teaching...

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

Your frustration compels me to offer an example of how I want the program to work. When given 1, the program should return 2: $f(1) = 2$. Frustrations settle and you rightly suspect the correct program to compute $f(x) = x + 1$.

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.

1.4 Why are we better?

1.4.1 Programmer assistants, not programmers

Rosette, Synquid, and Myth are state of the art. Yet they each only learn small fractions of the programs that we learn. For example, I'm most interestered in computers which learn not only programs, but programming languages from examples.

To learn a programming language is to learn its compiler or interpreter. Compilers and interpreters are themselves nothing more than programs. But it's impossible to learn compilers from examples using a language like Myth. At best, Myth lets you learn simple interpreters for languages which let you do basic arithmetic, e.g. "2+1" evaluates to "3". And even then, only with lots of help from the data.

Across the board, computers mostly learn simple programs. They are more programmer assistants than full-fledged programmers. They manage simple tasks, letting programmers focus on the heavy lifting of programming. Is this all they'll ever be?

1.4.2 Seemingly, an impasse

At first glance, computability theory appears to settle the debate [25]. Computers will never program like us.

Theorem 1.4.1 (Rice's Theorem)**.**
For Turing-complete languages, all non-trivial semantic properties of programs are undecidable.

Semantic properties are about a program's behavior. Does it halt? Does $f(1) = 2$? And non-trivial properties are those which aren't true of every program, or false of every program.

Recall when I taught you the program which computes $f(x) = x + 1$ from examples. Given 1 the program should return 2: $f(1) = 2$. Checking that any program satisfies even this simple example is undecidable. Similar issues arise if you try to avoid examples, and go the route of more precise types like Synquid. There's no way around it if a computer programs in a Turing-complete language. Learning all programs is undecidable.

But I program in OCaml, a Turing-complete language. Recently, I learned how to write a compiler in it. Something's off. If my brain is a computer, then Rice's theorem applies. Learning all OCaml programs is undecidable.

1.4.3 A sensible sacrifice

Rice's theorem forces the hand. The brain isn't solving undecidable problems. So then how did I learn to write a compiler in OCaml? A way forward is through a sensible sacrifice.

Learning all OCaml programs is undecidable. But I only ever learn a subset of programs in Turing-complete languages. In fact, I can only claim to have learned a subset—same as anyone else. Maybe my brain uses a programming language which isn't Turing-complete, but which can learn useful subsets of Turing-complete languages. A sensible sacrifice.

The sacrifice lets us design languages for which Rice's Theorem doesn't hold. Non-trivial semantic properties can be decided. And I can learn to write compilers in OCaml. This is possible in System F.

Chapter 2

System F

So we are led to endless improvement, in order to be able to consider, besides the booleans, the integers, lists, trees, etc. Of course, all this is done to the detriment of conceptual simplicity and modularity.

Jean-Yves Girard
Proofs and Types (1989)

2.1 System T, then F

In the Dialectica interpretation, Kurt Gödel constructs System T. With it, he proves the consistency of Heyting arithmetic: a logical framework for reasoning about natural numbers [2]. That is, the axioms for Heyting arithmetic do not lead to contradictions about natural numbers.

Despite success, logicians like Girard expressed misgivings about System T. In order to reason over natural numbers, System T explicitly introduces machinery for reasoning over things like booleans and pairs. Because they aren't inherently representable in System T, you end up with the endless improvements Girard mentions in the epigraph. To do something new in System T, you add something new. What results is a language which quickly loses "conceptual simplicity and modularity."

To amend these misgivings, Girard constructs System F [9]. It's powerful enough to represent all machinery for the Dialectica interpretation, whilst preserving conceptual simplicity and modularity.

Similar misgivings about languages which learn programs led my retreat to System F. To learn something new, you add something new. Very quickly, these languages get complex. The key contribution of this work shows that not only does System F provide all machinery for the Dialectica interpretation, but also for learning from examples. To do something new in System F, I add nothing new.

The remainder of the chapter presents System F, with notation near identical to its presentation in [21]. My presentation is terse, in order to provide intuitions for readers unfamiliar with lambda calculi. These intuitions ground the work developed hereafter. For comprehensive coverage, I defer to [21].

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*			

Figure 2.1: Syntax in System F

2.2 SYNTAX

A convenience of System F is its minimal syntax. It's only marginally more complex than the simplest typed languages. Figure 2.1 presents its grammar in Backus-Naur form [21]. Beyond aesthetics, the minimalism is a mathematical convenience. Proofs about System F's behavior need only worry about a handful of language constructs.

To ease presentation, we use encodings for natural numbers from the start. These encodings are standard, and are shown in [9].

2.2.1 TYPES

Types describe the behavior of programs.

a) *nat* is the type for natural numbers. A program of this type is a natural number.

b) $nat \to nat$ is the function type from *nat* to *nat*. A program of this type has an input of type *nat* and an output of type *nat*.

c) $nat \to nat \to nat$ is the function type from *nat* and *nat* to *nat*. A program of this type has two inputs of type *nat* and an output of type *nat*.

d) $\forall\alpha.\alpha \to \alpha \to \alpha$ is the polymorphic function type from α and α to α. A program of this type has two inputs of the same type and an output of that type. Polymorphic abstraction, denoted by $\forall\alpha$, lets the function work for any type represented by α.

e) $\forall\alpha.\alpha \to \alpha \to nat$ is the polymorphic function type from α and α to *nat*. A program of this type has two inputs of the same type and an output which is always type *nat*.

2.2.2 TERMS

Terms are programs.

a) $\lambda x{:}nat.x$ is a program which takes a natural number x as input, and returns it. The variable which comes after λ in a term denotes the input. What comes after is the term's body, where its input is used for computation.

b) $(\lambda x{:}nat.x)1$ applies the previous program to 1.

c) $\Lambda\alpha.\lambda x{:}\alpha.x$ is a program which takes an x of any type, and returns it. It's a polymorphic, or generic, version of the first program.

d) $(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil nat\rceil$ applies the previous program to type nat.

e) $\lambda f{:}nat \to nat.\lambda x{:}nat.fx$ is a program which takes a function f of type $nat \to nat$ and a natural number x as input. It returns the application of f to x.

2.2.3 Values

Values are programs which have finished computing.

a) $(\lambda x{:}nat.x)1$ is not a value. The program is an application, which can't be a value. Applications means there's computing left to do, namely the application of the program to its argument.

b) $(\lambda x{:}nat)$ is a value. The program is applied to no arguments. There's no computing left to do.

c) 1 is a value.

2.2.4 Contexts

Contexts carry type information about variables.

a) $x{:}nat, y{:}nat$ is a valid context. It says x and y have type nat.

b) α is a valid context. It says α is a type variable.

c) $\cdot$ is a valid context. It says nothing. At times, when other information is present in the context, we omit $\cdot$—which technically, is always present in the context.

2.3 Evaluating, a relation

System F's syntax tells us what programs look like. But doesn't say anything about how they run, how they compute. Suppose I have the program $(\lambda x{:}nat.x)1$. It applies a program which returns its input to the number 1. It should return 1. But how exactly? We want a relation which gives us the following behavior.

$$(\lambda x{:}nat.x)1 \to_\beta x[1/x] \to_\beta 1$$

First, 1 is bound to the input x of $\lambda x{:}nat.x$. Then 1 substitutes x, resulting in 1. The notation $x[1/x]$ denotes 1 replacing x in the program x. A relation of this behavior is defined in Figure 2.2.

(E-App1) says that if a program e_1 evaluates to e_1', then e_1e_2 evaluates to $e_1'e_2$. For example:

$$\frac{(\lambda x{:}nat \to nat.x)(\lambda y{:}nat.y) \to_\beta (\lambda y{:}nat.y)}{((\lambda x{:}nat \to nat.x)(\lambda y{:}nat.y))1 \to_\beta (\lambda y{:}nat.y)1} \text{ (E-App1)}$$

(E-App2) says that if a program e_2 evaluates to e_2', then e_1e_2 evaluates to e_1e_2'. For example:

$$\frac{(\lambda x{:}nat.x)1 \to_\beta 1}{(\lambda x{:}nat.x)((\lambda x{:}nat.x)1) \to_\beta (\lambda x{:}nat.x)1} \text{ (E-App2)}$$

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

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

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

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

Figure 2.2: Evaluating in System F

(E-Sub) says that if a program $\lambda x{:}\tau.e$ is applied to value v, then replace all instances of x in e with v. For example:

$$(\lambda x{:}nat.x)1 \to_\beta x[1/x]\ \text{(E-Sub)}$$

(E-TApp) says that if a program e evaluates to e', then $e\lceil\tau\rceil$ evaluates to $e'\lceil\tau\rceil$. For example:

$$\frac{\Lambda\alpha.(\lambda x{:}\alpha\to\alpha.x)(\lambda x{:}\alpha.x) \to_\beta \Lambda\alpha.\lambda x{:}\alpha.x}{(\Lambda\alpha.(\lambda x{:}\alpha\to\alpha.x)(\lambda x{:}\alpha.x))\lceil nat\rceil \to_\beta (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil nat\rceil}\ \text{(E-TApp)}$$

(E-TSub) says that if a program $\Lambda\alpha.\lambda x{:}\alpha.e$ is applied to type τ, then replace all instances of α in $\lambda x{:}\alpha.e$ with τ. For example:

$$(\Lambda\alpha.\lambda x{:}\alpha.x)\lceil nat\rceil \to_\beta (\lambda x{:}\alpha.x)[nat/\alpha]\ \text{(E-TSub)}$$

The evaluating relation comes in many forms. Here, we use what's referred to as a call-by-value evaluation strategy [23]. More exist, and each impact how programs evaluate in System F, e.g. call-by-name and call-by-need [23, 1]. These alternatives aren't discussed, but we do consider the reflexive, symmetric, transitive closure of the defined evaluating relation, $=_\beta$. It denotes program equality. Because this evaluating relation has the normalization property, deciding program equality is decidable. In upcoming chapters, we'll see how program equality is used to learn from examples.

2.4 Typing, a relation

With syntax, we saw what programs look like. With evaluation, we saw how programs execute. So then why do we need typing? Without typing, we can write the following program in System F. The grammar in Figure 2.1 permits it.

$$(\lambda x{:}nat.x)(\lambda y{:}nat.y)$$

But this program is nonsense. The left-hand side of the application is a program which returns its argument x, which must be of type nat. The right-hand side is the same program. If we were to naively evaluate this program, it would substitute $\lambda x{:}nat.x$ for the argument x on the left-hand side. Yet the argument wouldn't be of type nat. $\lambda x{:}nat.x$ is the function type $nat \to nat$. The types don't align for evaluation, to do computation.

To actually do something, the left-hand side needs an argument of the correct type.

$$(\lambda x{:}nat \to nat.x)(\lambda y{:}nat.y)$$

This is why we need typing. It lets us build programs which make sense, for which types align to do computation. Like evaluation, we define typing as a relation.

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

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

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

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

Figure 2.3: Typing in System F

(T-Var) says that if x is bound to type τ in the context Γ, then the program x is of type τ.

$$\frac{x{:}nat \in x{:}nat}{x{:}nat \vdash x : nat}\ \text{(T-Var)}$$

(T-Abs) says that if x is bound to type τ_1 in the context Γ and that a program e_2 is of type τ_2, then the program $\lambda x{:}\tau_1.e_2$ is of type $\tau_1 \to \tau_2$ and x is removed from the context.

$$\frac{\Gamma, x{:}nat \vdash \lambda y{:}nat.y : nat \to nat}{\Gamma \vdash \lambda x{:}nat.\lambda y{:}nat.y : nat \to nat \to nat}\ \text{(T-Abs)}$$

(T-App) says that if a program e_1 is of type $\tau_1 \to \tau_2$ and a program e_2 is of type τ_1, then $e_1 e_2$ is of type τ_2.

$$\frac{\Gamma \vdash \lambda x{:}nat.x : nat \to nat \qquad \Gamma \vdash 1 : nat}{\Gamma \vdash (\lambda x{:}nat.x)1 : nat}\ \text{(T-App)}$$

(T-TAbs) says that if α is in the context, and a program e is of type τ, then the program $\Lambda\alpha.e$ is of type $\forall\alpha.\tau$ and α is removed from the context.

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

(T-TApp) says that if a program e is of type $\forall\alpha.\tau_1$, then the program $e\lceil\tau_2\rceil$ is of type $[\tau_2/\alpha]\tau_1$.

$$\frac{\Gamma \vdash \Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha \to \alpha}{\Gamma \vdash (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil nat\rceil : nat \to nat}\ \text{(T-Abs)}$$

As with evaluating, there are many ways to construct typing relations on System F. Each offers their own take on what constitutes a well-typed program in System F, e.g. System F extended with subtyping [4]. The behavior of the typing relation is especially important for learning, as learning will depend on key properties of the typing relation: namely progress and preservation.

2.5 Metatheory

Well-typed terms can be thought of as theorems constructed through the typing relation, whose proof are sound proof trees. Hence, statements about these theorems, or the typing relation in general, are metatheoretic statements. I briefly review key metatheoretic properties of System F essential for learning. Proofs ommitted, but easily found in [9].

2.5.1 Programs don't get stuck

Theorem 2.5.1 (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 $e \to_\beta e'$.

Programs shouldn't get stuck in the middle of computation. We want well-typed programs to always have something to do, even if they don't ever stop. Progress ensures this. And with preservation, it lets us prove typing is sound, that we can prove all true theorems implied by the typing relation.

2.5.2 Programs don't change types

Theorem 2.5.2 (Preservation in Typing)**.**
If $\Gamma \vdash e : \tau$ and $e \to_\beta e'$, then $\Gamma \vdash e' : \tau$.

Programs shouldn't suddenly switch types in the middle of computation. Remember, the point of the type is to let us know kind of computation to expect from a program. If the type switches during computation, we're losing that information. With progress, preservation lets us prove typing is sound. By extension, they are essential to prove learning sound.

2.5.3 Programs always halt

Theorem 2.5.3 (Normalization in Evaluation)**.**
Well-typed programs in System F always evaluate to a value, to a normal form.

Progress doesn't guarantee programs ever stop computing. They could get stuck in infinite loops. Under certain type systems it's impossible to guarantee programs ever stop, e.g. recursive types [21]. But in System F we know all programs halt, or stop. For learning from examples, introduced in subsequent chapters, this property is essential. With normalization, program equivalence becomes decidable—a key part of the learning procedure.

Chapter 3

Learning from types

> *The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point.*
>
> Claude Shannon
> The Mathematical Theory of Communication (1948)

3.1 Types aid communication

Programs communicate. They speak to machines, what we tell them to. For machines, the communication is simple. Programs have all they need to communicate intent. But the programmer bears the burden of thought. In order to say something, the programmer must be precise—every instruction a meticulous curation.

Humans communicate far differently. At times, communication can feel meticulous. But often it is simple. The things we say aren't precise formal specifications of what we mean to say, but are rather just precise enough to convey useful ideas. The burden of thought is balanced between speaker and listener. For example,

A glork smashed my car.

You don't know what a glork is, but you at least know it smashed my car. I'm precise enough to say something useful, that glorks smash things. But imprecise enough that the listener must guess what a glork is. Note however, that you can't just guess anything. A glork isn't likely a worm, unless you know of worms which smash cars. If we think of glorks as having a type, it's type might be "smasher", which heavily constrains the kinds of things a listener can plug in for the meaning of glork. Maybe it's an elephant, maybe it's a hurricane.

We can harness types to build languages which allow for this sort of productive, yet imprecise kind of communication. We provide a type, and let the computer learn what we mean.

3.2 Learning, a relation

As with typing and evaluation, I describe learning as a relation:

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

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

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

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

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

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

Figure 3.1: Learning from types in System F

This relation is actually equivalent to the typing relation. The only real difference is one of notation. For instance, note the symmetries between Figure 3.1 and Figure 2.3. When discussing metatheory, we prove the equivalence. Despite this, the learning relation is not redundant. The new notation forms the core of an extended learning relation defined in the next chapter, to learn from examples. Additionally, it shows how to exploit the typing relation to yield the core of a learning relation, learning from types.

(L-VAR) says that if x is bound to type τ in the context Γ, then you can learn the program x of type τ.

$$\frac{x{:}nat \in x{:}nat}{x{:}nat \vdash nat \rightsquigarrow x}\ \text{(L-VAR)}$$

(L-ABS) says that if x is bound to type τ_1 in the context Γ 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.

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

(L-APP) says that if you can learn a program e_1 from type $\tau_1 \to \tau_2$ and a program e_2 from type τ_1, then you can learn $e_1 e_2$ from type τ_2.

$$\frac{\Gamma \vdash nat \to nat \rightsquigarrow \lambda x{:}nat.x \qquad \Gamma \vdash nat \rightsquigarrow 1}{\Gamma \vdash nat \rightsquigarrow (\lambda x{:}nat.x)1}\ \text{(L-APP)}$$

(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.

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

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

$$\frac{\Gamma \vdash \forall \alpha.\alpha \to \alpha \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x}{\Gamma \vdash nat \to nat \rightsquigarrow (\Lambda\alpha.\lambda x{:}\alpha.x)\lceil nat\rceil}\ \text{(T-ABS)}$$

The examples are symmetric to those shown for typing. We now show, beyond aesthetic symmetries, that learning from types and typing are equivalent.

3.3 Metatheory

3.3.1 Typing and Learning are equivalent

Learning should obey progress, preservation, and normalization. By proving equivalence between typing and learning, we show that learning does obey these properties.

Lemma 3.3.1 (Completeness of Learning)**.**
If $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash \tau \rightsquigarrow e$

Proof. Induction on the typing rules. □

Lemma 3.3.2 (Soundness of Learning)**.**
If $\Gamma \vdash \tau \rightsquigarrow e$ *then* $\Gamma \vdash e : \tau$

Proof. Induction on the learning rules. □

Theorem 3.3.3 (Equivalence of Typing and Learning)**.**
If and only if $\Gamma \vdash \tau \rightsquigarrow e$ *then* $\Gamma \vdash e : \tau$

Proof. Directly from Lemmas 3.3.1 and 3.3.2. □

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.

3.3.2 Learned programs don't get stuck

Corollary 3.3.4 (Progress in Learning)**.**
If e *is a closed, learned program, then either* e *is a value or else there is some program* e' *such that* $e \rightarrow_\beta e'$.

Proof. Directly from Theorems 3.3.3 and 2.5.1. □

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.

3.3.3 Learned programs don't change type

Corollary 3.3.5 (Preservation in Learning)**.**
If $\Gamma \vdash \tau \rightsquigarrow e$ *and* $e \rightarrow_\beta e'$*, then* $\Gamma \vdash \tau \rightsquigarrow e'$.

Proof. Directly from Theorems 3.3.3 and 2.5.2. □

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.

3.3.4 Learned programs always halt

Corollary 3.3.6 (Normalization in Evaluation)**.**
Learned programs in System F always evaluate to a value, to a normal form.

Proof. Directly from Theorems 3.3.3 and 2.5.3. □

We shouldn't be able to learn programs which never finish computing. They must halt. This means we take a hit on the expressivity of programs we can learn. But for this sensible sacrifice, we can learn from examples in a decidable way. And despite the sacrifice in expressivity, there is still many useful programs you can learn from types. The next section shows that within System F you can learn the encodings for lists, products, and sum types. These encodings then let you learn many of the programs that programmers are interested in, e.g. operations on lists, tuples, and pattern matching.

3.4 Learning lists, products, and sums

System F can learn the encodings for lists, products, and sums. In subsequent sections I prove these encodings are learnable. This has two aims. First, to illustrate how I go about proving that programs are learnable. Second, because the encodings for lists, products, and sums are used in the definition of learning from examples, the extended learning relation presented next chapter. All encodings shown come from those in [9].

3.4.1 Learning the list encoding

The type $List\ \tau$ is a list of type τ, a finite sequence $[x_1, \ldots, x_n]$ where each element is type τ. We want programs like this to be learnable in System F:

$$\Gamma \vdash List\ nat \rightsquigarrow [1, 2, 3]$$

To show this, System F must be able to learn the list constructors. These are Nil and $Cons$. For lists of type τ, Nil denotes the empty list of type τ. $Cons$ is an operation which takes an element of type τ and appends it to the head of a list of type τ. To learn the list $[1, 2, 3]$, you need to learn $Cons(1, (Cons(2, Cons(3, Nil))))$.

Let the following be encodings for lists of type τ and the constructors Nil and $Cons$:

$$\begin{aligned} List\ \tau &\equiv \forall\alpha.\alpha \to (\tau \to \alpha \to \alpha) \to \alpha \\ Nil &\equiv \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}(\tau \to \alpha \to \alpha).x \\ Cons\ h\ t &\equiv \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}(\tau \to \alpha \to \alpha).yh(t\alpha xy) \end{aligned}$$

Lemma 3.4.1 (Nil is learnable)**.**
$\cdot \vdash List\ \tau \rightsquigarrow Nil$

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{x{:}\alpha \in \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha)}{\alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \rightsquigarrow x}\text{(L-Var)}}{\alpha, x{:}\alpha \vdash (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow \lambda y{:}(\tau \to \alpha \to \alpha).x}\text{(L-Abs)}}{\alpha \vdash \alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow \lambda x{:}\alpha.\lambda y{:}(\tau \to \alpha \to \alpha).x}\text{(L-Abs)}}{\cdot \vdash \forall\alpha.\alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}(\tau \to \alpha \to \alpha).x}\text{(L-TAbs)}$$

□

Lemma 3.4.2 (Cons is learnable)**.**
$h{:}\tau, t{:}List\ \tau \vdash List\ \tau \rightsquigarrow Cons\ h\ t$

Proof.

(i) $h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \tau \to \alpha \to \alpha \rightsquigarrow y$

$$\dfrac{\dfrac{y{:}(\tau \to \alpha \to \alpha) \in h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha)}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash (\tau \to \alpha \to \alpha) \rightsquigarrow y}\ \text{(L-Var)}}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \tau \to \alpha \to \alpha \rightsquigarrow y}\ \text{(L-TApp)}$$

(ii) $h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \to \alpha \rightsquigarrow yh$

$$\dfrac{\textit{(i)}\quad \dfrac{h{:}\tau \in h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha)}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \tau \rightsquigarrow h}\ \text{(L-Var)}}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \to \alpha \rightsquigarrow yh}\ \text{(L-TApp)}$$

(iii) $h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \forall\alpha.\alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow t$

$$\dfrac{t{:}List\ \tau \in h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha)}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \forall\alpha.\alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow t}\ \text{(L-Var)}$$

(iv) $h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow t\lceil\alpha\rceil$

$$\dfrac{\textit{(iii)}\qquad h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \rightsquigarrow \alpha}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow t\lceil\alpha\rceil}\ \text{(L-TApp)}$$

(v) $h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow t\lceil\alpha\rceil x$

$$\dfrac{\textit{(iv)}\quad \dfrac{x{:}\alpha \in h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha)}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \rightsquigarrow x}\ \text{(L-Var)}}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow t\lceil\alpha\rceil x}\ \text{(L-App)}$$

(vi) $h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \rightsquigarrow t\lceil\alpha\rceil xy$

$$\dfrac{\textit{(v)}\quad \dfrac{y{:}(\tau \to \alpha \to \alpha) \in h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha)}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \rightsquigarrow y}\ \text{(L-Var)}}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \rightsquigarrow t\lceil\alpha\rceil xy}\ \text{(L-App)}$$

(vii) $h{:}\tau, t{:}List\ \tau \vdash List\ \tau \rightsquigarrow Cons\ h\ t$

$$\dfrac{\dfrac{\dfrac{\dfrac{\textit{(ii)}\qquad \textit{(vi)}}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha, y{:}(\tau \to \alpha \to \alpha) \vdash \alpha \rightsquigarrow yh(t\lceil\alpha\rceil xy)}\ \text{(L-App)}}{h{:}\tau, t{:}List\ \tau, \alpha, x{:}\alpha \vdash (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow \lambda y{:}(\tau \to \alpha \to \alpha).yh(t\lceil\alpha\rceil xy)}\ \text{(L-Abs)}}{h{:}\tau, t{:}List\ \tau, \alpha \vdash \alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow \lambda x{:}\alpha.\lambda y{:}(\tau \to \alpha \to \alpha).yh(t\lceil\alpha\rceil xy)}\ \text{(L-Abs)}}{h{:}\tau, t{:}List\ \tau \vdash \forall\alpha.\alpha \to (\tau \to \alpha \to \alpha) \to \alpha \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}(\tau \to \alpha \to \alpha).yh(t\lceil\alpha\rceil xy)}\ \text{(L-TAbs)}$$

□

Using this encoding, it's possible to construct the list $[1, 2, 3]$ by constructing the program $Cons(1, (Cons(2, Cons(3, Nil))))$. We omit the proof of useful, yet tertiary list operations like *Fold*, *Map*, and *Reduce*—the sort often used in functional programming languages. These are learnable in System F, but not strictly necessary for presenting learning from examples.

3.4.2 Learning the product encoding

The type $\tau_a \times \tau_b$ is a tuple $\langle a, b \rangle$ where a is type τ_a and b is type τ_b. Tuples, unlike lists, can have elements of different types. We want programs like this to be learnable in System F:

$$\Gamma \vdash nat \times nat \rightsquigarrow \langle 1, 2 \rangle$$

To show this, System F must be able to learn the product constructor $\langle a, b \rangle$. Its an operation which takes an element a of type τ_a and joins it in a tuple with an element b of type τ_b. Additionally, I show that System F can learn the projection functions, $\pi_1 t$ and $\pi_2 t$. Each take as input a tuple and project the first and second element respectively. For instance, $\pi_1 \langle 1, 2 \rangle \rightarrow_\beta 1$.

Let the following be encodings for tuples of type $\tau_a \times \tau_b$ and the projections :

$$\begin{aligned}
\tau_a \times \tau_b &\equiv \forall \alpha.(\tau_a \to \tau_b \to \alpha) \to \alpha \\
\langle a, b \rangle &\equiv \Lambda \alpha.\lambda x{:}(\tau_a \to \tau_b \to \alpha).xab \\
\pi_1 t &\equiv t \lceil \tau_a \rceil (\lambda x{:}\tau_a.\lambda y{:}\tau_b.x) \\
\pi_2 t &\equiv t \lceil \tau_b \rceil (\lambda x{:}\tau_a.\lambda y{:}\tau_b.y)
\end{aligned}$$

Lemma 3.4.3 (Products are learnable)**.**
$a{:}\tau_a, b{:}\tau_b \vdash \tau_a \times \tau_b \rightsquigarrow \langle a, b \rangle$

Proof.

(i) $a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash (\tau_a \to \tau_b \to \alpha) \rightsquigarrow x$

$$\frac{x{:}(\tau_a \to \tau_b \to \alpha) \in a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash \tau_a \rightsquigarrow a}{a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash (\tau_a \to \tau_b \to \alpha) \rightsquigarrow x}\ \text{(L-Var)}$$

(ii) $a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash \tau_b \to \alpha \rightsquigarrow xa$

$$\frac{\dfrac{a{:}\tau_a \in a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha)}{a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash \tau_a \rightsquigarrow a}\ \text{(L-Var)} \quad \textit{(i)}}{a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash \tau_b \to \alpha \rightsquigarrow xa}\ \text{(L-App)}$$

(iii) $a{:}\tau_a, b{:}\tau_b \vdash \tau_a \times \tau_b \rightsquigarrow \langle a, b \rangle$

$$\frac{\dfrac{\dfrac{\dfrac{b{:}\tau_b \in a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha)}{a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash \tau_b \rightsquigarrow b}\ \text{(L-Var)} \quad \textit{(ii)}}{a{:}\tau_a, b{:}\tau_b, \alpha, x{:}(\tau_a \to \tau_b \to \alpha) \vdash \alpha \rightsquigarrow xab}\ \text{(L-App)}}{a{:}\tau_a, b{:}\tau_b, \alpha \vdash (\tau_a \to \tau_b \to \alpha) \to \alpha \rightsquigarrow \lambda x{:}(\tau_a \to \tau_b \to \alpha).xab}\ \text{(L-Abs)}}{a{:}\tau_a, b{:}\tau_b \vdash \forall \alpha.(\tau_a \to \tau_b \to \alpha) \to \alpha \rightsquigarrow \Lambda \alpha.\lambda x{:}(\tau_a \to \tau_b \to \alpha).xab}\ \text{(L-TAbs)}$$

□

Lemma 3.4.4 (First projection is learnable).
$t{:}\tau_a{\times}\tau_b \vdash \tau_a \rightsquigarrow \pi_1 t$

Proof.

(i) $t{:}\tau_a{\times}\tau_b \vdash \tau_a{\times}\tau_b \rightsquigarrow t$

$$\frac{t{:}\tau_a{\times}\tau_b \in t{:}\tau_a{\times}\tau_b}{t{:}\tau_a{\times}\tau_b \vdash \tau_a{\times}\tau_b \rightsquigarrow t}\ \text{(L-Var)}$$

(ii) $t{:}\tau_a{\times}\tau_b \vdash \tau_a \rightsquigarrow t\lceil\tau_a\rceil(\lambda x{:}\tau_a.\lambda y{:}\tau_b.x)$

$$\frac{\dfrac{(i)\quad t{:}\tau_a{\times}\tau_b \vdash \tau_a \rightsquigarrow \tau_a}{t{:}\tau_a{\times}\tau_b \vdash (\tau_a \to \tau_b \to \tau_a) \to \tau_a \rightsquigarrow t\lceil\tau_a\rceil} \qquad \dfrac{\dfrac{\dfrac{x{:}\tau_a \in t{:}\tau_a{\times}\tau_b, x{:}\tau_a, y{:}\tau_b}{t{:}\tau_a{\times}\tau_b, x{:}\tau_a, y{:}\tau_b \vdash \tau_a \rightsquigarrow x}\ \text{(L-Var)}}{t{:}\tau_a{\times}\tau_b, x{:}\tau_a \vdash \tau_b \to \tau_a \rightsquigarrow \lambda y{:}\tau_b.x}\ \text{(L-Abs)}}{t{:}\tau_a{\times}\tau_b \vdash \tau_a \to \tau_b \to \tau_a \rightsquigarrow \lambda x{:}\tau_a.\lambda y{:}\tau_b.x}\ \text{(L-Abs)}}{t{:}\tau_a{\times}\tau_b \vdash \tau_a \rightsquigarrow t\lceil\tau_a\rceil(\lambda x{:}\tau_a.\lambda y{:}\tau_b.x)}\ \text{(L-App)}$$

□

Lemma 3.4.5 (Second projection is learnable).
$t{:}\tau_a{\times}\tau_b \vdash \tau_a \rightsquigarrow \pi_2 t$

Proof.

(i) $t{:}\tau_a{\times}\tau_b \vdash \tau_a{\times}\tau_b \rightsquigarrow t$

$$\frac{t{:}\tau_a{\times}\tau_b \in t{:}\tau_a{\times}\tau_b}{t{:}\tau_a{\times}\tau_b \vdash \tau_a{\times}\tau_b \rightsquigarrow t}\ \text{(L-Var)}$$

(ii) $t{:}\tau_a{\times}\tau_b \vdash \tau_b \rightsquigarrow t\lceil\tau_b\rceil(\lambda x{:}\tau_a.\lambda y{:}\tau_b.x)$

$$\frac{\dfrac{(i)\quad t{:}\tau_a{\times}\tau_b \vdash \tau_b \rightsquigarrow \tau_b}{t{:}\tau_a{\times}\tau_b \vdash (\tau_a \to \tau_b \to \tau_b) \to \tau_b \rightsquigarrow t\lceil\tau_b\rceil} \qquad \dfrac{\dfrac{\dfrac{y{:}\tau_b \in t{:}\tau_a{\times}\tau_b, x{:}\tau_a, y{:}\tau_b}{t{:}\tau_a{\times}\tau_b, x{:}\tau_a, y{:}\tau_b \vdash \tau_b \rightsquigarrow y}\ \text{(L-Var)}}{t{:}\tau_a{\times}\tau_b, x{:}\tau_a \vdash \tau_b \to \tau_b \rightsquigarrow \lambda y{:}\tau_b.y}\ \text{(L-Abs)}}{t{:}\tau_a{\times}\tau_b \vdash \tau_a \to \tau_b \to \tau_b \rightsquigarrow \lambda x{:}\tau_a.\lambda y{:}\tau_b.y}\ \text{(L-Abs)}}{t{:}\tau_a{\times}\tau_b \vdash \tau_b \rightsquigarrow t\lceil\tau_b\rceil(\lambda x{:}\tau_a.\lambda y{:}\tau_b.y)}\ \text{(L-App)}$$

□

Using this encoding, it's possible to construct the tuple $\langle 1, 2\rangle$ and project its first and second element.

3.4.3 Learning the sum encoding

The type $\tau_a + \tau_b$ is a sum of types τ_a and τ_b. Many types can naturally be framed as sums. Consider booleans, which are always either $true$ or $false$. Their encodings in System F are $\Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.x$ and $\Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.y$ respectively, each of type $bool \equiv \forall\alpha.\alpha \to \alpha \to \alpha$. Sums are useful because they let us do case analysis. If we treat $true$ and $false$ as comprising as sum type $bool + bool$ then we can construct programs which do different things depending on whether its input is $true$ or $false$. We want programs like this to be learnable in System F:

$$\Gamma \vdash bool + bool \rightsquigarrow case(e)\ of\ \iota_1(true) \mapsto false \mid \iota_2(false) \mapsto true$$

This program is *not*, which takes a boolean and inverts it. It does a case analysis on its input e. If it's $true$ wrapped by the injector ι_1, it returns $false$. If it's $false$ wrapped by the injector ι_2, it returns $true$. The injectors construct sum types. Typically, $true$ is of type $bool$. But wrapped with an injector, it's type becomes $bool + bool$, permitting us to do case analysis. To learn this program, we must show System F can learn the encodings for case analysis and injectors.

Let the following be encodings for sums of type $\tau_a + \tau_b$:

$$\begin{aligned}
\tau_a + \tau_b &\equiv \forall\alpha.(\tau_a \to \alpha) \to (\tau_b \to \alpha) \to \alpha \\
\iota_1(e) &\equiv \Lambda\alpha.\lambda f{:}\tau_a \to \alpha.\lambda g{:}\tau_b \to \alpha.fe \\
\iota_2(e) &\equiv \Lambda\alpha.\lambda f{:}\tau_a \to \alpha.\lambda g{:}\tau_b \to \alpha.ge \\
case(e)\ of\ \iota_1(x) \mapsto z_1 \mid \iota_2(y) \mapsto z_2 &\equiv e\lceil\tau_c\rceil(\lambda x{:}\tau_a.z_1)(\lambda y{:}\tau_b.z_2)
\end{aligned}$$

Lemma 3.4.6 (First injector is learnable)**.**
$e{:}\tau_a \vdash \tau_a + \tau_b \rightsquigarrow \iota_1(e)$

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f{:}\tau_a \to \alpha \in e{:}\tau_a, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha}{e{:}\tau_a, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha \vdash \tau_a \to \alpha \rightsquigarrow f}\text{(L-Var)} \quad \dfrac{e{:}\tau_a \in e{:}\tau_a, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha}{e{:}\tau_a, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha \vdash \tau_a \rightsquigarrow e}\text{(L-Var)}}{e{:}\tau_a, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha \vdash \alpha \rightsquigarrow fe}\text{(L-App)}}{e{:}\tau_a, \alpha, f{:}\tau_a \to \alpha \vdash (\tau_b \to \alpha) \to \alpha \rightsquigarrow \lambda g{:}\tau_b \to \alpha.fe}\text{(L-Abs)}}{e{:}\tau_a, \alpha \vdash (\tau_a \to \alpha) \to (\tau_b \to \alpha) \to \alpha \rightsquigarrow \lambda f{:}\tau_a \to \alpha.\lambda g{:}\tau_b \to \alpha.fe}\text{(L-Abs)}}{e{:}\tau_a \vdash \forall\alpha.(\tau_a \to \alpha) \to (\tau_b \to \alpha) \to \alpha \rightsquigarrow \Lambda\alpha.\lambda f{:}\tau_a \to \alpha.\lambda g{:}\tau_b \to \alpha.fe}\text{(L-TAbs)}$$

□

Lemma 3.4.7 (Second injector is learnable)**.**
$e{:}\tau_b \vdash \tau_a + \tau_b \rightsquigarrow \iota_2(e)$

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{g{:}\tau_b \to \alpha \in e{:}\tau_b, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha}{e{:}\tau_b, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha \vdash \tau_b \to \alpha \rightsquigarrow g}\text{(L-Var)} \quad \dfrac{e{:}\tau_b \in e{:}\tau_b, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha}{e{:}\tau_b, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha \vdash \tau_b \rightsquigarrow e}\text{(L-Var)}}{e{:}\tau_b, \alpha, f{:}\tau_a \to \alpha, g{:}\tau_b \to \alpha \vdash \alpha \rightsquigarrow ge}\text{(L-App)}}{e{:}\tau_b, \alpha, f{:}\tau_a \to \alpha \vdash (\tau_b \to \alpha) \to \alpha \rightsquigarrow \lambda g{:}\tau_b \to \alpha.ge}\text{(L-Abs)}}{e{:}\tau_b, \alpha \vdash (\tau_a \to \alpha) \to (\tau_b \to \alpha) \to \alpha \rightsquigarrow \lambda f{:}\tau_a \to \alpha.\lambda g{:}\tau_b \to \alpha.ge}\text{(L-Abs)}}{e{:}\tau_b \vdash \forall\alpha.(\tau_a \to \alpha) \to (\tau_b \to \alpha) \to \alpha \rightsquigarrow \Lambda\alpha.\lambda f{:}\tau_a \to \alpha.\lambda g{:}\tau_b \to \alpha.ge}\text{(L-TAbs)}$$

□

Lemma 3.4.8 (Case analysis is learnable)**.**
$e{:}\tau_a+\tau_b, x{:}\tau_a, y{:}\tau_b \vdash \tau_c \rightsquigarrow case(e)\ of\ \iota_1(x) \mapsto z_1 \mid \iota_2(y) \mapsto z_2$

Proof.

(i) $e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash (\tau_a \to \tau_c) \to (\tau_b \to \tau_c) \to \tau_c \rightsquigarrow e\lceil\tau_c\rceil$

$$\dfrac{\dfrac{e{:}\tau_a+\tau_b \in e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c}{e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_a+\tau_b \rightsquigarrow e}\ \text{(L-Var)} \qquad e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_c \rightsquigarrow \tau_c}{e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash (\tau_a \to \tau_c) \to (\tau_b \to \tau_c) \to \tau_c \rightsquigarrow e\lceil\tau_c\rceil}\ \text{(L-App)}$$

(ii) $e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash (\tau_b \to \tau_c) \to \tau_c \rightsquigarrow e\lceil\tau_c\rceil(\lambda x{:}\tau_a.z_1)$

$$\dfrac{\textit{(ii)} \qquad \dfrac{\dfrac{z_1{:}\tau_c \in e{:}\tau_a+\tau_b, x{:}\tau_a s, z_1{:}\tau_c, z_2{:}\tau_c}{e{:}\tau_a+\tau_b, x{:}\tau_a, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_a \to \tau_c \rightsquigarrow z_1}\ \text{(L-Var)}}{e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_a \to \tau_c \rightsquigarrow \lambda x{:}\tau_a.z_1}\ \text{(L-Abs)}}{e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash (\tau_b \to \tau_c) \to \tau_c \rightsquigarrow e\lceil\tau_c\rceil(\lambda x{:}\tau_a.z_1)}\ \text{(L-App)}$$

(iii) $e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_c \rightsquigarrow e\lceil\tau_c\rceil(\lambda x{:}\tau_a.z_1)(\lambda y{:}\tau_b.z_2)$

$$\dfrac{\textit{(iii)} \qquad \dfrac{\dfrac{z_2{:}\tau_c \in e{:}\tau_a+\tau_b, y{:}\tau_b, z_1{:}\tau_c, z_2{:}\tau_c}{e{:}\tau_a+\tau_b, y{:}\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_c \rightsquigarrow z_2}\ \text{(L-Var)}}{e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_b \to \tau_c \rightsquigarrow \lambda y{:}\tau_b.z_2}\ \text{(L-Abs)}}{e{:}\tau_a+\tau_b, z_1{:}\tau_c, z_2{:}\tau_c \vdash \tau_c \rightsquigarrow e\lceil\tau_c\rceil(\lambda x{:}\tau_a.z_1)(\lambda y{:}\tau_b.z_2)}\ \text{(L-App)}$$

□

Using this encoding, it's possible to construct the program described earlier, $case(e)\ of\ \iota_1(true) \mapsto false \mid \iota_2(false) \mapsto true$. You can construct and deconstruct sums with this encoding.

Chapter 4

Learning from examples

Now the learning process may be regarded as a search for a form of behaviour which will satisfy the teacher...

Alan Turing
Computing Machinery and Intelligence (1950)

4.1 Examples aid communication

Remember, programs communicate. But the burden lay entirely on the human. They must be precise to say something—every instruction a meticulous curation. Last chapter we saw how types helped shift the burden. Instead of writing out a full program, we can just state a type and let the machine learn a program that fits the type. We're closer to a language which permits this sort of imprecise yet productive use:

A glork smashed my car.

This sentence has useful information, even if you don't know what a glork is. But what if you wanted to know? Surely you can't avoid glorks if you don't even know what they are. Maybe glorks are elephants. Maybe they're hurricanes.

Enter examples. They aid communication. For teachers they're a weapon of choice, of necessity. Types combined with examples offer precision.

A glork smashed my car. This morning it was fine.
But now there's a dent in the shape of a humungous footprint.

With the help of an example (of what happened), it's clearer what a glork is. It's likely not a hurricane, but an elephant or some other animal with huge feet. So we still don't get complete precision, but we often get enough from just a few examples.

We can harness examples to build languages which allow for this sort of productive and precise enough kind of communication. We provide a type and examples, then let the machine learn what we mean.

4.2 Learning, a relation

As before, I describe learning as a relation. This relation extends the presentation of learning from last chapter, now with support for examples.

$$\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$$
Given a context Γ *, type* τ *, and examples* $[\chi_1, \ldots, \chi_n]$*, I can learn program* e*.*

The main difference is that types can now be further specified by examples—think when I described a glork by its type "smasher" and what it did to my car. Importantly, examples $[\chi_1, \ldots, \chi_n]$ are not a new syntactic form in System F. It's only a notational convenience. In actuality, examples are tuples of inputs and outputs, e.g. examples for the identity function on natural numbers could be $\langle 1, 1\rangle$. And last chapter, we saw that tuples are learnable in System F.

4.2.1 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.

4.2.2 Learning, informally

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 ■ 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.

4.2.3 Learning, formally

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

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

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

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

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

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

Figure 4.1: Learning from examples in System F

The informal process of learning described can be made formal via the relation presented in Figure 4.1.

(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.

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

(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 $0 \leq i \leq n$, then we can use each χ_i as an example output. For instance, $(\lambda x{:}\mathit{nat}.x)1 =_\beta 1$. This means we can use 1 as an example output for $(\lambda x{:}\mathit{nat}.x)$. When learning, this rule is the "base" case. After exhausting the example information, this rule turns the learning process into learning from types.

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

(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 $0 \leq i \leq n$. Note that π_1 is the first projection of an example tuple.

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

(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 $e \equiv \mathit{case}(b)\ \mathit{of}\ \iota_1(\mathit{true}) \mapsto \mathit{false} \mid \iota_2(\mathit{false}) \mapsto \mathit{true}$

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

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.

4.3 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 [20]. The mathematical convenience is afforded by not introducing any machinery into System F for learning.

4.3.1 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 4.3.1 (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:

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

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 4.3.2 (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 3.3.3, 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:

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

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 4.3.3 (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 4.3.4 (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 4.3.1, 4.3.2 and 4.3.3. □

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.

4.3.2 Learned programs still don't get stuck

Corollary 4.3.5 (Progress in Learning)**.**
If e is a learned program, then either e is a value or else there is some program e' such that $e \to_\beta e'$.

Proof. Directly from Theorems 4.3.4 and 2.5.1. □

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.

4.3.3 Learned programs still don't change type

Corollary 4.3.6 (Preservation in Learning)**.**
If $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e$ and $e \to_\beta e'$, then $\Gamma \vdash \tau \rhd [\chi_1, \ldots, \chi_n] \rightsquigarrow e'$.

Proof. Directly from Theorems 4.3.4 and 2.5.2. □

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.

4.3.4 Learned programs still always halt

Corollary 4.3.7 (Normalization in Evaluation)**.**
Learned programs in System F always evaluate to a value, to a normal form.

Proof. Directly from Theorems 4.3.4 and 2.5.3. □

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.

4.4 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 [20], 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.

4.4.1 Learning polymorphic identity

Lemma 4.4.1 (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) $\cdot \vdash \forall\alpha.\alpha \to \alpha \rhd [\Lambda\alpha.\langle z, z\rangle] \rightsquigarrow \Lambda\alpha.\lambda x{:}\alpha.x$

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

□

4.4.2 Learning boolean not

Lemma 4.4.2 (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) $true{:}bool \vdash bool \rightsquigarrow (\lambda x{:}bool.false)true$

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

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

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

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

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

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

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

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

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

□

4.4.3 Learning church successor

Lemma 4.4.3 (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) $\cdot \vdash church \to church \rhd [\langle\bar{0}, \bar{1}\rangle] \rightsquigarrow succ$

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

□

Chapter 5

To learn a language

> *If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master.*
>
> Hal Abelson
> Foreword to *Essentials of Programming Languages* (2008)

5.1 Languages are learnable

Can a machine learn a programming language?

The question which started this thesis. And to which there's been essentially no work. Despite a surging interest in machines which learn programs [11], interest escapes machines which learn programming languages.

A programming language is itself, just another program. Either it's an interpreter, or it's a compiler. Interpreters take programs in a language and interpret them, give them a value or meaning. Compilers take programs in a language and translate them to another, where they are then interpreted, given a value or meaning. If machines can learn programs, why not interpreters or compilers? Because of completeness of learning in System F, you can.

5.1.1 Language, the ultimate abstraction

Programs are beholden to the designer of the language they are written in. If the designer decides that you must attend to memory allocation, you must. Or if the designer decides that all statements must be wrapped in curly brackets, then {they must}.

But it's impossible for these decisions to suit everyone's needs. It's why we constantly see the proliferation of new programming languages. People design languages to include the abstractions which are helpful for them, and exclude those which aren't. They are the ultimate abstraction. They let us be productive and solve programs effectively, which would otherwise be difficult or impossible under the design constraints of another language. Because we can't anticipate all problems programmers will need to solve, it's essential we can create the abstractions we need, the programming languages we need.

Take for instance, the situation faced by me in writing this work. System F, while beautiful in many respects, suffers when trying to write programs. It's so minimal that writing simple

programs require good effort. And to write more complex programs, it's nearly prohibitive. For instance, I originally meant to present a proof of a compiler in System F. But the proof and program itself would've been lengthy and unnecessarily laborious. Instead I defer to the completeness of learning to show that it's possible. But had I been using an OCaml-like language [17], with constructs defined to make programming (and proving) easier, then I would've included the compiler proof.

5.1.2 A promethean gesture

Machines which learn programs do so constrained by the languages we imbue them with. They are beholden to our language decisions. Some work acknowledges this limitation [7]. Yet still, machines cannot design the languages they need for the problems they try to solve. The languages we imbue them with lack this ability.

But they shouldn't be doomed to our language decisions. We ought to design languages which give power to the machine, which let them design their own languages, abstractions—those best fit for the problem from their own perspective. System F does this, at least in principle. Learning is complete in System F. Any program which can be written in System F can be learned, including compilers or interpreters.

5.2 From theory to implementation

Without an implementation of learning it awaits to be seen whether the results of this work are borne out in practice and not just in theory. The learning relations described are highly nondeterministic, and do not themselves constitute learning procedures. They do however serve as the basis for a learning procedure. These are the project's next steps. Here I sketch out the difficulties to come.

5.2.1 Problems of search

Once learning exhausts the information in examples, learning a program becomes a game of search. For complex programs, the search space grows prohibitively. Programs like interpreters and compilers have this issue. In [20], where a simple interpreter is learned, the amount of time it took was significantly longer than other learned programs—because of the size of the search space.

In order to tackle the real issues of search, languages need not only develop frameworks for learning, but also clever algorithms for search. In this work, those issues are cast aside to focus to formulate learning. They are nevertheless, interesting and important problems necessary to transition theory to implementation.

5.2.2 Problems of data

To learn a program, I've shown that there exist examples that can be shown to teach the program. For many programs, there will be many sets of examples to do the job. But in practice, it may be that specific sets of examples make learning easier or more difficult. This, again, is observed in works like [20]. It's analagous to the problems faced by teachers. In principle, each student can learn the material—but what's the best way to present the data such that the student learns the right thing.

Similar to search, I elide the real problems of teaching. Nevertheless it's essential that languages which can learn programs facilitate practical teaching as well. Otherwise you have a language which can learn any program, but for which many programs teaching is an overwhelming burden.

5.3 Different typing, different learning

This work uses System F's typing relation to yield a learning relation, from types and examples. The same strategy could be used towards other typing relations, yielding new learning relations with their own interesting properties. Just within System F, we saw how sum types $\tau_a + \tau_b$ impose useful constraints on learning. When learning, and encountering a sum type, you distribute examples according to whether they're τ_a or τ_b. Other types ought to impose their own useful constraints on learning. As bountiful the literature on type systems and typing relations, I gesture that the same expanse of literature could exist for learning systems and learning relations.

5.3.1 Dependent Types

Dependent types are far more expressive than the types in System F [22]. They let types depend on values. For instance, you can define a type $Vect : Nat \rightarrow Type \rightarrow Type$, which describes a vector whose length varies its type; perhaps for memory considerations. What's important are that these types are precise. They aid the issues pointed out with problems of data, problems of teaching. Because we can be more precise in our languages about what we want the machine to learn, it impacts learning.

5.3.2 Linear Types

Linear types introduce resource-sensitivity to typing. You can have linear variables, which can only be used once in the typing derivation. Additionally, contexts split throughout the course of a typing derivation—keeping them smaller than they would be in typical type systems. Problems of search typically stem from the context growing too large over the course of learning, and this kind of resource-sensitivity ought to guide learning in a fruitful way. Recent work introducing linear types to a learning framework akin to Synquid show these interesting behaviors.

5.3.3 Differential Types

Differential linear logic extends linear logic with constructs for differentiating proofs, which correspond to programs written in a linear type system. A recent pre-print uses these constructs to explore learning encodings of Turing machines [6]. To date, these are works mostly explored in logic—which have not yet made ways into the programming languages writ large. Yet we already know the great promise of learning through differentiation, e.g. the current resurgence and success of deep neural networks [15]. The prospects are exciting for a language which combines learning in differentiable and non-differentiable ways.

Bibliography

[1] Zena M Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of functional programming*, 7(3):265–301, 1997.

[2] Jeremy Avigad. Gödel's functional ("dialectica") interpretation. *Handbook of proof theory*, 137:337–405, 1998.

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

[4] Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. An extension of system f with subtyping. In *International symposium on theoretical aspects of computer software*, pages 750–770. Springer, 1991.

[5] Andy Clark. The dynamical challenge. *Cognitive Science*, 21(4):461–481, 1997.

[6] James Clift and Daniel Murfet. Derivatives of turing machines in linear logic. *arXiv preprint arXiv:1805.11813*, 2018.

[7] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In *Advances in Neural Information Processing Systems*, pages 7805–7815, 2018.

[8] Tim Freeman. Refinement types for ml. Technical report, Carnegie-Mellon University, Department of Computer Science, No. CMU-CS-94-110., 1994.

[9] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7.

[10] Thomas L Griffiths, Nick Chater, Charles Kemp, Amy Perfors, and Joshua B Tenenbaum. Probabilistic models of cognition: Exploring representations and inductive biases. *Trends in cognitive sciences*, 14(8):357–364, 2010.

[11] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[12] Brian W Kernighan. Programming in c - a tutorial. *Unpublished internal memorandum, Bell Laboratories*, 1974.

[13] Daniel Kroening and Ofer Strichman. *Decision procedures*. Springer, 2016.

[14] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

[16] Gottfried Wilhelm Leibniz. Dissertation on the art of combinations. In *Philosophical Papers and Letters*, pages 73–84. Springer, 1989.

[17] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 54, 2014.

[18] Ada A Lovelace. Sketch of the analytical engine invented by charles babbage, by lf menabrea, officer of the military engineers, with notes upon the memoir by the translator. *Taylor's Scientific Memoirs*, 3:666–731, 1842.

[19] James L McClelland, Matthew M Botvinick, David C Noelle, David C Plaut, Timothy T Rogers, Mark S Seidenberg, and Linda B Smith. Letting structure emerge: connectionist and dynamical systems approaches to cognition. *Trends in cognitive sciences*, 14(8):348–356, 2010.

[20] Peter-Michael Santos Osera. Program synthesis with types. *University of Pennsylvania, Department of Computer Science. PhD Thesis.*, 2015.

[21] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[22] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2005.

[23] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[24] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Notices*, volume 51, pages 522–538. ACM, 2016.

[25] Michael Sipser et al. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[26] Armando Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008.

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

[28] Tim Van Gelder. The dynamical hypothesis in cognitive science. *Behavioral and brain sciences*, 21(5):615–628, 1998.

[29] Joey Velez-Ginorio, Max H Siegel, Joshua B Tenenbaum, and Julian Jara-Ettinger. Interpreting actions by attributing compositional desires. In *CogSci*, 2017.

[30] John Von Neumann. *The computer and the brain*. Yale University Press, 2012.

[31] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

[32] Bill Zorn, Dan Grossman, and Luis Ceze. Solver aided reverse engineering of architectural. *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.