Logical relations: safety of system F

Amin Timany

August 17, 2016

Note: in these notes, we simply ignore the issues regarding the clash between variable names, e.g., capturing, by assuming that bound variables are renamed whenever necessary to avoid such problems.

1 Language

1.1 Syntax

$$\begin{array}{lll} \textit{variables}(\mathsf{Var}) & x, y, z, \ldots & \\ \textit{expressions}(\mathsf{E}) & e & ::= \quad x \mid tt \mid (e, e) \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \lambda x.\ e \mid e\ e \mid \Lambda\ e \mid e\ \bullet \\ \textit{values}(\mathsf{Val}) & v & ::= \quad tt \mid (v, v) \mid \lambda x.\ e \mid\ \mid \Lambda\ e \end{array}$$

The set of values is a subset of the set of expressions: $\mathsf{Val} \subset \mathsf{E}$.

1.2 Types and typing

$$\begin{array}{ll} \textit{type variables}(\mathsf{TVar}) & \alpha, \delta, \zeta, \ldots \\ \textit{types}(\mathsf{Typ}) & \tau \quad ::= \quad \alpha \mid () \mid \tau \times \tau \mid \tau \to \tau \mid \forall \alpha.\ \tau \end{array}$$

The typing context that we consider is a pair of contexts Ω the context for type variables and Γ the context for term variables. The "compound" typing context (typing context for short) that we use in defining the typing rules is a pair of context written as $\Omega; \Gamma$.

$$\begin{array}{lll} \textit{typing context}(\mathsf{TCtx}) & \Gamma & ::= \quad \cdot \mid x : \tau, \Gamma \\ \textit{context of typing variables}(\mathsf{TCtx}) & \Omega & ::= \quad \cdot \mid \alpha, \Omega \end{array}$$

Typing rules for our system F are:

$$\frac{}{\Omega; \Gamma \vdash tt : ()}(\textsc{Unit}) \qquad \frac{\Omega; x : \tau \in \Gamma}{\Omega; \Gamma \vdash x : \tau}(\textsc{Var}) \qquad \frac{\Omega; \Gamma \vdash e_1 : \tau_1 \quad \Omega; \Gamma \vdash e_2 : \tau_2}{\Omega; \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}(\textsc{Prod})$$

$$\frac{\Omega; \Gamma \vdash e : \tau_1 \times \tau_2}{\Omega; \Gamma \vdash \mathit{fst}\ e : \tau_1}(\textsc{Fst}) \qquad \frac{\Omega; \Gamma \vdash e : \tau_1 \times \tau_2}{\Omega; \Gamma \vdash \mathit{snd}\ e : \tau_2}(\textsc{Snd}) \qquad \frac{\Omega; x : \tau_1, \Gamma \vdash e : \tau_2}{\Omega; \Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}(\textsc{Lam})$$

$$\frac{\Gamma \vdash e : \tau_1 \to \tau_2 \quad \Gamma \vdash e' : \tau_1}{\Gamma \vdash e\ e' : \tau_2}(\textsc{App}) \qquad \frac{\Omega; \Gamma \vdash e : \tau}{\Omega; \Gamma \vdash \lambda x.\ e : \forall \alpha.\ \tau}(\textsc{TLam}) \qquad \frac{\Omega; \Gamma \vdash e : \forall \alpha.\ \tau}{\Omega; \Gamma \vdash e\ \bullet : \tau[\tau'/\alpha]}(\textsc{TApp})$$

we write $e : \tau$ as a shorthand for $\cdot \vdash e : \tau$.

1.3 Operational semantics (CBV)

We describe the small-step call-by-value (CBV) operational semantics for system F. We do this in two steps. This is more-or-less the standard for describing the semantics of a CBV language. In the first step we give the head reduction relation ($\rightsquigarrow$). In the second step we extend this to non-head reductions using evaluation context (ECtx).

Head reduction:

$$\mathit{fst}\ (v_1, v_2) \rightsquigarrow v_1 \qquad \mathit{snd}\ (v_1, v_2) \rightsquigarrow v_2 \qquad (\lambda x.\ e)\ v \rightsquigarrow e[v/x] \qquad (\Lambda\ e)\ \bullet \rightsquigarrow e$$

Note that here v's are values *and not any expression*. $e[v/x]$ is the expression e where all instances of x are replaced with v. *Remember that all substitutions are capture avoiding*.

Non-head reduction: If the redex (what is being reduced) is not in the head position (see above) then evaluation contexts determine where in the term a reduction can happen.

$$\frac{e \rightsquigarrow e'}{\mathbf{K}[e] \rightsquigarrow \mathbf{K}[e']}$$

where $\mathbf{K}$ is an evaluation context, i.e., $\mathbf{K} \in \mathsf{ECtx}$ and $\mathbf{K}[e]$ is the expression where the single whole in the context $\mathbf{K}$ (see below) is substituted with e.

Evaluation Contexts:

$$\textit{evaluation contexts}(\mathsf{ECtx}) \quad \mathbf{K} \quad ::= \quad \cdot \mid \mathit{fst}\ \mathbf{K} \mid \mathit{snd}\ \mathbf{K} \mid (\mathbf{K}, e) \mid (v, \mathbf{K}) \mid \mathbf{K}\ e \mid v\ \mathbf{K} \mid \mathbf{K}\ \bullet$$

Remark: In the sequel, we will use the word "context" to refer to both typing contexts and evaluation contexts whenever the distinction is clear from the discussion at hand.

Example: The following is the only possible reduction for the expression:

$$\mathit{fst}\ ((\lambda x.\ ((\lambda y.\ tt)\ x, (\lambda y.\ x)\ tt))\ tt)$$

$$\mathit{fst}\ ((\lambda x.\ ((\lambda y.\ tt)\ x, (\lambda y.\ x)\ tt))\ tt) \rightsquigarrow \mathit{fst}\ ((\lambda y.\ tt)\ tt, (\lambda y.\ tt)\ tt) \rightsquigarrow \mathit{fst}\ (tt, (\lambda y.\ tt)\ tt)$$

$$\rightsquigarrow \mathit{fst}\ (tt, tt) \rightsquigarrow tt$$

Exercise: Determine the evaluation context for each step of the reduction above.

We use $\rightsquigarrow^*$ to denote reflexive and transitive closure of $\rightsquigarrow$.

2 Type safety

We say a language is type safe or has the type safety property if:

$$\forall e, \tau.\ e : \tau \Rightarrow \mathbf{Safe}(e)$$

where

$$\mathbf{Safe}(e) \triangleq \forall e'.\ e \rightsquigarrow^* e' \Rightarrow e' \in \mathsf{Val} \vee \exists e''.\ e \rightsquigarrow e''$$

For system F, type safety can be proven using the well-known progress and preservation technique. In these notes we ignore this well-known technique for the sake of developing a simple use cases of logical relations.

3 Logical relations

It seems only natural, given the statement of type safety (with the form $e : \tau$ implying safety), to try proving this statement with induction on the typing derivation. As we will see below, applying induction directly on the statement of type safety does not allow us to prove it.

What we do in this section is attempting to apply induction to type safety and each time this fails we change the statement of the theorem that we prove so that induction is applicable. In each step, we show that the statement that we consider implies type safety and hence proving the theorem considered gives us the desired result.

Attempt 1: If we simply apply induction on the the typing derivation to prove type safety as stated above, we immediately run into a problem in the case APP. In this case we have to prove $\mathbf{Safe}(e\ e')$ and the only thing that we know is that $\mathbf{Safe}(e)$ and $\mathbf{Safe}(e')$. Particularly, notice the case where $e = \lambda x.\ e''$. Since e is already a value, its safety, says absolutely nothing about e''. Note that in this case, we can have $e' \rightsquigarrow^* v'$ and $(\lambda x.\ e'')\ v' \rightsquigarrow^* e''[v'/x]$. In this case, we do not have enough information to show that

$$e''[v'/x] \in \mathsf{Val} \vee \exists e_2.\ e''[v'/x] \rightsquigarrow e_2$$

This is the key observation motivating logical relations. The essence of lambdas is that when applied, we substitute the term they are applied to with the lambda variable. The intuition being, lambdas are functions and when applied (just as is the case in ordinary mathematical functions) we substitute the term they are applied to for the function argument.

The main idea of logical relations is simply just that. In fact, we could summarize the technique of logical relations into the slogan: *Lambdas are functions*. In this regard, we define a relation (in this case a unary relation, i.e., a predicate) defined inductively on the types of the programming language. This predicate, in the case of function types (whose values are lambdas) specifically says that terms are in the predicate if they (vaguely speaking) lambdas that when applied to terms that have the desired property, result in a term with the desired property. We make this clear below.

Note: This issue is not observed in the case of TAPP. In this case, $e \bullet \rightsquigarrow e$ and the safety of e is simply the induction hypothesis.

3.1 The logical predicates for type safety

We define these predicates in two stages. First, we define a set of predicates on values defined recursively on types. In the second stage we extend these predicates to all expressions. Notice that since these predicates are defined by recursion on types and types can have free variables, we need to consider the interpretation of free type variables. We use a partial mapping ξ which maps type variables to predicates.

Safety logical predicates on values ($[\![\Omega \vdash \tau]\!]$)**:** Let us have a firs naïve attempt at defining these logical predicates.

$$
\begin{array}{lcl}
[\![\Omega \vdash \alpha]\!]^{\xi} & \triangleq & \xi(\alpha) \\
[\![\Omega \vdash ()]\!]^{\xi}(v) & \triangleq & v = tt \\
[\![\Omega \vdash \tau_1 \times \tau_2]\!]^{\xi}(v) & \triangleq & \exists v_1, v_2.\ v = (v_1, v_2) \wedge [\![\Omega \vdash \tau_1]\!]^{\xi}(v_1) \wedge [\![\Omega \vdash \tau_1]\!]^{\xi}(v_2) \\
[\![\Omega \vdash \tau_1 \to \tau_2]\!]^{\xi}(v) & \triangleq & \exists e, v = \lambda x.\ e \wedge \forall v'.\ [\![\Omega \vdash \tau_1]\!]^{\xi}(v') \Rightarrow \mathbf{Safe}_{[\![\Omega \vdash \tau_2]\!]^{\xi}}(e[v/x]) \\
[\![\Omega \vdash \forall \alpha.\ \tau]\!]^{\xi}(v) & \triangleq & \exists e, v = \Lambda\ e \wedge \forall \tau'.\ \mathbf{Safe}_{[\![\Omega \vdash \tau[\tau'/\alpha]]\!]^{\xi}}(e) \quad \textit{(incorrect)}
\end{array}
$$

where $[\alpha \mapsto P]\xi$ is extending the partial mapping ξ by additionally mapping α to P and $\mathbf{Safe}_P(e)$ for a predicate P on values is defined as

$$\mathbf{Safe}_P(e) \triangleq \forall e'.\ e \rightsquigarrow^* (e' \Rightarrow e' \in \mathsf{Val} \wedge P(e')) \vee \exists e''.\ e \rightsquigarrow e''$$

The problem with the case $[\![\Omega \vdash \forall \alpha.\ \tau]\!]^{\xi}$ is that this case is not *exactly recursive*. In other words, τ' is not a (inductively-defined) constituent of $\forall \alpha.\ \tau$. Notice that it is provable (and we will indeed use this fact later) that

$$[\![\Omega \vdash \tau[\tau'/\alpha]]\!]^{\xi} \equiv [\![\alpha, \Omega \vdash \forall \alpha.\ \tau]\!]^{[\alpha \mapsto [\![\Omega \vdash \tau'.\ \tau]\!]^{\xi}]\xi}$$

and it might seem that if we replace the left hand side with the right hand side, in the definition above, the problem is solved. However, the problem is deeper than a syntactical problem with the predicate not being syntactically recursively defined. In fact, there is nothing in this definition restricting the choice of τ'. We can for example take τ' to be $\forall \alpha.\ \tau$. This unrestricted choice of τ' is the essence of the problem.

There are two ways that we can address this issue. A syntactical approach and a rather meta theoretical observation. Syntactically we can stratify types into an infinite hierarchy of types.

$$\mathsf{Tp}_0, \mathsf{Tp}_1, \ldots$$

We would then restrict that Tp_0 has no types that include polymorphic types (types of the form $\forall \alpha.\ \tau$). Furthermore, for any type $\forall \alpha.\ \tau$ that belongs to Tp_{n+1}, only a τ' in Tp_n can be used to substitute α in the typing rules. This allows us define the logical relations of system F by recursion on the Tp's and the structure of types in a nested fashion. This approach will certainly work and allow us to prove type safety (and when the logical predicates are adjusted appropriately) to prove (strong) normalization for system F. This way of treating polymorphic types is usually referred to as *predicative* polymorphism. This is the approach taken in languages such as Coq and Agda.

We can however, treat polymorphic types in an *impredicative* way, i.e., not restrict what can be substituted for in a polymorphic type (τ' above). This is based on a (as alluded to earlier) meta theoretical observation (akin to the fact lambdas are functions above). This meta level observation is that polymorphic types are not quantifying over all of the types of the language but rather they are quantifying over all possible types (including those that possibly have no syntactical counterpart in our language). The new logical predicates for system F is as follows:

$$
\begin{array}{lcl}
[\![\Omega \vdash \alpha]\!]^{\xi} & \triangleq & \xi(\alpha) \\
[\![\Omega \vdash ()]\!]^{\xi}(v) & \triangleq & v = tt \\
[\![\Omega \vdash \tau_1 \times \tau_2]\!]^{\xi}(v) & \triangleq & \exists v_1, v_2.\ v = (v_1, v_2) \wedge [\![\Omega \vdash \tau_1]\!]^{\xi}(v_1) \wedge [\![\Omega \vdash \tau_1]\!]^{\xi}(v_2) \\
[\![\Omega \vdash \tau_1 \to \tau_2]\!]^{\xi}(v) & \triangleq & \exists e, v = \lambda x.\ e \wedge \forall v'.\ [\![\Omega \vdash \tau_1]\!]^{\xi}(v') \Rightarrow \mathbf{Safe}_{[\![\Omega \vdash \tau_2]\!]^{\xi}}(e[v/x]) \\
[\![\Omega \vdash \forall \alpha.\ \tau]\!]^{\xi}(v) & \triangleq & \exists e, v = \Lambda\ e \wedge \forall P \subseteq 2^{\mathsf{Val}}.\ \mathbf{Safe}_{[\![\alpha, \Omega \vdash \tau]\!]^{[\alpha \mapsto P]\xi}}(e)
\end{array}
$$

Safety logical predicates on expressions $\left([\![\Omega \vdash \tau]\!]^{\xi}_{\mathsf{E}}\right)$:

$$[\![\Omega \vdash \tau]\!]^{\xi}_{\mathsf{E}} \triangleq \mathbf{Safe}_{[\![\Omega \vdash \tau]\!]^{\xi}}(e)$$

Note: It is obvious that $[\![\Omega \vdash \tau]\!]^{\xi}_{\mathsf{E}}$ defined above implies $\mathbf{Safe}(e)$.

Attempt 2: Let us have another attempt at proving this property by proving the statement below which implies type safety.

$$\forall e, \tau.\ e : \tau \Rightarrow [\![\tau]\!]_{\mathsf{E}}(e)$$

Case APP: Let us consider the case of APP which was problematic before. We need to show that $[\![\tau_2]\!]_{\mathsf{E}}(e\ e')$ knowing that $[\![\tau_1 \to \tau_2]\!]_{\mathsf{E}}(e)$ and $[\![\tau_1]\!]_{\mathsf{E}}(e')$. We know that either **(case 1)** $e\ e' \rightsquigarrow^* e''\ e'$ if $e \rightsquigarrow^* e''$ or **(case 2)** to $e\ e' \rightsquigarrow^* e\ e''$ if e is already a value and $e' \rightsquigarrow^* e''$.

Let us consider **(case 1)**. In **(case 1)**, safety of e implies that e'' is either **(case 1.1)** a value (this case is analogous to **(case 2)** above) or **(case 1.2)** there exists an e_2 such that $e'' \rightsquigarrow e_2$. In **(case 1.2)**, we have $e''\ e' \rightsquigarrow e_2\ e'$ which proves the right hand side case of the disjunction in $\mathbf{Safe}_{[\![\tau_2]\!]_{\mathsf{E}}}(e\ e')$.

Now, let us consider **(case 2)** (and **(case 1.1)** by analogy). In this case, e is a value and therefore we have $[\![\tau_1 \to \tau_2]\!]_{\mathsf{E}}(e)$ and thus $e = \lambda x.\ e_1$. In this case, $e\ e' \rightsquigarrow^* e\ e''$ if $e' \rightsquigarrow^* e''$. Safety of e' now implies that either e'' is a value **(case 2.1)** or **(case 2.2)**there is an e_2 such that $e'' \rightsquigarrow e_2$. In **(case 2.1)** we need to show that $[\![\tau_1 \to \tau_2]\!]_{\mathsf{E}}((\lambda x.\ e_1)\ e'')$ where e'' is a value. This simply follows from the fact that $(\lambda x.\ e_1)\ e'' \rightsquigarrow e_1[e''/x]$ and the definition of $[\![\tau_1 \to \tau_2]\!]_{\mathsf{E}}$. In **(case 2.2)**, we have $e'' \rightsquigarrow e_2$ and therefore $e\ e'' \rightsquigarrow e\ e_2$ which proves the right hand disjunct of $\mathbf{Safe}_{[\![\tau_2]\!]_{\mathsf{E}}}(e\ e')$.

Case LAM: In proving the case LAM we immediately hit a problem. There is no induction hypothesis. Notice that the induction is on $e : \tau$ which is a shorthand for $\cdot \vdash e : \tau$. Now $e = \lambda x.\ e_1$ and $\tau = \tau_1 \to \tau_2$. The premise, therefore, is $x : \tau_1 \vdash e : \tau_2$. This differs from the

statement of the induction. In other words, we have not accounted for variables in the context (free variables) as our original statement was over closed terms.

Therefore, we have to strengthen the statement that we have to prove once more. We proceed by defining sequences of terms in the predicate. Then, we say that any term that is well-typed under a context, if its free variables are filled with terms in the respective predicates, the resulting term is in the predicate.

Sequences of terms in the predicate A sequence of terms $vs = v_1, \ldots, v_n$ is said to be in the predicates for a context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, written as $[\![\Gamma]\!](vs)$ if

$$\begin{array}{lcl} [\![\cdot]\!](vs) & \triangleq & |vs| = 0 \\ [\![x : \tau, \Gamma]\!](vs) & \triangleq & vs = v_1, vs' \wedge [\![\tau]\!](v_1) \wedge [\![\Gamma]\!](vs') \end{array}$$

Attempt 3 In this final attempt we prove the following theorem, known as the fundamental theorem (or fundamental lemma) of logical relations.

Theorem 1 (Fundamental theorem of logical relations)**.** *For any e, Γ and τ such that $\Gamma \vdash e : \tau$ we have:*

$$\forall vs.\ [\![\Gamma]\!](vs) \Rightarrow [\![\tau]\!]_{\mathsf{E}}(e[vs/xs])$$

where xs is the sequence of variables of Γ and $e[v_1, \ldots, v_n/x_1, \ldots, x_n]$ is the term e where v_i's are substituted with x_is simultaneously

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. The case of APP is very similar to its proof in the previous attempt. The only thing to note is that $(e\ e')[vs/xs] = e[vs/xs]\ e'[vs/xs]$.

Let us consider the case LAM. In this case, we have to show that

$$\forall vs.\ [\![\Gamma]\!](vs) \Rightarrow [\![\tau_1 \to \tau_2]\!]_{\mathsf{E}}((\lambda x.\ e)[vs/xs])$$

Since $(\lambda x.\ e)$ is already a value, we have to show that:

$$\forall vs.\ [\![\Gamma]\!](vs) \Rightarrow [\![\tau_1 \to \tau_2]\!]((\lambda x.\ e)[vs/xs])$$

and equivalently

$$\forall vs.\ [\![\Gamma]\!](vs) \Rightarrow [\![\tau_1 \to \tau_2]\!](\lambda x.\ e[vs/xs])$$

which is equivalent to

$$\forall vs.\ [\![\Gamma]\!](vs) \Rightarrow \forall v.\ [\![\tau_1]\!](v) \Rightarrow [\![\tau_2]\!](e[vs/xs])[x/v]$$

and by induction hypothesis we have

$$\forall vs.\ [\![x : \tau, \Gamma]\!](vs) \Rightarrow [\![\tau_2]\!](e[vs/xs])$$

It is simple to see that $(e[vs/xs])[x/v] = e[v, vs/x, xs]$ which concludes the case LAM.

We leave the rest of the cases as an easy exercise. □