Typed closure conversion using Logical Relations

Aarti Kashyap
University of British Columbia
19115724

Contents

1 Introduction

Logical relations was first introduced by Gordon Plotkin. Logical relations are type indexed inductive relations. In this work, I attempt to perform typed closure conversion using cross-language logical relation. The final goal was supposed to prove full abstraction. However, due to time constraints, I stick to proving type preservation for two languages.

1.1 Logical Relations

The first part of the project is to understand how to define a logical relation. Logical relations can be split into two categories: logical predicates and logical relations.

1.1.1 Logical predicates

Logical predicates are unary relations. These are defined by $\mathrm{P}_\tau(\mathrm{e})$. This means that some predicate P holds for some expression e over type τ. These types of relations can be used to prove strong normalization and type safety.

1.1.2 Logical Relations

The second type of logical relations are termed as binary relations. These are described as $\mathrm{R}_\tau(\mathrm{e1,e2})$. This expression says that there exists some relation R between terms e1 and e2 of the two languages. This type of relation can be used for showing program equivalence or some form of program equivalence.

1.2 Why do we need logical relations

Using logical relations is very useful. Properties such as strong normalization and type-safety are very useful in languages with a well-defined type-system. However, the most interesting part about using LRs is in the case of equivalence of programs.
Equivalence of programs within a programming language has a lot of benefits. We can use it to check the equivalence after we apply optimizations to a program. It is necessary to make sure the program behaves the same as it did before optimizations. We can also use it for security properties. To make sure that information flow policies hold in a language.

1.3 Defining a logical relation

In general, for a logical predicate P τ and an expression e, e is accepted by the predicate if it satisfies the following properties.

1. $\bullet \vdash$ e: τ
2. The property we wish e to have.
3. The condition is preserved by eliminating forms.

2 Simply Types Lambda Calculus (STLC)

The simply Typed Lambda calculus is defined as follows.

2.1 Language Definition

$$\begin{array}{rcl} e & := & v \mid e\ e \mid \\ & \mid & \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \\ \\ v & := & x \mid \lambda x.\ e \mid n \\ \\ \tau & := & \mathit{int} \mid \tau \to \tau \end{array}$$

The typing context is defined as

2.2 Semantics

$$\frac{e_1 \Downarrow \lambda x.\ e' \qquad e_2 \Downarrow v \qquad \cdots}{e_1\ e_2 \Downarrow e'[v/x]}\ \textsc{App} \qquad \frac{e_1 \Downarrow \textsf{false} \qquad e_3 \Downarrow v}{\textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 \Downarrow v}$$

2.3 Typing Judgements

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

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Gamma \vdash e_1 : \mathit{int} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 : \tau}$$

The STLC with minimum features and types.

3 Strong-normalization for STLC using Logical Relations

In this section, I try to show that simply typed lambda calculus holds strong normalization which means that every term reduces to its normal form. In our case, the normal forms of the language are the values of the language for STLC.

3.1 Why do we need logical predicates?

Theorem (Strong Normalization)

If $\bullet \vdash$ e: τ then e $\Downarrow$

I first try to prove the above property using induction on the typing derivation. And I see that it fails.

Proof. The proof fails for the application case.

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \; e_2 : \tau'}$$

By induction hypothesis we get that $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$. By the type of e_1, we conclude $e_1 \Downarrow \lambda x : \tau_2 .\ e'$. What I need to show is $e_1 e_2 \Downarrow$.

However, I run into an issue as we do not know anything about e'. This tells us that our induction hypothesis is not strong enough.

3.2 Defining Logical predicate

I define a logical predicate SN $_\tau$(e). The entire point of defining SN $_\tau$(e) is such that it accepts expressions of type τ that are strongly normalizing.

Properties to be kept in mind when defining logical predicates

SN $_{\text{int}}$(e) $\iff \bullet \vdash e : \mathit{int} \wedge e \Downarrow$

3.3 SN using Logical predicates

The proof is done in two steps:

1. $\bullet \vdash e : \tau \implies$ SN $_\tau$ (e)
2. SN $_\tau$ $(e) \implies e \Downarrow$

The structure of proof is common to proofs that use logical relations. When we try to use logical relations in order to prove compiler correctness we will arrive at similar results.

Theorem (Generalized version of 1.) If $\Gamma \vdash e : \tau$ and $\gamma \models \Gamma$, then $\mathrm{SN}_\tau(\gamma(e))$

The theorem basically says that if e is well typed with respect to some type τ and we have some closing substitution that satisfies the typing environment, then if e is closed with respect to γ, then this expression can be termed as SN_τ.

Lemma (Substitution Lemma) If $\Gamma \vdash e : \tau$ and $\gamma \models \Gamma$, then $\bullet \vdash \gamma(e) : \tau$

Lemma (SN preserved by forward/backward reduction)

After proving the lemmas, generalized proof can be proved by proof by induction on $\Gamma \vdash e : \tau$

The proof is not explained in detail here but it can be done using induction. The purpose of this part was to get familiar with logical relations. Constructing the definition of logical relations is more important than the proof. The reason is because as we can see from the above methodology, we construct the predicate such that the proof follows easily from it.

Similary, we can prove type-safety and equivalence of programs using logical relations.

Now, with a little background on how logical relations work while proving properties in a single programming language the next step is taken. The next part is the interesting part, where an attempt is made to use binary logical properties to prove something useful.

4 Compiler Correctness

The initial project goal was to use logical relations to prove full abstraction compiler correctness all the way down from a subset of C to a subset of Ocaml. The Project has briefly given an overview as to how to construct logical relations.The intuition behind constructing logical relations remains the same as it did for unary relations.

Compiler translation In this work the compiler translation we consider is typed closure conversion.

4.1 Closure conversion

The source language is the STLC described in Section 2. When we are computing in a different language, we start by translating to the language. Suppose, we have $e_1, e_2 and e_3$ put together. We can translate them individually. However, e_2 might contain a function with free variables. Hence, we need closure conversion. After doing that, the free variables can be hoisted up to the top.

4.1.1 Example

$\lambda x : int(x + y + z)$

After applying closure conversion

$\lambda((z : \tau_e, x : int, x + \pi_1 y + \pi_1 z), < y, w >)$

π_1 is the first component of the new new environment defined.

π_2 is the second component of the new new environment defined.

4.1.2 General form of closure conversion

So in general for representing all functions in the form as shown in the above example, a general format can be defined.

$\exists\alpha((\alpha, int) \to int)x\alpha$

Hence, we can see that the target language should have existential types. Instead of defining the source and target languages and then beginning the logical relations. I started with the simplest source language with a type system. Based on the translation, I defined my target language. Using existential types I can implement an interface which allows closure conversion. This is a generic way of representing any function as we want.

4.2 Target Language

$$
\begin{array}{rcl}
e & := & v \ | \ e - e \ | \ < e_1, ...e_n > \\
 & | & \textsf{if } e \textsf{ then } e \textsf{ else } e \ | \ \pi_1 e \\
 & | & pack(\tau, e) as \exists \alpha.\tau \ | \ e(\tau) \\
 & | & unpack(\alpha, x) = e_1 in e_2 \\
 & & \\
v & := & x \ | \ \lambda x : \tau.\ e \ | \ n \ | \ < v_1, ..., v_n > \\
 & & \\
\tau & := & int \ | \ (\tau_1, ...\tau_n) \to \tau' \ | \ < \tau_1, ..., \tau_n > \\
 & | & \alpha \ | \ \exists \alpha.\tau \ | \ \forall \alpha.\tau
\end{array}
$$

$\triangle, \Gamma \vdash e : \tau$

4.3 Typing rules

Pack is how we create something of existential type, it is our introduction form. I think I stumbled a bit on this part during the presentation.

I also need an elimination form which is unpack. unpack takes apart a package so that we can use its components. A package consists of a witness type and an expression that implements an existential type. The following typing rules are defined for the two constructs pack and unpack.

$$\frac{\triangle, \Gamma \vdash e : \tau[\tau'/\alpha] \qquad \triangle \vdash \tau'}{\triangle, \Gamma \vdash pack < \tau', e > as \exists \alpha.\tau : \exists \alpha.\tau}$$

$$\frac{\triangle, \Gamma \vdash e_1 : \exists \alpha.\tau \qquad \triangle, \Gamma \vdash e_1 : \exists \alpha.\tau \qquad \triangle \vdash \tau_2}{\triangle, \Gamma \vdash unpack < \alpha, x >= e_1 in e_2 : \tau_2}$$

4.4 Types Translation

Γ^+

I will be defining the type translation with the above symbol.

int $^+$ = int

$(\tau_1 \to \tau_2) = \exists \alpha.\text{<}((\alpha, {\tau_1}^+) \to {\tau_2}^+), \alpha\text{>}$

The point of producing such translations so that it satisfies a type-preserving compilation.

$\Gamma \vdash: \tau \curvearrowright \mathbf{e}$

We want to show type-preserving compiler.

My report was exceeding the page limit, hence, I removed some obvious details. I can add them again, if you want.

Most of the type translations are straight forward which are the variables and the numbers. The interesting cases are the lambda and the application.

The Lamda case makes use of pack in order to translate. The application uses unpack. We just need to make sure that after the translation the types of the translated term are in the same environment.

4.5 Logical relations

For every source language, we want to relate it to the target language. The translation can be doing anything, but we do not care about that, so long after the translations the properties are being maintained.

My aim here is to use the method of defining unary logical relations to extend to binary logical relations.

[1]. V[[τ]] = (Vs,Vt) | $\bullet \vdash$ Vs : $\tau \wedge \bullet \vdash$ Vt : τ^+ ...

The methodology of defining the LR is very similar to how it was done in Part 1 of my work.

The first part is the relation. The second part is the property we are concerned about and finally something else. What the property says is that if source has some type τ then the target should have some type τ $^+$

By doing induction over type derivatives we can make sure it works. I have made parts from the second language in bolds. For eg. in the second typing rule, the part inside the pack is from the target language.

[2]. V[[int]] = (n,**n**)

[3]. V[[$\tau_1 \rightarrow \tau_2$]] = ($\lambda x : \tau_1.e_1$, **pack(**$\tau_e$, ($\lambda$($z$: τ_e, x:τ^+)e, V_e) | $\forall$ (V, **V**) E V [[τ_1]]. (e[v/x], **e[v**$_e$**/z][v/x]** E E[[τ_2]])

This is for the value relations.After doing it for as shown in the above two relations, we have to create the same for expressions of the two languages.

NOTE While writing these relations, it is very important to note that these should not be written by looking at the term by term translation relations. We just look at the end states of the compiler and build the type indexed inductive relations also called as logical relations.

[4]. E$[[\tau]]$ = (e,**e**) $|\forall$ v e $\rightarrow^*$ v $\Longrightarrow \exists$ **v e** $\rightarrow^*$ **v** $\wedge$ (v,v) E V$[[\tau]]$

NOTE As shown in the above definition we can also have a symmetric definition, exchange the source and the target. Usually such symmetric exchanges are not possible in all the languages. However, in this case it is possible because of the way my two languages are defined.

The entire point of using logical relations is that the proof becomes very easy to follow after the relations have been defined. After defining a relation between two things, we do not need to know about the translation. With just the final results we can prove things about a compiler correctness.

The final part about this is the general relation which we define.

[5]. G[[.]] = ϕ

[6]. G$[[\Gamma$, x: $\tau]]$ = γ[x $\rightarrow$ (v,**v**)] | γ E G$[[\tau]]$ $\wedge$ (v,**v**) E V$[[\tau]]$

Finally, this gets us to the final stage.

$\Gamma \vdash e_s \backsimeq \mathbf{e}_t$**:** τ =

$\Gamma \vdash e_s$: $\tau \wedge \bullet$; $\Gamma^+ \vdash \mathbf{e}_t : \tau^+ \wedge \forall \gamma$ E G$[[\tau]]$ (γ_s(e$_s$), γ_t**(e**$_t$**))** E E$[[\tau]]$

Theorem: Fundamental property (Compiler Correctness)

If [$\Gamma \vdash$ e$_s$:$\tau \curvearrowright \mathbf{e}_t$]

then [$\Gamma \vdash$ e$_s \backsimeq \mathbf{e}_t$:$\tau$]

This helps us do type preservation for the two languages over closure conversion.

5 Extensions

In the above section we proved compiler correctness by showing that it preserves the types during closure conversion. This is just a beginning. The eventual goal was to prove full abstraction correctness for the two languages. The next step for this would be to show compositional compiler correctness for the two languages that I have described above. After than we can extend that to show full abstraction step by step.

From my current understanding I can see that we cannot directly jump to proving full correctness. It's a step by step process. We select our languages for which we feel that it's important to prove such properties. It can be a DSL or any form of languages. But, more importantly the question we should ask ourselves before selecting the languages should be - "Do these languages need this property to hold and why". Once that is done,we should proceed in a step by step fashion. For proving a really strong property for a system, we should first make sure that they hold some weak ones. It's not always necessary to do so (Obviously). However, according to me it's always a good thing to proceed systematically. Start small and make it big rather than targeting that big golden property.

I am aware from the current research work, that a lot of work has already been done in this area. People have been using logical relations since forever. People have also used logical relations to prove compiler correctness properties. Despite this, there is a lot of work which can be done in this area. Proving it for real languages will be so much more interesting. Languages which offer much more expressiveness than my described source and target languages. This is the reason that operation semantics when combined with denotational semantics can help us proving much stronger properties. Since operation semantics can give us the expressiveness and the denotational semantics can give us the formalism.

My goal is to someday be able to prove full abstraction for a cyber physical system which in some ways is more tricky because CPS usually don't just use one language. They have multiple languages hence the linking of different languages also become an important part of this area. I want to automate the entire process for Cyber physical system so that the developers don't have to worry about languages. They can use different languages for different applications in their CPS or what people call it nowdays IoT without having security concerns.