Lecture Notes on Simple Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 3
Tue Sep 2, 2025

1 Introduction

Since Church's original attempt at a pure calculus of functions at the foundations of mathematics was shown to be inconsistent, he developed the notion of *type* for his calculus, following the ideas of Whitehead and Russell [1910–13] who had developed a *ramified theory of types*. He removed some of the complications and called the resulting system the *simple theory of types* [Church, 1940]. The simple theory of types has been proved consistent in multiple ways (see, for example, Andrews [1986]) and is therefore sufficient to avoid a version of Russells's paradox. Versions of this theory are still at the core of systems for the formalization of mathematics like HOL, HOL Light, and Isabelle/HOL.

In this course we do not pursue the simple theory of types as a *logical theory* but look once again at its influence in the theory of programming languages. From either perspective it is quite natural. A function that takes argument of type τ and returns results of type σ is given type $\tau \to \sigma$. This allows us to characterize the *normal forms* of various types to a sufficient degree that we can recognize the Booleans and the natural numbers in their Church encodings. We will see in the next lecture that there are some significant limits in their use to characterize *computations* on expressions of these types, so we will have to take some steps beyond simple types.

After introducing the type system itself in a rigorous way, we formalize the notion of *computation* in the form of reduction. Then we tie them together via the theorem of *subject reduction* which requires the all-important technique of rule induction.

The eventual representation theorems for Booleans will say something along the following lines (perhaps to be refined):

> If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ and e does not reduce, then $e = \mathit{true} = \lambda x.\, \lambda y.\, x$ or $e = \mathit{false} = \lambda x.\, \lambda y.\, y$.

How do we use a programming language to perform computation? We provide some definitions and then an expression e. The implementation will then normalize the expression e by reducing it until it can no longer be reduced, at least conceptually. In actuality, it may translate or compile the expression and then execute the compiled form for efficiency.

With types, we check that our expression e has the type we intended (for example, $\cdot \vdash e : \tau$) before we reduce it, because only typed expressions are seen as meaningful. Also, we usually require the context to be empty because we might view a free variable as an "undefined function".

So just as in the representation of Booleans and natural numbers, we focus on the computation with closed terms.

Now we would like to be assured that the result of the computation, that is, the normal form e' of e (with $e \longrightarrow^* e'$) is still of the same type! It would not make sense if we start an expression $e : \mathit{nat}$ and are presented with an answer of $\mathit{true} : \mathit{bool}$. To prevent this situation, we would like to prove for our programming language (so far, just the λ-calculus) that

If $\cdot \vdash e : \tau$ and $e \longrightarrow^* e'$ and e' does not reduce, then $\cdot \vdash e' : \tau$.

We usually breaks this down into a *subject reduction* also called *type preservation* for the single-step reduction relation, since multi-step reduction then follows by a simple induction (see Exercise 1).

Conjecture 1 (Subject Reduction, v1) *If $\cdot \vdash e : \tau$ and $e \longrightarrow e'$ then $\cdot \vdash e' : \tau$.*

Before we investigate its proof and refine its statements, let's consider typing and its limits.

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? $\mathit{true} = \lambda x.\, \lambda y.\, x$ 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*:

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

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 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

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

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

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.

$$\cfrac{\cfrac{\cfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\ \mathsf{tp/var}}{x : \alpha \vdash \lambda y.\, x : \alpha \to \alpha}\ \mathsf{tp/lam}}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to (\alpha \to \alpha)}\ \mathsf{tp/lam}$$

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:

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

The worry is that applying the rule tp/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:

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

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 Type Inference

An important property of the typing rules we have so far is that they are *syntax-directed*, that is, for every form of expression there is exactly one typing rule that can be applied. We then perform *type inference* by constructing the skeleton of the typing derivation, filling it with *unknown types*, and reading off a set of equations that have be satisfied between the unknowns. Fortunately, these equations are relatively straightforward to solve with an algorithm called *unification*. This is the core of what is used in the implementation of modern functional languages such as Standard ML, OCaml, or Haskell.

We sketch how this process work, but only for a specific example; we might return to the general algorithm form in a future lecture. Consider the representation of 2:

$$\lambda s.\, \lambda z.\, s\,(s\,z)$$

We know it must have type $?\tau_1 \to (?\tau_2 \to ?\tau_3)$ for some unknown types $?\tau_1$, $?\tau_2$, and $?\tau_3$ where

$$s : ?\tau_1, z : ?\tau_2 \vdash s\,(s\,z) : ?\tau_3$$

Now, $s\,z$ applies s to z, so $?\tau_1 = ?\tau_2 \to ?\tau_4$ for some new $?\tau_4$. Next, the s is applied to the result of $s\,z$, so $?\tau_4 = ?\tau_2$. Also, the right-hand side is the same as the result type of s, so $?\tau_3 = ?\tau_4 = ?\tau_2$. Substituting everything out, we obtain

$$s : ?\tau_2 \to ?\tau_2, z : ?\tau_2 \vdash s\,(s\,z) : ?\tau_2$$

It is straightforward to write down the typing derivation for this judgment. Also, because we did not need to commit to what $?\tau_2$ actually is, we obtain

$$\lambda s.\, \lambda z.\, s\,(s\,z) : (\tau_2 \to \tau_2) \to (\tau_2 \to \tau_2) \qquad \text{for any type } \tau_2$$

We can express this by using a type variable instead, writing

$$\lambda s.\, \lambda z.\, s\,(s\,z) : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

If the the type of an expression contains type variables we can alway substitute arbitrary types for them and still obtain a valid type.

We find that

$$\vdash \overline{n} : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

even though some of the representations (such as $\overline{0} = \mathit{zero}$) also have other types. So our current hypothesis is that this type is a good candidate as a characterization of Church numerals, just as $\alpha \to (\alpha \to \alpha)$ is a characterization of the Booleans.

5 The Limits of Simple Types

We have proposed types as a way to classify functions, fixing their domain and their codomain, and making sure that functions are applied to arguments of the correct type. We also started to observe some patterns, such as *true* : $\alpha \to (\alpha \to \alpha)$ and *false* : $\alpha \to (\alpha \to \alpha)$, possibly using this type to characterize Booleans.

But what do we give up? Are there expressions that cannot be typed? From the historical perspective, this should definitely be the case, because types were introduced exactly to rule out certain "paradoxical" terms such as Ω that do not have a normal form.

One term that is no longer typeable is self-application $\omega = \lambda x.\, x\, x$. As a result, we also can type neither $\Omega = \omega\,\omega$ nor Y, which can be seen as achieving a goal from the logical perspective, but it does give up computational expressiveness. How do we prove that ω cannot be typed? We begin by creating the skeleton of a typing derivation, which is unique due to the syntax-directed nature of the rules (that is, for each language construct there is exactly one typing rule). We highlight in red rules whose constraints on types have not yet been considered. When all the rules are black, we know that every solution to the accumulated constraints leads to a valid typing derivation (and therefore a valid type in the conclusion).

$$\dfrac{\dfrac{\dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{tp/var} \qquad \dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{tp/var}}{x : \boxed{} \vdash x\,x : \boxed{}}\,\textsf{tp/app}}{\cdot \vdash \lambda x.\, x\, x : \boxed{}}\,\textsf{tp/lam}$$

The type in the final judgment must be $?\tau_1 \to ?\tau_2$ for some types $?\tau_1$ and $?\tau_2$.

$$\dfrac{\dfrac{\dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{tp/var} \qquad \dfrac{}{x : \boxed{} \vdash x : \boxed{}}\,\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\,\textsf{tp/app}}{\cdot \vdash \lambda x.\, x\, x : \boxed{?\tau_1 \to ?\tau_2}}\,\textsf{tp/lam}$$

Once the type of a variable is available in the context, this types is propagated upwards unchanged in a derivation, so we can fill in some more of the types.

$$\dfrac{\dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{}}\,\textsf{tp/var} \qquad \dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{}}\,\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\,\textsf{tp/app}$$
$$\dfrac{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}{\cdot \vdash \lambda x.\, x\,x : \boxed{?\tau_1 \to ?\tau_2}}\,\textsf{tp/lam}$$

In the tp/var rules the type of variable is just looked up in the context, so we can fill in those two types as well.

$$\dfrac{\dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\,\textsf{tp/var} \qquad \dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\,\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\,\textsf{tp/app}$$
$$\dfrac{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}{\cdot \vdash \lambda x.\, x\,x : \boxed{?\tau_1 \to ?\tau_2}}\,\textsf{tp/lam}$$

Finally, for the application of the tp/app rule to be correct, the type of x in the first premise must be a function type, expecting an argument of type $?\tau_1$ (the type of x in the second premise) and returning a result of type $?\tau_2$. That is:

$$\dfrac{\dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\,\textsf{tp/var} \qquad \dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\,\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\,\textsf{tp/app}$$
$$\dfrac{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}{\cdot \vdash \lambda x.\, x\,x : \boxed{?\tau_1 \to ?\tau_2}}\,\textsf{tp/lam}$$

provided $?\tau_1 = ?\tau_1 \to ?\tau_2$

Now we observe that there cannot be a solution to the required equation: there are no types τ_1 and τ_2 such that $\tau_1 = \tau_1 \to \tau_2$ since the right-hand side is always bigger (and therefore not equal) to the left-hand side.

To recover from this in full generality we would need so-called *recursive types*. In this example, we see

$$\tau_1 = F\,\tau_1$$

where $F = \lambda\alpha.\, \alpha \to \tau_2$ and we might then have a solution with $\tau_1 = Y\,F$. But such a solution is not immediately available to us. For one thing, we do not have function from types to types such as F. For another, we don't have a Y combinator at the level of types. However, it is perfectly possible to construct recursive types, and we will do so later in the course. We can also think of such recursive types as *infinite types*

$$\tau_1 = \tau_1 \to \tau_2 = (\tau_1 \to \tau_2) \to \tau_2 = ((\tau_1 \to \tau_2) \to \tau_2) \to \tau_2 = \cdots$$

Another way to recover some, but not all of the functions that can be typed in the λ-calculus is to introduce *polymorphism*, which we will also consider.

6 Reduction as a Judgment

Our characterization of normal forms so far is quite simple: they are terms that do not reduce. But this is a *negative* condition, and negative conditions can be difficult to work with in proofs. So we would like a *positive* definition normal forms. Just like typing, we tend to give such definitions in the form of inference rules. The property then holds if the judgment of interest (here, that an expression is normal) can be derived using the given rules. This is a form of *inductive definition*.

Before we get to defining normal forms by rules, we formally define β-reduction by inference rules. Previously, we just stated informally that a step of β-reduction can be "applied anywhere in an expression". Now we write this out. We refer to the last three rules as *congruence rules* because they allow the reduction of a subterm. The judgment here is $e \longrightarrow e'$ (omitting the β for brevity) expressing that e reduces to e'.

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \mathsf{red/beta} \qquad \frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{red/lam}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app}_1 \qquad \frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{red/app}_2$$

These rules are *not* syntax-directed: there are three rules for application (red/beta, $\mathsf{red/app}_1$, and $\mathsf{red/app}_2$), one for λ (red/lam) and none for variables. So if we use them to construct a derivation, we may have to backtrack or presciently make a choice. When constructing a derivation we usually assume e is given and we simultaneously construct a derivation and an expression e' such that $e \longrightarrow e'$. For example, with $K = \lambda x.\, \lambda y.\, x$ and $I = \lambda x.\, x$:

$$\frac{}{((\lambda x.\, \lambda y.\, x)\, K)\, I \longrightarrow \boxed{}}\ ?$$

We realize that this is not a redex at the top level, and we also know that I can't be reduced, so we use the $\mathsf{red/app}_1$ rule:

$$\frac{\dfrac{}{(\lambda x.\, \lambda y.\, x)\, K \longrightarrow \boxed{}}\ ?}{((\lambda x.\, \lambda y.\, x)\, K)\, I \longrightarrow \boxed{}}\ \mathsf{red/app}_1$$

Again, a priori three rules could be applied, but only red/beta will succeed in building a derivation:

$$\frac{\dfrac{}{(\lambda x.\, \lambda y.\, x)\, K \longrightarrow \boxed{}}\ \mathsf{red/beta}}{((\lambda x.\, \lambda y.\, x)\, K)\, I \longrightarrow \boxed{}}\ \mathsf{red/app}_1$$

At this point the structure of the derivation is complete, and we just need to fill in the blanks.

Starting from the top, we get $[K/x](\lambda y.\, x) = \lambda y.\, K$

$$\dfrac{\dfrac{}{(\lambda x.\, \lambda y.\, x)\, K \longrightarrow \lambda y.\, K}\ \mathsf{red/beta}}{((\lambda x.\, \lambda y.\, x)\, K)\, I \longrightarrow \boxed{}}\ \mathsf{red/app_1}$$

The final blank is the result from the previous line applied to I, and we obtain:

$$\dfrac{\dfrac{}{(\lambda x.\, \lambda y.\, x)\, K \longrightarrow \lambda y.\, K}\ \mathsf{red/beta}}{((\lambda x.\, \lambda y.\, x)\, K)\, I \longrightarrow (\lambda y.\, K)\, I}\ \mathsf{red/app_1}$$

7 Subject Reduction

Now we return to the main topic of this lecture, namely subject reduction. Recall our characterization of reduction:

$$\dfrac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{red/lam} \qquad \dfrac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app_1} \qquad \dfrac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{red/app_2}$$

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

And, for reference, here are the typing rules.

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

$$\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{app}$$

Conjecture 2 (Subject Reduction, v1)
If $\cdot \vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\cdot \vdash e' : \tau$.

Proof: (Attempt) [1] Proving this will require some analysis of the derivations of $\cdot \vdash e : \tau$ and $e \longrightarrow e'$. We hope that in all cases, $\cdot \vdash e' : \tau$. But just a proof by cases will not work, because the judgments of typing and reduction are defined *inductively* by a collection of inference rules. That is, for example, the judgment $e \longrightarrow e'$ holds *if and only if* there is a derivation for it. To mirror this inductive definition, we have to carry out a proof by induction. Such proofs are by induction over the structure of the derivation of a judgment. What this means is that for each rule, we get to assume our conjecture for all the premises of the rule (the *induction hypothesis*) and have to prove it for the conclusion. A rule with no premises (such as red/beta) is then a *base case* because we have no inductive hypotheses to work with. Rules with premises correspond to induction steps.

Another way to think about such a proof is: if a property of a judgment *preserved* by all rules of inference and holds for the axioms (the rules with no premises), then it must hold of all judgments.

[1] In lecture, we did actually not attempt to prove this, but went straight to Theorem 3.

So, let's get started. We call this form of induction *rule induction*. In this particular statement, we could use rule induction on the judgment $\cdot \vdash e : \tau$ or $e \longrightarrow e'$. In principle, we might also be able to use induction over the structure of e, e', or τ, but the derivations we have give us more information. So, generally speaking, it is unlikely that we perform induction over expressions or types when we have richer assumptions. Again, a general heuristic says that if we have a syntax-directed judgment and one that is not, it is often better to induct over the judgment that is not syntax-directed. This suggests a proof by rule induction over $e \longrightarrow e'$.

We have to distinguish the various rules for this judgment.

Case:

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app}_1$$

where $e = e_1\, e_2$ and $e' = e_1'\, e_2$. We start by restating what we know in this case.

$\cdot \vdash e_1\, e_2 : \tau$ Assumption

Now we perform a key step, namely *inversion*. We know that $\cdot \vdash e_1\, e_2 : \tau$ and we see that there is only inference rule whose conclusion matches this: tp/app. Since some instance of the rule must have been used, we conclude:

$\cdot \vdash e_1 : \tau_2 \to \tau$ and
$\cdot \vdash e_2 : \tau_2$ for some τ_2 By inversion

We have to be extremely careful, because it looks like we are using a rule of inference in the wrong direction. I guess that's why it is called *inversion*. So always make sure you know that a judgment is true, and there is only one (or, more generally, a finite number of) judgments that could have derived it.

At this point we have a type for e_1 and a reduction for e_1, so we can apply the induction hypothesis.

$\cdot \vdash e_1' : \tau_2 \to \tau$ By ind. hyp.

Now we can just apply the typing rule for application. Intuitively, in the typing for $e_1\, e_2$ we have replaced e_1 by e_1', which is okay since e_1' has the type of e_1.

$\cdot \vdash e_1'\, e_2 : \tau$ By rule tp/app

Case:

$$\frac{e_1 \longrightarrow e_1'}{\lambda x.\, e_1 \longrightarrow \lambda x.\, e_1'}\ \mathsf{red/lam}$$

where $e = \lambda x.\, e_1'$.

We proceed as before, applying inversion to the typing of e.

$\cdot \vdash \lambda x.\, e_1 : \tau$ Assumption
$x : \tau_2 \vdash e_1 : \tau_1$ and $\tau = \tau_2 \to \tau_1$ for some τ_1 and τ_2 By inversion

However, at this point we are stuck because we cannot apply the induction hypothesis! The problem is that in the typing for e_1 the context is not empty.

□

So it seems the property we want does not follow by rule induction! If we get stuck like that, there are usually three possibilities to consider:

(1) The conjecture is false. Maybe the failed proof attempt helps us find a counterexample so we can revise either our definitions or our conjecture.

(2) We need to generalize the induction hypothesis. Maybe the failed proof attempt helps us find the right generalization.

(3) We need to find a suitable lemma. Maybe the failed proof attempt helps us find such a lemma. In this case we can often keep the structure of the proof intact.

In this particular example, the fact that the context was nonempty in an attempted appeal to the induction hypothesis suggests we should allow arbitrary nonempty contexts in our induction hypothesis.

Theorem 3 (Subject Reduction, v2)
If $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : \tau$.

Proof: We try again: rule induction on the derivation of $e \longrightarrow e'$.

Case:

$$\frac{e_1 \longrightarrow e_1'}{\lambda x.\, e_1 \longrightarrow \lambda x.\, e_1'}\ \mathsf{red/lam}$$

where $e = \lambda x.\, e_1'$. In the critical step we can now appeal to the induction hypothesis using $\Gamma, x : \tau_2$ as our context.

$\Gamma \vdash \lambda x.\, e_1 : \tau$	Assumption
$\Gamma, x : \tau_2 \vdash e_1 : \tau_1$ and $\tau = \tau_2 \to \tau_1$ for some τ_1 and τ_2	By inversion
$\Gamma, x : \tau_2 \vdash e_1' : \tau_1$	By induction hypothesis
$\Gamma \vdash \lambda x.\, e_1' : \tau_2 \to \tau_1$	By rule tp/lam

Case:

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app_1}$$

where $e = e_1\, e_2$. We have to reconsider this case which worked before, now for the generalized induction hypothesis. But it works the same way.

$\Gamma \vdash e_1\, e_2 : \tau$	Assumption
$\Gamma \vdash e_1 : \tau_2 \to \tau$ and $\Gamma \vdash e_2 : \tau_2$ for some τ_2	By inversion

At this point we have a type for e_1 and a reduction for e_1, so we can apply the induction hypothesis.

$\Gamma \vdash e_1' : \tau_2 \to \tau$ By ind. hyp.

Now we can just apply the typing rule for application. Intuitively, in the typing for $e_1\, e_2$ we have replaced e_1 by e_1', which is okay since e_1' has the type of e_1.

$\Gamma \vdash e_1'\, e_2 : \tau$ By rule tp/app

Case:

$$\frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{red/app}_2$$

where $e = e_1\, e_2$. This proceeds completely analogous to the previous case.

Case:

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

where $e = (\lambda x.\, e_1)\, e_2$. In this case we apply inversion twice, since the structure of e is two levels deep.

$\Gamma \vdash (\lambda x.\, e_1)\, e_2 : \tau$ Assumption
$\Gamma \vdash \lambda x.\, e_1 : \tau_2 \to \tau$
and $\Gamma \vdash e_2 : \tau_2$ for some τ_2 By inversion
$\Gamma, x : \tau_2 \vdash e_1 : \tau$ By inversion

At this point we are once again truly stuck, because there is no obvious way to complete the proof.

To Show: $\Gamma \vdash [e_2/x]e_1 : \tau$

Fortunately, the gap that presents itself is exactly the content of the *substitution property*, stated below. The forward reference here is acceptable, since the proof of the substitution property does not depend on subject reduction.

$\Gamma \vdash [e_2/x]e_1 : \tau$ By the *substitution property* (Theorem 4)

□

The last case in the proof of subject reduction represents option (3) in the heuristic list of "ways out" if we get stuck in our induction proof.

Theorem 4 (Substitution Property)
If $\Gamma \vdash e : \tau$ *and* $\Gamma, x : \tau, \Gamma' \vdash e' : \tau'$ *then* $\Gamma, \Gamma' \vdash [e/x]e' : \tau'$

Proof sketch: By rule induction on the derivation of $\Gamma, x : \tau, \Gamma' \vdash e' : \tau'$. Intuitively, in this derivation we can use $x : \tau$ only at the leaves, and there to conclude $x : \tau$. Now we replace this leaf with the given derivation of $\Gamma \vdash e : \tau$ which concludes $e : \tau$. Luckily, $[e/x]x = e$, so this is the correct judgment.

There is only a small hiccup: we have to adjoin the additional variables declared in Γ'. This would be yet another lemma, namely, that we can always add unused variable and type to a typing derivation, a property called *weakening*. Since it is straightforward to prove, once again by induction, we will not formalize it even if we have multiple occasions to use it. □

We recommend you write out the cases of the substitution property in the style of our other proofs, just to make sure you understand the details.

The substitution property is so critical that we may elevate it to an intrinsic property of the turnstile ($\vdash$). Whenever we write $\Gamma \vdash J$ for any judgment J we imply that a substitution property for the judgments in Γ must hold. This is an example of a *hypothetical* and *generic* judgment [Martin-Löf, 1983]. We may return to this point in a future lecture, especially if the property appears to be in jeopardy at some point. It is worth remembering that, while we may not want to prove an explicit substitution property, we still need to make sure that the judgments we define *are* hypothetical/generic judgments.

Looking back at Conjecture 2, this also holds as a consequence of Theorem 3: we just use it for $\Gamma = (\cdot)$. So it was not wrong, we just couldn't prove it the way we wanted because the induction hypothesis for our rule induction wasn't strong enough.

Exercises

Exercise 1 We can definen the reflexive and transitive closure ($\longrightarrow^*$) of the single-step reduction ($\longrightarrow$) with the following rules:

$$\frac{}{e \longrightarrow^* e}\ \mathsf{red}^*/\mathsf{refl} \qquad \frac{e_1 \longrightarrow^* e_2 \quad e_2 \longrightarrow^* e_3}{e_1 \longrightarrow^* e_3}\ \mathsf{red}^*/\mathsf{trans} \qquad \frac{e_1 \longrightarrow e_2}{e_1 \longrightarrow^* e_2}\ \mathsf{red}^*/\mathsf{step}$$

However, it is more common to define multistep reduction with only two rules, as is done for the $\Longrightarrow$ judgment below:

$$\frac{}{e \Longrightarrow e}\ \mathsf{reds}/\mathsf{refl} \qquad \frac{e_1 \longrightarrow e_2 \quad e_2 \Longrightarrow e_3}{e_1 \Longrightarrow e_3}\ \mathsf{reds}/\mathsf{step}$$

Prove by rule induction that these two definitions are equivalent in the sense that $e \Longrightarrow e'$ iff $e \longrightarrow^* e'$.

Exercise 2 Recall the relation $\longrightarrow^*$ defined in Exercise 1. Prove by rule induction that if $\Gamma \vdash e : \tau$ and $e \longrightarrow^* e'$ then $\Gamma \vdash e' : \tau$. Here (as in general in the course), you may use theorems we have proved in the course (lecture or notes).

Exercise 3 Recall the relation $\Longrightarrow$ defined in Exercise 1. Prove by rule induction that if $\Gamma \vdash e : \tau$ and $e \Longrightarrow e'$ then $\Gamma \vdash e' : \tau$. Here (as in general in the course), you may use theorems we have proved in the course (lecture or notes).

Exercise 4 Define a new single-step relation $e \mapsto e'$ which means that e reduces to e' by *leftmost-outermost reduction*, using a collection of inference rules. Recall that I claimed this strategy is *sound*

(it only performs β-reductions) and *complete for normalization* (if e has a normal form, we can reach it by performing only leftmost-outermost reductions). Prove the following statements about your reduction judgment:

(i) If $e \mapsto e'$ then $e \longrightarrow e'$.

(ii) $\mapsto$ is *small-step deterministic*, that is, if $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

You should interpret $=$ as α-equality, that is, the two terms differ only in the names of their bound variables (which we always take for granted).

References

Peter B. Andrews. *An Introduction to Mathematial Logic and Type Theory: To Truth Through Proof*. Academic Press, Orlando, Florida, 1986.

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

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983. URL http://www.hf.uio.no/ifikk/forskning/publikasjoner/tidsskrifter/njpl/vol1no1/meaning.pdf.

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