1 TLDR

Logical Relations is a proof technique.

What: A logical relation $\mathbb{R}$ is a binary relation on terms within (or across) programming languages. Similarly, a logical predicate is a unary relation on terms in a language.

- For example, the pair $(\lambda x.\, x\ ,\ \lambda y.\, y)$ might be in a logical relation $\mathbb{R}$ for α-equivalence. This could be written as $(\lambda x.\, x\ ,\ \lambda y.\, y) \in \mathbb{R}$, or $(\lambda x.\, x\ ,\ \lambda y.\, y) \in \mathbb{R}$, or $(\lambda x.\, x)\ \mathbb{R}\ (\lambda y.\, y)$.

Why: Logical relations help prove properties like contextual equivalence.

When: If you need a *stronger hypothesis* in your proof by induction on the *operational semantics* of a language, setting up a *syntax*-directed logical relation might help.

- Read on to hear a few success stories!

Amal Ahmed's summer school slides give an excellent and less-verbose overview.

2 Game Plan

The point of this discussion is to show that logical relations are a *fundamental* and *powerful* proof technique. To this end,

- We first review the basic techniques for proving things about programs, and note that we cannot prove much without logical relations.
- We walk through a pair of basic logical relations proofs, one using logical predicates and the other using a binary relation, to get a feel for how logical relations are used.
- We critique logical relations and follow the relevant literature up to 2014.

3 Analyzing Programs

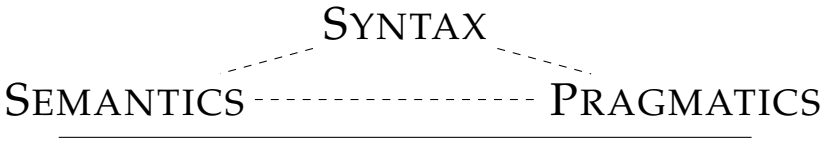
3.1 What is a computer program?

Computer programs have meaning in three senses [16]. First and most obvious is the **syntax** of a program. Syntax is the characters on the screen, the raw data that programmers and computers read.

Second is **semantics**. The semantics of a program is the idea it conveys; for instance, the semantic object represented by a chain of nested tuples might be a list. Semantics have little meaning for the computer executing a program, but they are crucial to the humans reading, writing, and reasoning about it.

Finally, and sometimes forgotten, is **pragmatics**. The pragmatics of a program describe how a machine executing the program will behave. Regardless of the ideas or appearance of a program, we should hope it is pragmatically useful when run.

This three-faceted interpretation of languages is important, so here's a picture:



A program

3.2 What might we want to prove about programs?

For now, let's stick with equality. We want to be able to prove that two programs are equivalent.

3.3 How do we prove properties about programs?

Given the three interpretations of a program, three notions of equality are immediate. We could ask for *syntactic equality*: that two programs are built of identical strings of characters. A much less strict notion would be *semantic equality*: that two programs embody the same ideas. Third we could have pragmatic equality: that two programs will operate the same when run by some machine.

This first notion, string equality, is not very useful. The other two are embodied by three fundamental techniques.

Operational Semantics model the execution of a program on an idealized machine. We construct a series of operational rules that describe an interpreter for the language, then reason about the behavior of programs under these rules.

Denotational Semantics represent programs as mathematical objects. Two programs are the same if they represent the same mathematical model; denotations precisely give the "deeper meaning" of a program.

Axiomatic Semantics represent programs by a set of constraining equations. Equal programs satisfy the same pre- and post-conditions. Unlike the operational and denotational rules, which each give only a single model of evaluation, axiomatic semantics are true over *all models* of evaluation [15]. For example, if you extend a programming language with a new keyword, you must add new operational rules and rebuild your denotational models, but the constraining equations still hold.

Today we will focus on the *operational semantics* of a program. In the next section, we consider a simple problem—does evaluation in the simply-typed λ-calculus deterministically terminate?—and attempt an operational proof. We soon find that operational semantics alone are not enough to prove this simple and obvious property. We need some help in the form of a logical relation.

3.4 Getting meta

Remember that there are many ways to reason about a program. Each has some ups and downs, but there is no "one true method" of reasoning about programs. It is important to think about the connections between models.

4 Strong Normalization

The simply-typed λ-calculus augmented with starter Pokémon is described by the following grammar:

$$\begin{array}{rcl} \tau & ::= & \mathit{pkmn} \mid \tau \to \tau \\ e & ::= & v \mid x \mid e\ e \\ v & ::= & \mathsf{Bulbasaur} \mid \mathsf{Squirtle} \mid \mathsf{Charmander} \mid \lambda x : \tau.\, e \\ \Gamma & ::= & \emptyset \mid \Gamma, x \mapsto v : \tau \end{array}$$

The environment Γ relates a variable x to a value v at type τ (you might implement it as a stack of 3-tuples). It only permits well-formed mappings; for example, the application Bulbasaur Charmander would be stuck. We use shorthands $\Gamma, x \mapsto v$ and $\Gamma, x : \tau$ and $\Gamma, v : \tau$ to talk about specify parts of a mapping.

Next we have typing judgments.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma, y : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e' : \tau}{\Gamma \vdash e : \tau \to \tau'} \over \Gamma \vdash e\ e' : \tau'$$

$$\frac{}{\Gamma \vdash \mathsf{Bulbasaur} : \mathit{pkmn}} \qquad \frac{}{\Gamma \vdash \mathsf{Squirtle} : \mathit{pkmn}} \qquad \frac{}{\Gamma \vdash \mathsf{Charmander} : \mathit{pkmn}}$$

Finally, the small-step operational semantics describe call-by-value β-reduction.

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

4.1 Problem statement

We will prove that the simply-typed lambda calculus is *strongly normalizing*. Normalization is a property of a language's operational semantics.

A language is *weakly normalizing* if it satisfies the diamond property; that is, if $\Gamma \vdash e \to v$ and $\Gamma \vdash e \to v'$ then $v = v'$.

Note that this definition assumes that evaluation terminates. If we have an expression $e^{\Uparrow}$ whose evaluation diverges, weak normalization does not tell us anything.

A language is *strongly normalizing* if for all well-typed terms $\Gamma \vdash e : \tau$, there exists a value $\Gamma \vdash v : \tau'$ such that if we begin with e and repeatedly step, we eventually reach v. In other words, the evaluation of e converges; we write this as $e \Downarrow$.

The property that the above τ and τ' are the same type is called *perservation*. Note also that strong normalization is slightly different from *progress*, which says that if we have $\Gamma \vdash e : \tau$ then either e is a value or it can step.

4.2 A first attempt

We might try proving strong normalization by structural induction on terms in our language. Indeed, the first few cases go smoothly:

- If our expression e is one of Bulbasaur, Squirtle, or Charmander, then our expression is already a value. We're done.
- If e is a variable x that is well-typed in context, then we're done because all variables map to values.
- If e is an abstraction $(\lambda x : \tau.\, e')$ then it is strongly normalizing because it is already a value.

However, we cannot prove the case where e is an application $e_1\ e_2$.

To prove: $\Gamma \vdash e_1\ e_2 : \tau$ implies that $(e_1\ e_2) \Downarrow$.

Proof attempt: Examining the operational semantics, we have three subcases: when e_1 takes a step, when e_1 is a value and e_2 takes a step, and finally when both are values and we perform a β-reduction. The first two are settled by the induction hypothesis; that the function and argument are well-typed and strongly normalizing. The value case is problematic. We know that $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash e_2 : \tau_1$, but this tells us nothing about the result of the application!

4.3 Logical Predicates to the rescue

Our first attempt got "stuck" because the proof required an "obvious fact" about terms in the language. Namely, that substitution preserves normalization behavior. Setting up a logical predicate on types in our language adds this fact as an assumption in our proof: the information we need will follow directly from the type of our expressions.

$$\begin{aligned} \mathsf{StronglyNormalizing}(e : \mathit{pkmn}) &\iff e \Downarrow \\ \mathsf{StronglyNormalizing}(e : \tau \to \tau') &\iff (e \Downarrow \ \wedge\ \forall (e' : \tau)\ .\ \mathsf{StronglyNormalizing}(e') \Rightarrow \\ &\qquad\qquad \mathsf{StronglyNormalizing}((e\ e') : \tau') \end{aligned}$$

With this relation in hand, we can finish the case for applications $e_1\ e_2$. From the induction hypothesis, we know that both e_1 and e_2 are strongly normalizing (because they are well-typed, and all well-typed terms are related by StronglyNormalizing, and $\mathsf{StronglyNormalizing}(e)$ implies $e \Downarrow$). Thus we use the logical relation for $\mathsf{StronglyNormalizing}(e_1 : \tau \to \tau')$ to finish the proof.

But there's a catch! Earlier we concluded that abstractions $(\lambda x : \tau.\, e) : \tau \to \tau'$ were strongly normalizing because they are values, but we must now show that for any choice of $e' : \tau$, the substitution $e[e'/x]$ is strongly normalizing. In effect, we must prove $\mathsf{StronglyNormalizing}(e : \tau')$, but because e has a free variable we must quantify over all possible substitutions. The proof of this "detail" is straightforward, but it is important not to forget it!

4.4 Discussion

Our choice of the logical predicate StronglyNormalizing may have seemed arbitrary. It was. The trick with logical relations proofs is to define a relation that:

- Helps finish your proof
- Follows immediately from the operational semantics

In fact, the proof in Section 4.3 omitted a few crucial lemmas relating the logical relation to the operational semantics; for instance, $\mathsf{StronglyNormalizing}(e)$ implies $e \Downarrow$. But the point is that such lemmas are easy to prove—we construct a logical relation precisely because we need help with a nontrivial proof.

5 Modularity

Existential types provide a mechanism for defining interfaces and hiding implementation details. Here is an existential type for picking a starter pokemon:

$$\begin{array}{rcll} \mathsf{I\_CHOOSE\_YOU} & = & \exists\alpha\,.\,\{ & \mathsf{Grass} : \alpha, \\ & & & \mathsf{Water} : \alpha, \\ & & & \mathsf{Fire} : \alpha, \\ & & & \mathsf{choose} : \alpha \to \mathsf{String}\ \} \end{array}$$

Suppose that we extended the simply-typed lambda calculus from Section 4 with existentials, a type String and a **match** statement. We could then implement this module as follows:

$$\begin{array}{rcll} \mathsf{Gen1} & = & \mathsf{provide\ I\_CHOOSE\_YOU}(\mathit{pkmn})\ \{ & \mathsf{Grass} = \mathsf{Bulbasaur}, \\ & & & \mathsf{Water} = \mathsf{Squirtle}, \\ & & & \mathsf{Fire} = \mathsf{Charmander}, \\ & & & \mathsf{choose} = \lambda x.\,\mathbf{match}\ x \\ & & & \quad \mid \mathsf{Grass} \to \textit{“Grass types are peaceful.”} \\ & & & \quad \mid \mathsf{Water} \to \textit{“Water types are agile.”} \\ & & & \quad \mid \mathsf{Fire} \to \textit{“Fire types are fierce.”}\ \} \end{array}$$

From here we might go on to build a whole video game that uses the values Grass, Water, and Fire abstractly. Suppose now that it is the year 2000 and a new generation of Pokémon has been introduced. We will extend our types τ and values v to accomodate the new Pokémon.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mathit{pkmn}' \\ v & ::= & \ldots \mid \mathsf{Chikorita} \mid \mathsf{Totodile} \mid \mathsf{Cyndaquil} \end{array}$$

While updating our old code for the new Pokémon, we will re-implement the interface I_CHOOSE_YOU:

$$\begin{array}{rcll} \mathsf{Gen2} & = & \mathsf{provide\ I\_CHOOSE\_YOU}(\mathit{pkmn}')\ \{ & \mathsf{Grass} = \mathsf{Chikorita}, \\ & & & \mathsf{Water} = \mathsf{Totodile}, \\ & & & \mathsf{Fire} = \mathsf{Cyndaquil}, \\ & & & \mathsf{choose} = \lambda x.\,\mathbf{match}\ x \\ & & & \quad \mid \mathsf{Grass} \to \textit{“Grass types are peaceful.”} \\ & & & \quad \mid \mathsf{Water} \to \textit{“Water types are agile.”} \\ & & & \quad \mid \mathsf{Fire} \to \textit{“Fire types are fierce.”}\ \} \end{array}$$

Unless we can prove that modules Gen1and Gen2are equivalent, we will need to rewrite all the code that depended on the abstract interface. At first glance, this seems obvious. Of course the modules are observably equal—the only difference between them is the choice of concrete type for α, but that choice is hidden within the existential.

Actually proving it is another story. We cannot “dive into” the modules and prove their equivalence at each field because the underlying values are in fact different:

$$\mathsf{Grass} = \mathsf{Bulbasaur} \quad \neq \quad \mathsf{Chikorita} = \mathsf{Grass}$$

The answer is to prove *contextual equivalence* with respect to a logical relation.

5.1 Contextual Equivalence

Contextual (or observational) equivalence reasons about the possible ways an expression e might be used. The first step to proving contextual equivalence, or $\equiv_C$, is to define the notion of an *evaluation context*.

Evaluation contexts describe the shape of terms in a language that can step. For the simply-typed lambda calculus, the contexts are:

$$\mathcal{C} \quad ::= \quad [\cdot] \mid \mathcal{C}\ e \mid v\ \mathcal{C}$$

The symbol $[\cdot]$ is called "the hole". Every context has one hole in it. This hole can be filled by any well-typed expression. Hence if we want to talk about all the ways an observer could use an expression e of type τ, we would quantify over all contexts $\mathcal{C}$ with a hole of type τ. We write $\mathcal{C}[e]$ to represent a context $\mathcal{C}$ whose hole has been filled by an expression e.

There is a strong connection between evaluation contexts and operational rules $e \to e'$. In fact, the "step" rule for contexts replaces the majority of the operational rules a language would otherwise need:

$$\frac{e \to e'}{\mathcal{C}[e] \to \mathcal{C}[e']}$$

Contexts are effectively a separation of concerns. Besides providing a way to reason about observable behaviors, they distill operational rules to the most interesting cases. For the simply-typed lambda calculus, those cases are the rule for contexts and the rule for β-reduction.

5.2 Choosing a logical relation

Now that we have a target, contextual equivalence, we can declare our goal:

To prove: The modules Gen1and Gen2are observationally equal.

The next step is to define observational equivalence at each type in our language. From here on, the vocabulary gets a little fuzzy. Instead of talking about two equal terms, we must talk about two *observably equal* terms (or, terms that are *related* or *behave* the same).

Our types are:

$$\tau \quad ::= \quad \mathit{pkmn} \mid \tau \to \tau' \mid \mathit{pkmn}' \mid \alpha \mid \mathsf{String}$$

That is, the type *pkmn* for original Pokémon, the type $\tau \to \tau$ for functions, the type *pkmn'* for new Pokémon, type variables α, and finally strings String.

Observational equivalence for values of type *pkmn*, *pkmn'*, or String is simple: two values at this type behave the same if they are the same value. Functions are observably equivalent if they take related inputs to related outputs. Type variables are equivalent according to *some choice* of logical relation $\mathbb{R}$. With this intuition, we set up the binary relation ValueRelation to show when two values are related.

Note that ValueRelation is parameterized by another relation, $\mathbb{R}$, that tells when two values are related at type α. The relation $\mathbb{R}$ is abstracting some information—just as the module signature hides the implementation for the type α in modules Gen1 and Gen2, using $\mathbb{R}$ gives us a choice of implementation for α. If we wanted to prove more properties about our language, we could re-use ValueRelation unchanged and simply plug in a new $\mathbb{R}$ to suit our interests.

$$\begin{array}{rcl}
\mathsf{ValueRelation}(\mathit{pkmn}, \mathbb{R}, v_1, v_2) & = & (v_1 = v_2) \wedge (v_1 \in \{\mathsf{Bulbasaur}\ ,\ \mathsf{Squirtle}\ ,\ \mathsf{Charmander}\}) \\
\mathsf{ValueRelation}(\tau \to \tau', \mathbb{R}, f_1, f_2) & = & \forall (v_1, v_2).\mathsf{ValueRelation}(\tau, \mathbb{R}, v_1, v_2) \Rightarrow \mathsf{ValueRelation}(\tau', \mathbb{R}, f_1 v_1, f_2 v_2) \\
\mathsf{ValueRelation}(\mathit{pkmn}', \mathbb{R}, v_1, v_2) & = & (v_1 = v_2) \wedge (v_1 \in \{\mathsf{Chikorita}\ ,\ \mathsf{Totodile}\ ,\ \mathsf{Cyndaquil}\}) \\
\mathsf{ValueRelation}(\alpha, \mathbb{R}, v_1, v_2) & = & \mathbb{R}(v_1, v_2) \\
\mathsf{ValueRelation}(\mathsf{String}, \mathbb{R}, v_1, v_2) & = & (v_1 = v_2)
\end{array}$$

To be fully precise, we also need a rule relating two existential types, but we skip this formal step and instead state it as the goal of our proof. Also, the applications $f\ v$ should techically be judged according to an expression relation, but we skip this detail.

Assume: Some relation $\mathbb{R}$ relating terms of type *pkmn* to terms of type *pkmn'*.

To prove: Given $\mathbb{R}$, each field of module Gen1 is related to the corresponding field of module Gen2.

We are free to instantiate $\mathbb{R}$ however we like, but only one choice of relation will complete the proof:

$$\mathsf{PkmnRelation} \quad = \quad \{(\mathsf{Bulbasaur}, \mathsf{Chikorita}) \ , \ (\mathsf{Squirtle}, \mathsf{Totodile}) \ , \ (\mathsf{Charmander}, \mathsf{Cyndaquil})\}$$

Fix $\mathbb{R} = \mathsf{PkmnRelation}$. Now we have 4 subcases to consider.

To prove: Each field of module Gen1 is related to the corresponding field of module Gen2. i.e.:

- $\mathsf{ValueRelation}(\alpha, \mathsf{PkmnRelation}, \mathsf{Gen1.Grass}, \mathsf{Gen2.Grass})$
- $\mathsf{ValueRelation}(\alpha, \mathsf{PkmnRelation}, \mathsf{Gen1.Water}, \mathsf{Gen2.Water})$
- $\mathsf{ValueRelation}(\alpha, \mathsf{PkmnRelation}, \mathsf{Gen1.Fire}, \mathsf{Gen2.Fire})$
- $\mathsf{ValueRelation}(\alpha \to \mathsf{String}, \mathsf{PkmnRelation}, \mathsf{Gen1.choose}, \mathsf{Gen2.choose})$

The first three cases are simple:

Case Grass :

$$\begin{array}{c}
\mathsf{ValueRelation}(\alpha, \mathsf{PkmnRelation}, \mathsf{Gen1.Grass}, \mathsf{Gen2.Grass}) \\
\Downarrow \\
\mathsf{PkmnRelation}(\mathsf{Gen1.Grass}, \mathsf{Gen2.Grass}) \\
\Downarrow \\
\mathsf{PkmnRelation}(\mathsf{Bulbasaur}, \mathsf{Chikorita}) \\
\Downarrow \\
(\mathsf{Bulbasaur}, \mathsf{Chikorita}) \in \mathsf{PkmnRelation} \\
\Downarrow \\
(\mathsf{Bulbasaur}, \mathsf{Chikorita}) \in \{(\mathsf{Bulbasaur}, \mathsf{Chikorita}), \ldots\}
\end{array}$$

Case Water :

$$\begin{array}{c}
\mathsf{ValueRelation}(\alpha, \mathsf{PkmnRelation}, \mathsf{Gen1.Water}, \mathsf{Gen2.Water}) \\
\Downarrow \\
\mathsf{PkmnRelation}(\mathsf{Gen1.Water}, \mathsf{Gen2.Water}) \\
\Downarrow \\
\mathsf{PkmnRelation}(\mathsf{Squirtle}, \mathsf{Totodile}) \\
\Downarrow \\
(\mathsf{Squirtle}, \mathsf{Totodile}) \in \mathsf{PkmnRelation} \\
\Downarrow \\
(\mathsf{Squirtle}, \mathsf{Totodile}) \in \{(\mathsf{Squirtle}, \mathsf{Totodile}), \ldots\}
\end{array}$$

Case Fire :

$$\begin{array}{c}
\mathsf{ValueRelation}(\alpha, \mathsf{PkmnRelation}, \mathsf{Gen1.Fire}, \mathsf{Gen2.Fire}) \\
\Downarrow \\
\mathsf{PkmnRelation}(\mathsf{Gen1.Fire}, \mathsf{Gen2.Fire}) \\
\Downarrow \\
\mathsf{PkmnRelation}(\mathsf{Charmander}, \mathsf{Cyndaquil}) \\
\Downarrow \\
(\mathsf{Charmander}, \mathsf{Cyndaquil}) \in \mathsf{PkmnRelation} \\
\Downarrow \\
(\mathsf{Charmander}, \mathsf{Cyndaquil}) \in \{(\mathsf{Charmander}, \mathsf{Cyndaquil}), \ldots\}
\end{array}$$

Finally, we compare the implementations of choose.

Case choose :

$$\begin{array}{l}
\mathsf{ValueRelation}(\alpha \to \mathsf{String}, \mathsf{PkmnRelation}, \mathsf{Gen1.choose}, \mathsf{Gen2.choose}) \\
\Downarrow \\
\forall (v_1, v_2) \ . \ \mathsf{ValueRelation}(\alpha, \mathsf{PkmnRelation}, v_1, v_2) \Rightarrow \\
\quad \mathsf{ValueRelation}(\mathsf{String}, \mathsf{PkmnRelation}, \mathsf{Gen1.choose}\ v_1, \mathsf{Gen2.choose}\ v_2)
\end{array}$$

Here we need to split into subcases for each possible choice of (v_1, v_2). Because these values must be related at type α, it suffices to split on the members of PkmnRelation. This gives us 3 sub-cases to prove, and each follows from string equality.

Case (Bulbasaur, Chikorita) :	ValueRelation(String, PkmnRelation, Gen1.choose Bulbasaur, Gen2.choose Chikorita) $\Downarrow$ ValueRelation(String, PkmnRelation, *"Grass types are peaceful."*, *"Grass types are peaceful."*) $\Downarrow$ *"Grass types are peaceful."* = *"Grass types are peaceful."*
Case (Squirtle, Totodile) :	ValueRelation(String, PkmnRelation, Gen1.choose Squirtle, Gen2.choose Totodile) $\Downarrow$ ValueRelation(String, PkmnRelation, *"Water types are agile."*, *"Water types are agile."*) $\Downarrow$ *"Water types are agile."* = *"Water types are agile."*
Case (Charmander, Cyndaquil) :	ValueRelation(String, PkmnRelation, Gen1.choose Charmander, Gen2.choose Cyndaquil) $\Downarrow$ ValueRelation(String, PkmnRelation, *"Fire types are fierce."*, *"Fire types are fierce."*) $\Downarrow$ *"Fire types are fierce."* = *"Fire types are fierce."*

This completes the proof! Hopefully walking through this illustrated 3 points:

- The proof was, after all, just a grind. Putting things all together was "obvious" after we had built up the proper machinery.
- Now that we have ValueRelation, we can use it to prove other useful equivalences.
- The real trick was choosing the right logical relation PkmnRelation. That one choice affected the entire flow of our proof. If we had chosen an incorrect relation, for example PkmnRelation$'$ $=$ $\{$(Bulbasaur, Cyndaquil)$\}$, we would have gotten stuck trying to prove that the strings *"Grass types are peaceful."* and *"Fire types are fierce."* were equivalent (not to mention geting stuck on the other fields).

6 The bad news

The logical relations we have used so far lack support for two fundamental language features: recursion and mutability. In the 90's this led to the criticism that logical relations are only useful for toy languages, and could not scale to real software.

6.1 Recursive Types

Recursion is a fundamental technique for building computer programs. Recursive types help us make static guarantees about recursive programs or data. There are two main approaches to recursive types: equirecursive types and isorecursive types. In an equirecursive system, types may be infinite. An isorecursive system allows only finite types, but a finite type may be isomorphic to an infinite one.

To illustrate the incompatibility of logical relations and recursive types, we extend our λ-calculus to with isorecursive types.

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mu\alpha.\tau \\ e & ::= & \ldots \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \\ v & ::= & \ldots \mid \mathsf{fold}\ v \end{array}$$

A recursive type $\mu\alpha.\tau$ is essentially a thunk. The operations fold and unfold create and force a thunk, respectively. They are described by the operational rules below:

$$\frac{\Gamma \vdash e : \tau\{\mu\alpha.\tau/\alpha\}}{\Gamma \vdash \mathsf{fold}\ e : \mu\alpha.\tau} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau\{\mu\alpha.\tau/\alpha\}}$$

We can try extending our logical relation ValueRelation (from Section 5.2) to include recursive types.

$$\begin{array}{rcl} & \vdots & \\ \mathsf{ValueRelation}(\tau \to \tau', \mathbb{R}, f_1, f_2) & = & \forall(v_1, v_2).\mathsf{ValueRelation}(\tau, \mathbb{R}, v_1, v_2) \Rightarrow \mathsf{ValueRelation}(\tau', \mathbb{R}, f_1 v_1, f_2 v_2) \\ \mathsf{ValueRelation}(\alpha, \mathbb{R}, v_1, v_2) & = & \mathbb{R}(v_1, v_2) \\ \mathsf{ValueRelation}(\mu\alpha.\tau, \mathbb{R}, \mathsf{fold}\ v_1, \mathsf{fold}\ v_2) & = & \mathsf{ValueRelation}(\tau\{\mu\alpha.\tau/\alpha\}, \mathbb{R}, \mathsf{unfold}\ (\mathsf{fold}\ v_1), \mathsf{unfold}\ (\mathsf{fold}\ v_2)) \end{array}$$

But the case for $\mu\alpha.\tau$ is not well-founded! All our logical relations until now (ValueRelation, PkmnRelation, StronglyNormalizing) were well-founded with respect to the size of their argument type. Recursive calls were made only on smaller types, thus evaluating a relation eventually terminated. This is no longer true.

6.2 Mutability

In Section 5, we saw how to prove contextual equivalence for two simple modules. Here is another simple module and implementation [2]. Ordinary logical relations cannot prove that the implementation never raises an index error.

```
interface SYMBOLSET =
  type t
  val insert : string -> t
  val lookup : t -> string
end
module (AsList:SYMBOLSET) =
  let size = ref 0
  let table = ref []

  type t = int
  let insert str =
    size :- !size + 1;
    table := str :: !table;
    !size
  let lookup n = List.nth !table (!size - n)
end
```

The module is correct because the type **`t`** used in **`lookup`** is hidden behind the interface **`SYMBOLSET`**. The only possible values an observer could pass to **`lookup`** are those created by **`insert`**. Logical relations cannot reason about the interactions of this newly-generated type with the mutable data inside **`AsList`**. We need(ed) a new idea.

7 An alternative: Bisimulations

Getting logical relations to work with recursive types was a big question for a long time. People like Pitts, Birkedal, Harper, and Crary [18, 6, 7] were trying some wild techniques, like biorthogonality.

Biorthogonality (aka $\top\top$-closure or $\bot\bot$-closure) is a technique for building a complete logical relation. "Complete" means that everything true is provable: in this case, every pair of contextually equivalent terms is represented by the logical relation. This is a tall order. Biorthogonality makes it work by pulling the definition of contextual equivalence into the logical relation. It is complicated and it (allegedly, I don't really know) has trouble with examples, but the proof is solid.

At any rate, that is just one of the proposed solutions for helping logical relations cope with recursive types. The trouble with this and related techniques is that they burden the end user. One must prove *unwinding properties* or calculate the $\top\top$-closure of their relation before proceeding.

After nearly a decade of these increasingly-complex solutions, the team of Eijiro Sumii and Benjamin Pierce presented a brand new technique, based on bisimulations, for proving contextual equivalence of recursive programs.

7.1 Bisimulation

Generally, two sequences are bisimilar with respect to a relation R if:

- Their current states are related by R.
- They are either complete, or both step to bisimilar next-states.

(As a historical note, bisimulations first crept into computer science via Milner, who used bisimulations to reason about the contextual equivalence of concurrent programs [13]. The technique was useful because it compared programs up to interleaving: if two executions had the same result but a different order of execution, the bisimulation still related them.)

Unlike logical relations, which must be well-founded, bisimulations are a *co-inductive* proof technique. Nothing in the definition of a bisimulation says that the sequences must terminate—they need only be synchronized. Sumii and Pierce leveraged this to handle recursive functions, which similarly may not terminate.

The trick was adapting bisimulations to reason about information hiding, like the example from Section 5. So whereas a traditional bisimulation relation is a set of pairs, Sumii and Pierce's bisimulations were sets of relations. Moreover, their bisimulations were not closed under union. Rather, one would define a bisimulation, much like our PkmnRelation, that described the information an observer could learn by interacting with an abstract type.

7.2 An Example

The bisimulation for our Gen1 and Gen2 modules would be:

$$B \quad = \quad \{(\emptyset, \mathcal{R}_0),\ (\Delta, \mathcal{R}_1),\ (\Delta, \mathcal{R}_2)\}$$

where:

$$\begin{array}{rcl}
\Delta & = & \{(\alpha, pkmn, pkmn')\} \\
\mathcal{R}_0 & = & \{(\mathsf{Gen1}, \mathsf{Gen2}, \tau)\} \\
\mathcal{R}_1 & = & \mathcal{R}_0 \cup \{(\langle \mathsf{Bulbasaur}, \mathsf{Squirtle}, \mathsf{Charmander}, \lambda x.\ \textbf{match}\ x \\
 & & \quad \mid \mathsf{Grass} \to \textit{“Grass types are peaceful.”} \\
 & & \quad \mid \mathsf{Water} \to \textit{“Water types are agile.”} \\
 & & \quad \mid \mathsf{Fire} \to \textit{“Fire types are fierce.”}\rangle, \\
 & & \langle \mathsf{Chikorita}, \mathsf{Totodile}, \mathsf{Cyndaquil}, \lambda x.\ \textbf{match}\ x \\
 & & \quad \mid \mathsf{Grass} \to \textit{“Grass types are peaceful.”} \\
 & & \quad \mid \mathsf{Water} \to \textit{“Water types are agile.”} \\
 & & \quad \mid \mathsf{Fire} \to \textit{“Fire types are fierce.”}\rangle, \\
 & & \alpha \times \alpha \times \alpha \times \alpha \to \mathsf{String})\} \\
\mathcal{R}_2 & = & \mathcal{R}_1 \cup \{(\mathsf{Bulbasaur}, \mathsf{Chikorita}, \alpha),\ (\mathsf{Squirtle}, \mathsf{Totodile}, \alpha),\ (\mathsf{Charmander}, \mathsf{Cyndaquil}, \alpha), \\
 & & \quad (\lambda x.\ \ldots, \lambda x.\ \ldots, \alpha \to \mathsf{String})\}
\end{array}$$

Again, the intuition is that each successive $\mathcal{R}_i$ shows what an observer could see after i steps. Note that the relation PkmnRelation appears as a subset of $\mathcal{R}_2$.

7.3 Nagging doubts

Bisimulations are a little dirty because they are not constructive. In the end, you get something proven correct, but little guidance on how to build the artifact.

8 Step Indexing

Logical relations might have fallen out of vogue, but Appel and McAllester saved the day with a technique called *step indexing* [5]. Recall that the trouble with recursive types was that all our logical relations were well-founded by induction on the type structure. The idea with step-indexing is to induct over the natural numbers instead.

Yes, that's it.

Instead of using domain theory or requiring the user to prove admissibility, step-indexing allows the same logical relations as before, just tagged with a counter. The counter represents the number of remaining computational steps. When it hits zero, you stop, but in practice you only use these indices for the proof. Ultimately, any proof by step indexing is initially completed for an arbitrary starting index k, then abstracted for any choice of index. Just as logical relations gave our induction hypothesis the little push it needed to finish a proof, indices give logical relations a little extra power.

8.1 At a glance

Here is a step-indexed logical relation for the λ-calculus. Note how the indices pop up *everywhere*.

$$
\begin{array}{rcl}
 & \vdots & \\
\mathsf{ValueRelation}_k(\tau \to \tau', \mathbb{R}, f_1, f_2) & = & \forall j < k \,.\, \forall (v_1, v_2) \,. \\
 & & \quad \mathsf{ValueRelation}_j(\tau, \mathbb{R}, v_1, v_2) \Rightarrow \mathsf{ValueRelation}_j(\tau', \mathbb{R}, f_1 v_1, f_2 v_2) \\
\mathsf{ValueRelation}_k(\alpha, \mathbb{R}, v_1, v_2) & = & \mathbb{R}(v_1, v_2) \\
\mathsf{ValueRelation}_k(\mu\alpha.\tau, \mathbb{R}, \mathsf{fold}\ v_1, \mathsf{fold}\ v_2) & = & \forall j < k \,. \\
 & = & \quad \mathsf{ValueRelation}_j(\tau\{\mu\alpha.\tau/\alpha\}, \mathbb{R}, \mathsf{unfold}\ (\mathsf{fold}\ v_1), \mathsf{unfold}\ (\mathsf{fold}\ v_2))
\end{array}
$$

The rule for λ abstractions bears further mention. If we want to show that two values f_1 and f_2 are related for k steps, we need to reason about what happens when they are applied to related arguments. Before, this was relatively straightforward. We simply demand related arguments and prove related results.

With the addition of steps, we need to be conscious of *when* we apply our functions. Suppose, for example, we did not require $\forall j < k$, but instead used $(k-1)$ as the index on our consequence. Then our proofs would get stuck whenever the argument to any function was not already in normal form. Because we need the freedom to apply at any point in the future, we quantify over all $j < k$.

8.2 In summary

8.2.1 Good parts

Step-indexed logical relations have been extremely successful. They are simple to understand and implement, and have proven useful in a variety of applications [12, 2, 3]. Regarding their implementation, Appel and McAllester first invented step-indexed relations for the Foundational Proof-Carrying Code project. The goal there was to build a small trusted computing base and then prove all other code correct before executing it. In this setting, the simplicity of step-indices was a huge bonus over techniques that required complicated theories.

8.2.2 The downsides

Step-indexing has a very "ad-hoc" feeling. Logical relations were already a bit vague—you just augment your proof with an extra relation—and the steps with their fuzzy arithmetic are even worse. Also, the little indices pollute the logical relation and are easy to mess up.

9 Possible Worlds

The other troublesome language feature mentioned in Section 6 was mutable references. Without going into details, the way to handle updates to mutable cells is to approximate the contents of a cell with an index. Suppose we had a cell $x =$ `ref(ref(ref false))`. The actual type of x is `bool ref ref ref`, but if we only had 2 computation steps available we could safely assume that x had type `string ref ref`.

Additionally, we need a technique called *Kripke models* (or *possible worlds*). When reasoning about the shape of heap memory, we quantify over the possible futures reachable from our current state and use minimal assumptions about the well-formedness of these futures.

10 Annotated Bibliography

The earliest work on logical relations seems to have developed around System F, the polymorphic λ-calculus. Tait [24], Girard [9], Plotkin [19], and Statman [22] all seem to have been involved, but I have

not read those papers. Reynolds [20] and Mitchell [14] used logical relations in their foundational work on parametricity (another subfield with a questionable name. . . Strachey [23] and Wadler [27] are to blame).

Amal Ahmed and Derek Dreyer have given excellent tutorials on logical relations and parametricity (respectively) at the Oregon Programming Languages Summer School. Shouts out to Oregon.

In fact, Amal Ahmed is the contemporary experts on logical relations. She brought the technique to maturity [3, 1] and has since proved a variety of extensions; for example, parametricity for dynamic languages [12], modules and mutable state [2], and correctness of a multi-language compiler [17].

On the other hand, Derek Dreyer (along with Nick Benton, Gil Hur, and Georg Neis) has been pushing the limits of logical relations. The most recent efforts on this front are parametric bisimulations, which combine the best parts of Sumii and Pierce's bisimulations with possible-worlds models [10]. These techniques have been used by Appel's group in their efforts towards a compositional CompCert compiler [11, 8].

Logical relations also have consequences for security typed languages. See Zdancewic [25] or Myers [21] for more.

Just a guess, but the future of logical relations might be in the unified (read: **not** ad-hoc) structure of homotopy type theory [26]. We'll see.

The foundational proof-carrying code project was started by George Necula and Peter Lee. Read the summary by Appel [4] for more.

Finally, Morris [16] and Moggi [15] have more to say about the foundations of program correctness. Highly recommended.

References

[1] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems*, pages 69–83. Springer, 2006.

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM SIGPLAN Notices*, volume 44, pages 340–353. ACM, 2009.

[3] Amal Jamil Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[4] Andrew W Appel. Foundational proof-carrying code. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 247–256. IEEE, 2001.

[5] Andrew W Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23:657–683, 2001.

[6] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. *Information and computation*, 155(1):3–63, 1999.

[7] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electronic Notes in Theoretical Computer Science*, 172:259–299, 2007.

[8] Gordon Stewart Lennart Beringer Santiago Cuellar and Andrew W Appel. Compositional compcert.

[9] Jean-Yves Girard. Interprétation fonctionelle et élimination des coupures de larithmétique dordre supérieur. 1972.

[10] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. Parametric bisimulations: A logical step forward. 2013.

[11] Xavier Leroy. The compcert c verified compiler, 2012.

[12] Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Programming Languages and Systems*, pages 16–31. Springer, 2008.

[13] Robin Milner. A calculus for communicating processes, volume 92 of lecture notes in computer science, 1980.

[14] John C Mitchell. Representation independence and data abstraction. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 263–276. ACM, 1986.

[15] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93:55–92, 1991.

[16] James H Morris. Lambda-calculus models of programming languages. Technical report, DTIC Document, 1968.

[17] James T Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *Programming Languages and Systems*, pages 128–148. Springer, 2014.

[18] Andrew M Pitts. Existential types: Logical relations and operational equivalence. In *Automata, Languages and Programming*, pages 309–326. Springer, 1998.

[19] Gordon D Plotkin. *Lambda-definability and logical relations*. School of Artificial Intelligence, University of Edinburgh, 1973.

[20] John C Reynolds. Types, abstraction and parametric polymorphism. 1983.

[21] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21:5–19, 2003.

[22] Richard Statman. Logical relations and the typed¡ i¿ λ¡/i¿-calculus. *Information and Control*, 65(2):85–97, 1985.

[23] Christopher Strachey. Fundamental concepts in programming languages, lecture notes for the international summer school in computer programming. *Copenhagen, August*, 1967.

[24] William W Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32:198–212, 1967.

[25] Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *ACM SIGPLAN Notices*, volume 39, pages 115–125. ACM, 2004.

[26] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[27] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.