Logical Relations Part 4

Lecture 4 Wednesday, September 13, 2017

We continue exploring logical relations. This lecture is based on lectures by Prof Amal Ahmed at the Oregon Programming Languages Summer School, 2015[1], as reported in the notes by Lau Skorstengaard.[2]

1 Recursive Types

What is the type of a data structure such as a tree? In a programming language like Java, we would define it as follows:

```
class Tree {
  int val;
  Tree left;
  Tree right;
}
```

This type is *recursive*: the type Tree is defined in terms of the type Tree. Similarly, nn an ML-like language, a declaration of a tree type would be recursive:

```
type tree = Leaf
          | Node of int * tree * tree
```

So how would we extend the lambda calculus to let us express recursive types? Let's think about what we would like for a **tree** type. A **tree** value can be either a leaf, or a node with an int and two subtrees. We can represent a leaf with a unit value (since the leaf carries no information). So we would like the **tree** type to be something like

$$\textbf{tree} = \textbf{unit} + (\textbf{int} \times \textbf{tree} \times \textbf{tree}).$$

Of course, this is recursive: it's not a definition of the type **tree**, rather it is an equation that we would like the type **tree** to satisfy. That is, if X is a variable that will be equal to the type for trees, we would like the following equivalences to hold.

$$\begin{aligned}
X &= \textbf{unit} + (\textbf{int} \times X \times X) \\
&= \textbf{unit} + (\textbf{int} \times (\textbf{unit} + (\textbf{int} \times X \times X)) \times (\textbf{unit} + (\textbf{int} \times X \times X))) \\
&= \textbf{unit} + (\textbf{int} \times (\textbf{unit} + (\textbf{int} \times (\textbf{int} \times X \times X) \times (\textbf{int} \times X \times X))) \times (\textbf{unit} + (\textbf{int} \times (\textbf{int} \times X \times X) \times (\textbf{int} \times X \times X)))) \\
&\ldots
\end{aligned}$$

Each time we "expand" the type variable X, the type gets bigger, with the limit being an infinite tree, which is, conceptually, the type that we have in mind for the type **tree**.

Let's define a function for which we want to find a fixed point. The function F will take a type X, and return another type.

$$F = \lambda X :: \mathsf{type}.\ \textbf{unit} + (\textbf{int} \times X \times X)$$

A fixed point of F is, by definition, a type τ such that $\tau = F(\tau)$. Let's call that type **tree**, i.e., we want $\textbf{tree} = F(\textbf{tree})$.

Se can write the fixed point as $\mu X.\ F(X)$, where μ is the fixed-point constructor. By definition $F(\mu X.\ F(X)) = \mu X.\ F(X)$. If we write τ for $F(X)$, then the fixed point would be $\mu X.\ \tau$, and $F(\mu X.\ F(X)) = F(\mu X.\ \tau) = \tau\{\mu X.\ \tau/X\}$. That is, the type $\mu X.\ \tau$ is equivalent to the type $\tau\{\mu X.\ \tau/X\}$.

This step of substituting the fixed point $\mu X.\ \tau$ for the type variable X in τ "unfolds" the recursive type $\mu X.\ \tau$. Going from $\mu X.\ \tau$ to $\tau\{\mu X.\ \tau/X\}$ and vice-versa is acheived by the fold and unfold operations.

[1] Videos available at `https://www.cs.uoregon.edu/research/summerschool/summer15/curriculum.html`.
[2] Available at `https://www.cs.uoregon.edu/research/summerschool/summer16/notes/AhmedLR.pdf`.

Consider a simply-typed lambda calculus extended with recursive types.

$$e ::= n \mid x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e$$
$$v ::= n \mid \lambda x{:}\tau.\, e \mid \mathsf{fold}\ v$$
$$\tau ::= \mathbf{int} \mid \tau_1 \to \tau_2 \mid \mu X.\, \tau \mid X$$
$$E ::= [\cdot] \mid E\ e \mid v\ E \mid \mathsf{fold}\ E \mid \mathsf{unfold}\ E$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \frac{}{(\lambda x{:}\tau.\, e)\ v \longrightarrow e\{v/x\}} \qquad \frac{}{(\mathsf{unfold}\ \mathsf{fold}\ v) \longrightarrow v}$$

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

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Gamma \vdash e : \tau\{\mu X.\, \tau/X\}}{\Gamma \vdash \mathsf{fold}\ e : \mu X.\, \tau} \qquad \frac{\Gamma \vdash e : \mu X.\, \tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau\{\mu X.\, \tau/X\}}$$

Note that we assume (without providing appropriate typing rules) that our types have no free type variables.

Returning to our tree example, we could define the type of **tree** as follows (assuming we had sum, product, int, and unit types).

$$\mathbf{tree} \triangleq \mu X.\, \mathbf{unit} + (\mathbf{int} \times X \times X)$$

We could write the type of integer lists as follows.

$$\mathbf{intlist} \triangleq \mu X.\, \mathbf{unit} + (\mathbf{int} \times X)$$

Note that we can also type the non-terminating term $\Omega = (\lambda x..\, x\ x)\ (\lambda x..\, x\ x)$ (provided that we insert appropriate folds and unfolds).

$$\vdash \mathsf{fold}\ (\lambda x.\, {:}\mu X.\, X \to X.\, (\mathsf{unfold}\ x)\ x) : \mu X.\, X \to X$$

Note that x has type $\mu X.\, X \to X$, and so $\mathsf{unfold}\ x$ has type $(\mu X.\, X \to X) \to (\mu X.\, X \to X)$.

So our lambda calculus with recursive types now has non-terminating computations.

2 Type Safety and Step-indexed logical relations

Let's consider proving type safety for this language. Informally, type safety means that a well-typed computation won't get stuck. That is, a computation can take some number of steps, and if it can't take any more steps, then it must be a value, i.e., it is never the case that we have a non-value expression that is irreducible.

Let's define a couple of useful predicates to let us express type safety.

$$\mathrm{irred}(e) =\!\triangleq \nexists e'.\, e \longrightarrow e'$$
$$\mathrm{safe}(e) =\!\triangleq \forall e'.\, e \longrightarrow^* e' \implies e' \text{ is a value} \vee \exists e''.\, e \longrightarrow e''$$

We will prove type safety by defining a logical relation. Let's first consider the logical relation for the simply typed lambda calculus (i.e., without recursive types), and then consider how to modify the logical relation to deal with recursive types.

We will again define a family of value relations $\mathcal{V}_\tau$ and expression relations $\mathcal{E}_\tau$ indexed by type τ. Relation (or, rather, predicate) $\mathcal{V}_\tau$ will be a set of closed values, and $\mathcal{E}_\tau$ will be a set of closed terms.

$$\begin{aligned}
\mathcal{V}_{\mathsf{int}} &= \{n \mid n \in \mathbb{Z}\} \\
\mathcal{V}_{\tau_1 \to \tau_2} &= \{\lambda x. : \tau_1.\, e \mid \forall v \in \mathcal{V}_{\tau_1}.\, e\{v/x\} \in \mathcal{E}_{\tau_2}\} \\
\mathcal{E}_{\tau} &= \{e \mid \forall e'.\, e \longrightarrow^* e' \wedge \mathrm{irred}(e') \Longrightarrow e' \in \mathcal{V}_{\tau}\}
\end{aligned}$$

We define our semantic notion of type safety by providing an interpretation of variable contexts, and then use that semantic notion of type safety to state the fundamental property.

Given a variable context Γ, we define the interpretation of Γ as a set of substitutions that are consistent with Γ.

$$\begin{aligned}
\mathcal{G}[\![\bullet]\!] &= \{\emptyset\} \\
\mathcal{G}[\![\Gamma, x{:}\tau]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!], v \in \mathcal{V}_{\tau}\} \\
\Gamma \vDash e{:}\tau &\triangleq \forall \gamma \in \mathcal{G}[\![\Gamma]\!].\, \gamma(e) \in \mathcal{E}_{\tau}
\end{aligned}$$

The proof of type safety is in two steps. First, well-typed expressions are in the relation:

Theorem 1 (Fundamental property)**.** *If* $\Gamma \vdash e{:}\tau$ *then* $\Gamma \vDash e{:}\tau$.

Second, expressions in the relation are type safe.

Theorem 2. *If* $\Gamma \vDash e{:}\tau$ *then* $\mathrm{safe}(e)$.

The proofs of these theorems are fairly straightforward. Let's turn our attention to extending the logical relations to handle recursive types.

Note that values of type $\mu X.\, \tau$ are of the form $\mathsf{fold}\ v$. A first (incorrect) attempt to define the appropriate value relation might be the following, based on the idea that the result of de-constructing the value (i.e., applying unfold to it) is in the appropriate relation.

$$\mathcal{V}_{\mu X.\, \tau} = \{\mathsf{fold}\ v \mid \mathsf{unfold}\ \mathsf{fold}\ v \in \mathcal{E}_{\tau\{\mu X.\, \tau/X\}}\}$$

We can simplify this a bit, since $\mathsf{unfold}\ \mathsf{fold}\ v$ steps to v, and v is a value.

$$\mathcal{V}_{\mu X.\, \tau} = \{\mathsf{fold}\ v \mid v \in \mathcal{V}_{\tau\{\mu X.\, \tau/X\}}\}$$

However, we now run into well-foundedness issues! The value relation $\mathcal{V}_{\mu X.\, \tau}$ is defined in terms of the value relation $\mathcal{V}_{\tau\{\mu X.\, \tau/X\}}$, which is not strictly smaller. Thus, the definition of $\mathcal{V}_{\mu X.\, \tau}$ may not be a definition per se, but rather a recursive equation that we would like the set $\mathcal{V}_{\mu X.\, \tau}$ to satisfy.

To solve this issue, we index the interpretation by a natural number k: $\mathcal{V}^k_{\tau} = \{v \mid \ldots\}$. The intuition is that $\mathcal{V}^k_{\tau}$ means that "v belongs to the interpretation of τ for k steps." That is, if we run a computation that uses v for k or fewer steps, then we will not notice that it does not have type τ. If we run a computation for *more* than k steps, we might notice that v does not have type τ, which means that we might get stuck. In essence, we are using these indexed relations to mean that the value/expression is type safe for at least k steps.

$$\begin{aligned}
\mathcal{V}^k_{\mathsf{int}} &= \{n \mid n \in \mathbb{Z}\} \\
\mathcal{V}^k_{\tau_1 \to \tau_2} &= \{\lambda x. : \tau_1.\, e \mid \forall j \leq k.\, \forall v \in \mathcal{V}^j_{\tau_1}.\, e\{v/x\} \in \mathcal{E}^j_{\tau_2}\} \\
\mathcal{V}^k_{\mu X.\, \tau} &= \{\mathsf{fold}\ v \mid \forall j < k.\, v \in \mathcal{V}^j_{\tau\{\mu X.\, \tau/X\}}\} \\
\mathcal{E}^j_{\tau} &= \{e \mid \forall j < k.\, \forall e'.\, e \longrightarrow^j e' \wedge \mathrm{irred}(e') \Longrightarrow e' \in \mathcal{V}^{k-j}_{\tau}\}
\end{aligned}$$

Note that we do not need a value interpretation for type variables X, since we never need to interpret a type variable (since recursive types are always closed).

We also need to lift the interpretation of type environments to step-indexing, and we can then lift the definition of semantic type safety.

$$\begin{aligned}
\mathcal{G}[\![\bullet]\!]_k &= \{\emptyset\} \\
\mathcal{G}[\![\Gamma, x:\tau]\!]_k &= \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!]_k, v \in \mathcal{V}^k_\tau\} \\
\Gamma \vDash e:\tau &\triangleq \forall k \geq 0.\, \forall \gamma \in \mathcal{G}[\![\Gamma]\!]_k.\, \gamma(e) \in \mathcal{E}^k_\tau
\end{aligned}$$