Logical Relations

What are logical relations? They are a proof method. This is a lot like the method you use to prove type safety, by progress and proservation, which every PL researcher knows.

This is a proof method you really ought to know.

THINGS YOU CAN PROVE

unary
- Strong normalization of STLC
  logical relations ✓ induction X ← doesn't work
- Type Safety
  esp. non-terminating languages
  SLTC + recursive types

compiler correctness: major topic. Not Compcert!

binary
- Equivalence of programs

to show correctness of optimizations... or do a transformation (lambda lifting) where the type of expressions CHANGES (or an IR-to-IR transformation) ★

- Non-interference
  (confidentiality in IFC systems)

$p : int^L \times int^H \to int^L$

$p(v_L, v_{H1}) \approx_L p(v_L, v_{H2})$

int x : int^L
int y : int^H
$x = y$ X error
if $y > 0$ then $x = 0$ else $x = 1$ X error

As long as you feed in the same low security input, no matter what the high security inputs are, the low security output should "look" the same.

good for navigating the literature

Q: Can you do this in cryptography?
A: This is a pretty simplistic non-interference theorem. I'm not sure; you should Google it, I am not sure what technicalities you run into. In crypto, you have situations where there is no leak of information at all, and some where you leak a little bit of information.

There are two categories of logical relations: unary logical relations (strong normalization and type safety) and binary logical relations (equivalence and non-interference). Unary relations (predicates) have a single expression with a property P(e) we are interested in; these relations have always been built out of types (though nowadays, you can do relations for untyped languages). Binary relations are about two terms P(e, e'); both have the same type and are related; possibly equivalence, but also different. These are more sophisticated than unary relations.

Note about type safety

The very first time logical relations were used were for strong normalization. For the longest time, they were defined by induction on types, and when you had recursive types or mutable references (which cause cycles in memory), there were circular advanced typing features which caused problems for logical relations. So how do you break this circularity? The key is STEP-INDEXED LOGICAL RELATIONS.

SLTC (CBV)

$\tau ::= \text{bool} \mid \tau_1 \to \tau_2$

$e ::= \text{true} \mid \text{false} \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid x \mid \lambda x{:}\tau.e \mid e_1\ e_2$

$v ::= \text{true} \mid \text{false} \mid \lambda x{:}\tau.e$

evaluation context → $E ::= [\cdot] \mid \text{if } E \text{ then } e_1 \text{ else } e_2 \mid E\ e \mid v\ E$

operational semantics

$\text{if true then } e_1 \text{ else } e_2 \longmapsto e_1$

$\text{if false then } e_1 \text{ else } e_2 \longmapsto e_2$

$(\lambda x{:}\tau.e)\ v \longmapsto e[v/x]$

$$\frac{e \longmapsto e'}{E[e] \longmapsto E[e']}$$

typing rules

$\Gamma ::= \cdot \mid \Gamma, x{:}\tau$

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

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

Strong normalization: $\cdot \vdash e : \tau \Rightarrow e\Downarrow$

$e \Downarrow v \overset{\text{def}}{=} e \overset{*}{\longmapsto} v$

$e\Downarrow \overset{\text{def}}{=} \exists v.\ e \Downarrow v$

Q: Isn't this weak normalization?
A: This happens every year! Well, because of determinacy, it's strong. Recall weak normalization says there exists a path which terminates; since the language is deterministic, all paths do this.

$\boxed{P_\tau(e)}$

- e has type τ
- e has property of interest
- L.R. is preserved by elimination form for type τ

★ the strengthened inductive hypothesis

So what is the problem with induction? We might say the induction hypothesis is normalizes(e). We will fail when we get to the application rule:

$$e_1\ e_2 \longmapsto^* (\lambda x.e')\ e_2 \longmapsto^* (\lambda x.e')\ v \longmapsto [v/x]e'$$

does this normalize?

The idea of logical relations is we want a STRONGER induction hypothesis, that is to say, there is some structure of the expression under the binder which we want to capture. Define a new predicate:

$SN_\tau(e)$ e is strongly normalizing and has type tau

$$SN_{bool}(e) \iff \vdash e : bool \wedge e\Downarrow$$

$$SN_{\tau_1 \to \tau_2}(e) \iff \vdash e : \tau_1 \to \tau_2 \wedge e\Downarrow \wedge \left(\forall e_1.\, SN_{\tau_1}(e_1) \Rightarrow SN_{\tau_2}(e\ e_1)\right)$$

When you do the elim-form for that type, the logical relation will be preserved. The elimination form for bools is if...then...else, while the elimination rule for functions is application. For booleans, we don't need anything else, since the relation is "obviously" preserved. (Recall this worked for normal induction.)

The function has something built-in which makes sure the logical relation is preserved by application. So if you give me a strongly normalizing argument, I will get a result which is strongly normalizing.

Recall that we said LRs are defined on induction of the type. So let's check that things are well founded. Bool has no problem; for T1 -> T2, I'm using SN of T1 and SN of T2; these are all smaller, so everything is OK! (The inner references to LRs have smaller types.)

just like generalizing an inductive hypothesis

Proof idea to show every well-typed thing terminates:

A) $\cdot \vdash e : \tau \Rightarrow SN_\tau(e)$ (hard part!)

B) $SN_\tau(e) \Rightarrow e\downarrow$ (trivial by induction on τ: immediate by definition)

Q: The condition you wrote down, the way I read the very last part is that if the argument is strongly normalizing, then the result strongly normalizes.
A: Well, it says that the application strongly normalizes. We'll see in just a minute that strong normalization is preserved by reduction. As we reduce it, it will remain in SN.
C: This is a bigger thing, because T2 could be a big type. This is not really normalization, but reducibility. You need more than just strong normalization of e1, which is that if it's applied.
A: Yes, that is what I am trying to capture. The third clause gives me my stronger induction hypothesis.
C: Well, there's a confusion here, because you're calling SN strong normalization, and it's not actually strong normalization.
A: You're right, I'm calling it SN to be suggestive, it is NOT strong normalization. It gives me a strong induction hypothesis, and it implies strong normalization.

It will be helpful to mentally replace "strong normalization" with "SN" in later sections.

So, how do we prove A? If we try induction on the derivation of the type, we will still run into the lambda typing rule, since it's no longer a closed context. We CANNOT prove A directly. We need a more general lemma which works with open terms. So we need to generalize AGAIN.

Lemma (SN preserved by backward and forward reduction)

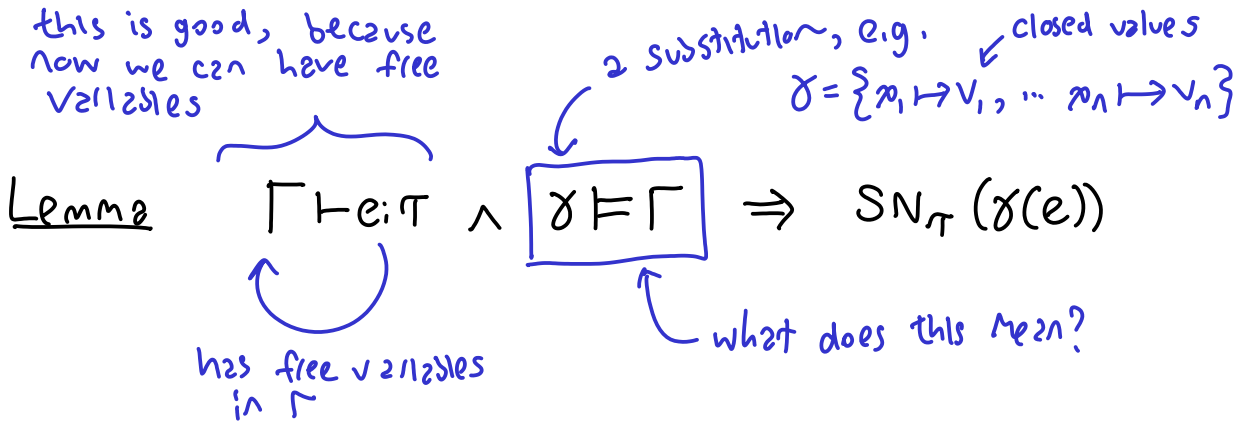
Suppose you have $\cdot \vdash e : \tau$ and $e \longmapsto e'$. Then:

a) $SN_\tau(e') \Rightarrow SN_\tau(e)$

b) $SN_\tau(e) \Rightarrow SN_\tau(e')$

we have managed to do induction on the operational rules!

Ex) Prove at least one of these. Hard case is application. In (b), you will need the preservation property too.



Idea: we want gamma to be strongly normalizing; similar to lambda, except there are an arbitrary number of free variables.

Another note: the substitution needs to preserve typing; Benjamin proves this in TAPL.

$$\gamma \models \Gamma \stackrel{\text{def}}{=} dom(\gamma) = dom(\Gamma) \wedge \forall x \in dom(\Gamma).\ SN_{\Gamma(x)}(\gamma(x))$$

i.e. give me a substitution, and they all should be strongly normalizing.

Note: aren't values strongly normalizing? Yes: we're punning here, what we actually mean is SN.

Subst lemma $\Gamma \vdash e : \tau \wedge \gamma \models \Gamma \Rightarrow \vdash \gamma(e) : \tau$

Proof by induction on size of gamma, referring to the actual substitution lemma when gamma is size one.

of the lemma at top of previous term

Proof: induction on typing derivation

case $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$:

suppose $\gamma \models \Gamma$

contains $\{ \cdots x \mapsto v \cdots \}$ thus

show $SN_\tau(\gamma(x))$ is v so show $SN_\tau(v)$

but this is by defn ✓

trivial

since we got stuck on it

case $\dfrac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$:

suppose $\gamma \models \Gamma$

show $SN_\tau(\gamma(e_1\ e_2))$

By induction hypothesis: $SN_{\tau_2 \to \tau}(\gamma(e_1))$

$SN_{\tau_2}(\gamma(e_2))$

this is interesting!

$e' := \gamma(e_2)$

$$SN_{\tau_2 \to \tau}(\gamma(e)) \iff \gamma(e) : \tau_2 \to \tau \wedge \gamma(e) \Downarrow \wedge \left(\forall e' . SN_{\tau_2}(e') \Rightarrow SN_\tau(\gamma(e)\ e') \right)$$

very close to what we want

thus conclude $SN_\tau(\gamma(e_1)\ \gamma(e_2))$

$= SN_\tau(\gamma(e_1\ e_2))$ ✓

Wait, this was really easy! What got harder? The function case.

case $\dfrac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}$

Supp $\gamma \models \Gamma$

Show $SN_{\tau_1 \to \tau_2}(\gamma(\lambda x{:}\tau.\, e))$

i.e. $\lambda x{:}\tau.\, \gamma(e)$

① $\vdash \gamma(\lambda x{:}\tau_1.\, e) : \tau_1 \to \tau_2$ by substitution lemma ✓

② $\gamma(\lambda x{:}\tau.\, e) \Downarrow$ easy

Q: When you're pushing the substitution, don't you need some technical details?
A: At the very outset, I said that whenever I write Gamma, x : tau, x doesn't occur in Gamma. So that's fine.

HARD

③ Supp e_1 s.t. $SN_{\tau_1}(e_1)$

Show $SN_{\tau_2}((\lambda x{:}\tau.\, \gamma(e))\ e_1)$

we need this to be a value. But we know $SN_{\tau_1}(e_1)$

The fact that generalizing the induction hypothesis made the introduction case harder and the elimination case easier is a pattern that shows up again and again in logical relations. If you don't get it right, one will work while the other will fall apart.

So, let $e_1 \Downarrow v_1$ (the existential lemma)

by forward reduction lemma, know $SN_{\tau_1}(v_1)$

$$(\lambda x{:}\tau.\, \gamma(e))\ e_1 \longmapsto^* (\lambda x{:}\tau.\, \gamma(e))\ v_1 \longmapsto \gamma(e)[v_1/x]$$

suffices to show $SN_{\tau_2}(\gamma(e)[v_1/x])$ by backwards red.

$\equiv SN_{\tau_2}(\gamma[x \mapsto v_1](e))$

we "run" the opsems forward and then backwards

Notice we haven't applied our induction hypothesis yet.

What is the inductive hypothesis?

Have: $\Gamma, x:\tau_1 \vdash e:\tau_2$ Need: $\gamma \models \Gamma, x:\tau_1$

notice this gamma is bigger

$\gamma \not\models \Gamma, x:\tau_1$ not true

So we need a new γ, e.g.

$\gamma[x \mapsto ?] \models \Gamma, x:\tau_1$

For this we need

$\gamma \models \Gamma$ (easy)

(from $\models$ dfn.) $\wedge$ $SN_{\tau_1}(?)$

let's have this be v_1

So is $SN_{\tau_1}(v_1)$? Yes!

now we get the result of the IH:

$SN_{\tau_2}((\gamma[x \mapsto v_1])(e))$ → now we're done

the mysterious case is the λ-case and the app-case.

If you can do that, you'll be set for life!

To sum up: we wanted to use the induction hypothesis, but we couldn't immediately do it. We had an application, but we couldn't stick it into the context until it became a value. This is actually a valuable source of operational insight. We were in the lambda case and we wanted to apply it. So we carefully followed the operational semantics, and reduced it to a value, then beta-reduced it, and then finally proved something about the term at the end of the beta-reduction step,

Logical relations are like a big jigsaw puzzle. If you define your relations just right, the proofs will just work out. Some people say the proofs are boring, but if you don't do the proofs, then you won't know why it's hard.

Q: Have you ever tried this with a non-deterministic language?
A: Well, I've done it with CBN and CBV (it's just a different operational semantics) but I haven't done it for nondeterminism.

Q: What about polymorphism?
A: Let's talk about that on Saturday. Polymorphism is...interesting. This takes us to work by Girard.

Some historical context: strong normalization was used for this proof. This is often called Tait's method, since he was the one who did it. Reynolds and Girard came up with the polymorphic lambda calculus (or the second-order lambda calculus); they ended up being the same. But Girard was also famous for the candidates method, where he showed strong normalization for System F using a logical relation technique, called "reducibility". We are not going to do Girard's method, but we will talk about parametricity for System F.

Bonus: cheat sheet

One might say that the key to understanding why logical relations works is definition unwinding. This is not too surprising, since the INGENUITY of logical relations comes from the definition of the logical relation. So I've collected all the definitions in one place for easy reference here.

The relation:

$$SN_{bool}(e) := \vdash e : bool \wedge e\Downarrow$$

$$SN_{\tau_1 \to \tau_2}(e) := \vdash e : \tau_1 \to \tau_2 \wedge e\Downarrow \wedge (\forall e_1 . SN_{\tau_1}(e_1) \Rightarrow SN_{\tau_2}(e\ e_1))$$

Substitutions on contexts:

$$\gamma \vDash \Gamma := dom(\gamma) = dom(\Gamma) \wedge \forall x \in dom(\Gamma).\ SN_{\Gamma(x)}(\gamma(x))$$

The lemmas:

SN-preserved by backwards/forwards reduction

Given $\cdot \vdash e : \tau$ and $e \longmapsto e'$

1) $SN_\tau(e') \Rightarrow SN_\tau(e)$
2) $SN_\tau(e) \Rightarrow SN_\tau(e')$

Subst lemma

$$\Gamma \vdash e : \tau \wedge \gamma \vDash \Gamma \Rightarrow \cdot \vdash \gamma(e) : \tau$$

The core lemma

$$\Gamma \vdash e : \tau \wedge \gamma \vDash \Gamma \Rightarrow SN_\tau(\gamma(e))$$