Typed closure conversion using Logical Relations

Aarti Kashyap
University of British Columbia
19115724

Contents

1 Introduction

Logical relations was first introduced by Gordon Plotkin who was also the first to introduce operational semantics. 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 compositional compiler verification. However, due to time constraints, I stick to the initial part like I explain above.

1.1 Logical Relations

The first part of the project was 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 the unary relations. These are defined by $P_\tau(e)$. 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 $R_\tau(e1,e2)$. 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 us it to check the equivalence after we apply optimizations, to check the correctness of the program. We can also use it for security properties. To make sure that information flow policies hold.

1.3 Defining a logical relation

In general, for a logical predicate P and an expression e, we want e to be accepted by the predicate if it satisfies the following properties.

1. $\bullet \; \Gamma \vdash \; : \; e\tau$
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

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

The typing context is defined as

2.2 Semantics

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

2.3 Typing Judgements

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

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

The STLC is defined in the simplest way possible.

3 Strong-normalization for STLC using Logical Relations

In this section, I try to show that simply typed lambda calculus has strong normalization which means that every term is strongly normalizing. If a term is strongly normalizing, then it 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)

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.

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

By induction hypothesis we get that $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$. By the type of e_1, we . 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 construction methodology remains pretty similar.

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

$$\begin{array}{lll}
e & := & v \ \mid \ e - e \ \mid \ < e_1, ...e_n > \\
 & \mid & \textsf{if } e \textsf{ then } e \textsf{ else } e \ \mid \ \pi_1 e \\
 & \mid & pack(\tau, e) as \exists \alpha.\tau \ \mid \ e(\tau) \\
 & \mid & unpack(\alpha, x) = e_1 ine_2 \\
 & & \\
v & := & x \ \mid \ \lambda x : \tau.\ e \ \mid \ n \ \mid \ < v_1, ..., v_n > \\
 & & \\
\tau & := & int \ \mid \ (\tau_1, ...\tau_n) \rightarrow \tau' \ \mid \ < \tau_1, ..., \tau_n > \\
 & \mid & \alpha \ \mid \ \exists \alpha.\tau \ \mid \ \forall \alpha.\tau
\end{array}$$

The evaluation function is defined as

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

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.

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

$$\frac{\triangle, \Gamma \vdash e_1 : \exists \alpha.\tau \qquad \triangle, \Gamma \vdash e_1 : \exists \alpha.\tau \qquad \triangle \vdash \tau_2}{\triangle, \Gamma \vdash unpack < \alpha, x >= e_1 ine_2 : \tau_2}$$

4.4 Types Translation

Γ^+

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

int $^+$ = int

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

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.

It was becoming more and more difficult to use Latex, hence I skip a lot of details here. If you feel I should have included more, I will add them. The details are the same which I showed on the blackboard during the presentation.

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.

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

The logical relation for the two languages is defined similar to how I had defined at the very beginning for proving strong normalization.

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.

V[[int]] = (n,n)

V[[$\tau_1 \rightarrow \tau_2$]]

5 Language Definitions

$$\begin{array}{rcl}
e & := & x \mid v \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \\
 & \mid & \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e \mid \mathit{unop}\ e \mid \mathit{binop}\ e\ e \\
 & & \\
v & := & \lambda x.\ e \mid \textsf{true} \mid \textsf{false} \\
 & & \\
\mathit{binop} & := & \textsf{and} \mid \textsf{or} \\
\mathit{unop} & := & \textsf{not} \\
 & & \\
\mathcal{E} & := & \mathcal{E}\ e \mid v\ \mathcal{E} \mid \textsf{let}\ x = \mathcal{E}\ \textsf{in}\ e \\
 & \mid & \textsf{if}\ \mathcal{E}\ \textsf{then}\ e\ \textsf{else}\ e \mid \mathit{unop}\ \mathcal{E} \mid \mathit{binop}\ \mathcal{E}\ e \mid \mathit{binop}\ v\ \mathcal{E} \\
 & & \\
\tau & := & \texttt{bool} \mid \tau \to \tau
\end{array}$$

6 Semantics One

$$\frac{e_1 \Downarrow \lambda x.\ e' \qquad e_2 \Downarrow v \qquad \cdots}{e_1\ e_2 \Downarrow e'[v/x]}\ \textsc{App} \quad \frac{e_1 \Downarrow v \qquad \cdots}{\textsf{let}\ x = e_1\ \textsf{in}\ e_2 \Downarrow e_2[v/x]}$$

$$\frac{e_1 \Downarrow \textsf{true} \qquad e_2 \Downarrow v}{\textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 \Downarrow v} \quad \frac{e_1 \Downarrow \textsf{false} \qquad e_3 \Downarrow v}{\textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 \Downarrow v}$$

$$\frac{e_1 \Downarrow \textsf{true} \qquad e_2 \Downarrow \textsf{true}}{\textsf{and}\ e_1\ e_2 \Downarrow \textsf{true}} \quad \frac{e_1 \Downarrow \textsf{false}}{\textsf{and}\ e_1\ e_2 \Downarrow \textsf{false}}$$

$$\frac{e_1 \Downarrow \textsf{true}}{\textsf{or}\ e_1\ e_2 \Downarrow \textsf{true}} \quad \frac{e_1 \Downarrow \textsf{false} \qquad e_2 \Downarrow v}{\textsf{or}\ e_1\ e_2 \Downarrow v}$$

$$\frac{e \Downarrow \textsf{true}}{\textsf{not}\ e \Downarrow \textsf{false}} \quad \frac{e \Downarrow \textsf{false}}{\textsf{not}\ e \Downarrow \textsf{true}}$$

7 Semantics Two

$$\begin{array}{rcll}
(\lambda x.\ e)\ v & \to & e[v/x](\cdots) & \mathit{App} \\
\textsf{let}\ x = v\ \textsf{in}\ e & \to & e[v/x](\cdots) & \\
\textsf{if true then}\ e_2\ \textsf{else}\ e_3 & \to & e_2 & \\
\textsf{if false then}\ e_2\ \textsf{else}\ e_3 & \to & e_3 & \\
\textsf{and false}\ e_2 & \to & \textsf{false} & \\
\textsf{and true}\ e_2 & \to & e_2 & \\
\textsf{or false}\ e_2 & \to & e_2 & \\
\textsf{or true}\ e_2 & \to & \textsf{true} & \\
\textsf{not false} & \to & \textsf{true} & \\
\textsf{not true} & \to & \textsf{false} & \\
 & & & \\
\mathcal{E}[e] & \mapsto & \mathcal{E}[e']\ (\text{if } e \to e') &
\end{array}$$

8 Typing Judgments

$$\frac{}{\Gamma \vdash \textsf{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \textsf{false} : \texttt{bool}}$$

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

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Gamma, x : \tau' \vdash e_2 : \tau \qquad \Gamma \vdash e_1 : \tau'}{\Gamma \vdash \textsf{let}\ x = e_1\ \textsf{in}\ e_2 : \tau}$$

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

$$\frac{\Gamma \vdash \mathit{binop} : \tau_1 \to \tau_2 \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathit{binop}\ e_1\ e_2 : \tau}$$