Lecture Notes on Simple Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 3
Tuesday, September 10, 2019

1 Introduction

We have experienced the expressive power of the λ-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 than 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 λ-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 λ-expressions. We also explore how typing interacts with computation.

2 Simple Types, Intuitively

Since our language of expression consists only of λ-abstraction to form functions, juxtaposition to apply functions, and variables, we would expect our language of types τ 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 α, β, γ, 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 τ. For example, the identity function takes an argument of arbitrary type α and returns a result of the same type α. 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? *true* $= \lambda x.\, \lambda y.\, x$ is a function that takes an argument of some arbitrary type α, a second argument y of a potentially different type β and returns a result of type α. We can similarly analyze *false*:

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

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*:

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

The type $\alpha \to (\alpha \to \alpha)$ then becomes our candidate as a type of Booleans in the λ-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 τ_1 if e_1 maps arguments of type τ_2 to results of type τ_1 and e_2 has type τ_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^τ. 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 Γ contains declarations for the free variables in e. It is defined by the following three rules

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

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

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.

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

In this construction we exploit that the rules for typing are *syntax-directed*: for every form of expression there is exactly one rule we can use to infer its type.

How about the expression $\lambda x.\, \lambda x.\, x$? This is α-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 α-conversion, so we do it here, renaming x to x'. We can then apply the rule:

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

A final observation here about type variables: if $\cdot \vdash e : \alpha \to (\beta \to \beta)$ then also $\cdot \vdash e : \tau_1 \to (\tau_2 \to \tau_2)$ for any types τ_1 and τ_2. In other words, we can *substitute* arbitrary types for type variables in a typing judgment $\Gamma \vdash e : \tau$ and still get a valid judgment. In particular, the expressions *true* and *false* have *infinitely many types*.

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 β-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 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 β-reduced. They play the role that *values* play in many programming language.

Conjecture 3 (Representation of Booleans, v3)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* e *is a normal form, then* $e =$ *true or* $e =$ *false.*

We will later combine this with the following theorems which yields 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 4 (Termination) *If* $\Gamma \vdash e : \tau$ *then* $e \longrightarrow_\beta^* e'$ *for a normal form* e'.

Theorem 5 (Subject reduction) *If* $\Gamma \vdash e : \tau$ *and* $e \longrightarrow_\beta e'$ *then* $\Gamma \vdash e' : \tau$.

We will prove subject reduction in 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 subterm.

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{red/lam} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app}_1 \qquad \frac{e_1 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{red/app}_2$$

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

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 β-redices $(\lambda x.\, e_1)\, e_2$, but we would like to describe normal forms via inference rules so we can easily prove inductive theorems on them. 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. To capture something like this, we define a judgment $e\ \mathit{nf}$ by using inference rules. First, a λ-abstraction is normal if its body is normal (the only place where a reduction might be applied). Also, variables by themselves are certainly normal.

$$\frac{e\ \mathit{nf}}{\lambda x.\, e\ \mathit{nf}}\ \mathsf{nf/lam} \qquad \frac{}{x\ \mathit{nf}}\ \mathsf{nf/var}$$

Applications $e_1\, e_2$ are trickier. Certainly, both e_1 and e_2 need to be normal. In addition, e_1 cannot be a λ-abstraction because then the term would be a β-redex.

$$\frac{e_1 \neq \lambda \_ \quad e_1\ \mathit{nf} \quad e_2\ \mathit{nf}}{e_1\, e_2\ \mathit{nf}}\ \mathsf{nf/app}$$

Of course, we are now obligated to check that this definition is correct, that is, $e\ \mathit{nf}$ if and only if there is no e' such that $e \longrightarrow e'$. We now prove one direction because it introduces the ubiquitous techniques of *rule induction* and *inversion*.

We write $e \not\longrightarrow$ (pronounced e *does not reduce*) if there is no e' such that $e \longrightarrow e'$. Before we launch into any proof, how do we establish $e \not\longrightarrow$? Because it is a negation, we *assume* instead that $e \longrightarrow e'$ for some e' and then derive a contradiction from that. Also, because e' is not significant in this study of normal forms, we may write $e \longrightarrow$ (pronounced e *reduces*) to express that there is an e' such that $e \longrightarrow e'$.

Theorem 6 (Normal forms do not reduce) *For all expressions e, if $e\ \mathit{nf}$ then $e \not\longrightarrow$.*

Proof: By *rule induction* on the derivation of $e\ \mathit{nf}$. We can also think of this as *structural induction* over the derivation of $e\ \mathit{nf}$. This is a sound principle because the rules we give exactly *define* the judgment $e\ \mathit{nf}$.

Since there are three inference rules for $e\ \mathit{nf}$, we distinguish three cases.

Case:

$$\frac{e_1\ \mathit{nf}}{\lambda x.\, e_1\ \mathit{nf}}\ \mathsf{nf/lam}$$

where $e = \lambda x.\, e_1$. Then we start reasoning as follows:

$e_1 \not\longrightarrow$ By induction hypothesis

We have to show that $\lambda x.\, e_1 \not\longrightarrow$. For that purpose we assume that $\lambda x.\, e_1$ reduces and then derive a contradiction.

$\lambda x.\, e_1 \longrightarrow e'$ for some e'	Assumption

Now we can examine rules for reduction and see there is only one rule that could possibly arrive at this conclusion, namely red/lam. Therefore, there must also be a derivation of the premise. We call this kind of reasoning *inversion* because we obtain a derivation of the premise from knowing the conclusion is true.

> *In my experience, incorrect application of inversion is the main source of error by novices when constructing "proofs" in the theory of programming languages. To apply inversion, you*
>
> *(a) You must already know that there is the derivation of a judgment, and*
>
> *(b) you must carefully determine* all *the possible rules that could derive this judgment, and*
>
> *(c) you must distinguish all these cases, and finally*
>
> *(d) in each case you may assume that the premise(s) hold.*

In this case, we find that

$e_1 \to e_1'$ for some e_1' and $e' = \lambda x.\, e_1'$	By inversion

But this is a contradiction, since we already learned from the induction hypothesis that $e_1 \not\longrightarrow$.

In summary:

$e_1 \not\longrightarrow$	By induction hypothesis
$\lambda x.\, e_1 \longrightarrow e'$ for some e'	Assumption
$e_1 \to e_1'$ for some e_1' and $e' = \lambda x.\, e_1'$	By inversion
Contradiction	Since $e_1 \not\longrightarrow$
$\lambda x.\, e_1 \not\longrightarrow$	Since $\lambda x.\, e_1 \longrightarrow$ is contradictory

Case:

$$\frac{}{x\ \mathit{nf}}\ \mathsf{nf/var}$$

where $e = x$. Then

$x \longrightarrow e'$ for some e'	Assumption
Contradiction	By inversion (no rule allows this conclusion)
$x \not\longrightarrow$	since $x \longrightarrow$ is contradictory

Case:

$$\frac{e_1 \neq \lambda \_ \quad e_1 \; nf \quad e_2 \; nf}{e_1 \, e_2 \; nf} \; \mathsf{nf/app}$$

where $e = e_1 \, e_2$. Then

$e_1 \not\longrightarrow$	By induction hypothesis
$e_2 \not\longrightarrow$	By induction hypothesis
$e_1 \, e_2 \longrightarrow e'$ for some e'	Assumption

At this point we would like to apply inversion, and there are three rules matching the conclusion $e_1 \, e_2 \longrightarrow e'$.

$e_1 \, e_2 \longrightarrow e_1' \, e_2$	
where $e_1 \longrightarrow e_1'$	First subcase (rule $\mathsf{red/app}_1$)
Contradiction	Since $e_1 \not\longrightarrow$
$e_1 \, e_2 \not\longrightarrow$	since $e_1 \, e_2 \longrightarrow$ is contradictory

$e_1 \, e_2 \longrightarrow e_1 \, e_2'$	
where $e_2 \longrightarrow e_2'$	Second subcase (rule $\mathsf{red/app}_2$)
Contradiction	Since $e_2 \not\longrightarrow$
$e_1 \, e_2 \not\longrightarrow$	since $e_1 \, e_2 \longrightarrow$ is contradictory

$(\lambda x.\, e_3) \, e_2 \longrightarrow [e_2/x]e_3$	
where $e_1 = (\lambda x.\, e_3)$	Third subcase (rule beta)
Contradiction	Since $e_1 \neq \lambda \_$ by the first premise of $\mathsf{nf/app}$

□

Exercises

Exercise 1 Fill in the blanks in the following typing judgments so the resulting judgment holds, or indicate there is no way to do so. You do not need to justify your answer or supply a typing derivation, and the types do not need to be "most general" in any sense. Remember that the function type constructor associates to the right, so that $\tau \to \sigma \to \rho = \tau \to (\sigma \to \rho)$.

(i) $\boxed{} \vdash y\, x : \alpha$

(ii) $\boxed{} \vdash x\, x : \boxed{}$

(iii) $\cdot \vdash \boxed{} : (\alpha \to \alpha) \to \alpha$

(iv) $\cdot \vdash (\lambda z.\, z)\, (\lambda x.\, \lambda y.\, \lambda p.\, p\, x\, y) : \boxed{}$

(v)

$$\cdot \vdash \lambda f.\, \lambda g.\, \lambda x.\, (f\, x)\, (g\, x)$$
$$: (\alpha \to \boxed{}) \to (\alpha \to \boxed{}) \to (\alpha \to \boxed{})$$

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.