When we introduced type systems, we asserted that STLC is fundamentally less powerful than untyped λ-calculus, because every well-typed STLC program terminates. We proved that we required fancy type constructs like recursive types to give a type to Ω, but we did not prove that there was no other way to generate an infinite loop. Today we will see how to do that.

Recall the definition of STLC.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \mathsf{unit} \mid \tau_1 \to \tau_2 \\ \text{Terms} & e & ::= & () \mid x \mid e_1\ e_2 \mid \lambda x{:}\tau.\, e \\ \text{Values} & v & ::= & () \mid \lambda x{:}\tau.\, e \end{array}$$

$$\text{[UNIT]}\ \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \text{[VAR]}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{[ABS]}\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad \text{[APP]}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$E \quad ::= \quad [\cdot] \mid E\ e \mid v\ E \qquad \text{[CTX]}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \text{[}\beta\text{-REDUCE]}\ \frac{}{(\lambda x{:}\tau.\, e)\ v \longrightarrow e[x \mapsto v]}$$

1 Strong Normalization

Recall that a *normal form* is a λ-term that cannot take any more steps. The Church–Rosser property (confluence) proves that, if a term has a normal form, that normal form is unique up to α-equivalence. It does not, however, say anything about whether or not a term has a normal form.

Definition 1 (Strong Normalization)**.** A term e is *strongly normalizing* if every β-reduction sequence eventually produces a normal form—it eventually terminates.[1]

A *language* is strongly normalizing if every term in that language is strongly normalizing.

To say that we cannot write an infinite loop in STLC is then to say that STLC—the language of well-typed closed terms—is strongly normalizing. That is, $\vdash e : \tau$ implies that every reduction sequence of e eventually terminates, which we will write $e{\Downarrow}$.

2 Logical Relations

We will prove that STLC is strongly normalizing using a technique called *logical relations*. While they are not strictly necessary for this proof, they are a powerful technique that makes the proof relatively simple, and this context provides a nice introduction to them.

A logical relation is a relation on terms parameterized by a type. It defines a semantic meaning for the types, so we use the notation $[\![\tau]\!]$, as this is our normal syntax for a denotation, or meaning. To prove strong normalization, we will use a *unary* logical relation—it only relates one value—meaning we can think of the relation as a predicate over expressions parameterized by a type. In general, logical relations can have any arity, and binary logical relations are common.

When defining a logical relation, there are three principles we typically follow.

[1] There is also a notion of *weak normalization* which says that *some* reduction sequence terminates.

1. The relation should contain only terms of interest. For this proof, we are only concerned with closed well-typed terms, so $e \in [\![\tau]\!]$ should imply that $\vdash e : \tau$.
2. The property of interest (in this case strong normalization) should be baked into the relation.
3. The property of interest should be preserved by elimination forms for each type τ. Intuitively, this means we can continue after consuming a value of type τ.

To prove strong normalization of STLC, we can follow these principles and define the relation as follows.

Definition 2 (Normalization Logical Relation)**.**

$$\begin{array}{rcl} e \in [\![\mathsf{unit}]\!] & \stackrel{\triangle}{\iff} & \vdash e : \mathsf{unit} \quad \wedge \quad e\Downarrow \\ e \in [\![\tau_1 \to \tau_2]\!] & \stackrel{\triangle}{\iff} & \vdash e : \tau_1 \to \tau_2 \quad \wedge \quad e\Downarrow \quad \wedge \quad \forall e' \in [\![\tau_1]\!].\, (e\ e') \in [\![\tau_2]\!] \end{array}$$

It is simple to check that the three principles above are satisfied by this definition.

Notably, this definition may at first appear circular, as the definition of $[\![\tau_1 \to \tau_2]\!]$ refers to the logical relation itself. However, it is not. Because $[\![\tau_1 \to \tau_2]\!]$ is defined in terms of $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$, and τ_1 and τ_2 are strict subterms of $\tau_1 \to \tau_2$, the relation remains well-defined. This would not be true if we added recursive types, in which case more complicated techniques would be necessary to make the logical relation well-defined (and also strong normalization would be false).

3 Proving Normalization of STLC

To prove strong normalization of STLC using the relation above, we need to prove two facts:

1. If $\vdash e : \tau$ then $e \in [\![\tau]\!]$.
2. If $e \in [\![\tau]\!]$, then $e\Downarrow$.

Because we followed design principle (2) above, the second point here is totally trivial. So it remains to prove the first, which is generally called the *fundamental theorem* of the logical relation.

We might try to prove the fundamental theorem by induction on the typing judgment $\vdash e : \tau$. The UNIT case is immediate, the VAR case cannot happen, and the APP case is a straightforward application of induction. Unfortunately, ABS poses a problem. To prove this case, we need to show that $\vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2$, which holds by assumption, $(\lambda x{:}\tau_1.\, e)\Downarrow$, which is immediate, and $\forall e' \in [\![\tau_1]\!].\, (\lambda x{:}\tau_1.\, e)\ e' \in [\![\tau_2]\!]$. This last condition poses a problem. The premise of ABS gives us $x{:}\tau_1 \vdash e : \tau_2$, but our inductive hypothesis only applies when $\Gamma = \varnothing$, so it tells us nothing about the behavior of e.

To fix this problem, we need to generalize our inductive hypothesis to allow for open terms! However, open terms can get stuck in general, and our theorem only talks about closed terms. So what we really care about is *all closed terms that we can acquire by substituting free variables for values in our relation*. We do this with a substitution γ, and use the notation $\gamma(e) = e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$.

To ensure that a substitution γ substitutes precisely the relevant variables and all values are in the logical relation, we extend our logical relation to a relation on substitutions γ parameterized by a typing context Γ. With a minor, but fairly common, abuse of notation, we define

$$\gamma \in [\![\Gamma]\!] \stackrel{\triangle}{\iff} \mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma) \ \wedge\ \forall x \in \mathrm{dom}(\gamma).\, \gamma(x) \in [\![\Gamma(x)]\!]\,.$$

We can now use this to generalize our theorem to the full version of the fundamental theorem.

Theorem 1 (Fundamental Theorem)**.** *If $\Gamma \vdash e : \tau$, then for any substitution $\gamma \in [\![\Gamma]\!]$, $\gamma(e) \in [\![\tau]\!]$.*

The proof will rely on standard substitution and preservation lemmas that we will not prove. Both are straightforward.

Lemma 1 (Type Substitution)**.** *If* $\Gamma \vdash e : \tau$ *and* $\gamma \in [\![\Gamma]\!]$*, then* $\vdash \gamma(e) : \tau$*.*

Lemma 2 (Preservation of Inclusion)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow^* e'$*, then* $e \in [\![\tau]\!]$ *if and only if* $e' \in [\![\tau]\!]$*.*

Proof of Theorem 1. This will be a proof by induction on the derivation of $\Gamma \vdash e : \tau$.

Case Unit**:** Here $e = ()$ and $\gamma(e) = \mathsf{unit}$, $()$ cannot step.

Case Var**:** Here $e = x \in \mathrm{dom}(\Gamma)$ and $\tau = \Gamma(x)$. Since $\gamma \in [\![\Gamma]\!]$, that means $x \in \mathrm{dom}(\gamma)$, so

$$\gamma(e) = \gamma(x) \in [\![\Gamma(x)]\!] = [\![\tau]\!] .$$

Case App**:** Here $e = e_1\ e_2$, and $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash e_2 : \tau_1$. By induction, $\gamma(e_1) \in [\![\tau_1 \to \tau_2]\!]$ and $\gamma(e_2) \in [\![\tau_2]\!]$. Noting that substitution distributes across application and the definition of the logical relation on function types proves

$$\gamma(e_1\ e_2) = \gamma(e_1)\ \gamma(e_2) \in [\![\tau_2]\!] .$$

Case Abs**:** This is the most complicated case.

Unfolding the definition of $[\![\tau_1 \to \tau_2]\!]$ from Definition 2 tells us that we need to show,

1. $\vdash \gamma(\lambda x : \tau_1 .\ e) : \tau_1 \to \tau_2$,
2. $\gamma(\lambda x : \tau_1 .\ e)\Downarrow$, and
3. if $e' \in [\![\tau_1]\!]$, then $\gamma(\lambda x : \tau_1 .\ e)\ e' \in [\![\tau_2]\!]$.

The first follows from Lemma 1. The second is immediate because we already have a value. That leaves us a need to prove (3).

Because $e' \in [\![\tau_1]\!]$, we know that $e' \longrightarrow^* v$. Additionally, $\gamma(\lambda x : \tau_1 .\ e) = \lambda x : \tau_1 .\ (\gamma - \{x\})(e)$, meaning

$$\begin{array}{rcl} \gamma(\lambda x : \tau_1 .\ e)\ v & = & (\lambda x : \tau_1 .\ (\gamma - \{x\})(e))\ v \\ & \longrightarrow & ((\gamma - \{x\})(e))[x \mapsto v] \\ & = & (\gamma[x \mapsto v])(e). \end{array}$$

Moreover, Lemma 2 proves that $v \in [\![\tau_1]\!]$, meaning $\gamma[x \mapsto v] \in [\![\Gamma, x : \tau_1]\!]$.

The premise of the Abs rule is that $\Gamma, x : \tau_1 \vdash e : \tau_2$, meaning our inductive hypothesis immediately proves $(\gamma[x \mapsto v])(e) \in [\![\tau_2]\!]$. Therefore,

$$\gamma(\lambda x : \tau_1 .\ e)\ e' \longrightarrow^* (\gamma[x \mapsto v])(e) \in [\![\tau_2]\!] .$$

The assumption that $e' \in [\![\tau_1]\!]$ means $\vdash e' : \tau_1$. Coupled with the proof above of point (1), App proves $\vdash \gamma(\lambda x : \tau_1 .\ e)\ e' : \tau_2$, so Lemma 2 completes the case. □

A special case of this lemma is the statement we originally cared about. By taking $\Gamma = \varnothing$, we know that if $\vdash e : \tau$ then $e \in [\![\tau]\!]$. Because the logical relation was constructed such that $e \in [\![\tau]\!]$ implies that e is strongly normalizing, this theorem proves that STLC is strongly normalizing.

4 Extending the Proof

We have seen how to add a variety of features to languages, so one might naturally wonder if this proof and the logical relations techniques extend to those. Both extend in a very straightforward way to numeric types, booleans, sums, and products. That proof is left as an exercise. With some (fairly nontrivial) modifications to ensure things are well-defined, the technique of logical relations extends to recursive types and state, though the proof above very much does not as both introduce nontermination.

Polymorphism is a bit more interesting. System F *is* strongly normalizing, and the standard proof makes use of logical relations. However, handling polymorphism is very complicated, so the associated logical relation and the proof are beyond the scope of this course.