Structural Logical Relations with Case Analysis and Equality Reasoning

Ulrik Rasmussen
DIKU, University of Copenhagen
dolle@diku.dk

Andrzej Filinski
DIKU, University of Copenhagen
andrzej@diku.dk

Abstract

Formalizing proofs by logical relations in the Twelf proof assistant is known to be notoriously difficult. However, as demonstrated by Schürmann and Sarnat [In *Proc. of 23rd Symp. on Logic in Computer Science*, 2008] such proofs can be represented and verified in Twelf if done so using a Gentzen-style auxiliary assertion logic which is subsequently proved consistent via cut elimination.

We demonstrate in this paper an application of the above methodology to proofs of observational equivalence between expressions in a simply typed lambda calculus with a call-by-name operational semantics. Our use case requires the assertion logic to be extended with reasoning principles not present in the original presentation of the formalization method. We address this by generalizing the assertion logic to include dependent sorts, and demonstrate that the original cut elimination proof continues to apply without modification.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems, Proof theory

Keywords logical frameworks, logical relations, Twelf, cut elimination, observational equivalence

1. Introduction

Logical relations are a proof technique frequently employed in the study of programming languages based on typed lambda calculi. Apart form Tait's original normalization argument [14], they have been used to prove several properties, including, for example, completeness of equivalence checking [2], or observational equivalence of expressions [5].

The Twelf proof assistant [7] is a meta-logical framework for representing and verifying meta-theorems for deductive systems represented in the LF logical framework [3]. As LF provides a very lightweight way of representing binders and variables via *higher-order abstract syntax*, Twelf is especially well-suited for mechanizing the meta-theory of programming languages, as long as meta-theorems can be expressed in the form of ∀∃-statements. This restriction has still allowed a surprisingly large body of proofs to be represented in Twelf, including, for example, type-safety of the Standard ML programming language [6]. However, proofs by logical relations have been notoriously difficult to formalize, due to their relation-preserving definition at function types which is incompatible with the restriction of meta-theorems to only ∀∃-statements [4].

1.1 Structural logical relations

The technique of *structural logical relations* [13] enables the formalization of certain proofs by logical relations by explicitly representing and reasoning about an auxiliary logic, called the *assertion logic*. Certain core parts of the proofs are then represented in this logic, which is subsequently proved consistent via cut-elimination. The technique additionally gives some interesting insight into the structure of a logical-relations based proof, as it defines a clear separation between the different levels of reasoning: induction lives exclusively on the meta-level (Twelf), while implication and object-language judgments live in the assertion logic.

We will illustrate the general methodology by an example. Consider the following language defined as a Twelf signature:

```
tp : type.
tbool : tp.
tarrow : tp -> tp -> tp.

exp : type.
true : exp.
false : exp.
app : exp -> exp -> exp.
lam : (exp -> exp) -> exp.

of : exp -> tp -> type.
%{ standard typing rules ... }%

eval : exp -> exp -> type.
eval/app : eval E1 (lam E0)
       -> eval (E0 E2) V
       -> eval (app E1 E2) V.
%{ other standard evaluation rules ... }%
```

Suppose that we wanted to prove that every ground-typed expression terminate (i.e., has an evaluation derivation), which we would express as the following meta-theorem:

```
term : of E tbool -> eval E V -> type.
%mode term +OP -EP.
```


LFMTP '13, September 23, 2013, Boston, MA, USA.

ACM 978-1-4503-2382-6/13/09. . . $15.00.
http://dx.doi.org/10.1145/2503887.2503891

However, we cannot prove this by simple induction over the typing derivation, but must use a logical-relations argument, which is problematic to represent directly in Twelf. Instead, we introduce an auxiliary logic with the following definition of formulas and judgments:

```
form : type.
forall : (exp -> form) -> form.
exists : (exp -> form) -> form.
==> : form -> form -> form. %infix right 1 ==>.
@eval : exp -> exp -> form.

pf : form -> type.
```

In the above, the `pf` type family represents the proof judgment for first-order logic. E.g., we could define `pf` as a natural deduction system, formulating the rules for implication and conjunction as follows:

```
forallI : ({x:exp} pf (A x)) -> pf (forall A).
forallE : {x:exp} pf (forall A) -> pf (A x).
existsI : {x:exp} pf (A x) -> pf (exists A).
existsE : pf (exists A)
       -> ({x:exp} pf (A x) -> pf C)
       -> pf C.
impI : (pf A -> pf B) -> pf (A ==> B).
impE : pf (A ==> B) -> pf A -> pf B.
```

The rules for the connective `@eval` should be introduction rules only, and should be defined such that `@eval` characterizes evaluation. For example, we may formulate a set of introduction rules similar in structure to the rules for the `eval` judgment:

```
@eval/app : pf (@eval E1 (lam E0))
          -> pf (@eval (E0 E2) V)
          -> pf (@eval (app E1 E2) V).
%{ remaining rules elided ... }%
```

The logical relation that we need for our proof is then expressed in terms of the formulas of the assertion logic. For our termination proof, we might define it as follows:

```
lr : {T:tp} (exp -> form) -> type.
lr/bool : lr tbool ([e] exists [v] @eval e v).
lr/arr : lr T2 R2 -> lr T0 R0
         -> lr (tarrow T2 T0)
               ([e] forall [e2]
                     R2 e2 ==> R0 (app e e2)).
```

Belonging to the logical relation at base type implies by definition an assertion-logic proof of termination.

We then prove the *fundamental theorem* for our logical relation, namely that all well-typed terms satisfy it:

```
fund : of E T -> lr T R -> pf (R E) -> type.
%mode fund +OP +LP -PF.
```

From this, an assertion-logic proof of termination at boolean types follows as a special case:

```
bool-eval : of E bool
          -> pf (exists [v] @eval E v) -> type.
%mode bool-eval +OP -PF.
```

At this point, we could in principle stop and declare that we have proved that all well-typed expressions terminate. However, this requires us to trust several aspects of the proof. First of all, we have to believe that a derivation of `pf (exists [v] @eval E v)` indeed implies the existence of a derivation of `eval E V` for some `V`. Assertion-logic proofs are not necessarily on normal form, e.g., a derivation of `pf (exists [v] @eval E v)` could possibly end in a use of `impE` operating on proofs of much larger formulas, which significantly complicates a soundness proof.

Instead, we define an alternative proof judgment:

```
pfn : form -> type.
```

The `pfn` system is defined to be similar to `pf`, but restricted such that only proofs on normal form can be expressed. This allows the following *extraction* theorem to be proved by straightforward induction over derivations:

```
ext : pfn (exists [v] @eval E v) -> eval E V -> type.
%mode ext +PF -EP.
```

It is not practical to construct normalized proof derivations directly in the proof of the fundamental theorem, however, so we still have to prove the following normalization theorem:

```
norm : pf A -> pfn A -> type.
%mode norm +PF -PF'.
```

Our main theorem can now be formulated and verified by the Twelf meta-theorem prover:

```
term : of E bool -> eval E V -> type.
%mode term +OP -EP.
- : term OP EP
      <- bool-eval OP PF
      <- norm PF PFN
      <- ext PFN EP.
```

Interestingly, the normalization proof for the assertion logic can also be considered a termination proof, but for a logic which admits a structural proof.

We refer to the original presentation of structural logical relations [13] for a more detailed example of the formalization of proofs of weak normalization and completeness of equivalence checking, using the same proof structure as sketched above, and with a Gentzen-style sequent calculus for the assertion logic. Showing that assertion-logic proofs can be normalized thus reduces to showing that cut can be eliminated from proofs, which has previously been formalized in Twelf by Pfenning [8].

This methodology works well for formalizing proofs about a typed lambda calculus with only application and abstraction. However, the method does not generalize in a straightforward way to the formalization of meta-theory for even slightly more expressive programming languages, such as, for example, a language with natural numbers and case expressions.

The reason is that core parts of the proof of the fundamental theorem must be conducted in the assertion logic, which needs to be extended with additional reasoning principles when working with a more expressive language. Specificaly, in the termination example, the mathematical proof for case-expressions distinguishes between the two possible evaluation results of a base-typed subexpression, requiring a form of case-analysis principle in the assertion logic as well. However, we cannot just add arbitrary rules to the assertion logic, as we must be able to prove that proofs normalize. In the case of a Gentzen-style calculus, this means that we must not destroy the cut-admissibility property.

1.2 Contributions

The original presentation of the technique of structural logical relations was concerned only with the formalizations of weak normalization and equivalence checking for a minimal typed λ-calculus, using a similarly minimal assertion logic. In this

paper, we will apply the technique to a proof of observational equivalence for expressions in a simply typed λ-calculus with a call-by-name operational semantics. It turns out that the assertion logic needs to be extended with several reasoning principles when scaling the technique to proofs about programming languages. Specifically, adding inhabitants to base types, together with elimination constructs, calls for an assertion logic with reasoning principles for doing case analysis and equality reasoning for derivations. We demonstrate how to achieve this while preserving normalizability of the assertion logic, by extending it with quantification over objects in a dependent theory of sorts. Interestingly, the structural cut-elimination proof of Pfenning [8] continues to apply without requiring modification. We further add structural witnesses for doing case analysis on objects, and show how to formulate the theory of sorts to accommodate equality reasoning. The resulting system further distinguishes the levels of reasoning in a proof by logical relations: induction lives on the meta-level, implication and case analysis live in the assertion logic, and equality reasoning and object-language judgments live in the underlying theory of sorts.

The full Twelf proofs using the methodology presented here can be found in the electronic appendix [11].

2. Extended structural logical relations

In this section, we present an extension of the methodology of structural logical relations which admits more advanced reasoning principles in the assertion logic. The method is best described in the context of a concrete example, which will be the formalization of a soundness proof of an equational reasoning system for a simple programming language.

The programming language that we work with is defined in Figure 1, and is a simply typed lambda calculus with a call-by-name operational semantics, natural numbers, a case construct, and possibility of failure. In Figure 4, we have defined the syntactic equational reasoning system for which we will prove soundness. Note that exchange is implicitly assumed for contexts Γ, and that when writing $\Gamma, x : \tau$, we tacitly imply that x does not occur in the domain of Γ. We will write $\mathrm{dom}(\Gamma)$ for the domain of Γ. Given an expression e, we will write $\mathrm{FV}(e)$ for the set of free variables of e. A variable x is *free* if it does not occur as descendent of a binder for x.

The Twelf representation of the object language and the equational reasoning system is straightforward, and we will therefore only show some representative parts of it:

```
nat : type
z : nat.
s : nat -> nat.

tp : type.
tnat : tp.
tarrow : tp -> tp -> tp.

exp : type.
%{ declarations elided }%

eval : exp -> exp -> type.
eval/app : eval E1 (lam E0) -> eval (E0 E2) V
       -> eval (app E1 E2) V.
%{ remaining declarations elided }%

of : exp -> tp -> type.
of/lam : ({x} of x T2 -> of (E0 x) T0)
        -> of (lam E0) (tarrow T2 T0).
%{ remaining declarations elided }%

sim : exp -> exp -> tp -> type.
sim/lam : ({x} sim x x T2
          -> sim (E0 x) (E0' x) T0)
       -> sim (lam E0) (lam E0') (tarrow T2 T0).
%{ remaining declarations elided }%
```

The representation is *adequate*, and we will assume the existence of a compositional translation $\ulcorner\cdot\urcorner$ from our original definitions to LF types and terms. For example, we have $\ulcorner \mathsf{Exp} \urcorner = \mathtt{exp}$ and $\ulcorner \mathsf{app}\ e_1\ e_2 \urcorner = \mathtt{app} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$, where the typewriter typeface is used to denote LF expressions.

Our assertion logic is presented in Figure 2, and is formulated as a Gentzen-style sequent calculus as in the original presentation of the methodology. The definition deserves some explanation: A derivation of a judgment of the form $\Xi | \Delta \vdash^c A$ is an assertion-logic proof of the validity of the formula A, with *parameters* from an *ordered* mapping from meta-variables to sorts Ξ, and using hypothetical assumptions from an *unordered* set of hypotheses Δ. The proof sequent is further parameterized by a *cut tag* c, which denotes whether the proof is normalized (i.e., cut-free); $\bullet$ is the tag for potentially cutful proofs; $\circ$ is the tag for cut-free proofs. The important difference between the original methodology and ours is that we do not add judgments as atomic formulas to the assertion logic (e.g., the `@eval` formula from the introduction), but instead quantify over *dependent sorts* whose objects represent the object language derivations that we are interested in. The logic quantifies over four different sorts, namely expressions Exp; naturals Nat; binders $(\mathsf{Exp})\mathsf{Exp}$; and *data derivations* of the form $\Vdash D$ – the latter is our means of representing object language judgments, which we will get back to in a moment. The definition also mentions a set of unspecified rules related to the *structural witness* predicate Data^+, whose rules will also be defined later when we specify the rules for the judgment $\Vdash D$.

It is tacitly assumed that meta-variables may occur anywhere in objects, e.g., we may form an object of the form $\mathsf{app}\ x\ \alpha$, where x is an expression-variable and α is a meta-variable standing for some arbitrary expression. In the introduction rule for existential quantification and the elimination rules for universal quantification, a concrete object from the appropriate sort must be substituted for a meta-variable. Here, we require not that it is well-formed, but that its *LF encoding* in some signature Σ is a canonical form at the appropriate LF type family. We will take Σ to be the LF representation of our object language syntax as well as the judgment $\Vdash D$. For example, if $\Xi = \cdot, \alpha : \mathsf{Exp}$, then we would have $\ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathsf{app}\ \alpha\ \mathsf{fail} \urcorner : \mathtt{exp}$, but <u>not</u> $\ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \mathtt{s}\ \alpha : \mathtt{nat}$.

The judgment $\Vdash D$ denotes proofs of *data formulas* D in a *data representation logic* which is defined in the next section. The logic is used to encode the object language judgments we are interested in. Data formulas D may contain objects belonging to the Exp sort as well as meta-variables standing for expressions. This effectively equips the assertion logic with dependent sorts.

The Twelf representation of the assertion logic is formulated as follows:

```
% Data formulas and repr. logic judgment.
dform : type.
data : dform -> type.

tag : type.
cutful : tag.
cutfree : tag.
```

$$
\begin{array}{llcll}
\text{Types:} & \tau & :: & \mathsf{Tp} & ::= \ \mathsf{nat} \mid \tau_1 \rightharpoonup \tau_2 \\
\text{Naturals:} & n & :: & \mathsf{Nat} & ::= \ \mathsf{z} \mid \mathsf{s}\ n \\
\text{Expressions:} & e, v & :: & \mathsf{Exp} & ::= \ x \mid \mathsf{app}\ e_1\ e_2 \mid \mathsf{lam}\ x.\, e_0 \mid \mathsf{num}\ n \mid \mathsf{case}(e_0,\, e_1,\, x.\, e_2) \mid \mathsf{fail} \\
\text{Binders:} & b & :: & \mathsf{(Exp)Exp} & ::= \ x.\, e_0 \\
\text{Evaluation:} & \mathcal{E} & :: & \boxed{e \Downarrow v} & :
\end{array}
$$

$$\frac{e_1 \Downarrow \mathsf{lam}\ x.\, e_0 \quad e_0[e_2/x] \Downarrow v}{\mathsf{app}\ e_1\ e_2 \Downarrow v}\ \mathsf{e\_app} \qquad \frac{}{\mathsf{lam}\ x.\, e_0 \Downarrow \mathsf{lam}\ x.\, e_0}\ \mathsf{e\_lam} \qquad \frac{}{\mathsf{num}\ n \Downarrow \mathsf{num}\ n}\ \mathsf{e\_num}$$

$$\frac{e_0 \Downarrow \mathsf{num}\ \mathsf{z} \quad e_1 \Downarrow v}{\mathsf{case}(e_0,\, e_1,\, x.\, e_2) \Downarrow v}\ \mathsf{e\_case0} \qquad \frac{e_0 \Downarrow \mathsf{num}\ (\mathsf{s}\ n) \quad e_2[\mathsf{num}\ n/x] \Downarrow v}{\mathsf{case}(e_0,\, e_1,\, x.\, e_2) \Downarrow v}\ \mathsf{e\_case1}$$

$$\text{Typing:} \qquad \mathcal{T} \ :: \ \boxed{\Gamma \vdash e : \tau} \ :$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightharpoonup \tau_0 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{app}\ e_1\ e_2 : \tau_0}\ \mathsf{t\_app} \qquad \frac{\Gamma, x : \tau_2 \vdash e_0 : \tau_0}{\Gamma \vdash \mathsf{lam}\ x.\, e_0 : \tau_2 \rightharpoonup \tau_0}\ \mathsf{t\_lam} \qquad \frac{}{\Gamma \vdash \mathsf{num}\ n : \mathsf{nat}}\ \mathsf{t\_num}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}(e_0,\, e_1,\, x.\, e_2) : \tau}\ \mathsf{t\_ifz} \qquad \frac{}{\Gamma \vdash \mathsf{fail} : \tau}\ \mathsf{t\_fail}$$

Figure 1. Simply typed λ-calculus.

$$
\begin{array}{llcll}
\text{Cut tag:} & c & :: & \mathsf{Tag} & ::= \ \bullet \mid \circ \\
\text{Formulas:} & A, B, C & :: & \mathsf{Form} & ::= \ \top \mid \forall \alpha : \mathsf{Exp}.\, A \mid \forall \alpha : \mathsf{Nat}.\, A \mid \exists \alpha : (\Vdash D).\, A \mid A \vee B \mid A \wedge B \mid A \supset B \\
 & & & & \quad \mid \mathsf{Data}^{+}(\mathcal{D} : D) \\
\text{Parameters:} & \Xi & :: & \mathsf{Parms} & ::= \ \cdot \mid \Xi, \alpha : \mathsf{Exp} \mid \Xi, \alpha : \mathsf{Nat} \mid \Xi, \alpha : \mathsf{(Exp)Exp} \mid \Xi, \alpha : (\Vdash D) \\
\text{Assumptions:} & \Delta & :: & \mathsf{Assm} & ::= \ \cdot \mid \Delta, A \\
\text{Data formulas:} & D & :: & \mathsf{DForm} & \text{(Defined in Figure 3.)} \\
\text{Data judgment:} & \mathcal{D} & :: & \boxed{\Vdash D} & \text{(Defined in Figure 3.)} \\
\text{Sequents:} & \mathcal{S} & :: & \boxed{\Xi | \Delta \vdash^{c} A} \ : &
\end{array}
$$

Initial sequent and cut:

$$\frac{}{\Xi | \Delta, A \vdash^{c} A}\ \mathsf{ax} \qquad \frac{\Xi | \Delta \vdash^{c} A \quad \Xi | \Delta, A \vdash^{c} C}{\Xi | \Delta \vdash^{\bullet} C}\ \mathsf{cut}$$

Right rules:

$$\frac{}{\Xi | \Delta \vdash^{c} \top}\ \mathsf{topR} \qquad \frac{\Xi | \Delta, A \vdash^{c} B}{\Xi | \Delta \vdash^{c} A \supset B}\ \mathsf{impR} \qquad \frac{\Xi | \Delta \vdash^{c} A \quad \Xi | \Delta \vdash^{c} B}{\Xi | \Delta \vdash^{c} A \wedge B}\ \mathsf{andR} \qquad \frac{\Xi | \Delta \vdash^{c} A}{\Xi | \Delta \vdash^{c} A \vee B}\ \mathsf{orR1} \qquad \frac{\Xi | \Delta \vdash^{c} B}{\Xi | \Delta \vdash^{c} A \vee B}\ \mathsf{orR2}$$

$$\frac{\Xi, \alpha : \mathsf{Exp} | \Delta \vdash^{c} A}{\Xi | \Delta \vdash^{c} \forall \alpha : \mathsf{Exp}.\, A}\ \mathsf{allR\_e} \qquad \frac{\Xi, \alpha : \mathsf{Nat} | \Delta \vdash^{c} A}{\Xi | \Delta \vdash^{c} \forall \alpha : \mathsf{Nat}.\, A}\ \mathsf{allR\_n} \qquad \frac{\Xi | \Delta \vdash^{c} A[\mathcal{D}/\alpha] \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathcal{D} \urcorner : \ulcorner \Vdash D \urcorner}{\Xi | \Delta \vdash^{c} \exists \alpha : (\Vdash D).\, A}\ \mathsf{exiR\_d}$$

(+ right-rules for structural witness. Defined in Section 2.2.)

Left rules:

$$\frac{\Xi | \Delta, A \supset B \vdash^{c} A \quad \Xi | \Delta, A \supset B, B \vdash^{c} C}{\Xi | \Delta, A \supset B \vdash^{c} C}\ \mathsf{impL} \qquad \frac{\Xi | \Delta, A \wedge B, A \vdash^{c} C}{\Xi | \Delta, A \wedge B \vdash^{c} C}\ \mathsf{andL1} \qquad \frac{\Xi | \Delta, A \wedge B, B \vdash^{c} C}{\Xi | \Delta, A \wedge B \vdash^{c} C}\ \mathsf{andL2}$$

$$\frac{\Xi | \Delta, A \vee B, A \vdash^{c} C \quad \Xi | \Delta, A \vee B, B \vdash^{c} C}{\Xi | \Delta, A \vee B \vdash^{c} C}\ \mathsf{orL} \qquad \frac{\Xi | \Delta, \forall \alpha : \mathsf{Exp}.\, A, A[e/\alpha] \vdash^{c} C \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner e \urcorner : \mathtt{exp}}{\Xi | \Delta, \forall \alpha : \mathsf{Exp}.\, A \vdash^{c} C}\ \mathsf{allL\_e}$$

$$\frac{\Xi | \Delta, \forall \alpha : \mathsf{Nat}.\, A, A[n/\alpha] \vdash^{c} C \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner n \urcorner : \mathtt{nat}}{\Xi | \Delta, \forall \alpha : \mathsf{Nat}.\, A \vdash^{c} C}\ \mathsf{allL\_n} \qquad \frac{\Xi, \alpha' : (\Vdash D) | \Delta, \exists \alpha : (\Vdash D).\, A, A[\alpha'/\alpha] \vdash^{c} C}{\Xi | \Delta, \exists \alpha : (\Vdash D).\, A \vdash^{c} C}\ \mathsf{exiL\_d}$$

(+ left-rule for structural witness. Defined in Section 2.2.)

LF translation of parameters:

$$
\begin{array}{rcl@{\qquad}rcl}
\ulcorner . \urcorner & = & . & \ulcorner \Xi, \alpha : \mathsf{Exp} \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_{\alpha} : \mathtt{exp} \\
\ulcorner \Xi, \alpha : \mathsf{(Exp)Exp} \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_{\alpha} : \mathtt{exp\ ->\ exp} & \ulcorner \Xi, \alpha : (\Vdash D) \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_{\alpha} : \mathtt{data}\ \ulcorner D \urcorner
\end{array}
$$

Figure 2. Assertion logic.

```
form : type.
top : form.
forall_e : (exp -> form) -> form.
forall_n : (nat -> form) -> form.
exists_d : (data D -> form) -> form.
==> : form -> form -> form. %infix right 1 ==>.
\/ : form -> form -> form. %infix left 2.
/\ : form -> form -> form. %infix left 3.
data+ : data D -> form.

hyp : form -> type.
% Proof judgment:
conc : tag -> form -> type.
%{ rules elided }%
```

Note that in the above, assumptions are modeled using the `hyp` type family to distinguish assumptions from derivations. For example, the left and right rules for implication look as follows:

```
impr : (hyp F -> conc T G)
       -> conc T (F ==> G).
impl : conc T F -> (hyp G -> conc T C)
       -> (hyp (F ==> G) -> conc T C).
```

Parameter contexts Ξ are represented directly using the LF context, which is reflected in the higher-order representation of the quantifiers. Our requirement that concrete objects must correspond to well-typed canonical LF terms is also directly represented; e.g., the left-rule for universal quantification over expressions looks as follows:

```
foralll_e : {e:exp} (hyp (A e) -> conc T C)
  -> (hyp (forall_e A) -> conc T C).
```

Case-analysis principles are provided via structural witnesses, which are proofs of the atomic predicate Data^+. The exact rules are yet to be specified, but will provide a way to observe the structure of certain representation-logic derivations within assertion-logic proofs. To be able to reason from the expression equalities that follow from each possible case, we need to structure the representation logic carefully; this will be covered in the next section.

2.1 Representation logic

In Twelf, equality reasoning is often done implicitly via unification of LF variables when pattern-matching on possible cases. More important, impossible cases are in most cases automatically ruled out by the coverage-checking algorithm (although in general the problem is undecidable), and need not be covered in proofs. However, any case-analysis rules we add to our assertion logic will not have access to any meta-level facilities of Twelf, and must therefore reason from explicit syntactic equality proofs whose structure we cannot observe. When we cannot observe the structure of an equality proof, we have to give it meaning in some other way, namely by adding conversion rules which allows us to convert derivations involving equal objects.

The representation logic, defined in Figure 3, gives us a way to represent object language judgments (in this case, the evaluation judgment) while enabling such equality reasoning. Specifically, it allows equality proofs to be inserted at any point in a derivation, allowing equality conversions to be carried out without relying on unification of meta-variables by observing the structure of equality proofs. We write all formulas of this logic with banana brackets $(\!|\cdot|\!)$ to visually distinguish them from the original judgments they encode. Note how all syntactic constraints in the original rules of the evaluation judgment are expressed via explicit equality premises in the representation logic – this is indeed the main technical feature of the representation logic, and all evaluation rules can be derived mechanically from their original definitions. There is a large number of rules for reasoning about equalities, so only a few representatives are shown. Note that we also identify absurd equalities via $(\!|\,\mathsf{void}\,|\!)$, from which any equality can be derived – and hence also any evaluation via one of the rules that only have equality premises, such as `de_num`.

The representation logic is represented as follows in Twelf:

```
dform : type.
dvoid : dform.
dqe : exp -> exp -> dform.
dqe2 : (exp -> exp) -> (exp -> exp) -> dform.
deval : exp -> exp -> dform.

data : dform -> type.
deval/app : data (deval E1 (lam E0))
            -> data (deval (E0 E2) Ev)
            -> data (dqe X1 (app E1 E2))
            -> data (dqe X2 Ev)
            -> data (deval X1 X2).
%{ remaining rules elided }%
```

The logic is sound and complete with respect to the original definition of evaluation, i.e., we can prove the following meta-theorems:

Lemma 1 (Isomorphism)**.** *For any expressions e,e', we have $e \Downarrow e' \iff \Vdash (\!| e \Downarrow e' |\!)$.*

```
emb : eval E E' -> data (deval E E') -> type.
%mode emb +EP -DP.

unemb : data (deval E E') -> eval E E' -> type.
%mode unemb +DP -EP.
```

Proof. By induction over derivations. The "if" direction requires mutual induction with a soundness proof for equality derivations as well. □

Note that unlike the assertion logic, all representation-logic derivations are on normal form and need not be normalized as part of the proof for `unemb`.

2.2 Case analysis

It remains to define a case-analysis principle for the assertion logic, such that we can express proofs that consider all the ways in which a given representation-logic derivation could have been constructed. To do so, we define a set of rules for the Data^+ predicate, which acts as a witness for the structure of a given derivation.

Due to the presence of judgments-in-judgments, the paper definition of the rules are rather cluttered and spacious, so we will only show a representative part of their Twelf representations. The idea is to introduce a right-rule for each possible representation-logic rule that we want to consider in case-analysis proofs. For our purposes, it suffices to add five right-rules, one for each `de`-prefixed rule, which enables case analysis on evaluation derivations. For example, the right-rule for `de_app` looks as follows:

```
data+R_app :
  conc T (data+ DP1)
  -> conc T (data+ DP2)
  -> conc T (data+ (deval/app DP1 DP2 DP3 DP4)).
```

Data formulas: $D :: \mathsf{DForm} ::= (\!|\mathsf{void}|\!) \mid (\!|e \stackrel{\star}{=} e'|\!) \mid (\!|x.\, e \stackrel{\star}{=}_2 x'.\, e'|\!) \mid (\!|e \Downarrow v|\!)$

Judgment: $\mathcal{D} :: \boxed{\Vdash D} :$

Evaluation:

$$\frac{\Vdash (\!|e_1 \stackrel{\star}{=} \mathsf{lam}\ x.\, e_0|\!) \quad \Vdash (\!|e_2 \stackrel{\star}{=} \mathsf{lam}\ x.\, e_0|\!)}{\Vdash (\!|e_1 \Downarrow e_2|\!)}\ \mathsf{de\_lam} \qquad \frac{\Vdash (\!|e_1 \stackrel{\star}{=} \mathsf{num}\ n|\!) \quad \Vdash (\!|e_2 \stackrel{\star}{=} \mathsf{num}\ n|\!)}{\Vdash (\!|e_1 \Downarrow e_2|\!)}\ \mathsf{de\_num}$$

$$\frac{\Vdash (\!|e_1' \Downarrow \mathsf{lam}\ x.\, e_0|\!) \quad \Vdash (\!|e_0[e_2'/x] \Downarrow v|\!) \quad \Vdash (\!|e_1 \stackrel{\star}{=} \mathsf{app}\ e_1'\ e_2'|\!) \quad \Vdash (\!|e_2 \stackrel{\star}{=} v|\!)}{\Vdash (\!|e_1 \Downarrow e_2|\!)}\ \mathsf{de\_app}$$

$$\frac{\Vdash (\!|e_0' \Downarrow \mathsf{num}\ \mathsf{z}|\!) \quad \Vdash (\!|e_1' \Downarrow v|\!) \quad \Vdash (\!|e_1 \stackrel{\star}{=} \mathsf{case}(e_0', e_1', x.\, e_2')|\!) \quad \Vdash (\!|e_2 \stackrel{\star}{=} v|\!)}{\Vdash (\!|e_1 \Downarrow e_2|\!)}\ \mathsf{de\_case0}$$

$$\frac{\Vdash (\!|e_0' \Downarrow \mathsf{num}\ (\mathsf{s}\ n')|\!) \quad \Vdash (\!|e_2'[\mathsf{num}\ n'/x] \Downarrow v|\!) \quad \Vdash (\!|e_1 \stackrel{\star}{=} \mathsf{case}(e_0', e_1', x.\, e_2')|\!) \quad \Vdash (\!|e_2 \stackrel{\star}{=} v|\!)}{\Vdash (\!|e_1 \Downarrow e_2|\!)}\ \mathsf{de\_case1}$$

Equality and falsehood (representative subset):

$$\frac{\Vdash (\!|\mathsf{void}|\!)}{\Vdash (\!|e \stackrel{\star}{=} e'|\!)}\ \mathsf{dqe\_void} \qquad \frac{}{\Vdash (\!|e \stackrel{\star}{=} e|\!)}\ \mathsf{dqe\_id} \qquad \frac{\Vdash (\!|e \stackrel{\star}{=} e'|\!)}{\Vdash (\!|e' \stackrel{\star}{=} e|\!)}\ \mathsf{dqe\_sym} \qquad \frac{\Vdash (\!|e \stackrel{\star}{=} e'|\!) \quad \Vdash (\!|e' \stackrel{\star}{=} e''|\!)}{\Vdash (\!|e \stackrel{\star}{=} e''|\!)}\ \mathsf{dqe\_trans}$$

$$\frac{\Vdash (\!|x.\, e_0 \stackrel{\star}{=}_2 x'.\, e_0'|\!) \quad \Vdash (\!|e \stackrel{\star}{=} e'|\!)}{\Vdash (\!|e_0[e/x] \stackrel{\star}{=} e_0'[e'/x']|\!)}\ \mathsf{dqe\_subst} \qquad \frac{\Vdash (\!|\mathsf{lam}\ x.\, e_0 \stackrel{\star}{=} \mathsf{lam}\ x'.\, e_0'|\!)}{\Vdash (\!|x.\, e_0 \stackrel{\star}{=}_2 x'.\, e_0'|\!)}\ \mathsf{dqe\_cvrs\_lam}$$

$$\frac{\Vdash (\!|\mathsf{app}\ e_1\ e_2 \stackrel{\star}{=} \mathsf{app}\ e_1'\ e_2'|\!)}{\Vdash (\!|e_1 \stackrel{\star}{=} e_1'|\!)}\ \mathsf{dqe\_cvrs\_app1} \qquad \frac{\Vdash (\!|\mathsf{app}\ e_1\ e_2 \stackrel{\star}{=} \mathsf{app}\ e_1'\ e_2'|\!)}{\Vdash (\!|e_2 \stackrel{\star}{=} e_2'|\!)}\ \mathsf{dqe\_cvrs\_app2}$$

$$\frac{\Vdash (\!|\mathsf{case}(e_0, e_1, x.\, e_2) \stackrel{\star}{=} \mathsf{case}(e_0', e_1', x'.\, e_2')|\!)}{\Vdash (\!|e_0 \stackrel{\star}{=} e_0'|\!)}\ \mathsf{dqe\_cvrs\_case0}$$

(dqe_cvrs_case1 and dqe_cvrs_case2 are defined similarly.)

$$\frac{\Vdash (\!|\mathsf{app}\ e_1\ e_2 \stackrel{\star}{=} \mathsf{lam}\ x.\, e_0|\!)}{\Vdash (\!|\mathsf{void}|\!)}\ \mathsf{dqe\_app\_lam} \quad \cdots \quad \text{(Similar rules for all 15 pairs of distinct constructors.)}$$

$$\frac{}{\Vdash (\!|x.\, e_0 \stackrel{\star}{=}_2 x.\, e_0|\!)}\ \mathsf{dqe2\_id} \quad \cdots \quad \text{(Similar rules as above.)}$$

Figure 3. Representation logic

Note that, although `deval/app` has four premises, we only require structural witnesses for the subderivations concerned with the evaluation judgment, as we are not interested in the structure of equality proofs. We define four similar right-rules for de_lam, de_num, de_case0 and de_case1. Finally, we define a single left-rule which acts as an elimination rule for the structural witness Data^+. The rule is similar to elimination for disjunction, in that it has a premise per introduction rule. The rule is represented in Twelf as follows:

```
data+L :
  (%{case for de_lam ...}%)
  -> (%{case for de_num}%
      {N0}
      {q1:data (dqe X1 (num N0))}
      {q2:data (dqe X2 (num N0))}
        conc V C)
  -> (%{case for de_app}%
      {Ev}{E1}{E2}{E0}
      {dp1:data (deval E1 (lam E0))}
      {dp2:data (deval (E0 E2) Ev)}
      {h1:hyp (data+ dp1)}{h2:hyp (data+ dp2)}
      {dp3:data (dqe X1 (app E1 E2))}
      {dp4:data (dqe X2 Ev)}
        conc V C)
  -> (%{case for de_case0 ...}%)
  -> (%{case for de_case1 ...}%)
  -> (hyp (data+ (DP : data (deval X1 X2)))
       -> conc V C).
```

For space reasons, we have elided the full types for the cases for de_lam, de_case0 and de_case1; their definitions should be clear from the context.

The above definition should also make the motivation for introducing the representation logic more clear: Note how the meta-variables `X1` and `X2` appear both in the type of the representation-logic derivation that we are doing case analysis on, as well as in the types of each proof case, where specifically they only appear in equality formulas. If the equality constraints had not been expressed via explicit equality proofs, we would have run into scoping issues in the definition of the above rule.

In practice, we want structural-witness proofs to "travel" with all evaluation derivations. Using Twelf's support for definitions, we thus define the following formula abbreviation to denote assertion-logic proofs of evaluation:

```
#eval : exp -> exp -> form
  = [e][v] existsd [dp:deval e v] data+ dp.
```

2.3 Cut-elimination

In a sequent calculus, cut-free proofs are exactly the derivations on normal form. Proving that all derivations normalize therefore reduces to proving *cut-elimination*, i.e., proving that if $\Xi|\Delta \vdash^\bullet A$, then also $\Xi|\Delta \vdash^\circ A$. For the fragment of the assertion logic without rules for structural witnesses, we can use the method of Pfenning [8] without any alterations. We first prove *cut-admissibility* for the cut-free fragment of the logic, i.e.:

Lemma 2 (Cut-admissibility)**.** *If we have* $\Xi|\Delta \vdash^\circ A$ *and* $\Xi|\Delta, A \vdash^\circ C$*, then also* $\Xi|\Delta \vdash^\circ C$*.*

```
ca : {A} conc cutfree A -> (hyp A -> conc cutful C)
  -> conc cutful C -> type.
%mode ca +F +PF1 +PF2 -PF'.
```

Proof sketch. By lexicographic induction, first on the cut formula A, followed by simultaneous induction on the two proof derivations. □

Uses of the cut rule can then be eliminated by a bottom-up procedure which works by simple induction over derivations:

Lemma 3 (Cut-elimination)**.** *If* $\Xi|\Delta \vdash^\bullet A$*, then also* $\Xi|\Delta \vdash^\circ A$*.*

```
ce : conc cutful A -> conc cutfree A -> type.
%mode ce +PF -PF'.
```

Proof sketch. By induction on derivations, appealing to Lemma 2 in the case for cut. □

Although Pfenning's cut-admissibility proof was originally formulated for a single-sorted logic, it generalizes directly to the dependently-sorted case without requiring alterations. However, the addition of rules for reasoning by case-analysis (i.e., the rules for the Data^+ connective) requires the termination measure to be strengthened. The cut-admissibility proof uses the cut formula as the outermost termination measure, relying on the fact that the premises of right-rules always prove strict subformulas of the full derivation. However, this property does not hold for the right-rules for the Data^+ connective, where both the premises and the full derivation end in formulas of the form $\mathsf{Data}^+(\mathcal{D})$ for some representation logic derivation $\mathcal{D}$, and the formulas of the premises hence fail to get smaller. The inner representation-logic derivations *do* get smaller, though, which we can capture using an auxiliary measure on formulas. First, we define a syntax for formula skeletons:

```
skel : type.
kzero : skel.
kunary : skel -> skel.
kbinary : skel -> skel -> skel.
```

Our measure is a total relation transforming formulas to skeletons and data derivations. We show three representative rules, which looks as follows:

```
msre : form -> skel -> data D -> type.
msre/and :
  msre F1 K1 DP1
  -> msre F2 K2 DP2
  -> msre (F1 /\ F2) (kbinary K1 K2) dqe_id.
msre/forall_e :
  ({x:exp} msre (F x) K (DP x : data (D x))
  -> msre (forall_e F) (kunary K) dqe_id.
msre/data+ : msre (data+ DP) kzero DP.
%{ ... }%
```

The measure is total, which can be proved as an *effectiveness lemma* by trivial induction over formulas.

If we are given measure derivations `msre A1 K1 DP1` and `msre A2 K2 DP2`, where `A1` is a subformula of `A2`, then `K1` is also a sub-skeleton of `K2`, justifying the use of skeletons as a drop-in replacement for formulas as the outermost termination metric in the cut-admissibility proof. The measure is also defined such that if we are given a derivation of `msre A K DP`, then `DP` is equal to the "dummy" value `dqe_id` for all formulas, except for formulas of the form `data+ DP'`, in which case `DP` is chosen to be `DP'` and `K` is chosen to be `kzero`. This allows us to continue using representation logic derivations as a termination metric when formulas do not suffice.

We can thus refine the Twelf formulation of the cut-admissiblity theorem as follows:

```
ca : {A}{K}{DP} measure A K DP
  -> conc cutfree A -> (hyp A -> conc cutful C)
  -> conc cutful C -> type.
%mode ca +F +K +DP +M +PF1 +PF2 -PF'.
```

The proof proceeds by lexicographic induction on formula skeletons `K`, followed by representation logic derivations `DP`, and then simultaneous induction over the two assertion-logic derivations. The proof for the cut-elimination theorem requires only minor adjustments, requiring an extra invocation of the effectiveness lemma for the `msre` type family when appealing to the cut-admissibility lemma.

3. Observational equivalence for simply typed λ-expressions

In this section, we will describe the formalization of a soundness proof for the system defined in Figure 4. That is, we need to prove that a derivation in the axiomatic equivalence system implies *observational equivalence*. The development follows that of Harper [5], but adapted slightly to be representable in Twelf. We will first define what we *mean* when we say that two expressions are observationally equivalent.

3.1 Observational equivalence

Following Harper, we introduce the notion of *expression contexts*, which are expressions with a single "hole", $\circ$, standing for another expression. We write $\mathcal{C}$ when referring to expression contexts, and $\mathcal{C}\{e\}$ for the expression resulting from replacing e for $\circ$ in $\mathcal{C}$. Replacement is different from substitution, in that expressions replaced for holes may *capture* any variables in scope at hole position (e.g., $(\mathsf{lam}\ x.\circ)\{\mathsf{app}\ x\ y\} \equiv \mathsf{lam}\ x.\mathsf{app}\ x\ y$). We will write $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\Gamma' \rhd \tau')$ when $\mathcal{C}$ is an expression context and for any e where $\Gamma \vdash e : \tau$, we have $\Gamma' \vdash C\{e\} : \tau'$. We will not give the definition of context typing here, but it can be straightforwardly derived from the typing rules in Figure 1. We say that an expression context $\mathcal{C}$ is a *program context* iff $\mathcal{C} : (\Gamma \rhd \tau) \rightsquigarrow (\cdot \rhd \mathsf{nat})$, i.e., the result of substituting an expression for the hole in $\mathcal{C}$ results in a closed expression with type nat.

Definition 4 (Kleene equivalence)**.** We say that e and e' are *Kleene equivalent*, written $e \simeq e'$, iff $\forall n.\, e \Downarrow \mathsf{num}\ n \iff e' \Downarrow \mathsf{num}\ n$.

Expression equivalence: $\mathcal{Q} :: \boxed{\Gamma \vdash e \cong e' : \tau} :$

$$\frac{\Gamma \vdash e \cong e' : \tau}{\Gamma \vdash e' \cong e : \tau}\ \mathsf{q\_sym} \qquad \frac{\Gamma \vdash e \cong e' : \tau \quad \Gamma \vdash e' \cong e'' : \tau}{\Gamma \vdash e \cong e'' : \tau}\ \mathsf{q\_trans} \qquad \frac{}{\Gamma, x : \tau \vdash x \cong x : \tau}\ \mathsf{q\_var} \qquad \frac{}{\Gamma \vdash \mathsf{num}\ n \cong \mathsf{num}\ n : \mathsf{nat}}\ \mathsf{q\_num}$$

$$\frac{}{\Gamma \vdash \mathsf{fail} \cong \mathsf{fail} : \tau}\ \mathsf{q\_fail} \qquad \frac{\Gamma, x : \tau_2 \vdash e_0 \cong e_0' : \tau_0}{\Gamma \vdash \mathsf{lam}\ x.\, e_0 \cong \mathsf{lam}\ x.\, e_0' : \tau_2 \rightharpoonup \tau_0}\ \mathsf{q\_lam} \qquad \frac{\Gamma \vdash e_1 \cong e_1' : \tau_2 \rightharpoonup \tau_0 \quad \Gamma \vdash e_2 \cong e_2' : \tau_2}{\Gamma \vdash \mathsf{app}\ e_1\ e_2 \cong \mathsf{app}\ e_1'\ e_2' : \tau_0}\ \mathsf{q\_app}$$

$$\frac{\Gamma \vdash e_0 \cong e_0' : \mathsf{nat} \quad \Gamma \vdash e_1 \cong e_1' : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_2 \cong e_2' : \tau}{\Gamma \vdash \mathsf{case}(e_0,\ e_1,\ x.\, e_2) \cong \mathsf{case}(e_0',\ e_1',\ x.\, e_2') : \tau}\ \mathsf{q\_ifz}$$

$$\frac{\Gamma \vdash e_1 \cong e_1 : \tau}{\Gamma \vdash \mathsf{case}(\mathsf{num}\ \mathsf{z},\ e_1,\ x.\, e_2) \cong e_1 : \tau}\ \mathsf{q\_ifz0} \qquad \frac{\Gamma, x : \mathsf{nat} \vdash e_2 \cong e_2 : \tau}{\Gamma \vdash \mathsf{case}(\mathsf{num}\ (\mathsf{s}\ n),\ e_1,\ x.\, e_2) \cong e_2[\mathsf{num}\ n/x] : \tau}\ \mathsf{q\_ifz1}$$

$$\frac{\Gamma, x : \tau_2 \vdash e_0 \cong e_0 : \tau_0 \quad \Gamma \vdash e_2 \cong e_2 : \tau_2}{\Gamma \vdash \mathsf{app}\ (\mathsf{lam}\ x.\, e_0)\ e_2 \cong e_0[e_2/x] : \tau_0}\ \mathsf{q\_beta} \qquad \frac{\Gamma \vdash e \cong e : \tau_2 \rightharpoonup \tau_0}{\Gamma \vdash e \cong \mathsf{lam}\ x.\, \mathsf{app}\ e\ x : \tau_2 \rightharpoonup \tau_0}\ \mathsf{q\_eta}$$

Figure 4. Equational reasoning.

Definition 5 (Observational equivalence)**.** We say that two (possibly open) expressions e and e' with $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ are *observationally equivalent* in Γ iff for any $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\cdot \triangleright \mathsf{nat})$, we have $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$.

That is, two expressions are observationally equivalent when any conceivable (well-typed) program context cannot observe their difference, i.e., yields identical results. This definition of observational equivalence is rather hard to work with in proofs, though, as we have to take all conceivable contexts into consideration when proving that two expressions are equivalent. However, it turns out that any relation satisfying the following properties implies observational equivalence:

Definition 6 (Congruence)**.** A family of type-indexed relations on open expressions $\Gamma \vdash e\ \mathcal{R}\ e' : \tau$ is said to be *congruent* iff $\Gamma \vdash e\ \mathcal{R}\ e' : \tau$ implies $\Gamma \vdash \mathcal{C}\{e\}\ \mathcal{R}\ \mathcal{C}\{e'\} : \tau'$ for any $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\Gamma' \triangleright \tau')$.

Definition 7 (Consistency)**.** A type-indexed family of relations on expressions $\Gamma \vdash e\ \mathcal{R}\ e' : \tau$ is said to be *consistent* iff $\cdot \vdash e\ \mathcal{R}\ e' : \mathsf{nat}$ implies $\forall n.\, e \Downarrow \mathsf{num}\ n \iff e' \Downarrow \mathsf{num}\ n$.

It can easily be verified that if *any* family of relations $\mathcal{R}$ satisfies the above, then $\Gamma \vdash e\ \mathcal{R}\ e' : \tau$ implies that e and e' are observationally equivalent: Just observe that since $\mathcal{R}$ is a congruence, we have $\cdot \vdash \mathcal{C}\{e\}\ \mathcal{R}\ \mathcal{C}\{e'\} : \mathsf{nat}$ for any program context $\mathcal{C} : (\Gamma \triangleright \tau) \rightsquigarrow (\cdot \triangleright \mathsf{nat})$, and by consistency of $\mathcal{R}$, we hence have $\mathcal{C}\{e\} \simeq \mathcal{C}\{e'\}$.

The system in Figure 4 is a congruent equivalence relation *by definition*. To prove soundness, it therefore suffices to prove that it is also consistent, which by the presence of symmetry (q_sym) reduces to the following:

Theorem 8 (Soundness)**.** *For any e, e', if $\cdot \vdash e \cong e' : \mathsf{nat}$, then for any n, if $e \Downarrow \mathsf{num}\ n$, then also $e' \Downarrow \mathsf{num}\ n$.*

```
sim-sound : sim E E' tnat -> eval E (num N)
            -> eval E' (num N) -> type.
%mode sim-sound +SIP +EP -EP'.
```

We cannot prove this directly by induction on derivations, but must use a logical-relations based argument. In the following we will define a binary logical relation which implies Kleene equivalence at base types. We then prove the above theorem by showing that a derivation of $\Gamma \vdash e \cong e' : \tau$ implies that e and e' are logically related at τ, assuming that variables from Γ are related to themselves.

3.2 Logical equivalence

We define *logical equivalence* and its representation in Twelf as follows:

Definition 9 (Logical equivalence)**.** We say that two closed expressions e and e' are *logically equivalent* at type τ iff $e \sim_\tau e'$, where

$$\begin{aligned} e \sim_{\mathsf{nat}} e' &\iff e \simeq e', \\ e \sim_{\tau_2 \rightharpoonup \tau_0} e' &\iff \forall e_2.\, \forall e_2'.\\ &\qquad e_2 \sim_{\tau_2} e_2' \Rightarrow \mathsf{app}\ e\ e_2 \sim_{\tau_0} \mathsf{app}\ e'\ e_2'. \end{aligned}$$

```
% Kleene equivalence
keq : exp -> exp -> form
  = [e][e'] forall_n [n] #eval e (num n)
                          <==> #eval e' (num n).

lr : tp -> (exp -> exp -> form) -> type.
lr/tnat : lr tnat keq.
lr/tarrow :
  lr T2 R2 -> lr T0 R0
  -> lr (tarrow T2 T0)
        ([e][e']
        forall_e [e2] forall_e [e2']
          R2 e2 e2'
            ==> R0 (app e e2) (app e' e2')).
```

In the following, we will present the lemmas stating that all the properties of axiomatic equivalence also hold for logical equivalence, as well as their formulations in Twelf. For brevity, we will write `conc*` as an abbreviation for `conc cutful`.

Logical equivalence is a partial equivalence relation:

Lemma 10. *Logical equivalence is symmetric and transitive.*

```
lr-sym : lr T R -> conc* (R E E')
             -> conc* (R E' E) -> type.
%mode lr-sym +LP +PF -PF'
lr-trans : lr T R -> conc* (R E E')
                -> conc* (R E' E'')
                -> conc* (R E E'') -> type.
%mode lr-trans +LP +PF1 +Pf2 -PF3.
```

Proof. By meta-level induction on τ. □

Open expressions are logically equivalent if substituting related closed expressions for free variables yields related results. To give a formal definition of this, we introduce the notion of *binary* typing contexts and closing substitutions:

Definition 11 (Binary typing context)**.** A *binary typing context* is a mapping $\Psi = (x_1, x_1') : \tau_1, \ldots, (x_n, x_n') : \tau_n$ from $2n$ variables to n types.

Definition 12 (Closing substitution)**.** Given a binary typing context $\Psi = (x_1, x_1') : \tau_1, \ldots, (x_n, x_n') : \tau_n$, a *closing substitution* for Ψ is a finite function $\psi = [x_1 \mapsto e_1, x_1' \mapsto e_1', \ldots, x_n \mapsto e_n, x_n' \mapsto e_n']$ assigning *closed* expressions for variables in Ψ.

For any such ψ, we write $\hat{\psi}(e)$ for the substitution $e[e_1/x_1, e_1'/x_1', \ldots, e_n/x_n, e_n'/x_n']$.

Definition 13 (Open logical equivalence)**.** Suppose e and e' are (open) expressions and $\Psi = (x_1, x_1') : \tau_1, \ldots, (x_n, x_n') : \tau_n$ is a binary typing context. *Open logical equivalence*, written $\Psi \vdash e \sim e' : \tau$, is defined to mean that for any closing substitution ψ for Ψ, if $\psi(x_i) \sim_{\tau_i} \psi(x_i')$ for every i, then $\hat{\psi}(e) \sim_\tau \hat{\psi}(e')$ and $\mathrm{FV}(e) \subseteq \{x_1, \ldots, x_n\}$ and $\mathrm{FV}(e') \subseteq \{x_1', \ldots, x_n'\}$.

Note that the definition of open logical equivalence does not map straightforwardly to our definition of axiomatic equivalence in Figure 4, as that one uses a unary typing context. We could also have chosen to define open logical equivalence in the style of Harper [5], which uses unary typing contexts and two closing substitutions instead of one, restricted to having equal domains and pointwise related codomains. Harper's formulation is a bit simpler for paper proofs, but unfortunately, does not have a compositional representation in LF. This has some implications on the proof of Theorem 8, which we will return to in Section 3.3.

We can represent open logical equivalence in Twelf using the LF context, by defining the encoding

$$\begin{aligned}&\ulcorner \Psi, (x_i, x_i') : \tau_i \urcorner \\ &\quad = \ulcorner \Psi \urcorner, \mathtt{x}_i : \mathtt{exp}, \mathtt{x}_i' : \mathtt{exp}, \mathtt{u}_i : \mathtt{conc*}\ (\ulcorner \sim_{\tau_i} \urcorner\ \mathtt{x}_i\ \mathtt{x}_i'),\end{aligned}$$

where $\ulcorner \sim_\tau \urcorner$ denotes the formula representation of the logical relation at type τ.

Symmetry and transitivity (Lemma 10) transfers directly to open logical equivalence. It remains to prove that open logical equivalence is a congruence, and that it respects β-reduction and η-expansion. To do that, we need some auxiliary results and a notion of *weak equivalence*; a slight generalization of Kleene equivalence:

Definition 14 (Weak equivalence)**.** We write $e \approx e'$ iff $\forall v.\, e \Downarrow v \iff e' \Downarrow v$.

```
weq : exp -> exp -> form
  = [e][e'] forall_e [v] #eval e v <==> #eval e' v.
```

Lemma 15 (Closure under weak equivalence)**.** *If $e \approx e'$ and $e' \sim_\tau e''$, then $e \sim_\tau e''$.*

```
cweq : lr T R -> conc* (weq E E')
  -> conc* (R E' E'') -> conc* (R E E'') -> type.
%mode cweq +LP +PF1 +PF2 -PF3.
```

Proof sketch. By Lemma 10, it suffices to show that if $e \approx e'$ and $e' \sim_\tau e'$, then $e \sim_\tau e'$. We proceed by meta-level induction on τ. In the base case, the result follows directly. In the case for function types, we appeal to the induction hypothesis, which we justify by proving $\mathsf{app}\ e\ e_2 \approx \mathsf{app}\ e'\ e_2$ for some e_2: This result follows by assumption and *case analysis on possible derivations*. For example, in the forward direction, we need to prove that if $\mathsf{app}\ e\ e_2 \Downarrow v$ (for some v), then also $\mathsf{app}\ e'\ e_2 \Downarrow v$. Since the given evaluation can only end in e_app, we obtain a derivation of $e \Downarrow \mathsf{lam}\ x.\, e_0$ for some $x.\, e_0$, which by $e \approx e'$ implies $e' \Downarrow \mathsf{lam}\ x.\, e_0$, so by e_app, we are done. □

The Twelf proof of the above lemma depends crucially on our ability to reason by case analysis and equality in the assertion logic. Case analysis allows us to proceed by cases on evaluation derivations, while the equality principles of the representation logic allows us to derive falsehood in all cases but the one for e_app.

Lemma 16 (Application commutes over case)**.** *For any e_0,e_1,$x.\, e_2$ and e', where $x \notin \mathrm{FV}(e')$, we have*

$$\mathsf{app}\ \mathsf{case}(e_0,\ e_1,\ x.\, e_2)\ e' \approx \mathsf{case}(e_0,\ \mathsf{app}\ e_1\ e',\ x.\, \mathsf{app}\ e_2\ e').$$

```
weq-app :
  pf (weq (app (case E0 E1 E2) E')
          (case E0 (app E1 E') ([x] app (E2 x) E')))
  = %{ ... }%
```

Proof sketch. By case analysis on derivations. □

We need to prove that logical equivalence is congruent with respect to all six syntactic constructors of the object language, namely app, lam, case, num, fail and variables. We will prove this in the following, but to conserve space, we only show the Twelf representation of a representative subset of the lemmas.

Lemma 17 (Congruence at application)**.** *Suppose that $\Psi \vdash e_1 \sim e_1' : \tau_2 \rightharpoonup \tau_0$ and $\Psi \vdash e_2 \sim e_2' : \tau_2$. Then, we also have $\Psi \vdash \mathsf{app}\ e_1\ e_2 \sim_{\tau_0} \mathsf{app}\ e_1'\ e_2'$.*

```
lr-app : lr T2 R2 -> lr T0 R0
     -> conc* (forall_e [e2] forall_e [e2']
                R2 e2 e2'
                  ==> R0 (app E1 e2) (app E1' e2'))
     -> conc* (R2 E2 E2')
     -> conc* (R0 (app E1 E2) (app E1' E2'))
     -> type.
%mode lr-app +LP2 +LP0 +PF1 +PF2 -PF3.
```

Proof. Direct, by definition of logical equivalence at function type. □

Lemma 18 (Congruence at abstraction)**.** *Suppose that $\Psi, (x, x') : \tau_2 \vdash e \sim e' : \tau_0$.*

Then also $\Psi \vdash \mathsf{lam}\ x.\, e \sim \mathsf{lam}\ x'.\, e' : \tau_2 \rightharpoonup \tau_0$.

Proof sketch. By Lemma 15 on the assumption, justified by proving that $\mathsf{app}\ (\mathsf{lam}\ x.\, e)\ e_2 \approx e[e_2/x]$ for any e, e_2 by case analysis on possible derivations, followed by applications of Lemma 10 where appropriate. □

Lemma 19 (Congruence at case)**.** *Suppose $\Psi \vdash e_0 \sim e_0' : \mathsf{nat}$, and $\Psi \vdash e_1 \sim e_1' : \tau$, and $\Psi, (x, x') : \mathsf{nat} \vdash e_2 \sim e_2' : \tau$. Then also $\Psi \vdash \mathsf{case}(e_0,\ e_1,\ x.\, e_2) \sim \mathsf{case}(e_0',\ e_1',\ x'.\, e_2') : \tau$.*

```
lr-case : lr T R -> conc* (keq E0 E0')
  -> conc* (R E1 E1')
  -> ({x}{x'} conc* (keq x x')
       -> conc* (R (E2 x) (E2' x')))
  -> conc* (R (case E0 E1 E2) (case E0' E1' E2'))
  -> type.
%mode lr-case +LP +PF0 +PF1 +PF2 -PF'.
```

Proof sketch. By induction on τ. For $\tau = \mathsf{nat}$, it suffices to prove that Kleene equivalence is congruent with respect to case, which follows by case analysis on possible derivations. For $\tau = \tau_2 \rightharpoonup \tau_0$, we proceed by induction, and using Lemma 15 and Lemma 16. □

Lemma 20 (Reflexivity at numerals)**.** *For any n, we have* $\Psi \vdash \mathsf{num}\ n \sim \mathsf{num}\ n : \mathsf{nat}$.

```
lr-num : pf (keq (num N) (num N)) = %{ ... }%
```

Proof. Directly by definition. □

The proof for congruence at fail needs a slightly more general induction hypothesis to go through, requiring the introduction of a new form of context:

Definition 21 (Applicative context)**.** An *applicative context* is an expression with a single hole, generated by the following grammar:

$$\text{Applicative contexts:} \quad \mathcal{A} \quad ::= \quad \circ \mid \mathsf{app}\ \mathcal{A}\ e_2.$$

```
ctx : (exp -> exp) -> type.
ctx/id : ctx ([x] x).
ctx/app : ctx E1 -> ctx ([x] app (E1 x) E2).
```

Lemma 22 (Strictness of applicative contexts)**.** *For any applicative context $\mathcal{A}$, we have $\mathcal{A}\{\mathsf{fail}\} \not\Downarrow$, i.e., an expression with* fail *in reduction position does not evaluate.*

```
eval-strict : ctx A
    -> conc* (forall_e [v]
              #eval (A fail) v
                 ==> existsd [dp:data dvoid] top)
    -> type.
%mode eval-strict +CP -PF.
```

Proof sketch. By meta-level induction on $\mathcal{A}$. □

Lemma 23 (Strictness of logical equivalence)**.** *For any applicative contexts $\mathcal{A}$, $\mathcal{A}'$, and for any type τ, we have* $\Psi \vdash \mathcal{A}\{\mathsf{fail}\} \sim \mathcal{A}'\{\mathsf{fail}\} : \tau$.

```
lr-strict : ctx A -> ctx A' -> lr T R
  -> conc* (R (A fail) (A' fail)) -> type.
%mode lr-strict +CP +CP' +LP -PF.
```

Proof sketch. By meta-level induction on τ, appealing to Lemma 22 in the case for $\tau = \mathsf{nat}$. □

Congruence at fail follows as a special case of the above.

The case for congruence at variables needs no proof, due to the use of higher-order abstract syntax to represent open logical equivalence.

We have now proved that logical equivalence is a congruent equivalence relation. It remains to show that logical equivalence also supports $\beta\eta$-conversion and reduction of case constructs:

Lemma 24 (β-conversion)**.** *If $\Psi \vdash \mathsf{lam}\ x.\, e_0 \sim \mathsf{lam}\ x'.\, e_0' : \tau_2 \rightharpoonup \tau_0$ and $\Psi \vdash e_2 \sim e_2' : \tau_2$, then we also have* $\Psi \vdash \mathsf{app}\ (\mathsf{lam}\ x.\, e_0)\ e_2 \sim e_0'[e_2'/x'] : \tau_0$.

```
lr-beta : lr T0 R0 -> lr T2 R2
  -> (conc* (forall_e [e2] forall_e [e2']
            R2 e2 e2' ==> R0 (app (lam E0) e2)
                             (app (lam E0') e2')))
  -> conc* (R2 E2 E2')
  -> conc* (R0 (app (lam E0) E2) (E0' E2'))
  -> type.
%mode lr-beta +LP0 +LP2 +PF1 +PF2 -PF'.
```

Proof sketch. Follows by Lemma 17, and Lemma 15 on a proof of $\mathsf{app}\ (\mathsf{lam}\ x.\, e_0)\ e_2 \approx e_0[e_2/x]$, which follows by case analysis on possible derivations. □

Lemma 25 (η-conversion)**.** *If $\Psi \vdash e \sim e' : \tau_2 \rightharpoonup \tau_0$, then also $\Psi \vdash e \sim \mathsf{lam}\ x.\, (\mathsf{app}\ e'\ x) : \tau_2 \rightharpoonup \tau_0$.*

Proof sketch. Follows by Lemma 18 and Lemma 15. □

Lemma 26 (Case reduction at zero numeral)**.** *If $\Psi \vdash e_1 \sim e_1' : \tau$, then also $\Psi \vdash \mathsf{case}(\mathsf{num}\ \mathsf{z},\, e_1,\, x.\, e_2) \sim e_1 : \tau$.*

Lemma 27 (Case reduction at non-zero numeral)**.** *If $\Psi, (x, x') : \mathsf{nat} \vdash e_2 \sim e_2' : \tau$, then also*

$$\Psi \vdash \mathsf{case}(\mathsf{num}\ (\mathsf{s}\ n'),\, e_1,\, x.\, e_2) \sim e_2'[\mathsf{num}\ n'/x'] : \tau.$$

This concludes the proofs of properties required to show that axiomatic equivalence implies logical equivalence. In the following section, we will return to the soundness proof (Theorem 8).

3.3 Context separation

We will prove Theorem 8 by showing that $\cdot \vdash e \cong e' : \mathsf{nat}$ implies $\cdot \vdash e \sim e' : \mathsf{nat}$, which by definition implies Kleene equivalence and hence consistency of axiomatic equivalence. However, the proof needs to work in non-empty contexts as well in order to go through.

One attempt would be to prove that if

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \cong e' : \tau,$$

then also

$$(x_1^L, x_1^R) : \tau_1, \ldots, (x_n^L, x_n^R) : \tau_n \vdash e[x_1^L/x_1, \ldots, x_n^L/x_n] \sim e'[x_1^R/x_1, \ldots, x_n^R/x_n] : \tau,$$

for some $x_1^L, x_1^R, \ldots, x_n^L, x_n^R$. That is, for every x_i, we introduce two fresh, distinct variables x_i^L, x_i^R, effectively "separating" the context. This approach is possible, but results in a surprisingly complex Twelf proof, due to the extra overhead of keeping track of "left" and "right" versions of each variable assumption.

An easier approach is to divide the soundness proof into two steps. In the first step, we show that we can translate derivations in the original axiomatic equivalence system into an alternative version of the system that uses binary contexts. We then prove consistency of this system instead.

A representative subset of the definition of the alternative reasoning system can be seen in Figure 5. The only differences is in the rules for variables and in the rules that extend the context. Note that when extending the context, variables are renamed such that the context is only extended with *distinct* variables.

The Twelf representation of the alternative system looks as follows:

```
sim* : exp -> exp -> tp -> type.
sim*/lam : ({l}{r} sim l r T2
           -> sim (E0 l) (E0' r) T0)
        -> sim (lam E0) (lam E0') (tarrow T2 T0).
%{ ... remaining rules elided ... }%
```

Conversion is shown by first proving that a single assumption with identical variables can be rewritten such that the variables are distinct:

Lemma 28 (Doubling)**.** *If $\Psi, (x, x) : \tau' \vdash^\star e \cong e' : \tau$, then also*

$$\Psi, (x^L, x^R) : \tau' \vdash^\star e[x^L/x] \cong e'[x^R/x] : \tau,$$

for some $x^L \neq x^R$.

Expression equivalence, alt.: $\mathcal{Q}$ :: $\boxed{\Psi \vdash^\star e \cong e' : \tau}$:

$$\frac{\Psi \vdash^\star e \cong e' : \tau}{\Psi \vdash^\star e' \cong e : \tau}\ \mathsf{q\_sym}^\star$$

$$\frac{\Psi \vdash^\star e \cong e' : \tau \quad \Psi \vdash^\star e' \cong e'' : \tau}{\Psi \vdash^\star e \cong e'' : \tau}\ \mathsf{q\_trans}^\star$$

$$\frac{}{\Psi, (x, x') : \tau \vdash^\star x \cong x' : \tau}\ \mathsf{q\_var}^\star$$

$$\frac{\Psi, (x^L, x^R) : \tau_2 \vdash^\star e_0[x^L/x] \cong e_0'[x^R/x'] : \tau_0}{\Psi \vdash^\star \mathsf{lam}\ x.\, e_0 \cong \mathsf{lam}\ x'.\, e_0' : \tau_2 \rightharpoonup \tau_0}\ {}^{(x^L \neq x^R)}\ \mathsf{q\_lam}^\star$$

$$\vdots$$

Figure 5. Alternative expression equivalence, representative subset.

```
sim*-double :
  ({x} sim* x x T -> sim* (E x) (E' x) T')
  -> ({l}{r} sim* l r T -> sim* (E l) (E' r) T')
  -> type.
%mode sim*-double +SIP -SIP'.
```

Proof sketch. By induction on derivations. The cases for $\mathsf{q\_trans}^\star$ and $\mathsf{q\_sym}^\star$ involves an extra inner induction proof to convert uses of the rewritten assumptions. For example, in the case for $\mathsf{q\_sym}^\star$, we get by the induction hypothesis that $\Psi, (x^L, x^R) : \tau' \vdash^\star e'[x^L/x] \cong e[x^R/x] : \tau$, but we need $\Psi, (x^L, x^R) : \tau' \vdash^\star e'[x^R/x] \cong e[x^L/x] : \tau$ in order to be able to reapply $\mathsf{q\_sym}^\star$ and obtain the desired goal. By inner induction, we apply $\mathsf{q\_sym}^\star$ to all uses of the assumption $(x^L, x^R) : \tau'$, and we are done. □

The conversion lemma proceeds by a straightforward bottom-up conversion of derivations:

Lemma 29 (Conversion)**.** *Suppose* $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ *and* $\Psi = (x_1, x_1) : \tau_1, \ldots, (x_n, x_n) : \tau_n$*. If* $\Gamma \vdash e \cong e' : \tau$*, then also* $\Psi \vdash^\star e \cong e' : \tau$*.*

```
sim=>sim* : sim E E' T -> sim* E E' T -> type.
%mode sim=>sim* +SIP -SIP'.
```

Proof sketch. By straightforward induction on derivations. In all cases where the context is extended, e.g. $\mathsf{q\_lam}$, we appeal to the induction hypothesis followed by Lemma 28, which ensures that we can apply the alternative rule (e.g., $\mathsf{q\_lam}^\star$) to the result. □

Note that the conversion lemma does not rewrite existing contexts, only derivations. Since we are only interested in proving consistency, which involves an empty context, this is not a problem.

Soundness of the alternative system follows by straightforward induction on derivations:

Lemma 30 (Soundness, alternative)**.** *Suppose* $\Psi = (x_1, x_1') : \tau_1, \ldots, (x_n, x_n') : \tau_n$*, where* $x_i \neq x_i'$ *for every* i*. If* $\Psi \vdash^\star e \cong e' : \tau$*, then also* $\Psi \vdash e \sim e' : \tau$*.*

```
sim*-sound : sim* E E' T -> lr T R
          -> conc* (R E E') -> type.
%mode sim*-sound +SIP -LP -SP.
```

Proof sketch. By straightforward induction over derivations. Cases $\mathsf{q\_sym}$, $\mathsf{q\_trans}$ are covered by Lemma 10; $\mathsf{q\_app}$ by Lemma 17; $\mathsf{q\_lam}$ by Lemma 18; $\mathsf{q\_case}$ by Lemma 19; $\mathsf{q\_num}$ by Lemma 20; $\mathsf{q\_fail}$ by Lemma 23; $\mathsf{q\_case0}$ by Lemma 26; $\mathsf{q\_case1}$ by Lemma 27; $\mathsf{q\_beta}$ by Lemma 24; and $\mathsf{q\_eta}$ by Lemma 25. The case for $\mathsf{q\_var}$ follows directly. □

The soundness theorem now follows as a corollary. Recall that we needed to prove that for any e,e', if $\cdot \vdash e \cong e' : \mathsf{nat}$, then for any n, if $e \Downarrow \mathsf{num}\ n$, then $e' \Downarrow \mathsf{num}\ n'$.

Proof of Theorem 8.

1. By Lemma 29, we obtain $\cdot \vdash^\star e \cong e' : \mathsf{nat}$.
2. By Lemma 30, we obtain $\cdot \vdash e \sim e' : \mathsf{nat}$, or, by definition, $\cdot|\cdot \vdash^\bullet e \simeq e'$.
3. By Lemma 1 and assumption, we obtain $\Vdash (\!| e \Downarrow \mathsf{num}\ n |\!)$.
4. By $\mathsf{allL\_n}$, impL, $\mathsf{exiR\_d}$, $\mathsf{andL1}$ and ax, we obtain a cutful derivation of $\cdot|\cdot \vdash^\bullet \exists \alpha : (\Vdash (\!| e' \Downarrow \mathsf{num}\ n |\!))$.
5. By Lemma 3, we obtain $\cdot|\cdot \vdash^\circ \exists \alpha : (\Vdash (\!| e' \Downarrow \mathsf{num}\ n |\!))$.
6. As the above derivation can only end in $\mathsf{exiR\_d}$, we obtain $\vdash^{\mathsf{LF}}_{\Sigma} \mathtt{M} : \ulcorner \Vdash (\!| e' \Downarrow \mathsf{num}\ n |\!) \urcorner$ for some LF term $\mathtt{M}$. By adequacy of encoding, there exists a derivation of $\Vdash (\!| e' \Downarrow \mathsf{num}\ n |\!)$.
7. By Lemma 1, we obtain $e' \Downarrow \mathsf{num}\ n$, and we are done. □

```
sim-sound : sim E E' tnat -> eval E (num N)
         -> eval E' (num N) -> type.
%mode sim-sound +SIP +EP -EP'.
eval-sound : conc cutfree (#eval E V)
          -> eval E V -> type.
%mode eval-sound +PF -EP.

- : sim-sound (SIP : sim E E' tnat) EP EP'
    <- sim=>sim* SIP SIP'
    <- sim*-sound SIP' lr/tnat SP
    <- emb EP DP SP+
    <- ce (cut SP
           (foralll_n N
             (andl_1 (impl (existsdr DP SP+) ax))))
          SP'
    <- eval-sound SP' EP'.

% Separate lemma necessary due to coverage check
- : eval-sound (existsdr DP _) EP
   <- unemb DP EP.
```

4. Conclusion and future work

We have presented an extension of the method of structural logical relations that admits case analysis and equality reasoning to be used in proofs, while still allowing the consistency of the assertion logic to be proved as a meta-theorem. We have demonstrated that the technique allows the formalization of proofs of meta-theory for programming languages, specifically proofs of observational equivalence between expressions in a call-by-name language. The method has also been used to prove observational equivalence for a call-by-value language with non-determinism [11]. Although the meta-theory for such a system is more complex, the methodology required to formalize the results in Twelf remains the same.

The Twelf formalizations presented here involve quite a bit of boilerplate code, much of which is very mechanical. For example, whenever the evaluation judgment is extended with new rules, we have to add corresponding rules to the representation-logic formalization as well, which again

requires the assertion logic and the cut-admissibility proof to be extended. This means that adding a single new syntactic construct to the object language may require a substantial amount of boilerplate to be written as well. A topic for future work would be to develop a streamlined way of generating assertion logics and representation logics from an LF signature, possibly by utilizing the experimental Twelf module system [9].

While adding case-analysis principles to the assertion logic is evidently feasible, Twelf is not proof-theoretically strong enough to prove normalization of a logic with a general induction principle [12]. If we want to avoid having to trust the consistency of the assertion logic, we therefore have to do without an induction principle. Alternatively, we can still write a normalization procedure for such a logic, and tell Twelf that we believe that it terminates. Having to believe in the termination of a normalization proof for a relatively standard first-order logic may in many cases be an acceptable trade-off for a more expressive assertion logic.

While we have succesfully formalized the meta-theory of an object language with more complex features than the one used in the original presentation by Schürmann and Sarnat [13], there are still certain aspects that are not present in our formalization. Importantly, the original presentation of structural logical relations dealt with a language which allowed reductions under abstractions, whereas our evaluation judgment is only defined for closed expressions. This means that our representation-logic encoding of the object-language semantics never has to deal with (object-level) variables. It is not immediately clear, and a topic for future work, whether the methodology presented here can be extended to work for such systems as well.

It has previously been demonstrated by Abel [1] that a normalization proof for a simply typed lambda calculus can be formalized in Twelf without formulating an assertion logic, yielding a simpler proof. His proof does, however, still require one to prove properties akin to cut-elimination, albeit for a specialized syntactic relation instead of a general assertion logic. It would be interesting to investigate whether some of the ideas presented here, specifically the addition of case analysis and equality reasoning, can be transferred to his methodology.

4.1 Related work

The Twelf module system has recently been extended with a notion of logical relations [9, 10]. The module system allows the definition of isolated signatures and *morphisms*, which are transformations of types and terms in one signature to types and terms in another. A theory of logical relations is then built on top of the module system, allowing one to express logical relations as an n-ary relation between morphisms, i.e., relations between interpretations of expressions.

This gives a compact way of representing proofs of the fundamental theorem, e.g., we could use it to prove that all well-typed expressions of a terminating language are related to an assertion-logic proof proving that they evaluate. However, we would still have to trust the consistency of the assertion logic. The module system is not integrated with the Twelf meta-logic at all, so a logical-relations argument formulated in this way cannot be given an operational interpretation as a Twelf meta-theorem.

Acknowledgments

We would like to thank the anonymous reviewers for comprehensive and helpful comments and recommendations for improvement. We would also like to thank Carsten Schürmann for helpful discussions on the topic.

References

[1] Andreas Abel. Normalization for the simply-typed lambda-calculus in Twelf. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, volume 199 of Electronic Notes in Theoretical Computer Science, February 2008.

[2] Karl Crary. Logical relations and a case study in equivalence checking. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages,* pages 223–243, 2005.

[3] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for definining logics. *Journal of the ACM,* 40(1):143–184, 1993.

[4] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

[5] Robert Harper. *Practical Foundations for Programming languages,* pages 413–432, ISBN 978-1-107-02957-6, Cambridge University Press, 2013.

[6] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* POPL'07, pages 173–184, New York, NY, USA, 2007.

[7] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16),* pages 202–206, Trento, Italy, June 1999.

[8] Frank Pfenning. Structural cut elimination: I. Intuitionistic and classical logic. *Information and Computation,* 157(1-2):84–141, 2000.

[9] Florian Rabe and Carsten Schürmann. A practical module system for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP),* volume LFMTP'09 of ACM International Conference Proceeding Series, pages 40–48. ACM Press, 2009.

[10] Florian Rabe and Kristina Sojakova. Logical relations for a logical framework. *ACM Transactions on Computational Logic,* 2013. to appear; see `http://kwarc.info/frabe/Research/RS_logrels_12.pdf`.

[11] Ulrik Rasmussen. *Formalization of proofs by logical relations in a logical framework.* M.Sc. Thesis, Department of Computer Science, University of Copenhagen, Denmark; can be obtained electronically at `http://utr.dk/slr/`, 2013.

[12] Jeffrey Sarnat. *Syntactic Finitism in the Metatheory of Programming Languages.* PhD thesis, Yale University, May 2010.

[13] Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science,* LICS'08, pages 69–80, 2008.

[14] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic,* 32(2):198–212, 1967.