# Lecture Notes on Simple Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 3
Tuesday, September 11, 2018

## 1 Introduction

We have experienced the expressive power of the $\lambda$-calculus in multiple ways. We followed the slogan of *data as functions* and represented types such as Booleans and natural numbers. On the natural numbers, we were able to express the same set of partial functions as with Turing machines, which gave rise to the Church-Turing thesis that these are all the effectively computable functions.

On the other hand, Church's original purpose of the pure calculus of functions was a new foundations of mathematics distinct from set theory [Chu32, Chu33]. Unfortunately, this foundation suffered from similar paradoxes as early attempts at set theory and was shown to be *inconsistent*, that is, every proposition has a proof. Church's reaction was to return to the ideas by Russell and Whitehead [WR13] and introduce *types*. The resulting calculus, called *Church's Simple Theory of Types* [Chu40] is much simpler that Russell and Whitehead's *Ramified Theory of Types* and, indeed, serves well as a foundation for (classical) mathematics.

We will follow Church and introduce *simple types* as a means to classify $\lambda$-expressions. An important consequence is that we can recognize the representation of Booleans, natural numbers, and other data types and distinguish them from other forms of $\lambda$-expressions. We also explore how typing interacts with computation.

## 2 Simple Types, Intuitively

Since our language of expression consists only of $\lambda$-abstraction to form functions, juxtaposition to apply functions, and variables, we would expect our language of types $\tau$ to just contain $\tau ::= \tau_1 \to \tau_2$. This type might be considered "empty" since there is no base case, so we add type variables $\alpha$, $\beta$, $\gamma$, etc.

$$\begin{array}{lll} \text{Type variables} & \alpha & \\ \text{Types} & \tau & ::= \quad \tau_1 \to \tau_2 \mid \alpha \end{array}$$

We follow the convention that the function type constructor "$\to$" is *right-associative*, that is, $\tau_1 \to \tau_2 \to \tau_3 = \tau_1 \to (\tau_2 \to \tau_3)$.

We write $e : \tau$ if expression $e$ has type $\tau$. For example, the identity function takes an argument of arbitrary type $\alpha$ and returns a result of the same type $\alpha$. But the type is not unique. For example, the following two hold:

$$\begin{array}{lcl} \lambda x.\, x & : & \alpha \to \alpha \\ \lambda x.\, x & : & (\alpha \to \beta) \to (\alpha \to \beta) \end{array}$$

What about the Booleans? $\mathit{true} = \lambda x.\, \lambda y.\, x$ is a function that takes an argument of some arbitrary type $\alpha$, a second argument $y$ of a potentially different type $\beta$ and returns a result of type $\alpha$. We can similarly analyze *false*:

$$\begin{array}{lcll} \mathit{true} & = & \lambda x.\, \lambda y.\, x & : \quad \alpha \to (\beta \to \alpha) \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y & : \quad \alpha \to (\beta \to \beta) \end{array}$$

This looks like bad news: how can we capture the Booleans by their type if *true* and *false* have a different type? We have to realize that types are not unique and we can indeed find a type that is shared by *true* and *false*:

$$\begin{array}{lcll} \mathit{true} & = & \lambda x.\, \lambda y.\, x & : \quad \alpha \to (\alpha \to \alpha) \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y & : \quad \alpha \to (\alpha \to \alpha) \end{array}$$

The type $\alpha \to (\alpha \to \alpha)$ then becomes our candidate as a type of Booleans in the $\lambda$-calculus. Before we get there, we formalize the type system so we can rigorously prove the right properties.

## 3 The Typing Judgment

We like to formalize various judgments about expressions and types in the form of inference rules. For example, we might say

$$\frac{e_1 : \tau_2 \to \tau_1 \quad e_2 : \tau_2}{e_1\ e_2 : \tau_1}$$

We usually read such rules from the conclusion to the premises, pronouncing the horizontal line as "*if*":

> *The application $e_1\ e_2$ has type $\tau_1$ if $e_1$ maps arguments of type $\tau_2$ to results of type $\tau_1$ and $e_2$ has type $\tau_2$.*

When we arrive at functions, we might attempt

$$\frac{x_1 : \tau_1 \quad e_2 : \tau_2}{\lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ ?$$

This is (more or less) Church's approach. It requires that each variable $x$ intrinsically has a type that we can check, so probably we should write $x^\tau$. In modern programming languages this can be bit awkward because we might substitute for type variables or apply other operations on types, so instead we record the types of variable in a *typing context*.

$$\text{Typing context} \quad \Gamma \quad ::= \quad x_1 : \tau_1, \ldots, x_n : \tau_n$$

Critically, we always assume:

> *All variables declared in a context are distinct.*

This avoids any ambiguity when we try to determine the type of a variable. The typing judgment now becomes

$$\Gamma \vdash e : \tau$$

where the context $\Gamma$ contains declarations for the free variables in $e$. It is defined by the following three rules

$$\frac{\Gamma, x_1 : \tau_2 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ \textsf{lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsf{var}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \textsf{app}$$

As a simple example, let's type-check *true*. Note that we always construct such derivations bottom-up, starting with the final conclusion, deciding on rules, writing premises, and continuing.

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\ \textsf{var}}{x : \alpha \vdash \lambda y.\, x : \alpha \to \alpha}\ \textsf{lam}}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to (\alpha \to \alpha)}\ \textsf{lam}$$

How about the expression $\lambda x.\, \lambda x.\, x$? This is $\alpha$-equivalent to $\lambda x.\, \lambda y.\, y$ and therefore should check (among other types) as having type $\alpha \to (\beta \to \beta)$. It appears we get stuck:

$$\dfrac{\dfrac{??}{x : \alpha \vdash \lambda x.\, x : \beta \to \beta}\ \mathsf{lam}??}{\cdot \vdash \lambda x.\, \lambda x.\, x : \alpha \to (\beta \to \beta)}\ \mathsf{lam}$$

The worry is that applying the rule lam would violate our presupposition that no variable is declared more than once and $x : \alpha, x : \beta \vdash x : \beta$ would be ambiguous. But we said we can "*silently*" apply $\alpha$-conversion, so we do it here, renaming $x$ to $x'$. We can then apply the rule:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, x' : \beta \vdash x' : \beta}\ \mathsf{var}}{x : \alpha \vdash \lambda x.\, x : \beta \to \beta}\ \mathsf{lam}}{\cdot \vdash \lambda x.\, \lambda x.\, x : \alpha \to (\beta \to \beta)}\ \mathsf{lam}$$

# 4 Characterizing the Booleans

We would now like to show that the representation of the Booleans is in fact correct. We go through a sequence of conjectures to (hopefully) arrive at the correct conclusion.

**Conjecture 1 (Representation of Booleans, v1)**
*If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e =$ *true or* $e =$ *false.*

If by "$=$" we mean mathematical equality that this is false. For example,

$$\cdot \vdash (\lambda z.\, z)\ (\lambda x.\, \lambda y.\, x) : \alpha \to (\alpha \to \alpha)$$

but the expression $(\lambda z.\, z)\ (\lambda x.\, \lambda y.\, x)$ represents neither true nor false. But it is in fact $\beta$-convertible to *true*, so we might loosen our conjecture:

**Conjecture 2 (Representation of Booleans, v2)**
*If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e =_\beta$ *true or* $e =_\beta$ *false.*

This speaks to equality, but since we are interested in programming languages and computation, we may want to prove something ostensibly stronger. Recall that $e \longrightarrow_\beta^* e'$ means that we can $\beta$-reduce $e$ to $e'$ in an arbitrary number of steps (including zero). In other words, $\longrightarrow_\beta^*$ is the *reflexive* and *transitive* closure of $\longrightarrow_\beta$.

**Conjecture 3 (Representation of Booleans, v3)**
*If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e \longrightarrow_\beta^* \mathit{true}$ *or* $e \longrightarrow_\beta^* \mathit{false}$.

This is actually quite difficult to prove, so we break it down into several propositions, some of which we can actually prove. The first one concerns *normal forms*, that is, expressions that cannot be $\beta$-reduced. They play the role that *values* play in many programming language.

**Conjecture 4 (Representation of Booleans as normal forms)**
*If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* $e$ *is a normal form, then* $e = \mathit{true}$ *or* $e = \mathit{false}$.

We will later combine this with the following theorems which yiels correctness of the representation of Booleans. These theorems are quite general (not just on Booleans), and we will see multiple versions of them in the remainder of the course.

**Theorem 5 (Termination)** *If* $\Gamma \vdash e : \tau$ *then* $e \longrightarrow_\beta^* e'$ *for a normal form* $e'$.

**Theorem 6 (Subject reduction)** *If* $\Gamma \vdash e : \tau$ *and* $e \longrightarrow_\beta e'$ *then* $\Gamma \vdash e' : \tau$.

We will prove subject reduction on Lecture 4, and we may or may not prove termination in a later lecture. Today, we will focus on the the correctness of the representation of normal forms.

## 5 Normal Forms

Recall the rules for reduction. We refer to the first three rules as *congruence rules* because they allow the reduction of a subterms.

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{lm} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{ap}_1 \qquad \frac{e_1 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{ap}_2$$

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \beta$$

A *normal form* is an expression $e$ such that there does not exists an $e'$ such that $e \longrightarrow e'$. Basically, we have to rule out $\beta$-redices $(\lambda x.\, e_1)\, e_2$, but we would like to describe normal forms via inference rules to we can easily prove inductive theorems. This definition should capture the form

$$\lambda x_1.\ \ldots\ \lambda x_n.\, ((x\, e_1) \ldots e_k)$$

where $e_1, \ldots e_k$ are again in normal form.

We can capture this intuition in two parts: the definition of normal forms allows us to descend through arbitrarily many $\lambda$-abstractions. We write $e$ *nf* for the judgment that $e$ is in normal form.

$$\frac{e \ \mathit{nf}}{\lambda x.\, e \ \mathit{nf}}$$

At some point we have to switch to an application. The key part of this is that if we keep looking at the function part it may again be an application, or it may be a variable, but it cannot be a $\lambda$-abstraction. We call such expressions *neutral* because when they are applied to an argument they do not reduce but remain a normal form.

$$\frac{e \ \mathit{neutral}}{e \ \mathit{nf}} \qquad \frac{e_1 \ \mathit{neutral} \quad e_2 \ \mathit{nf}}{e_1 \ e_2 \ \mathit{neutral}} \qquad \frac{}{x \ \mathit{neutral}}$$

## 6 Representation as Normal Forms

In the next lecture we will prove that every expression either reduces or is a normal form. In this lecture we will be concerned with the property that *closed normal forms* of type $\alpha \to (\alpha \to \alpha)$ are exactly *true* and *false*.

Many of our proofs will go by induction, either on the structure of expressions or the structure of deductions using inference rules. The latter is called *rule induction*. It states that if every rule preserves the property of a judgment we want to show, then the property must always be true since the rules are the *only* way to establish a judgment. We get to assume the property for the premise of the rule (the *induction hypothesis*) and have to show it for the conclusion.

A special case of rule induction is proof by cases on the rules. We try here a rule by cases.

**Conjecture 7 (Representation of Booleans as normal forms, v1)**
*For any expression $e$, if $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ and $e$ nf then $e =$ true or $e =$ false.*

**Proof attempt:** By cases on the deduction of $e$ *nf*. We analyze each rule in turn.

**Case:**

$$\frac{e_2 \ \mathit{nf}}{\lambda x_1.\, e_2 \ \mathit{nf}}$$

where $e = \lambda x_1.\, e_2$. Let's gather our knowledge.

$\cdot \vdash \lambda x_1.\, e_2 : \alpha \to (\alpha \to \alpha)$ Assumption

What can we do with this knowledge? Looking at the typing rules we see that there is only one possible rule that may have this conclusion. Since the judgment holds by assumption, it must have been inferred with this rule and the premise of that rule must also hold. We call this step *inversion*.

$x_1 : \alpha \vdash e_2 : \alpha \to \alpha$ By inversion

We also know that $e_2$ is a normal form (the premise of the rule in this case), so now we'd like to show that $e_2 = \lambda x_2.\, x_1$ or $e_2 = \lambda x_2.\, x_2$. We choose the generalize the theorem to make this property explicit as well (see version 2).

First, though, let's consider the second case.

**Case:**

$$\frac{e \textit{ neutral}}{e \textit{ nf}}$$

Again, restating our assumption, we have

$\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ By assumption

But there is no closed neutral expression because at the head of the left-nested expressions would have to be a variable, of which we have none. This property will be an instance of a more general property we will need shortly.

□

**Conjecture 8 (Representation of Booleans as normal forms, v2)**

*(i)* *If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* $e$ *nf then* $e =$ *true or* $e =$ *false.*

*(ii)* *If* $x : \alpha \vdash e : \alpha \to \alpha$ *and* $e$ *nf then* $e = \lambda y.\, x$ *or* $e = \lambda y.\, y$.

*(iii)* *If* $x : \alpha, y : \alpha \vdash e : \alpha$ *and nf then* $e = x$ *or* $e = y$.

**Proof attempt:** By cases on the given deduction of $e$ *nf*. Now parts (i) and (ii) proceed as in the previous attempt, (i) relying on (ii) and (ii) relying on (iii), analyzing the cases of normal forms. The last one is interesting: we know that $e$ cannot be $\lambda$-abstraction because that would have function type, so it must be a neutral expression. But if it is a neutral expression it should have to be one of the variables $x$ or $y$: it cannot be an application because we would have to find a variable of function type at the head of the left-nested applications. So we need a lemma before we can complete the proof. □

The insight in this proof attempt gives rise to the following lemma.

**Lemma 9 (Neutral expressions)**
*If* $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e : \tau$ *and* $e$ *neutral then* $e = x_i$ *and* $\tau = \alpha_i$ *for some* $i$.

**Proof:** By rule induction over the definition of $e$ *neutral*

**Case:**

$$\frac{}{x \textit{ neutral}}$$

where $e = x$.

| | |
|---|---|
| $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash x : \tau$ | Assumption |
| $x : \tau \in (x_1 : \alpha_1, \ldots, x_n : \alpha_n)$ | By inversion |
| $x = x_i$ and $\tau = \alpha_i$ for some $i$ | |

**Case:**

$$\frac{e_1 \textit{ neutral} \quad e_2 \textit{ nf}}{e_1\, e_2 \textit{ neutral}}$$

and $e = e_1\, e_2$. This case is impossible:

| | |
|---|---|
| $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e_1\, e_2 : \tau$ | Assumption |
| $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e_1 : \tau_2 \to \tau$ | |
| and $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e_2 : \tau_2$ for some $\tau_2$ | By inversion |
| $e_1 = x_i$ and $\tau_2 \to \tau = \alpha_i$ | By induction hypothesis |
| Contradiction, since $\tau_2 \to \tau \neq \alpha_i$ | |

□

Now we are ready to prove the representation theorem.

**Theorem 10 (Representation of Booleans as normal forms, v3)**

*(i)* *If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* $e$ *nf then* $e = true$ *or* $e = false$.

*(ii)* *If* $x : \alpha \vdash e : \alpha \to \alpha$ *and* $e$ *nf then* $e = \lambda y.\, x$ *or* $e = \lambda y.\, y$.

*(iii)* *If* $x : \alpha, y : \alpha \vdash e : \alpha$ *and nf then* $e = x$ *or* $e = y$.

**Proof:** By cases on the given deduction of $e$ *nf*.

**Case for (i):**

$$\frac{e_2 \ nf}{\lambda x.\, e_2 \ nf}$$

where $e = \lambda x.\, e_2$.

| | |
|---|---|
| $\cdot \vdash \lambda x.\, e_2 : \alpha \to (\alpha \to \alpha)$ | Assumption |
| $x : \alpha \vdash e_2 : \alpha \to \alpha$ | By inversion |
| $e_2 = \lambda y.\, x$ or $e_2 = \lambda y.\, y$ | By part (ii) |
| $e = \lambda x.\, \lambda y.\, x$ or $e = \lambda x.\, \lambda y.\, y$ | since $e = \lambda x.\, e_2$ |

**Case for (i):**

$$\frac{e \ neutral}{e \ nf}$$

| | |
|---|---|
| $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ | Assumption |
| Impossible, by the *neutral expression lemma* (9) | |

**Cases for (ii):** analogous to the cases for (i), appealing to part (iii).

**Case for (iii):**

$$\frac{e_2 \ nf}{\lambda z.\, e_2 \ nf}$$

| | |
|---|---|
| $x : \alpha, y : \alpha \vdash \lambda z.\, e_2 : \alpha$ | Assumption |
| Impossible by inversion (no typing rule matches this conclusion) | |

**Case for (iii):**

$$\frac{e \ neutral}{e \ nf}$$

| | |
|---|---|
| $x : \alpha, y : \alpha \vdash e : \alpha$ | Assumption |
| $e = x$ and or $e = y$ by the neutral expression lemma (9) | |

□

## References

[Chu32] A. Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

[Chu33] A. Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 34:839–864, 1933.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[WR13] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910–13. 3 volumes.