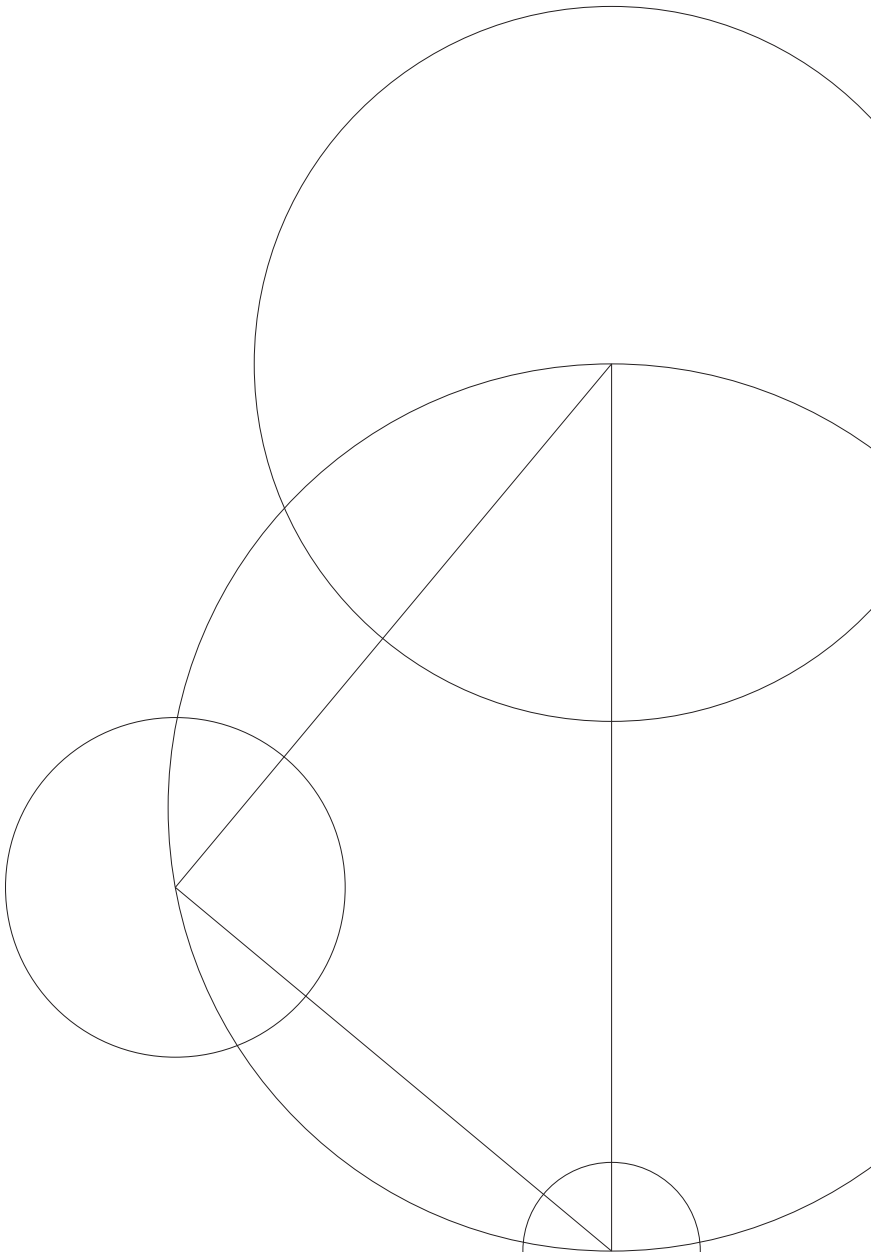
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Ulrik Rasmussen

Formalization of proofs by logical relations in a logical framework



Supervisor: Andrzej Filinski

June 6, 2013

Abstract

Logical relations are an important proof technique frequently employed in the study of programming languages based on typed lambda calculi, where they have been used to prove a broad range of foundational properties.

We present applications and extensions of the method of *structural logical relations* by Schürmann and Sarnat [*In Proc. of 23rd Symp. on Logic in Computer Science*, 2008], which enables syntactic and verifiable representations of proofs by logical relations in the Twelf proof assistant by reducing problems to a consistency proof of an auxiliary logic.

We apply the method in several case studies where we formalize logical relations for prototypical programming languages. We identify shortcomings of the existing method, which we address by extending the auxiliary logic with several new reasoning principles.

Resumé

Logiske relationer er en vigtig bevisteknik, ofte anvendt indenfor studiet af programmeringssprog baseret på typet lambdakalkule, hvor de har fundet anvendelse i adskillige beviser af fundamentale egenskaber.

Vi præsenterer anvendelser og udvidelser af metoden *structural logical relations* (strukturelle logiske relationer) af Schürmann og Sarnat [*In Proc. of 23rd Symp. on Logic in Computer Science*, 2008], som muliggør syntaktiske og verificerbare beviser ved hjælp af logiske relationer i bevisassistenten Twelf, ved at reducere problemer til et konsistensbevis af en hjælpelogik.

Vi anvender metoden i flere casestudier, hvor vi formaliserer logiske relationer for prototypiske programmeringssprog. Vi identificerer mangler ved den eksisterende metode, som afstedkommes ved at udvide hjælpelogikken med nye deduktionsprincipper.

Preface

This document constitutes the author's Master's thesis, submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science (*Datalogi*) at the University of Copenhagen.

I would like to thank my supervisor, Andrzej Filinski, for his dedication and many hours of advising. I would also like to thank Carsten Schürmann for answering some of the questions I had on Twelf and structural logical relations.

A electronic appendix with all Twelf formalizations is enclosed with this thesis [Ras13].

Contents

1 Introduction

How can we be sure that a lengthy and complex mathematical proof is actually true? Researchers in the field of programming languages often work with very large systems whose correctness is of critical importance, and would like to be able to make sure that programs never exhibit unexpected behaviour because of a mistake in the programming language design. Since programming languages are often large and complex systems, verifying proofs about them by hand can be a labor intensive and error-prone task. Worse, the specifications may change, meaning that every existing proof has to be verified again to make sure that it still holds. For this purpose, we can use *proof assistants* to mechanically verify that our reasoning is still sound. Believing in the correctness of the system thus reduces to believing in the correctness of the proof assistant and the specifications that we give it.

The choice of proof assistant to use is a decision that is driven by many factors, e.g., its expressive power; our trust in the correctness of its implementation; its specification language; etc. The Twelf proof assistant [PS99] has proved to be very effective for verifying the meta-theory of programming languages, and has, for example, been used to verify type-safety of the programming language Standard ML [LCH07].

In this thesis, we will focus on a class of proofs that *are not* easily mechanized in Twelf, namely proofs by logical relations. Logical relations is a popular technique which is used for proving properties about programming languages based on typed lambda calculi. It has been an open problem whether proofs by logical relations could be formalized in Twelf at all, until demonstrated in [SS08], where Twelf-verified proofs of weak normalization and completeness of equivalence checking for simply typed lambda calculus were presented. There does, however, not seem to be examples of Twelf formalizations of proofs by logical relations for actual programming languages, i.e., languages with an operational semantics and richer feature sets.

We will attempt to close this gap, by investigating a number of case studies where we use Twelf to formalize the meta-theory of programming languages, using proofs by logical relations. Our goal is to identify any shortcomings of existing formalization methodologies, and to investigate if, and how, these shortcomings can be addressed. Ultimately, we hope to shed some light on the question of whether Twelf is feasible as a tool for language researchers that wish to verify their logical relations based proofs.

1.1 Contributions

We present three case studies and their formalizations in Twelf: a termination proof for a language with booleans and natural numbers; observational equivalence of a call-by-name language; and observational equivalence for a call-by-value language.

We identify several shortcomings of the method of *structural logical relations* [SS08], most notably problems with regards to the formalization of proofs that use case analysis in their core arguments. We solve this problem by presenting new extensions to the technique that allows more powerful reasoning principles to be used in proofs, with a trade-off of more boilerplate code in the formalization.

The result is a methodology that enbled case analysis and equality reasoning to be used in core parts of the formalized proofs, which allows a larger body of proofs by logical relations to be formalized in Twelf.

There are still several areas that we have not investigated in detail, and we point out several possible possible subjects for future work.

The full Twelf formalizations of the case studies presented in this thesis can be found online at [Ras13].

1.2 Overview of the thesis

This section gives an overview of the material that we will cover in each chapter. We will gradually be extending our formalization technique in each chapter, so later chapters will generally depend on the preceding ones. The object-language meta-theory not related to the Twelf formalizations can in principle be read independently of each other, and in any order.

The rest of the thesis is structured as follows.

- The following chapter introduces some general notational conventions, and briefly introduces the LF logical framework, the LF methodology for representing syntax, and the Twelf proof assistant. Readers are expected to have some familiarity with these topics, as the chapter will primarily serve to establish the nomenclature of the rest of the thesis. Readers not familiar with the topics are advised to consult the provided references.

- Chapter 3 presents a termination proof for call-by-name (CBN) simply typed λ-calculus with booleans and the corresponding formalization in Twelf. We will also introduce the general method of structural logical relations, which is the method that will be used in the Twelf formalizations in the rest of the thesis. The object language will be extended with features step-by-step, first by adding an elimination construct for booleans, and then by adding natural numbers and a case construct. We will demonstrate how the method of structural logical relations needs to be extended to accommodate the added features.

- Chapter 4 covers a soundness proof for an axiomatic reasoning system for observational equivalence between expressions in CBN simply typed λ-calculus. We will see how the formalization of this proof requires an additional representational layer in order to correctly reason about equalities.
- Chapter 5 presents a soundness proof for a system similar to the one in the preceding chapter, but for a call-by-value (CBV) simply typed λ-calculus. The proof requires a monadic extension to the logical relation, but the formalization can be carried out without introducing further extensions to the underlying techniques. The formalization does however require some clever representation techniques to work around proof-theoretical limits of Twelf, but we consider these as orthogonal to structural logical relations.
- In Chapter 6, we summarize the results of the thesis and point out possible future work.
- The appendices contain selected parts of the Twelf source code. The full source code can be obtained in the electronic appendix [Ras13].

2 Preliminaries

2.1 Notation

2.1.1 Syntax

Whenever syntax is declared, we specify its informal name, the names of meta-variables, a "sort", and the grammar as so:

$$\text{Expressions:} \quad e, v \;::\; \mathsf{Exp} \;::=\; x \mid e_1\ e_2 \mid \lambda x.\, e_0$$

We write sorts with uppercase sans-serif font, and use infix :: to denote that an object belongs to a given sort, meaning that it has been constructed using only the rules of the corresponding grammar. For example, we might write $e :: \mathsf{Exp}$ to denote that the variable e stands for an expression. A variable x is said to be *free* in some expression e iff it occurs in e, but is not a descendant of a lambda abstraction binding x. The set of free variables for an expression e is denoted $\mathrm{FV}(e)$.

We disallow shadowing of bound variables, i.e., all bound variables are uniquely identified with their binder. We consider expressions equivalent up to α-conversion.

Binders

When talking about the syntax of object language expressions, we will sometimes need to refer to expressions with one or more named free variables. We will refer to these as *binders*, and will write $x_1\ x_2\ \cdots\ x_n.\, e$ for the binder that binds the variables $x_1\ x_2\ \cdots\ x_n$ in the expression e. For example, $x.\, x\ y$ is a binder that binds the variable x, but has y as a free variable. Substitutions avoid capturing bound variables.

When we write

$$e :: \underbrace{(\mathsf{Exp})(\mathsf{Exp}) \cdots (\mathsf{Exp})}_{n \text{ times}}\, \mathsf{Exp},$$

we denote that e is a binder, binding n variables; e.g., $x.\, x\ y :: (\mathsf{Exp})\mathsf{Exp}$.

We avoid generalizing the concept of binders to other syntactic categories for simplicity.

Substitution

We will write substitutions with postfix notation. For example, we will write $e[e_2/x]$ for the result of substituting e_2 for the variable x in e. Substitutions follow the scope of the language, and avoid capturing bound variables.

When substituting objects for bound variables in binders, the variable may be omitted. For example, then $(x.\, e)[e_2]$ is equivalent to $e[e_2/x]$. Assuming $x \notin \mathrm{FV}(e_2)$, then the substitution $(x.\, x\ y)[e_2/y]$ is equivalent to $x.\, e\ e_2$; and $(x.\, x\ y)[e_2/x]$ is equivalent to $x.\, x\ y$.

2.1.2 Judgments

Judgments are declared using the same conventions as syntax. Instead of specifying a sort, we write the judgment in a box when it is introduced. For example, we may write $\boxed{e \Downarrow v}$ when introducing an evaluation judgment. We will write derivations of judgments with a calligraphy typeface, and reuse the infix :: to denote that a derivation derives a given judgment. E.g., we may write $\mathcal{E} :: \lambda x.\, e_0 \Downarrow v$.

2.1.3 Notational conventions

Proof labels

We will overload the infix :: and use it to label subgoals, hypotheses and established truths in proofs, but where the labels may not necessarily range over syntactic objects. For example, we might write $h :: \forall x.\, P(x) \Rightarrow Q(x)$ if we have just established or assumed $\forall x.\, P(x) \Rightarrow Q(x)$, and wish to refer to this later by the label h.

2.2 The Edinburgh Logical Framework

In the following, we will give a brief overview of LF and the syntactical conventions that we will use when we talk about it. Some familiarity with LF is expected to be able to follow the later developments, and we refer to [HL07] for a more thorough introduction.

When mechanizing the metatheory of programming languages, we are met with the choice of how to represent the syntactical categories of our *object language* (the programming language that we are studying) in a way that is faithful to our original specification. The LF logical framework [HHP93] is a dependently typed lambda calculus in which syntax, judgments, and derivations of an object language are represented as LF types, LF type families and canonical LF terms, respectively. Canonical LF terms are essentially the β-short and η-long terms, i.e., terms where all β-redices have been reduced, and where terms with function types have been η-expanded as much as possible without introducing a new β-redex. An LF encoding of an object language is said to be *adequate* iff it defines a compositional bijection between well-formed objects in the object language

Kinds:	$\mathtt{K}$	$::=$	$\mathtt{type} \mid \Pi \mathtt{x{:}A}.\,\mathtt{K}$
Canonical type families:	$\mathtt{A}$	$::=$	$\mathtt{P} \mid \Pi \mathtt{x{:}A_2}.\,\mathtt{A}$
Atomic type families:	$\mathtt{P}$	$::=$	$\mathtt{a} \mid \mathtt{P\,M}$
Canonical terms:	$\mathtt{M}$	$::=$	$\mathtt{R} \mid \lambda \mathtt{x}.\,\mathtt{M}$
Atomic terms:	$\mathtt{R}$	$::=$	$\mathtt{x} \mid \mathtt{c} \mid \mathtt{R\,M}$
Signatures:	Σ	$::=$	$\cdot \mid \Sigma, \mathtt{c} : \mathtt{A} \mid \Sigma, \mathtt{a} : \mathtt{K}$
Contexts:	Γ	$::=$	$\cdot \mid \Gamma, \mathtt{x} : \mathtt{A}$

(a) *Syntax of LF*

Signature formation:	$\mathcal{L}^s$	$::$	$\boxed{\vdash^{\mathsf{LF}} \Sigma\ \mathtt{sig}}$
Context formation:	$\mathcal{L}^c$	$::$	$\boxed{\vdash^{\mathsf{LF}}_{\Sigma} \Gamma\ \mathtt{ctx}}$
Kind formation:	$\mathcal{L}^k$	$::$	$\boxed{\Gamma \vdash^{\mathsf{LF}}_{\Sigma} \mathtt{K\ kind}}$
Canonical type formation:	$\mathcal{L}^a$	$::$	$\boxed{\Gamma \vdash^{\mathsf{LF}}_{\Sigma} \mathtt{A\ type}}$
Atomic type formation:	$\mathcal{L}^p$	$::$	$\boxed{\Gamma \vdash^{\mathsf{LF}}_{\Sigma} \mathtt{P} \Rightarrow \mathtt{K}}$
Canonical term formation:	$\mathcal{L}^m$	$::$	$\boxed{\Gamma \vdash^{\mathsf{LF}}_{\Sigma} \mathtt{M} \Leftarrow \mathtt{A}}$
Atomic term formation:	$\mathcal{L}^r$	$::$	$\boxed{\Gamma \vdash^{\mathsf{LF}}_{\Sigma} \mathtt{R} \Rightarrow \mathtt{A}}$

(b) *Formation judgments*

Figure 2.1: *Canonical LF syntax and formations.*

and canonical forms in LF with the associated types. We will explain what this means in greater detail later in this section.

The presentation of LF given here closely follows that of [HL07], and is known as *Canonical LF*. It differs from the original presentation of LF, in that only β-short and η-long forms are considered well-typed; intermediate reducible forms are not even considered. This property is ensured by the notion of *hereditary substitution*, which is a decidable procedure for finding the canonical result of substituting one canonical term into another: If, as a result of substitution, a redex of the form $(\lambda \mathtt{x}.\,\mathtt{M}_0)\,\mathtt{M}_2$ would have been formed, hereditary substitution immediately proceeds by recursively computing the canonical result of substituting $\mathtt{M}_2$ into $\mathtt{M}_0$.

We will write LF syntax in an upright typewriter font to distinguish it from the syntax of object languages. The syntax of Canonical LF is given in Figure 2.1a. The grammar is written such that β-redices cannot even be formed, by separating the syntax of types and terms into canonical and atomic subclasses.

The type theory of LF has three levels: kinds $\mathtt{K}$; types $\mathtt{A}$, $\mathtt{P}$; and terms $\mathtt{M}$, $\mathtt{R}$. Functions

are given dependent function types $\Pi x{:}A_1.\, A_2$, where x may occur free in A_2. We write $A_1 \to A_2$ in the degenerate case where x does not occur in A_2. The letters a and c range over *type families* and *term constants*. Objects of LF are implicitly considered equivalent up to α-conversion of bound variables.

When we write $E_0[M_2/x]$, we refer to the *hereditary substitution* of M_2 for x in E_0, where M_2 is a term and E_0 is any LF expression from one of the three levels. Given a context Γ, we will write $\Gamma[M_2/x]$ for the context where M_2 has been hereditarily substituted for x in all typing assumptions.

We will not give a complete definition of all the formation judgments of the type theory. Instead, we present their judgment forms and meaning, and refer to [HL07] for details. The two judgments for signtaure and context formation, as well as the five formation judgments for characterizing well-formed objects within the LF type theory, can be seen in Figure 2.1b. All expression formation judgments are parameterized by both a *signature* Σ and a *context* Γ. Signatures contain type family and term constant declarations which must be well-formed, written as $\vdash^{\mathsf{LF}} \Sigma\ \mathtt{sig}$, meaning that each declaration should have a well-formed kind or type in the signature consisting of all preceding (well-formed) declarations. Contexts introduce hypothetical assumptions labeled by variables. A context Γ is well-formed in a signature Σ, written $\vdash^{\mathsf{LF}}_{\Sigma} \Gamma\ \mathtt{ctx}$ if each type assumption is well-formed using preceding assumptions. Since the type theory is dependent, exchange of assumptions within contexts is not, in general, permitted. The main formation judgment we care about is $\Gamma \vdash^{\mathsf{LF}}_{\Sigma} M \Leftarrow A$, which says that the term M is a canonical form with type A in the signature Σ, using hypothetical assumptions Γ. The expression formation judgments all presuppose that the signature and context are well-formed.

The arrows in the formation judgments indicate the flow of information in type checking: For atomic types and terms, the associated kind or type is checked against the structure of the given type or term, respectively. For canonical terms, the structure of the term is checked against the given type, which is presupposed to be well-formed.

The LF logical framework aims to precisely model the usual notions of abstract syntax, well-formedness and capture-avoiding substitution which are ubiquitous in most formal presentations of programming languages and logics. It has a rich metatheory, and importantly enables proofs to be conducted by induction over canonical forms. One important property we will rely on later is that well-formedness is preserved by hereditary substitution:

Proposition 2.1. *(Substitution) If*

1. $\Gamma_1 \vdash^{\mathsf{LF}}_{\Sigma} M_0 \Leftarrow A_0$, *and*
2. $\Gamma_1, x_0 : A_0, \Gamma_2 \vdash^{\mathsf{LF}}_{\Sigma} M \Leftarrow A$,

then also

1. $\Gamma_1, \Gamma_2[M_0/x] \vdash^{\mathsf{LF}}_{\Sigma} M[M_0/x] \Leftarrow A[M_0/x]$.

In the following, we will give an overview of the LF methodology for representing syntax, and sketch a method for formally proving that the representation is faithful to the original definition.

2.2.1 Representing syntax

In the LF methodology, syntax and judgments are encoded as an LF signature. For each syntactic class in a given object language, we declare an associated type family, inhabited by a set of constants representing the syntactic constructors. For example, consider a minimal lambda calculus with a unit expression:

$$\text{Expressions:} \quad e \;::\; \mathsf{Exp} \;::=\; x \mid e_1\ e_2 \mid \lambda x.\, e_0 \mid ()$$

We assume the usual notion of capture-avoiding substitution, and write $e[e'/x]$ for the result of substituting the expression e' for any *free* occurrences of the variable x. A variable x is free iff it is not a descendant of a lambda binder for x. We can represent this syntax by the following LF signature, Σ:

$$\begin{array}{rcl} \mathtt{exp} & : & \mathtt{type} \\ \mathtt{app} & : & \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp} \\ \mathtt{lam} & : & (\mathtt{exp} \to \mathtt{exp}) \to \mathtt{exp} \\ \mathtt{unit} & : & \mathtt{exp} \end{array}$$

Here, we have declared an LF type family `exp` representing the sort Exp, as well as a constant for each syntactic constructor, except for variables. We omit an explicit constructor for variables as LF supports an elegant representation of them by *reusing the binders and variables of LF*. The constant representing lambda abstraction constructs a closed expression from an LF term-level lambda abstraction, taking expressions to expressions. This form of encoding object languages is known as *higher-order abstract syntax*, and has the advantage of providing capture avoiding substitution "for free", since it is implicitly provided by hereditary substitution within LF.

To specify the relationship between the signature and our object language, we define a translation function $\ulcorner\cdot\urcorner$ on sorts and expressions, relating well-formed object language expressions to canonical LF terms in the signature Σ:

$$\ulcorner \mathsf{Exp} \urcorner \;=\; \mathtt{exp}$$

$$\begin{array}{rcl} \ulcorner x \urcorner & = & \mathtt{x}_x \\ \ulcorner e_1\ e_2 \urcorner & = & \mathtt{app}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner \\ \ulcorner \lambda x.\, e_0 \urcorner & = & \mathtt{lam}\, (\lambda \mathtt{x}_x.\, \ulcorner e_0 \urcorner) \\ \ulcorner () \urcorner & = & \mathtt{unit} \end{array}$$

Note the typeface for variables: We translate variables from our object language to variables in LF. We index LF variables by their object language counterparts to underline

the one-to-one correspondence between them. To convince ourselves that the encoding is correct, we may show a property known as *adequacy*, which consists of showing that the translation is a compositional bijection. This means that the encoding gives a one-to-one relationship between the well-formed expressions of our object language, and well-typed canonical forms of type `exp` in Σ. In other words, there also exists an inverse translation $\llcorner \cdot \lrcorner$ from *well-typed* canonical terms of type `exp` to well-formed expressions. Additionally, our translation should be compositional, meaning that the notion of substitution in our object language coincides with hereditary substitution in LF. Showing that our translation is a bijection amounts to showing that the encoding is *complete* with respect to the original system, i.e., that all well-formed expressions can be represented; and that the encoding is also *sound*, i.e., that all well-typed LF terms with the appropriate type translates back to a well-formed expression. The proof obligations thus look as follows:

Completeness. For any well-formed expression e with free variables among $x_1, \ldots, x_n$, we have $\mathtt{x_1} : \mathtt{exp}, \ldots, \mathtt{x_n} : \mathtt{exp} \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner e \urcorner \Leftarrow \mathtt{exp}$.

Soundness. For any canonical LF term `M`, if $\mathtt{x_1}$:`exp`, $\ldots$, $\mathtt{x_n}$:`exp` $\vdash^{\mathsf{LF}}_{\Sigma}$ `M` $\Leftarrow$ `exp`, then there exists an e with free varaibles among $x_1, \ldots, x_n$, and $\ulcorner e \urcorner = \mathtt{M}$.

Compositionality. For any e, e', we have $\ulcorner e[e'/x] \urcorner = \ulcorner e \urcorner [\ulcorner e' \urcorner / \mathtt{x}_x]$.

A proof of the above will usually proceed by induction over canonical forms, using some of the results from the meta-theory of LF to reason about hereditary substitution.

We have avoided the problem of transport of adequacy proofs: A proof of the above only shows that the encoding is adequate in the minimal signature Σ, but usually we will introduce further declarations for our object language as well. The translation of these will most likely also utilize the LF context, meaning that the results above are not general enough, since they assume that the LF context exclusively contain assumptions of the type `exp`. We have left out the concept of *subordination* in this presentation, which together with meta-theoretical results about LF allows us to transfer an adequacy proof of the above to certain larger signatures and contexts. Specifically, it says that if inhabitants of some type family $\mathtt{a}'$ can never appear in the constants of another family `a`, the adequacy proofs concerning `a` can be transferred directly to larger signatures and contexts containing $\mathtt{a}'$.

In the developments in this thesis, we will not dwelve on the details of proving adequacy of our object languages, as it is not within the scope of our goals. We will usually present the LF signature representing our system, implicitly assuming the presence of translation functions representing an adequate relationship between the signature and our object language.

2.2.2 Representing judgments

Suppose we introduce a judgment for determining whether expressions with free variables among some context γ are *closed*, given that we substitute closed expressions for

every variable in γ. It is essentially a degenerate typing judgment in the case where we have only a single type:

$$\begin{array}{lllll} \text{Contexts:} & \gamma & :: & \mathsf{Ctx} & ::= \quad \cdot \mid \gamma, x \\ \text{Closed expressions:} & \mathcal{C} & :: & \boxed{\gamma \vdash e \text{ closed}} & : \end{array}$$

$$\mathsf{c\_var}: \frac{}{\gamma, x \vdash x \text{ closed}} \qquad \mathsf{c\_unit}: \frac{}{\gamma \vdash () \text{ closed}} \qquad \mathsf{c\_lam}: \frac{\gamma, x \vdash e_0 \text{ closed}}{\lambda x.\, e_0 \vdash \text{closed}}$$

$$\mathsf{c\_app}: \frac{\gamma \vdash e_1 \text{ closed} \quad \gamma \vdash e_2 \text{ closed}}{\gamma \vdash e_1\ e_2 \text{ closed}}$$

We allow implicit exchange of variables within contexts γ, and when we write γ, x (context γ extended with variable x), we implicitly mean that x does not occur in γ. We can easily show that the following substitution principle holds:

Proposition 2.2 (Substitution)**.** *If* $\mathcal{C}_2 :: \gamma \vdash e_2$ closed *and* $\mathcal{C} :: \gamma, x \vdash e$ closed*, then there is a derivation* $\mathcal{C}' :: \gamma \vdash e[e_2/x]$ closed*.*

Proof sketch. By induction on $\mathcal{C}$, replacing all uses of the variable assumption x with $\mathcal{C}_2$. □

We can represent the judgment in LF using the *judgments-as-types* methodology, representing the judgment $\gamma \vdash e$ closed as a type family. We can also extend the method of higher-order abstract syntax to judgments, by using the LF context to represent hypothetical derivations. We append the following declarations to the signature Σ defined in the last section:

$$\begin{array}{rcl} \texttt{clos} & : & \texttt{exp} \to \texttt{type} \\ \texttt{clos/unit} & : & \texttt{clos unit} \\ \texttt{clos/lam} & : & \Pi \texttt{e}_0{:}\texttt{exp} \to \texttt{exp}.\, (\Pi \texttt{x}{:}\texttt{exp}.\, \texttt{clos x} \to \texttt{clos}\, (\texttt{e}_0\ \texttt{x})) \to \texttt{clos}\, (\texttt{lam}\, (\lambda \texttt{x}.\, \texttt{e}_0\ \texttt{x})) \\ \texttt{clos/app} & : & \Pi \texttt{e}_1{:}\texttt{exp}.\, \Pi \texttt{e}_2{:}\texttt{exp}.\, \texttt{clos}\ \texttt{e}_1 \to \texttt{clos}\ \texttt{e}_2 \to \texttt{clos}\, (\texttt{app}\ \texttt{e}_1\ \texttt{e}_2) \end{array}$$

Again, we have left out the rule for variables. Instead, we craft the representation of c_lam such that it introduces a hypothetical derivation of `clos x` whenever it extends the LF context with a variable `x : exp`. We do not represent variable contexts explicitly, but define a translation from them to LF contexts:

$$\begin{array}{rcl} \ulcorner \cdot \urcorner & = & \cdot \\ \ulcorner \gamma, x \urcorner & = & \ulcorner \gamma \urcorner, \texttt{x}_x : \texttt{exp}, \texttt{u}_x : \texttt{clos x}_x \end{array}$$

This ensures that there is a corresponding labeled hypothetical derivation for each use of c_var, and we can thus define

$$\ulcorner\ \mathsf{c\_var}: \frac{}{\gamma, x \vdash x \text{ closed}}\ \urcorner \quad = \quad \texttt{u}_x,$$

where $\mathtt{u}_x$ is the label that is associated with the hypothetical derivation for x. The translation of c_lam must also respect the representation of contexts, and the translation of the remaining rules and the judgment itself thus looks as follows:

$$\ulcorner e \text{ closed} \urcorner \;=\; \mathtt{clos}\, \ulcorner e \urcorner$$

$$\ulcorner\ \mathsf{c\_lam}\colon \frac{\begin{array}{c}\mathcal{C}'\\ \gamma, x \vdash e_0 \text{ closed}\end{array}}{\gamma \vdash \lambda x.\, e_0 \text{ closed}}\ \urcorner \;=\; \mathtt{clos/lam}\ (\lambda \mathtt{x}_x.\, \ulcorner e_0 \urcorner)\ (\lambda \mathtt{x}_x.\, \lambda \mathtt{u}_x.\, \ulcorner \mathcal{C}' \urcorner)$$

$$\ulcorner\ \mathsf{c\_app}\colon \frac{\begin{array}{cc}\mathcal{C}_1 & \mathcal{C}_2\\ \gamma \vdash e_1 \text{ closed} & \gamma \vdash e_2 \text{ closed}\end{array}}{\gamma \vdash e_1\ e_2 \text{ closed}}\ \urcorner \;=\; \mathtt{clos/app}\, \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner\, \ulcorner \mathcal{C}_1 \urcorner\, \ulcorner \mathcal{C}_2 \urcorner$$

$$\ulcorner\ \mathsf{c\_unit}\colon \frac{}{\gamma \vdash () \text{ closed}}\ \urcorner \;=\; \mathtt{clos/unit}$$

Note that the definition depends on a translation for expressions.

To justify the representation using hypothetical derivations, we observe that the judgment $\gamma, x \vdash e$ closed is equivalent to the following *hypothetical judgment*, which is parametric in x and hypothetical in u:

$$\begin{array}{c} u\,\overline{\gamma \vdash x \text{ closed}} \\ \vdots \\ \gamma \vdash e \text{ closed} \end{array}$$

in that we have a derivation of one *if and only if* we have a derivation of the other. We can therefore restate the substitution lemma as

Proposition 2.3 (Substitution, alternative formulation)**.** *If* $\mathcal{C}_2 :: \gamma \vdash e_2$ closed *and*

$$\mathcal{C} :: \begin{array}{c} u\,\overline{\gamma \vdash x \text{ closed}} \\ \vdots \\ \gamma \vdash e \text{ closed} \end{array}$$

then $\mathcal{C}[e_2/x][\mathcal{C}_2/u] :: \gamma \vdash e[e_2/x]$ closed*.*

The proof obligations for proving adequacy look similar to those for the LF representation of expressions. The difference is that we now have to take contexts of the judgment into consideration as well. The proof obligations for soundness and completeness would thus look as follows:

Completeness. For any derivation $\mathcal{C} :: \gamma \vdash e$ closed, we have $\ulcorner \gamma \urcorner \overset{\mathsf{LF}}{\vdash_\Sigma} \ulcorner C \urcorner \Leftarrow \mathtt{clos}\, \ulcorner e \urcorner$.

Soundness. Suppose γ is a variable context, and $\ulcorner \gamma \urcorner = \Gamma$. For any canonical LF terms $\mathtt{M}$, $\mathtt{M}'$, if $\Gamma \overset{\mathsf{LF}}{\vdash_\Sigma} \mathtt{M}' : \mathtt{exp}$ and $\Gamma \overset{\mathsf{LF}}{\vdash_\Sigma} \mathtt{M} \Leftarrow \mathtt{clos}\, \mathtt{M}'$, then there is a derivation $\mathcal{C} :: \gamma \vdash e$ closed such that $\ulcorner e \urcorner = \mathtt{M}'$ and $\ulcorner \mathcal{C} \urcorner = \mathtt{M}$.

Compositionality. For any $\mathcal{C}_2 :: \gamma \vdash e_2$ closed, and

$$\mathcal{C} :: \begin{array}{c} u\,\overline{\gamma \vdash x \text{ closed}} \\ \vdots \\ \gamma \vdash e \text{ closed} \end{array}$$

we have

$$\ulcorner \mathcal{C}[e_2/x][\mathcal{C}_2/u] \urcorner = \ulcorner \mathcal{C} \urcorner[\ulcorner e_2 \urcorner / \mathtt{x}_x][\ulcorner \mathcal{C}_2 \urcorner / \mathtt{u}_x]$$

2.3 The Twelf meta-logical framework

In this section, we will give a brief overview of the Twelf meta-logical framework [PS99]. For a more thorough introduction, we refer to Pfenning's lecture notes [Pfe01] and the Twelf User's Guide [PS02].

As we saw in the last section, the LF methodology provides a way of representing syntax and judgments as well-formed LF terms and types. But what about the representation of *meta-theorems* about the systems that we represent? If we have provided an adequate LF representation of a given object language, then surely we should be able to translate an argument by induction on derivations to an argument by induction on well-typed canonical forms.

The Twelf meta-logical framework is an implementation of the LF type theory, and thus supports the LF methodology for representing abstract syntax and judgments. Given an adequate encoding of a logical system, this provides mechanical verification of proofs *within* the system, by reducing verification to type-checking of LF terms. The syntax of Twelf closely follows the LF syntax. For example, the encoding of the expressions and judgments from the last section looks as follows:

```
% Expressions
exp : type.
app : exp -> exp -> exp.
lam : (exp -> exp) -> exp.
unit : exp.
```

```
% Judgments
clos : exp -> type.
clos/unit : clos unit.
clos/lam : {e0:exp -> exp}
           ({x:exp}
            clos x -> clos (e0 x))
         -> clos (lam [x] e0 x).
clos/app : clos E1
         -> clos E2
         -> clos (app E1 E2).
```

The syntax uses braces for kind and type level abstraction (Π in the type theory) and brackets for term-level abstraction. By writing variables with upper-case syntax, their type-level binders can often be omitted as long as Twelf can unambiguously infer their type from the context (we have illustrated this in the definition of `clos/app`). Likewise, type annotations can often be omitted. Since type inference for LF is in general undecidable, annotations will sometimes be needed. Type arrows may be reversed, thus writing `c <- b <- a` is equivalent to writing `a -> b -> c`.

Additionally, Twelf provides meta-level facilities for searching for inhabitants of a given type family. For example, we can determine whether $\cdot \vdash (\lambda x.\, x)\ ()$ closed by writing

```
%query 1 1 D : clos (app (lam [x] x) unit).
```

By which Twelf responds with the solution

```
D = clos/app (clos/lam ([x:exp] x) ([x:exp] [x1:clos x] x1)) clos/unit.
```

Proof search can be viewed as an operationalization of judgments. For example, if we encode an evaluation judgment on expressions, we can "run" the judgment as an interpreter by querying for an evaluation derivation given a ground expression for the input, leaving the output as a free meta-variable. Constants of a given type family can thus be viewed as the clauses of a logic program, and proof search as program execution.

Since proofs can be viewed as programs under the Curry-Howard correspondence, this gives us a way to represent *meta-theorems* via higher-order judgments. Additionally, Twelf has facilities for proving the totality of a judgment under the judgments-as-programs interpretation, i.e., proving that whenever certain arguments of a type family are instantiated with *ground* terms, proof search will always terminate successfully. This requires some extra annotations for specifying which type family arguments are inputs and which are outputs, as well as specifying a metric for justifying recursive calls. Meta-theorems may also extend the LF context, requiring the specification of *regular worlds* which asserts that the context is only extended in certain predefined, regular ways.

For example, we can prove that `clos` is complete, i.e., that for every closed expression `e`, we have a derivation `clos e`. This statement can be expressed as the following type family:

```
clos-tot : {e:exp} clos e -> type.
%mode clos-tot +E -CP.
```

The mode clause is a meta-level declaration, specifying that the expression is to be considered an input, and that the `clos`-derivation is an output. The proof cases of the theorem are given as constants inhabiting the type family. They are usually written in reverse arrow notation to follow the conventions of logic programming languages. As the constants will only be used for proof search, they need not be named:

```
- : clos-tot unit clos/unit.
- : clos-tot (app E1 E2) (clos/app CP1 CP2)
     <- clos-tot E1 CP1
     <- clos-tot E2 CP2.
- : clos-tot (lam E0) (clos/lam E0 CP0)
     <- ({x}{cp:clos x} clos-tot x cp -> clos-tot (E0 x) (CP0 x cp)).
```

In the case for `lam`, the meta-theorem proof needs to extend the LF context in order to recurse on a *closed* expression. This dynamically extends the `exp` type family in the enlarged context, implicitly introducing a new, uncovered proof case. To cover this case, we also need to extend the LF context with a new meta-theorem case, which covers the new case. Twelf can verify this kind of reasoning if we add annotations specifying that we only extend the LF context in this way:

```
%block bclos-tot : block {x:exp}{cp:clos x}{_:clos-tot x cp}.
%worlds (bclos-tot) (clos-tot _ _).
```

Finally, we can ask Twelf to verify the totality of `clos-tot`. To do this, we need to specify what we do induction over:

```
%total (E) (clos-tot E _).
```

The above declaration will first perform a *coverage check*, checking that the inhabitants of `clos-tot` covers all possible inputs, and that assumptions made on outputs from recursive calls are sufficiently general. Next, a termination check will be performed, checking that all recursive calls only occur on smaller arguments, ensuring that proof search must terminate at some point. Twelf supports induction by lexicographic ordering, corresponding to nested induction, as well as mutual induction on type families.

This methodology allows us to represent and verify a relatively large body of meta-theorems. Since meta-theorems must be expressed as dependently typed logic programs, there are restrictions on the form of meta-theorems we can represent. In general, Twelf supports proving meta-theorems of the form

$$\forall \mathtt{x}_1 : \mathtt{A}_1. \cdots \forall \mathtt{x}_n : \mathtt{A}_n. \exists \mathtt{y}_1 : \mathtt{B}_1. \cdots \exists \mathtt{y}_m : \mathtt{B}_m. \top,$$

where any variable x_i or y_j may occur in *later* types `A` or `B`, and where types may be uninhabited. Central is the restriction that quantifiers cannot be alternated, but must come in the order indicated above; we refer to theorems on this form as $\forall\exists$-theorems.

A surprisingly large amount of meta-theorems can actually be formulated this way, but as we shall see later, this does not, in general, apply for proofs by logical relations.

3 Termination for CBN simply typed λ-calculus

In this chapter, we will describe the formalization of a logical relations proof of termination of a simply typed lambda calculus (STLC) with a call-by-name operational semantics, and extended with various common programming language constructs such as booleans, natural numbers and branching constructs.

This chapter will serve both as an introduction to proofs by logical relations and to the methods for formalizing such proofs using the Twelf proof-assistant. We will start out with a minimal STLC having only application and abstraction and will progressively add object language features, showing how each feature affects the termination proof and the corresponding formalization.

The chapter is structured as follows. In Section 3.1, we introduce logical relations and give a simple example of a termination proof for a minimal STLC with booleans, but no if-construct. In Section 3.2, we describe how to formalize the termination proof in Twelf by conducting core parts of the proof in an auxiliary assertion logic. We then add an if-construct in Section 3.3, extending the termination proof and the formalization. In Section 3.4 we add both natural numbers and a case construct to the object language. In Section 3.5, we demonstrate how this requires the assertion logic to be extended with new reasoning principles.

3.1 A simple logical relation

A proof by logical relations relies on the definition of a family $\{R_\tau\}_{\tau \text{ type}}$ of type-indexed relations on expressions from the object language. The relation is usually defined such that the desired result follows immediately by the definition at base types, and crucially, such that the relation $R_{\sigma\to\tau}$ at function types is defined in terms of the relations R_σ, R_τ in a way such that closure under lambda abstraction and application is guaranteed. A common way to say this is that *"expressions related at function types should take related arguments to related results"*.

Using a logical relation is often necessary when one tries to prove a property of

Expressions: $e, v :: \mathsf{Exp} ::= x \mid e_1\ e_2 \mid \lambda x.\, e_0 \mid \mathsf{true} \mid \mathsf{false}$
Types: $\tau :: \mathsf{Tp} ::= \mathsf{bool} \mid \tau_2 \to \tau_0$
Contexts: $\Gamma :: \mathsf{Ctx} ::= \cdot \mid \Gamma, x : \tau$
Values: $\mathcal{V} :: \boxed{v\ \mathsf{value}}$:

$$\mathsf{v\_lam}: \frac{}{\lambda x.\, e_0\ \mathsf{value}} \qquad \mathsf{v\_true}: \frac{}{\mathsf{true\ value}} \qquad \mathsf{v\_false}: \frac{}{\mathsf{false\ value}}$$

Dynamic semantics: $\mathcal{E} :: \boxed{e \Downarrow v}$ (e closed) :

$$\mathsf{e\_true}: \frac{}{\mathsf{true} \Downarrow \mathsf{true}} \qquad \mathsf{e\_false}: \frac{}{\mathsf{false} \Downarrow \mathsf{false}} \qquad \mathsf{e\_lam}: \frac{}{\lambda x.\, e_0 \Downarrow \lambda x.\, e_0}$$

$$\mathsf{e\_app}: \frac{e_1 \Downarrow \lambda x.\, e_2 \quad e_1[e_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

Static semantics: $\mathcal{T} :: \boxed{\Gamma \vdash e : \tau}$:

$$\mathsf{t\_var}: \frac{}{\Gamma \vdash x : \tau}\ (\Gamma(x) = \tau) \qquad \mathsf{t\_true}: \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \mathsf{t\_false}: \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\mathsf{t\_lam}: \frac{\Gamma, x : \tau_2 \vdash e_0 : \tau_0}{\Gamma \vdash \lambda x.\, e_0 : \tau_2 \to \tau_0} \qquad \mathsf{t\_app}: \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_0 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_0}$$

Figure 3.1: *Syntax and semantics of* $\lambda^{\to,\mathsf{bool}^-}_{\mathsf{cbn}}$.

a typed lambda calculus but finds that the induction hypothesis is too weak for the proof to go through. Additionally, the technique tends to scale well. In the following, we will illustrate the technique by giving a termination proof of the simply typed lambda calculus with call-by-name semantics and booleans, or just $\lambda^{\to,\mathsf{bool}^-}_{\mathsf{cbn}}$. The minus superscript indicates that we have not yet added any elimination forms (e.g., if-constructs) to the language. The syntax and semantics are given in Figure 3.1. As a technical convenience, whenever we write $\Gamma, x : \tau$, it is implicit that we also mean $x \notin \mathrm{dom}(\Gamma)$. The corresponding encoding in LF is presented as a Twelf signature in Figure 3.2.

Proving that all well-typed expressions terminate specifically means proving that if $\mathcal{T} :: \cdot \vdash e : \tau$, then there exists a v such that $\mathcal{E} :: e \Downarrow v$. We cannot prove this property directly by induction over $\mathcal{T}$, as we will get stuck in the case for t_app; induction will only give us termination of the two subexpressions, which is not sufficient for proving that there exists an evaluation for the whole application.

We therefore need to formulate a stronger induction hypothesis from which termination follows as a special case. The solution is to use a *unary logical relation*, which in this case is a type-indexed predicate on expressions. We define the predicate such that it trivially implies termination, but using a slightly different evaluation judgment, defined in Figure 3.3, which turns out works a little smoother for the proof to come. Our logical

```
% Types
tp : type.
bool : tp.
=> : tp -> tp -> tp.
     %infix right 1 =>.

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

% Values
val : exp -> type.
val/lam : val (lam E0).
val/true : val true.
val/false : val false.
```

```
% Evaluation judgment
eval : exp -> exp -> type.
eval/lam : eval (lam E0) (lam E0).
eval/app : eval E1 (lam E0)
        -> eval (E0 E2) V
        -> eval (app E1 E2) V.
eval/true : eval true true.
eval/false : eval false false.

% Typing judgment
of : exp -> tp -> type.
of/lam : ({x} of x T2 -> of (E0 x) T0)
      -> of (lam E0) (T2 => T0).
of/app : of E1 (T2 => T0)
      -> of E2 T2
      -> of (app E1 E2) T0.
of/true : of true bool.
of/false : of false bool.
```

Figure 3.2: *Twelf signature for* $\lambda^{\rightarrow,\mathsf{bool}^-}_{\mathsf{cbn}}$

relation is defined as follows:

Definition 3.1 (Logical termination)**.**

$$P_{\mathsf{bool}}(e) \Leftrightarrow \exists v.\, e \searrow v,$$
$$P_{\tau_2 \to \tau_0}(e) \Leftrightarrow (\exists v.\, e \searrow v) \wedge \forall e_2.\, P_{\tau_2}(e_2) \Rightarrow P_{\tau_0}(e\ e_2). \qquad \diamondsuit$$

The relation can be seen as an interpretation of types as unary predicates on expressions. It is clear from the definition that for any e, if we can show $P_\tau(e)$, then $e \searrow v$ (for some v) follows directly. We will show in a moment that evaluation by reduction implies our original big-step formulation of evaluation, and hence that the logical relation implies termination. To show termination of well-typed expressions, it thus suffices to show that if $\cdot \vdash e : \tau$, then also $P_\tau(e)$.

The relation as defined expresses a semantic notion of well-typedness, namely that the expression in question evaluates. It is not the same as well-typedness as defined in Figure 3.1: Consider for example the expression $e_1 \stackrel{\text{def}}{=} (\lambda f.\, f\ f)\ \lambda x.\, x$ for which $P_{\mathsf{bool}\to\mathsf{bool}}(e_1)$ can easily be seen to be true, but where $\cdot \nvdash e_1 : \mathsf{bool} \to \mathsf{bool}$, due to the untypable subexpression $\lambda f.\, f\ f$.

Before we prove that any well-typed term satisfies the logical relation, we need to establish that it actually implies termination. To do that, we need to prove that evaluation by reduction is sound with respect to big-step evaluation, which requires the following two auxiliary lemmas. The first shows that big-step evaluation is closed under head reduction, and the second that values always evaluate to themselves:

Reduction context: $\mathcal{R} ::= \circ \mid \mathcal{R}\ e_2$

Weak head reduction: $\mathcal{H} :: \boxed{e \rightsquigarrow e'}$:

$$\textsf{whr\_beta}: \frac{}{(\lambda x.\, e_0)\ e_2 \rightsquigarrow e_0[e_2/x]}$$

Evaluation by reduction: $\mathcal{E} :: \boxed{e \searrow v}$:

$$\textsf{ev\_val}: \frac{v\ \text{value}}{v \searrow v} \qquad \textsf{ev\_whr}: \frac{e \rightsquigarrow e' \quad \mathcal{R}\{e'\} \searrow \mathcal{R}\{v\}}{\mathcal{R}\{e\} \searrow \mathcal{R}\{v\}}$$

(a) *Evaluation by reduction.*

```
    % Reduction contexts
ctx : (exp -> exp) -> type.
ctx/id : ctx [x] x.
ctx/app : ctx R
      -> ctx ([x] app (R x) E2).

% Weak head reduction
whr : exp -> exp -> type.
whr/beta :
       whr (app (lam E0) E2) (E0 E2).
```

```
% Evaluation by reduction
eval~ : exp -> exp -> type.
eval~/val : val V -> eval~ V V.
eval~/whr :  ctx RX
         -> whr E E'
         -> eval~ (RX E') V
         -> eval~ (RX E) V.
```

(b) *Twelf representation.*

Figure 3.3: *Evaluation by reduction together with the Twelf representation.*

Lemma 3.2 (Converse head reduction)**.** *If* $\mathcal{E} :: \mathcal{R}\{e'\} \Downarrow v$ *and* $\mathcal{H} :: e \rightsquigarrow e'$*, then* $\mathcal{R}\{e\} \Downarrow v$*.*

Proof. By induction on $\mathcal{R}$.

- Case $\mathcal{R} = \circ$. We proceed by cases on $\mathcal{H}$. The only possible case is whr_beta, so we have $\mathcal{E} :: e_0[e_2/x] \Downarrow v$ for some $x.\, e_0$ and e_2. But then we construct the goal by e_app and e_lam on $\mathcal{E}$.
- Case $\mathcal{R} = \mathcal{R}'\ e_2$. Then $\mathcal{E}$ must end in e_app, implying that we have $\mathcal{E}_1' :: \mathcal{R}'\{e'\} \Downarrow \lambda x.\, e_0$ and $\mathcal{E}_2 :: e_0[e_2/x] \Downarrow v$. By the induction hypothesis (IH) on $\mathcal{R}'$ with $\mathcal{E}_1'$, we get $\mathcal{E}_1 :: \mathcal{R}'\{e\} \Downarrow \lambda x.\, e_0$, and by e_app on $\mathcal{E}_1$,$\mathcal{E}_2$, we are done. □

Lemma 3.3 (Values evaluate)**.** *If* v value*, then* $v \Downarrow v$*.*

Proof. Immediate. □

The conversion lemma can then be proved by simple induction over the iterated weak head reduction:

Lemma 3.4 (Soudness of iterated reduction)**.** *If $\mathcal{E} :: e \searrow v$, then $e \Downarrow v$.*

Proof. By induction over $\mathcal{E}$, applying Lemma 3.3 in the case for ev_val, and Lemma 3.2 and IH in the case for ev_whr. □

We have now established that to show $e \Downarrow v$ for some v, it is sufficient to show $P_\tau(e)$. It remains to show that any well-typed expression satisfies the relation at the appropriate type. The proof depends crucially on the following lemma:

Lemma 3.5 (Closure under weak head expansion)**.** *If $a :: P_\tau(\mathcal{R}\{e'\})$ and $\mathcal{H} :: e \rightsquigarrow e'$, then $P_\tau(\mathcal{R}\{e\})$.*

Proof. By induction on τ.

- Case $\tau = \mathsf{bool}$. By assumption, we have $\mathcal{E} :: \mathcal{R}\{e'\} \searrow v$ for some v, and so we get $\mathcal{R}\{e\} \searrow v$ by ev_whr on $\mathcal{E}$ and $\mathcal{H}$.

- Case $\tau = \tau_2 \to \tau_0$. By assumption, we have $\mathcal{E} :: \mathcal{R}\{e'\} \searrow v$ for some v, and so we get $\mathcal{R}\{e\} \rightsquigarrow v'$ by ev_whr on $\mathcal{E}$ and $\mathcal{H}$. It remains to show that for any e_2, if $h :: P_{\tau_2}(e_2)$, then also $P_{\tau_0}(\mathcal{R}\{e\}\ e_2)$. By a on h, we get $P_{\tau_0}(\mathcal{R}\{e'\}\ e_2)$, and hence by IH on τ_0 with $\mathcal{H}$ we are done. □

To be able to formulate our induction hypothesis, we need to introduce a little bit of extra machinery. The reason for this is that the proof needs to work for typing derivations with non-empty contexts. In the following, we will use γ to range over finite maps from variables to closed expressions. If $\gamma = [x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ is one such map, we write $\hat{\gamma}(e)$ for the substitution $e[e_1/x_1, \ldots, e_n/x_n]$. We will write $P_\Gamma(\gamma)$ if $\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma)$, and for every $x \in \mathrm{dom}(\Gamma)$, we have $P_{\Gamma(x)}(\gamma(x))$.

We have now set up everything that is needed to prove the main result, namely that every well-typed term belongs to the logical relation at the appropriate type. This property is often referred to as the *fundamental theorem*:

Theorem 3.6 (Fundamental theorem)**.** *For any expression e, if $\mathcal{T} :: \Gamma \vdash e : \tau$, then for any substitution γ where $P_\Gamma(\gamma)$, it holds that $P_\tau(\hat{\gamma}(e))$.*

Proof. By induction on $\mathcal{T}$.

- Case $\mathcal{T}$ ends in t_var: We have $e = x$ and $x \in \mathrm{dom}(\Gamma)$. Since $P_\Gamma(\gamma)$ by assumption, it follows that $P_\tau(\gamma(x))$, and we are done.

- Case $\mathcal{T}$ ends in t_true or t_false: We cover the first case, as the second is analogous. We have $e = \mathsf{true}$, and it suffices to show $e \searrow v$ for some v. We choose $v = \mathsf{true}$, and construct the evaluation by ev_val and v_true, and we are done.

- Case $\mathcal{T}$ ends in t_lam: we have $e = \lambda x.\, e_0$ and $\tau = \tau_2 \to \tau_0$ and a derivation $\mathcal{T}_0 :: \Gamma, x : \tau_2 \vdash e_0 : \tau_0$. We thus have $\hat{\gamma}(e) = \lambda x.\, \hat{\gamma}(e_0)$, so we get $e \searrow v$ by ev_val and val_lam. It thus remains to show that for any e_2 where $P_{\tau_2}(e_2)$, we have $P_{\tau_0}(e\ e_2)$.

  We now construct $\gamma' = \gamma[x \mapsto e_2]$. Since $P_{\tau_2}(e_2)$ by assumption, we also have $P_{\Gamma, x:\tau_2}(\gamma')$, justifying the use of IH on $\mathcal{T}_0$ to obtain a proof of $P_{\tau_0}(\hat{\gamma}'(e_0))$, or equivalently, $P_{\tau_0}(\hat{\gamma}(e_0)[e_2/x])$. By Lemma 3.5, it thus suffices to show

  $$(\lambda x.\, \hat{\gamma}(e_0))\ e_2 \rightsquigarrow \hat{\gamma}(e_0)[e_2/x],$$

  which follows directly by whr_beta, and we are done.

- Case $\mathcal{T}$ ends in t_app: We have $e = e_1\ e_2$, and derivations $\mathcal{T}_1 :: \Gamma \vdash e_1 : \tau_2 \to \tau$ and $\mathcal{T}_2 :: \Gamma \vdash e_2 : \tau_2$. By IH on $\mathcal{T}_1$ and $\mathcal{T}_2$, we get $P_{\tau_2 \to \tau_0}(\hat{\gamma}(e_1))$ and $P_{\tau_2}(\hat{\gamma}(e_2))$, respectively. But then $P_{\tau_0}(\hat{\gamma}(e)\ \hat{\gamma}(e_2))$ follows directly by the definition of $P_{\tau_2 \to \tau_0}$, and we are done. □

Termination follows as a trivial corollary:

Corollary 3.7 (Termination)**.** *For any e where $\mathcal{T} :: \cdot \vdash e : \tau$, there exists a v such that $e \Downarrow v$.*

Proof. By Theorem 3.6, we get $P_\tau(e)$. Regardless of τ, this implies $\mathcal{E} :: e \searrow v$ for some v. But then by Lemma 3.4 on $\mathcal{E}$, we are done. □

The proof presented above is a very simple example of a logical-relations based proof, employing a unary logical relation. In general, there are no restrictions on the arity of the relation, although the practical applications of logical relations with arity beyond two is limited. We made things a little bit easier for ourselves in the definition of the logical relation, since we added our desired property (termination) explicitly at the definition for function types, instead of only including it for base types. The proof could be adapted to work for such a definition as well, although we would have to prove a separate *escape* lemma to show how to "get out" of the logical relation at function types.

The proof technique was first introduced by Tait [Tai67] and is thus sometimes referred to in the literature as *Tait's method*. Logical relations have a wide range of applications besides proving termination (which is a property that most programming languages lack anyway), including, but not limited to: contextual refinement proofs [TTA+13], completeness of equivalence checking [HP05] and proving observational equivalence [Har13].

3.2 Structural logical relations

In this section, we will describe how to formalize the termination proof in the Twelf proof assistant. As explained in Section 2.3, the Twelf meta-logical framework is designed to prove meta-theorems of the form

$$\forall x_1 : A_1. \cdots \forall x_n : A_n. \exists y_1 : B_1. \cdots \exists y_m : B_m.\ \top,$$

where any variable x_i or y_j may occur in *later* types A or B, and where types may be uninhabited. Under the judgments-as-types interpretation, this allows us to express a broad range of meta-theorems about object logics and programming languages in particular. However, for logical-relations based proofs, this model is problematic. The reason is due to the inherent alternation of quantifiers in the definition of the logical relation, which is incompatible with the restrictions on the forms of meta-theorems that can be verified by Twelf.

For example, recall the logical relation P_τ defined in Section 3.1. For some expression e, a proof of $P_{\mathsf{bool}}(e)$ can obviously be straightforwardly represented by showing the existence of an evaluation derivation for e. On the other hand, we cannot formulate $P_{(\mathsf{bool}\to\mathsf{bool})\to\mathsf{bool}}(e)$ as a meta-theorem, due to its form. We have:

$$\begin{aligned} P_{(\mathsf{bool}\to\mathsf{bool})\to\mathsf{bool}}(e) &\Leftrightarrow (\exists v.\, e \searrow v) \wedge \forall e_2.\, P_{\mathsf{bool}\to\mathsf{bool}}(e_2) \Rightarrow P_{\mathsf{bool}}(e\ e_2) \\ P_{\mathsf{bool}\to\mathsf{bool}}(e_2) &\Leftrightarrow (\exists v_2.\, e_2 \searrow v_2) \wedge \forall e_2'.\, P_{\mathsf{bool}}(e_2') \Rightarrow P_{\mathsf{bool}}(e_2\ e_2') \\ P_{\mathsf{bool}} &\Leftrightarrow \exists v.\, e \searrow v. \end{aligned}$$

As noted above, P_{bool} is obviously on the $\forall\exists$-form, and we could reformulate the statement $P_{\mathsf{bool}\to\mathsf{bool}}(e_2)$ as the following $\forall\exists$-statement:

$$\forall e_2'.\, \forall v_2'.\, \forall(\mathcal{E}_2' :: e_2' \searrow v_2').\, \exists v_2.\, \exists v''.\, \exists(\mathcal{E}_2 :: e_2 \searrow v_2).\, \exists(\mathcal{E}'' :: e_2\ e_2' \searrow v'').\, \top$$

However, what about $P_{(\mathsf{bool}\to\mathsf{bool})\to\mathsf{bool}}(e)$? It seems to be defined in terms of another $\forall\exists$-statement, namely $P_{\mathsf{bool}\to\mathsf{bool}}(e_2)$, which occurs on the *left* of an implication. This nesting of implications seems to go beyond the limits of what we can express as an $\forall\exists$-statement and hence as Twelf meta-theorems, as such a statement is no longer just hypothetical in syntactic derivations, but hypothetical in the truth of *another* meta-theorem.

In informal proofs we have the implicit understanding that types, judgments, implication and structural induction all exists on the same level, and thus does not impose the same restrictions on the things we consider provable. But this is not the case in Twelf, where there is a clear distinction between the level of representation (LF) and meta-theorems (the Twelf meta-logic).

At first, it would thus look like Twelf is too weak to support this kind of reasoning. However, it was demonstrated by Schürmann and Sarnat [SS08] that logical relations *can* be represented in Twelf after all, by introducing the methodology of *structural logical relations*. Importantly, the proofs conducted using this method remains verifiable in Twelf as meta-theorems. The method proceeds by explicitly representing an auxiliary *assertion logic* in which core parts of the proof is conducted. The logic imposes no restrictions on the way quantifiers are nested, but has some other limitations instead. In the following, we will define an assertion logic and describe its consistency proof. We will then return to our termination proof, and show how to formalize the logical relation.

3.2.1 The assertion logic

The first obstacle to overcome is the question of how to represent the logical relation at all. For this purpose, we define an auxiliary logic, henceforth referred to as the *assertion logic*, equipped with the logical connectives that we need to define the logical relation and for proving properties about it. The required strength of the assertion logic inherently depends on the object logic (i.e., the programming language that we are studying) and the properties that we are interested in proving. Since we are ultimately interested in proving that some judgment has a derivation, we will at least need a way to "embed" judgments and their rules inside the logic. The assertion logic that we use in this formalization is defined in Figure 3.4, and has been equipped with rules and formulas for quantifying over expressions and evaluation derivations.

The definition deserves some explanation. A derivation of a sequent of the form $\Xi|\Delta \vdash^{c}_{\Sigma} A$ is an assertion logic proof of the validity of the formula A, parametric in an ordered list of meta-variables Ξ, and hypothetical in an unordered set of assumptions Δ. Additionally, sequents are indexed by an LF signature Σ; we will return to the meaning of the parameter c shortly. The system allows for quantification over two sorts, namely expressions and evaluation derivations. In all rules where concrete objects are substituted for meta-variables (i.e., rules exidR, exieR and alleL), we have a restriction that says that the given object must have a well-typed LF encoding. Specifically, we assume that Σ is an *adequate* encoding of our object language, and that $\ulcorner \cdot \urcorner$ defines a suitable translation function. We will tacitly assume that meta-variables may occur anywhere in objects. The inclusion of the LF translation of the parameters Ξ in the LF well-typedness criteria effectively equips the logic with *dependent sorts*. We will often refer to the syntactic objects that occur in assertion logic proofs as *embedded* objects and judgments.

The tacit assumption that meta-variables can occur anywhere in objects means that we may, for example, form "expressions" of the form $e_1\ \alpha$, where α is a meta-variable standing for an expression. The translation to LF terms is still well-defined by extending all translations with the LF translation defined for meta-variables. When meta-variables occur in derivations, they are effectively standing for hypothetical derivations.

As a result of the substitution property of LF (Proposition 2.1), we can easily show the following property:

Proposition 3.8 (Substitution of well-formed encodings)**.** *Assume $\mathcal{S} :: \Xi|\Delta \vdash^{c}_{\Sigma} C$. We then have the following:*

1. *Suppose $\Xi = \Xi_1, \alpha : \mathsf{Exp}, \Xi_2$. For any expression e, if $\ulcorner \Xi_1 \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner e \urcorner \Leftarrow \mathtt{exp}$, then there is a derivation $\mathcal{S}' :: \Xi_1, \Xi_2[e/\alpha]|\Delta[e/\alpha] \vdash^{c}_{\Sigma} C[e/\alpha]$.*

2. *Suppose $\Xi = \Xi_1, \alpha : e \searrow v, \Xi_2$. For any derivation $\mathcal{E}$, if $\ulcorner \Xi_1 \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathcal{E} \urcorner \Leftarrow \ulcorner e \searrow v \urcorner$, then there is a derivation $\mathcal{S}' :: \Xi_1, \Xi_2|\Delta \vdash^{c}_{\Sigma} C$.*

Allowance of cut:	c	::	Allow	::=	$\mathsf{cut} \mid \mathsf{cf}$
Metavariables:	α				
Formulas:	$A, B, \ldots$	::	Form	::=	$\top \mid \forall \alpha : \mathsf{Exp}.\, A \mid \exists \alpha : \mathsf{Exp}.\, A$
					$\mid \exists \alpha : e \searrow v.\, A$
					$\mid A \vee B \mid A \wedge B \mid A \supset B$
Parameters:	Ξ	::	Parms	::=	$\cdot \mid \Xi, \alpha : \mathsf{Exp} \mid \Xi, \alpha : e \searrow v$
Assumptions:	Δ	::	Assm	::=	$\cdot \mid \Delta, A$
Proof judgment:	$\mathcal{S}$	::	$\boxed{\Xi \mid \Delta \vdash^{c}_{\Sigma} A}$		

Initial sequent and cut:

$$\mathsf{ax}: \frac{}{\Xi \mid \Delta, A \vdash^{c}_{\Sigma} A} \qquad \mathsf{cut}: \frac{\Xi \mid \Delta \vdash^{c}_{\Sigma} A \quad \Xi \mid \Delta, A \vdash^{c}_{\Sigma} C}{\Xi \mid \Delta \vdash^{\mathsf{cut}}_{\Sigma} C}$$

Right rules:

$$\mathsf{topR}: \frac{}{\Xi \mid \Delta \vdash^{c}_{\Sigma} \top} \qquad \mathsf{impR}: \frac{\Xi \mid \Delta, A \vdash^{c}_{\Sigma} B}{\Delta \vdash^{c}_{\Sigma} A \supset B} \qquad \mathsf{andR}: \frac{\Xi \mid \Delta \vdash^{c}_{\Sigma} A \quad \Xi \mid \Delta \vdash^{c}_{\Sigma} B}{\Xi \mid \Delta \vdash^{c}_{\Sigma} A \wedge B}$$

$$\mathsf{orR1}: \frac{\Xi \mid \Delta \vdash^{c}_{\Sigma} A}{\Xi \mid \Delta \vdash^{c}_{\Sigma} A \vee B} \qquad \mathsf{orR2}: \frac{\Xi \mid \Delta \vdash^{c}_{\Sigma} B}{\Xi \mid \Delta \vdash^{c}_{\Sigma} A \vee B} \qquad \mathsf{alleR}: \frac{\Xi, \alpha : \mathsf{Exp} \mid \Delta \vdash^{c}_{\Sigma} A}{\Xi \mid \Delta \vdash^{c}_{\Sigma} \forall \alpha : \mathsf{Exp}.\, A}$$

$$\mathsf{exidR}: \frac{\Xi \mid \Delta \vdash^{c}_{\Sigma} A[\mathcal{E}/\alpha] \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathcal{E} \urcorner \Leftarrow \ulcorner e \searrow v \urcorner}{\Xi \mid \Delta \vdash^{c}_{\Sigma} \exists \alpha : e \searrow v.\, A}$$

$$\mathsf{exieR}: \frac{\Xi \mid \Delta \vdash^{c}_{\Sigma} A[e/\alpha] \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner e \urcorner \Leftarrow \ulcorner \mathsf{Exp} \urcorner}{\Xi \mid \Delta \vdash^{c}_{\Sigma} \exists \alpha : \mathsf{Exp}.\, A}$$

Left rules:

$$\mathsf{andL1}: \frac{\Xi \mid \Delta, A \wedge B, A \vdash^{c}_{\Sigma} C}{\Xi \mid \Delta, A \wedge B \vdash^{c}_{\Sigma} C} \qquad \mathsf{andL2}: \frac{\Xi \mid \Delta, A \wedge B, B \vdash^{c}_{\Sigma} C}{\Xi \mid \Delta, A \wedge B \vdash^{c}_{\Sigma} C}$$

$$\mathsf{orL}: \frac{\Xi \mid \Delta, A \vee B, A \vdash^{c}_{\Sigma} C \quad \Xi \mid \Delta, A \vee B, B \vdash^{c}_{\Sigma} C}{\Xi \mid \Delta, A \vee B \vdash^{c}_{\Sigma} C}$$

$$\mathsf{impL}: \frac{\Xi \mid \Delta, A \supset B \vdash^{c}_{\Sigma} A \quad \Xi \mid \Delta, A \supset B, B \vdash^{c}_{\Sigma} C}{\Xi \mid \Delta, A \supset B \vdash^{c}_{\Sigma} C}$$

$$\mathsf{alleL}: \frac{\Xi \mid \Delta, \forall \alpha : \mathsf{Exp}.\, A, A[e/\alpha] \vdash^{c}_{\Sigma} C \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner e \urcorner \Leftarrow \ulcorner \mathsf{Exp} \urcorner}{\Xi \mid \Delta, \forall \alpha : \mathsf{Exp}.\, A \vdash^{c}_{\Sigma} C}$$

$$\mathsf{exieL}: \frac{\Xi, \alpha' : \mathsf{Exp} \mid \Delta, \exists \alpha : \mathsf{Exp}.\, A, A[\alpha'/\alpha] \vdash^{c}_{\Sigma} C}{\Xi \mid \Delta, \exists \alpha : \mathsf{Exp}.\, A \vdash^{c}_{\Sigma} C}$$

$$\mathsf{exidL}: \frac{\Xi, \alpha' : e \searrow v \mid \Delta, \exists \alpha : e \searrow v.\, A, A[\alpha'/\alpha] \vdash^{c}_{\Sigma} C}{\Xi \mid \Delta, \exists \alpha : e \searrow v.\, A \vdash^{c}_{\Sigma} C}$$

Meta-variable and parameter encoding (assuming an LF encoding $\ulcorner \cdot \urcorner$ for each sort):

$$\begin{array}{lcl} \ulcorner \alpha \urcorner & = & \mathtt{x}_{\alpha} \\ \\ \ulcorner \cdot \urcorner & = & \cdot \\ \ulcorner \Xi, \alpha : \mathsf{Exp} \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_{\alpha} : \mathtt{exp} \\ \ulcorner \Xi, \alpha : e \searrow v \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_{\alpha} : \mathtt{eval} \sim \ulcorner e \urcorner \ulcorner v \urcorner \end{array}$$

Figure 3.4: *The assertion logic presented as a sequent calculus.*

Proof sketch. By induction on $\mathcal{S}$. In the cases for rules exidR,exieR and alleL, we appeal to Proposition 2.1. □

The choice of embedding evaluation derivations as dependent sorts is different from the method originally presented in [SS08], where judgments were encoded as atomic propositions by effectively duplicating the derivation rules of the judgment as explicit right-rules. In the presentation we use here, the separation of concerns is more apparent, as the rules of the embedded judgments do not affect the presentation of the assertion logic.

Note that we have not specified any explicit structural rules. It can easily be proven that weakening and contraction for assumptions Δ is admissible from the system given in Figure 3.4. Weakening follows from the fact that extra assumptions can always be added at the inital sequents; contraction follows by observing that left-rules need only use one of two identical assumptions in the context. We do, however, assume the presence of an implicit exchange rule for assumptions. Parameter contexts Ξ are used exclusively to specify the structure of LF contexts, and hence have the structural properties that are admissible for its encoding $\ulcorner \Xi \urcorner$ in LF.

The rules of the system are given as a *sequent calculus*, rather than as a natural deduction system. The difference is subtle, but important: In a sequent calculus, all rules concerned with a particular logical connective can be classified as either a *left-rule* or a *right-rule*. Left-rules act on the assumptions of sequents, but not the conclusions, whereas right rules act on the conclusions, but not the assumptions (they may, however, add variables to the list of parameters, as is the case for allR.) The rules ax and cut fit in neither category; the first rule restricts initial sequents to only conclude formulas from the list of assumptions, while the cut-rule allows the proof of a lemma to be "cut into" the proof of some result. Note that the cut formula A occurs neither in the assumptions nor in the conclusion of the resulting sequent, but is "internal" to the rule.

The parameter c on a proof sequent determines whether the sequent has been derived using the cut rule or not: By inspection of the rules, we can see that there is no way of deriving a proof of the form $\Xi|\Delta \vdash^{\mathsf{cf}}_{\Sigma} A$ using the cut rule. Derivations of this form have the *subformula property*, meaning that all propositions occurring in the derivation are subformulas of A.

Cut-free sequent derivations of formulas only using the connectives $\forall\exists\wedge\vee$ are additionally *right-normal*, meaning that they only consist of right-rules. If we have a cut-free proof of a formula containing an existential quantifier, we know that there is exactly one corresponding instance of the introduction rule, together with an associated witness of the given sort. As an example, suppose that we are given a sequent derivation

$$\mathcal{S} :: \cdot|\cdot \vdash^{\mathsf{cf}}_{\Sigma} \forall \alpha : \mathsf{Exp}.\, \exists \alpha' : (\lambda x.\, v)\ \alpha \searrow v.\, \top$$

for some closed expression v. The only possible form of $\mathcal{S}$ is

$$\dfrac{\dfrac{}{\alpha : \mathsf{Exp} \mid \cdot \vdash^{\mathsf{cf}}_{\Sigma} \top}\ \mathsf{topR} \quad \mathtt{x}_\alpha : \mathtt{exp} \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathcal{E} \urcorner \Leftarrow \ulcorner (\lambda x.\, v)\ \alpha \searrow v \urcorner \quad \mathcal{L}^m}{\alpha : \mathsf{Exp} \mid \cdot \vdash^{\mathsf{cf}}_{\Sigma} \exists \alpha' : (\lambda x.\, v)\ \alpha \searrow v.\, \top}\ \mathsf{exidR}}{\cdot \mid \cdot \vdash^{\mathsf{cf}}_{\Sigma} \forall \alpha : \mathsf{Exp}.\, \exists \alpha' : (\lambda x.\, v)\ \alpha \searrow v.\, \top}\ \mathsf{alleR}$$

The assertion logic proof thus implies the existence of a derivation $\mathcal{E}$. We do not know whether this derivation is well-formed at all, but we are given a proof $\mathcal{L}^m$ saying that its *LF encoding* is well-typed in the context $\mathtt{x} : \ulcorner \mathsf{Exp} \urcorner$. Since we have assumed that Σ represents an *adequate* LF representation of our system, then $\ulcorner \cdot \urcorner$ is compositional with respect to substitution, implying that if we have a closed expression $e :: \mathsf{Exp}$ then also $\cdot \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner e \urcorner \Leftarrow \ulcorner \mathsf{Exp} \urcorner$, and hence

$$\cdot \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathcal{E} \urcorner [\ulcorner e \urcorner / \mathtt{x}_\alpha] \Leftarrow \ulcorner (\lambda x.\, v)\ \alpha \searrow v \urcorner [\ulcorner e \urcorner / \mathtt{x}_\alpha],$$

or, equivalently

$$\cdot \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathcal{E} \urcorner [\ulcorner e \urcorner / \mathtt{x}_\alpha] \Leftarrow \ulcorner (\lambda x.\, v)\ e \searrow v \urcorner.$$

But since the LF representation is adequate, this also implies that there exists a derivation of $(\lambda x.\, v)\ e \searrow v$. We thus rely on the the LF formation judgments and adequacy to ensure that we only extract well-formed derivations from the cut-free fragment of the assertion logic.

A sequent derivation of an implication is not necessarily right-normal, since impR introduces a hypothesis to the context. The cut rule can be used to "apply" an implication proof to a proof of its premises until we end up with a proof of an implication-free formula, which as we have seen admits extraction of relevant witnesses. If we can show that cut is admissible for the cut-free fragment of the logic, we can execute our proofs and extract the witnesses we are interested in. In the context of Twelf, this means that we can verify, on the meta-level, that our assertion logic proofs actually proves the existence of some judgment derivation. We will return to cut elimination shortly in the next subsection.

The Twelf representation of the assertion logic can be seen in Figure 3.5. Parameters can be represented by reusing the LF context, and hence require no explicit encoding. Since left rules can operate on assumptions, assumptions cannot be represented as hypothetical derivations in the LF context though. We therefore have to restrict their use by declaring a separate uninhabited type family hyp, to ensure that any occurrence must be an assumption. The structure of the parameters Ξ, expressions e and derivations $\mathcal{E}$ is never needed in the rules, only their LF translations. We can therefore directly represent parameters by reusing the LF context. The well-formedness criteria, that the concrete objects in the logic must be well-formed according to their sort modulo parameters, is implicitly represented because we reuse the LF context to encode parameter contexts Ξ.

```
% Formulas
form : type. %name form F.
top : form.
/\ : form -> form -> form.
%infix left 4 /\.

\/ : form -> form -> form.
%infix left 3 \/.

==> : form -> form -> form.
%infix right 2 ==>.

foralle : (exp -> form) -> form.
existse : (exp -> form) -> form.
existsev : (eval~ E V -> form) -> form.

% Allowance of cut
allow : type.
cutful : allow.
cutfree : allow.

% Proof judgment
conc : allow -> form -> type.

% Hypotheses
hyp : form -> type.

% Proof rules
ax : hyp A -> conc V A.
cut : conc V A -> (hyp A -> conc V C)
   -> conc cutful C.
```

```
% Proof rules, cont'd
topr : conc V top.
andr : conc V F -> conc V G
        -> conc V (F /\ G).
andl1 : (hyp F -> conc V C)
         -> (hyp (F /\ G) -> conc V C).
andl2 : (hyp G -> conc V C)
         -> (hyp (F /\ G) -> conc V C).
impr : (hyp F -> conc V G)
    -> conc V (F ==> G).
impl : conc V F -> (hyp G -> conc V C)
    -> (hyp (F ==> G) -> conc V C).
orr1 : conc V F
    -> conc V (F \/ G).
orr2 : conc V G
        -> conc V (F \/ G).
orl : (hyp F -> conc V C)
   -> (hyp G -> conc V C)
   -> (hyp (F \/ G) -> conc V C).
foraller : ({x:exp} conc V (C x))
        -> conc V (foralle C).
forallel : {x:exp}
           (hyp (F x) -> conc V C)
         -> (hyp (foralle F)
              -> conc V C).
existser : {x:exp} conc V (F x)
        -> conc V (existse F).
existsel : ({x:exp} hyp (F x)
             -> conc V C)
         -> (hyp (existse F)
             -> conc V C).
existsevr : {x:eval~ E V'} conc V (F x)
         -> conc V (existsev F).
existsevl : ({x:eval~ E V'}
             hyp (F x) -> conc V C)
         -> (hyp (existsev F)
             -> conc V C).
```

Figure 3.5: *Twelf signature for the assertion logic.*

3.2.2 Cut elimination

Cut elimination follows as a result of cut admissibility of the cut-free sequent calculus. The original proof is due to Gentzen [Gen35, Gen65] and has previously been formalized in Twelf by Pfenning [Pfe00]. Both assume that the logic is single-sorted with no dependencies, but the proof turns out to generalize to the dependent case as well. We will not describe it in detail here, but will just present the general strategy which follows Pfenning:

Theorem 3.9 (Cut admissibility)**.** *If* $\mathcal{S}_1 :: \Xi|\Delta \vdash^{\mathsf{cf}} A$ *and* $\mathcal{S}_2 :: \Xi|\Delta, A \vdash^{\mathsf{cf}} C$ *then also* $\Xi|\Delta \vdash^{\mathsf{cf}} C$*.*

Proof sketch. By nested structural induction on the cut formula A and the derivations $\mathcal{S}_1, \mathcal{S}_2$. We proceed by cases on $\mathcal{S}_1, \mathcal{S}_2$, and classify the possible cases in three categories:

Essential cases. When $\mathcal{S}_1$ ends in a left-rule and $\mathcal{S}_2$ ends in a right-rule for the same logical connective. This is where the actual normalization happens, since we eliminate the left-rule from $\mathcal{S}_2$ by inserting $\mathcal{S}_1$.

Left commutative cases. When $\mathcal{S}_1$ ends in a left-rule. We cannot eliminate this, since the hypothesis must be in Δ, and hence is also a hypothesis in the resulting proof. We apply the induction hypothesis on all subderivations and reapply the left rule to the results.

Right commutative cases. When $\mathcal{S}_2$ ends in a right-rule or in a left-rule that does not use the conclusion of $\mathcal{S}_1$ as a hypothesis. Again, we apply the induction hypothesis to subderivations and reapply the original rule. □

Cut elimination follows from the above result:

Theorem 3.10 (Cut elimination)**.** *If* $\Xi|\Delta \vdash^{\mathsf{cut}} A$ *then also* $\Xi|\Delta \vdash^{\mathsf{cf}} A$*.*

Proof sketch. By trivial induction on the proof derivation, using Theorem 3.9 in the case for `cut`. □

The cut elimination theorem is what allows us to "run" an assertion logic proof and extract its result. This also means that we can only extend the assertion logic with features that do not invalidate cut admissibility.

The Twelf formalization of the above theorems consists of the meta-theorems `ca` and `ce`, declared as follows:

```
ca : {A} conc cutfree A -> (hyp A -> conc cutfree C) -> conc cutfree C -> type.
%mode ca +A +SP1 +SP2 -SP'.
ce : conc cutful A -> conc cutfree A -> type.
%mode ce +SP* -SP'.

%{ ... proof cases elided ... }%
```

```
%worlds (bhyp | bexp | beval~) (ca _ _ _ _).
%total {A [SP1 SP2]} (ca A SP1 SP2 _).
%worlds (bhyp | bexp | beval~) (ce _ _ ).
%total (SP) (ce SP _).
```

The `%worlds` declarations says that both proofs has to extend the LF context with hypotheses, expressions and evaluation derivations.

3.2.3 Encoding the logical relation

We have now set up the necessary machinery for characterizing our logical relation using only syntactic methods. In the following, we will describe how to formalize the termination proof given in Section 3.1. Since the assertion logic has no notion of induction, all structural induction must occur at the meta-level. The separation between the assertion logic and the meta-level is clear: We cannot do meta-level induction on an object that only exists within the assertion logic. Since the assertion logic proofs at this point are not cut-free and hence not normalized, all objects inside assertion logic proofs are entirely hypothetical. We can, however, quantify over objects at the meta-level and use those objects inside the assertion logic.

Lemma 3.2 (Converse head reduction) and Lemma 3.4 (Iterated reduction) both live entirely the meta level, and are formalized as the following meta-theorems:

```
eval-cvrs : ctx RX
          -> eval (RX E') V
          -> whr E E'
          -> eval (RX E) V -> type.
%mode eval-cvrs +RP +EP +WP -EP'.
%{... proof cases elided ... }%
%worlds () (eval-cvrs _ _ _).
%total (RP) (eval-cvrs RP _ _ _).
```

```
eval~=>eval : eval~ E V
            -> eval E V -> type.
%mode eval~=>eval +EP -EP'.
%{... proof cases elided ...}%
%worlds () (eval~=>eval _ _).
%total (EP) (eval~=>eval EP _).
```

The logical relation is encoded as a function from types to formulas with a single free expression. The function is defined as a relation as follows:

```
lr : tp -> (exp -> form) -> type.
lr/bool : lr bool ([e] existse [v] existsev [ep:eval~ e v] top).
lr/=> : lr (T2 => T0) ([e] (existse [v] existsev [ep:eval~ e v] top)
                           /\ foralle [e2] R2 e2 ==> R0 (app e e2))
          <- lr T0 R0
          <- lr T2 R2.
```

Interestingly, we do not have to add quantifiers for the `whr` family to the assertion logic at all, it lives entirely on the meta-level. Lemma 3.5 (Closure under weak head reduction) can thus be formulated as follows:

```
cwhe : lr T R -> ctx RX -> whr E E'
          -> conc cutful (R (RX E')) -> conc cutful (R (RX E)) -> type.
%mode cwhe +LP +RP +SP +SPR -SP'.

%{ ... proof cases elided ... }%
%worlds (bexp | bconc) (cwhe _ _ _ _ _).
%total (LP) (cwhe LP _ _ _ _).
```

The fundamental theorem works by meta-level induction over the typing derivation, inferring the logical relation from the type:

```
fund : of E T -> lr T R -> conc cutful (R E) -> type.
%mode (fund +OP -LP -SP).
%{ ... proof cases elided ... }%
%block bfund : some {T':tp}{R':exp -> form}{LP':lr T' R'}
                block {x:exp}{op:of x T'}{sp:conc cutful (R' x)}
                {_:fund op LP' sp}.
%worlds (bfund) (fund _ _ _).
%total (OP) (fund OP _ _).
```

Note that in the syntactic formulation of the fundamental theorem, we construct an open derivation using hypothetical derivations instead of extending the set of assumptions in the sequent (by adding `conc`-assumptions to the LF context instead of `hyp`-assumptions). When we are working with the cutful sequent calculus, it does not really matter what approach we use, since a hypothetical derivation can be brought into the set of assumptions by applying the cut rule. The Twelf formalization goes a little smoother this way though.

We finally formulate a top-level meta-theorem for tying everything together. We first show that a cut-free proof of an expression belonging to the logical relation implies that it terminates. Here, we get a witness of an iterated evaluation derivation which we then convert to a big-step derivation on the meta-level:

```
ext : lr T R -> conc cutfree (R E) -> eval E V -> type.
%mode ext +LP +SP -EP.
- : ext lr/bool (existser V (existsevr EP _)) EP'
     <- eval~=>eval EP EP'.
- : ext (lr/=> _ _) (andr (existser V (existsevr EP _)) _) EP'
     <- eval~=>eval EP EP'.
%worlds () (ext _ _ _).
%total (EP) (ext EP _ _).
```

The final lemma invokes the fundamental theorem, cut elimination and extraction in succession. Due to cut elimination, the main theorem that we care about can be formulated without referring to the assertion logic:

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

Expressions: $e, v :: \mathsf{Exp} ::= x \mid e_1\ e_2 \mid \lambda x.\, e_0 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}(e_0,\ e_1,\ e_2)$
Types: $\tau :: \mathsf{Tp} ::= \mathsf{bool} \mid \tau_2 \to \tau_0$
Contexts: $\Gamma :: \mathsf{Ctx} ::= \cdot \mid \Gamma, x : \tau$
Values: $\mathcal{V} :: \boxed{v\ \mathrm{value}}$:

(v_lam,v_true and v_false are defined as before.)

Dynamic semantics: $\mathcal{E} :: \boxed{e \Downarrow v}$ (e closed)

(e_lam, e_app, e_true and e_false are defined as before.)

$$\mathsf{e\_ift}: \frac{e_0 \Downarrow \mathsf{true} \quad e_1 \Downarrow v}{\mathsf{if}(e_0,\ e_1,\ e_2) \Downarrow v} \qquad \mathsf{e\_iff}: \frac{e_0 \Downarrow \mathsf{false} \quad e_2 \Downarrow v}{\mathsf{if}(e_0,\ e_1,\ e_2) \Downarrow v}$$

Static semantics: $\mathcal{T} :: \boxed{\Gamma \vdash e : \tau}$

(t_var, t_lam and t_app, t_true and t_false are defined as before.)

$$\mathsf{t\_if}: \frac{\Gamma \vdash e_0 : \mathsf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}(e_0,\ e_1,\ e_2) : \tau}$$

Figure 3.6: *Syntax and semantics of* $\lambda^{\to,\mathsf{bool}}_{\mathsf{cbn}}$.

```
    <- fund OP LP SP
    <- ce SP SP'
    <- ext LP SP' EP.
%worlds () (term _ _).
%total {} (term _ _).
```

3.3 Adding full booleans

The basic machinery for representing logical relations has now been presented. The next step is to extend our programming language with more features to see how this affects the termination proof and the corresponding Twelf formalization.

In this section, we will see what happens when we extend $\lambda^{\to,\mathsf{bool}^-}_{\mathsf{cbn}}$ with an if-construct. The extended language, called $\lambda^{\to,\mathsf{bool}}_{\mathsf{cbn}}$, builds on the definitions from Figure 3.1 and can be seen in Figure 3.6. We also extend the notion of reduction contexts and weak head reduction in Figure 3.8, as well as the Twelf encoding, which can be seen in Figure 3.7.

We have now added a new language construct, but not any new types. One might therefore think that we can simply use the logical relation from Definition 3.1, but this would not work. Previously, we were satisfied with knowing that an expression

```
% Expressions
if : exp -> exp -> exp -> exp.

% Evaluation
eval/ift : eval E0 true
        -> eval E1 V
        -> eval (if E0 E1 E2) V.
eval/iff : eval E0 false
        -> eval E2 V
        -> eval (if E0 E1 E2) V.
```

```
% Typing
of/if : of E0 bool
     -> of E1 T
     -> of E2 T
     -> of (if E0 E1 E2) T.
```

Figure 3.7: *Twelf signature for $\lambda^{\rightarrow,\mathsf{bool}}_{\mathsf{cbn}}$, extended*

at boolean type *terminated*, but not what its result actually was—we would have been satisfied even if the result was of a different type. It turns out that for termination to go through, we need the logical relation to not just imply the existence of an evaluation, but also that the result of the evaluation is a well-typed value at type bool. This information is critical in the case for if in the fundamental theorem, as a well-typed if-expression only has an evaluation if evaluation preserves well-typedness for the test expression.

Our logical relation is now defined as follows:

Definition 3.11 (Logical termination, booleans)**.**

$$P_{\mathsf{bool}}(e) \Leftrightarrow e \searrow \mathsf{true} \vee e \searrow \mathsf{false},$$
$$P_{\tau_2 \to \tau_0}(e) \Leftrightarrow (\exists v.\, e \searrow v) \wedge \forall e_2.\, P_{\tau_2}(e_2) \Rightarrow P_{\tau_0}(e\ e_2). \qquad \diamondsuit$$

We have chosen a rather verbose definition at base types, essentially giving an extensional definition by listing all the possible outcomes—in this case two. Alternatively, we could have defined that for an expression e to be in the relation at booleans, it should satisfy $\exists v.\, e \searrow v \wedge (\cdot \vdash v : \mathsf{bool})$, which says the same thing. However, the proof case for if in the fundamental theorem depends on knowing the concrete values that v can be. In the paper proof, we can easily infer that these can only be true or false by looking at the possible typing derivations. However, since the assertion logic does not have any way to reason about the possible ways a derivation of $\cdot \vdash v : \mathsf{bool}$ was derived, this alternative definition would not work in the subsequent formalization.

We will need to extend Lemma 3.4 to show that the iterated evaluation judgment remains sound with regards to big-step evaluation after the addition of ev_ctx. To show this, we will need some additional lemmas about values. The proof details are trivial and thus omitted:

Lemma 3.12 (Results are values)**.** *If $e \Downarrow v$, then v* value*.*

Proof sketch. By induction on $\mathcal{E}$. We either get the result directly by the form of the evaluation result, or immediately by IH in the cases for e_app, e_ift and e_iff. □

Reduction context: $\mathcal{R} ::= \circ \mid \mathcal{R}\ e_2 \mid \mathsf{if}(\mathcal{R},\ e_1,\ e_2)$

Weak head reduction: $\mathcal{H} :: \boxed{e \rightsquigarrow e'}$:

(Rule whr_beta is defined as before.)

$$\text{whr\_ift:}\ \frac{}{\mathsf{if}(\mathsf{true},\ e_1,\ e_2) \rightsquigarrow e_1} \qquad \text{whr\_iff:}\ \frac{}{\mathsf{if}(\mathsf{false},\ e_1,\ e_2) \rightsquigarrow e_2}$$

Evaluation by reduction: $\mathcal{E} :: \boxed{e \searrow v}$:

(Rules ev_val and ev_whr are defined as before.)

$$\text{ev\_ctx:}\ \frac{e_0 \searrow v_0 \quad \mathcal{R}\{v_0\} \searrow v}{\mathcal{R}\{e_0\} \searrow v}$$

(a) *Iterated reduction, extended.*

```
% Reduction contexts
ctx/if : ctx R
      -> ctx ([x] if (R x) E1 E2).

% Weak head reduction
whr/ift : whr (if true E1 E2) E1.
whr/iff : whr (if false E1 E2) E2.
```

```
% Iterated reduction
eval~/ctx : ctx RX
         -> eval~ E0 V0
         -> eval~ (RX V0) V
         -> eval~ (RX E0) V.
```

(b) *Extended Twelf representation.*

Figure 3.8: *Evaluation by reduction, extended for* $\lambda^{\rightarrow,\mathsf{bool}}_{\mathsf{cbn}}$.

Lemma 3.13 (Value determinism)**.** *If* v value *and* $v \Downarrow v'$*, then* $v = v'$*.*

Proof sketch. Immediate, by the possible forms of the value derivation. □

Using these results, we can prove that big-step evaluation is closed under converse context reduction:

Lemma 3.14 (Converse evaluation)**.** *If* $\mathcal{E}_0 :: e_0 \Downarrow v_0$ *and* $\mathcal{E}' :: \mathcal{R}\{v_0\} \Downarrow v$*, then there is a derivation of* $\mathcal{R}\{e_0\} \Downarrow v$*.*

Proof. By induction on $\mathcal{R}$.

- Case $\mathcal{R} = \circ$. Then $\mathcal{E}'$ is a derivation of $v_0 \Downarrow v$, and it suffices to show $e_0 \Downarrow v$. By Lemma 3.12 on $\mathcal{E}$, we have v_0 value. This justifies Lemma 3.13 on $\mathcal{E}'$, implying $v = v_0$. But then $\mathcal{E}$ is a derivation of $e_0 \Downarrow v$ already, and we are done.
- Case $\mathcal{R} = \mathcal{R}'\ e_2$. Then $\mathcal{E}'$ is a derivation of $\mathcal{R}'\{v_0\}\ e_2 \Downarrow v$, and it suffices to show that $\mathcal{R}'\{e_0\}\ e_2 \Downarrow v$. $\mathcal{E}'$ must end in e_app, implying that we have $\mathcal{E}'_1 :: \mathcal{R}'\{v_0\} \Downarrow$

$\lambda x.\, e_0'$ and $\mathcal{E}_2' :: e_0'[e_2/x] \Downarrow v$. By IH on $\mathcal{R}'$ with $\mathcal{E}_0$, $\mathcal{E}_1'$, we obtain $\mathcal{E}_1 :: \mathcal{R}'\{e_0\} \Downarrow \lambda x.\, e_0'$. But then we can construct the goal by e_app on $\mathcal{E}_1$ and $\mathcal{E}_2'$.

- Case $\mathcal{R} = \mathsf{if}(\mathcal{R}',\ e_1,\ e_2)$. Then $\mathcal{E}'$ can either end in e_ift or e_iff. In either case, we convert the first subderivation by IH and reapply the original evaluation rule, as in the case above. □

Lemma 3.2 has to be extended with cases for whr_ift and whr_iff as well:

Lemma 3.15 (Converse head reduction)**.** *If $\mathcal{E} :: \mathcal{R}\{e'\} \Downarrow v$ and $\mathcal{H} :: e \rightsquigarrow e'$, then $\mathcal{R}\{e\} \Downarrow v$.*

Proof. By induction on $\mathcal{R}$, extending the proof of Lemma 3.2 for the extra cases.

- Case $\mathcal{R} = \circ$. We handle the new case where $\mathcal{H}$ ends in whr_ift of whr_iff. In both cases we get the result directly by e_ift, e_true or e_iff,e_false on $\mathcal{E}$.

- Case $\mathcal{R} = \mathsf{if}(\mathcal{R}',\ e_1,\ e_2)$. We see that $\mathcal{E}$ must end in e_ift of e_iff. In both cases, we apply IH on the first subderivation and reapplies the evaluation rule on the result. □

We extend Lemma 3.4 to cover the extra case introduced by the addition of ev_ctx:

Lemma 3.16 (Soundness of iterated reduction)**.** *If $\mathcal{E} :: e \searrow v$, then $e \Downarrow v$.*

Proof. Extension of the the proof of Lemma 3.4, handling the case for ev_ctx by IH followed by Lemma 3.14. The rest of the proof proceeds as before, but appealing to Lemma 3.15 where we used Lemma 3.2 before. □

This concludes the extension of the lemmas concerned with showing that our alternative evaluation judgment is sound. The remaining results are properties of the logical relation.

Lemma 3.5 needs to be changed to accommodate the new definition of the logical relation at base types:

Lemma 3.17 (Closure under weak head expansion)**.** *If $P_\tau(e')$ and $\mathcal{H} :: e \rightsquigarrow e'$, then $P_\tau(e)$.*

Proof. By induction on τ.

- Case $\tau = \mathsf{bool}$. We either have $e \searrow \mathsf{true}$ or $e \searrow \mathsf{false}$ by assumption. In both cases we apply ev_cvrs on $\mathcal{H}$ and the derivation for e, and we are done.

- The case for $\tau = \tau_2 \to \tau_0$ is identical to the proof of Lemma 3.5. □

Additionally, the logical relation is also closed under converse context reduction:

Lemma 3.18 (Closure under converse context reduction)**.**
If $\mathcal{E} :: e_0 \searrow v_0$ and $a :: P_\tau(\mathcal{R}\{v_0\})$ then $P_\tau(\mathcal{R}\{e_0\})$.

Proof. By induction on τ.

- Case $\tau = \mathsf{bool}$. We either have $\mathcal{R}\{v_0\} \searrow \mathsf{true}$ or $\mathcal{R}\{v_0\} \searrow \mathsf{false}$. In each case, the result follows immediately by ev_ctx and $\mathcal{E}$.
- Case $\tau = \tau_2 \to \tau_0$. By assumption, we have $\mathcal{R}\{v_0\} \searrow v$ for some v, and so we get $\mathcal{R}\{e_0\} \searrow v$ by ev_ctx and $\mathcal{E}$. It remains to show that for any e_2, if $h :: P_{\tau_2}(e_2)$, then also $P_{\tau_0}(\mathcal{R}\{e_0\}\ e_2)$. By a on h, we get $P_{\tau_0}(\mathcal{R}\{v_0\}\ e_2)$, and hence by IH on τ_0 and $\mathcal{E}$, we are done. □

We have now established the necessary lemmas needed for proving the additional case in Theorem 3.6; the cases for t_if. The remaining cases remain unchanged:

Theorem 3.19 (Fundamental theorem)**.** *For any expression e, if $\mathcal{T} :: \Gamma \vdash e : \tau$, then for any substitution γ where $P_\Gamma(\gamma)$, it holds that $P_\tau(\hat{\gamma}(e))$.*

Proof. By induction on $\mathcal{T}$.

- Case $\mathcal{T}$ ends in t_if. So $e = \mathsf{if}(e_0,\ e_1,\ e_2)$, and we have typing derivations $\mathcal{T}_0 :: e_0 : \Gamma \vdash \mathsf{bool}$, $\mathcal{T}_1 :: \Gamma \vdash e_1 : \tau$ and $\mathcal{T}_2 :: \Gamma \vdash e_2 : \tau$. By IH on each subderivation, we obtain $\mathcal{E}_0 :: \hat{\gamma}(e_0) \searrow \mathsf{true} \vee \hat{\gamma}(e_0) \searrow \mathsf{false}$, $h_1 :: P_\tau(\hat{\gamma}(e_1))$ and $h_2 :: P_\tau(\hat{\gamma}(e_2))$.

  In the case where we have $\mathcal{E}_0 :: \hat{\gamma}(e_0) \searrow \mathsf{true}$, we apply Lemma 3.17 on h_1 and whr_ift, and obtain $h_1' :: P_\tau(\mathsf{if}(\mathsf{true},\ e_1,\ e_2))$. By Lemma 3.18 on $\mathcal{E}_0$ and h_1', we are done.

  The case where we have $\hat{\gamma}(e_0) \searrow \mathsf{false}$ is analogous.
- The cases for t_app, t_lam, t_true and t_false are the same as in the proof of Theorem 3.6, but using Lemma 3.17 for all uses of Lemma 3.5. □

3.3.1 Extending the formalization

The formalization of the extended proof is relatively straightforward. The assertion logic does not need to be extended, and remains the same. As mentioned earlier, we did however make some choices about the structure of the proof in order to make the formalization go smoother.

The most important choice to point out is the inclusion of the rule ev_ctx in the judgment $\boxed{e \searrow v}$. This rule is actually *admissible*, in the sense that it could have been replaced by a lemma proving the conclusion from the premises. This approach would have been perfectly fine if we only cared about getting the proof to go through on paper. However, the property is needed in the proof of Lemma 3.18, which is formalized as the following Twelf meta-theorem (only one proof case shown):

```
lr-ctxred : lr T R -> ctx RX
          -> eval~ E0 V0 -> conc cutful (R (RX V0)) -> conc cutful (R (RX E0))
          -> type.
```

```
%mode lr-ctxred +LP +RP +EP +SP -SP'.
- : lr-ctxred lr/bool RP EP SP
      (cut SP
      (orl
        (existsevl [E'=>true][_]
          orr1 (existsevr (eval~/ctx RP EP E'=>true) topr))
        (existsevl [E'=>false][_]
          orr2 (existsevr (eval~/ctx RP EP E'=>false) topr)))).
%{ ... proof case for function type elided ... }%
%worlds (bexp | bconc | beval~) (lr-ctxred _ _ _ _ _).
%total (LP) (lr-ctxred LP _ _ _ _).
```

In the proof of the theorem, we apply ev_ctx (`eval~/ctx` in the above) to an evaluation derivation that is obtained from *within* the assertion logic proof. If we were to replace the rule with a lemma, that lemma would therefore have to be provable for evaluation derivations quantified over inside the assertion logic. We could prove it by meta-level induction over the reduction context, but the proof of the lemma would also require some form of case analysis on the evaluation derivation. This case analysis can *not* occur on the meta-level, and the assertion logic does not provide such a reasoning principle.

We get around the problem by explicitly axiomatizing the property through the ev_ctx rule, effectively postponing the proof of the lemma until after we have "escaped" from the assertion logic via cut elimination.

3.4 Infinite value domains

Up until now, the value domain at base types has been finite. We will now see what happens when we extend $\lambda_{\mathsf{cbn}}^{\rightarrow,\mathsf{bool}}$ with natural numbers and a case construct. The language, $\lambda_{\mathsf{cbn}}^{\rightarrow,\mathsf{bool},\mathsf{nat}}$, is an extension of the definition in Figure 3.6, and is defined in Figure 3.9. We must also extend the definition of reduction contexts and the notion of weak head reduction, as defined in Figure 3.11, as well as the Twelf encoding, defined in Figure 3.10.

The definition of the logical relation will have to be extended for the added base type, such that $P_{\mathsf{nat}}(e)$ entails that e evaluates to a well-typed value with type nat. We therefore need to characterize when a value is a proper numeral. Unlike the situation we had for booleans, we cannot just define this property as the disjunction of all the possible outcomes, as there are infinitely many: To specify that an expression is a numeral would thus require that we add recursive formulas to the assertion logic. Such an extension would have non-trivial implications for cut elimination, though, due to "unfolding" of recursively defined formulas.

We therefore have to give a more intensional definition at type nat, by just specifying that the result is *some* numeral. We will later see how to handle this in the formalization by adding case analysis on natural numbers to the assertion logic.

The logical relation for the extended language is defined as follows:

Natural numbers:	n	::	Nat	::=	$\mathsf{z} \mid \mathsf{s}(n)$
Expressions:	e, v	::	Exp	::=	$x \mid e_1\ e_2 \mid \lambda x.\, e_0 \mid \mathsf{true} \mid \mathsf{false} \mid \overline{n}$
					$\mid \mathsf{if}(e_0,\ e_1,\ e_2) \mid \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2)$
Types:	τ	::	Tp	::=	$\mathsf{bool} \mid \mathsf{nat} \mid \tau_2 \to \tau_0$
Contexts:	Γ	::	Ctx	::=	$\cdot \mid \Gamma, x : \tau$
Values:	$\mathcal{V}$	::	$\boxed{v\ \mathsf{value}}$:		

(v_lam, v_true and v_false are defined as before.)

$$\text{v\_num: } \frac{}{\overline{n}\ \mathsf{value}}$$

Dynamic semantics: $\mathcal{E}$ :: $\boxed{e \Downarrow v}$ (e closed)

(e_lam, e_app, e_true,e_false, e_ift and e_iff are defined as before.)

$$\text{e\_num: } \frac{}{\overline{n} \Downarrow \overline{n}}$$

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

Static semantics: $\mathcal{T}$ :: $\boxed{\Gamma \vdash e : \tau}$

(t_var, t_lam and t_app, t_true, t_false and t_if are defined as before.)

$$\text{t\_num: } \frac{}{\Gamma \vdash \overline{n} : \mathsf{nat}} \qquad \text{t\_case: } \frac{\Gamma \vdash e_0 : \mathsf{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2)}$$

Figure 3.9: *Syntax and semantics of* $\lambda^{\to,\mathsf{bool},\mathsf{nat}}_{\mathsf{cbn}}$.

Definition 3.20 (Logical termination, naturals)**.**

$$\begin{aligned} P_{\mathsf{bool}}(e) &\Leftrightarrow e \searrow \mathsf{true} \vee e \searrow \mathsf{false}, \\ P_{\mathsf{nat}}(e) &\Leftrightarrow \exists n.\, e \searrow \overline{n}, \\ P_{\tau_2 \to \tau_0}(e) &\Leftrightarrow (\exists v.\, e \searrow v) \wedge \forall e_2.\, P_{\tau_2}(e_2) \Rightarrow P_{\tau_0}(e\ e_2). \end{aligned}$$ ♢

We introduce no new concepts to the proof otherwise, but extend the existing lemmas to cover the new cases that we have introduced. The proofs for the new cases in Lemma 3.14 (Converse evaluation), Lemma 3.15 (Converse head reduction) and Lemma 3.16 (Soundness of iterated reduction) are all analogous to the cases for the if-construct, and hence we will not cover them here. Lemma 3.17 (Closure under weak head expansion) and Lemma 3.18 has new cases for $\tau = \mathsf{nat}$, but these are immediate by an argument analogous to the cases for $\tau = \mathsf{bool}$.

What remains is to extend the fundamental theorem with the new case for t_case:

```
% Natural numbers
nat : type.
z : nat.
s : nat -> nat.

% Types
nat' : tp.

% Expressions
num : nat -> exp.
case : exp
    -> exp
    -> (exp -> exp) -> exp.

% Values
val/num : val (num N).
```

```
% Evaluation
eval/num : eval (num N) (num N).
eval/case0 : eval E0 (num z)
          -> eval E1 V
          -> eval (case E0 E1 E2) V.
eval/case1 : eval E0 (num (s N))
          -> eval (E2 (num N)) V
          -> eval (case E0 E1 E2) V.

% Typing
of/num : of (num N) nat'.
of/case : of E0 nat'
       -> of E1 T
       -> ({x} of x nat'
           -> of (E2 x) T)
       -> of (case E0 E1 E2) T.
```

Figure 3.10: *Twelf signature for $\lambda^{\rightarrow,\mathsf{bool},\mathsf{nat}}_{\mathsf{cbn}}$, extended*

Theorem 3.21 (Fundamental theorem)**.** *For any expression e, if $\mathcal{T} :: \Gamma \vdash e : \tau$, then for any substitution γ where $P_\Gamma(\gamma)$, it holds that $P_\tau(\hat{\gamma}(e))$.*

Proof. By induction on $\mathcal{T}$.

- Case $\mathcal{T}$ ends in t_case. So $e = \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2)$, and we have typing derivations $\mathcal{T}_0 :: \Gamma \vdash e_0 : \mathsf{nat}$, $\mathcal{T}_1 :: \Gamma \vdash e_1 : \tau$ and $\mathcal{T}_2 :: \Gamma, x : \mathsf{nat} \vdash e_2 : \tau$. By IH on $\mathcal{T}_0$, we obtain $\mathcal{E}_0 :: \hat{\gamma}(e_0) \searrow \overline{n}$ (for some n), and by IH on $\mathcal{T}_1$, we obtain $h_1 :: P_\tau(\hat{\gamma}(e_1))$.

  We proceed by cases on n:

  - Subcase $n = \mathsf{z}$: By Lemma 3.17 on h_1 and whr_case0, we obtain

    $$h_1' :: P_\tau(\mathsf{ncase}(\overline{\mathsf{z}},\ e_1,\ x.\, e_2)),$$

    and by Lemma 3.14 on h_1' and $\mathcal{E}_0$, we are done (justified by $n = \mathsf{z}$).
  - Subcase $n = \mathsf{s}(n')$: We construct a substitution $\gamma_0 = \gamma[x \mapsto \overline{n'}]$, and observe that we trivially have $P_{\mathsf{nat}}(\overline{n'})$ by ev_val and val_num, implying $P_{\Gamma,x:\mathsf{nat}}(\gamma_0)$. But then by IH on $\mathcal{T}_2$ with γ_0, we obtain and $h_2 :: P_\tau(\hat{\gamma_0}(e_2))$, or, equivalently, $h_2 :: P_\tau(\hat{\gamma}(e_2)[\overline{n}/x])$. By Lemma 3.17 on h_2 and whr_case1, we obtain

    $$h_2' :: P_\tau(\mathsf{ncase}(\overline{\mathsf{s}(n')},\ e_1,\ x.\, e_2)),$$

    and by Lemma 3.14 on h_2' and $\mathcal{E}_0$, we are done (justified by $n = \mathsf{s}(n')$).

- The cases for t_app, t_lam, t_true, t_false and t_if are the same as in the proofs of Theorem 3.6 and Theorem 3.19. □

Reduction context: $\mathcal{R} ::= \circ \mid \mathcal{R}\ e_2 \mid \mathsf{if}(\mathcal{R},\ e_1,\ e_2) \mid \mathsf{ncase}(\mathcal{R},\ e_1,\ x.e_2)$

Weak head reduction: $\mathcal{H} :: \boxed{e \rightsquigarrow e'}$:

(Rules whr_beta, whr_ift and whr_iff are defined as before.)

whr_case0: $\dfrac{}{\mathsf{ncase}(\overline{z},\ e_1,\ x.e_2) \rightsquigarrow e_1}$ whr_case1: $\dfrac{}{\mathsf{ncase}(\overline{\mathsf{s}(n)},\ e_1,\ x.e_2) \rightsquigarrow e_2[\overline{n}/x]}$

Evaluation by reduction: $\mathcal{E} :: \boxed{e \searrow v}$:

(Rules ev_val, ev_whr and ev_ctx are defined as before.)

(a) *Evaluation by reduction, rule extensions*

```
% Reduction contexts
ctx/case : ctx R
        -> ctx ([x] case (R x) E1 E2).
```

```
% Weak head reduction
whr/case0 :
    whr (case (num z) E1 E2) E1.
whr/case1 :
    whr (case (num (s N)) E1 E2)
        (E2 (num N)).
```

(b) *Extended Twelf representation.*

Figure 3.11: *Evaluation by reduction, extended for* $\lambda^{\rightarrow,\mathsf{bool},\mathsf{nat}}_{\mathsf{cbn}}$.

The Twelf formalization turns out not to be so straightforward, as it requires new features to be added to the assertion logic. We will describe these extensions in the next section.

3.5 Case analysis

In the Twelf formalization, all proofs except the fundamental theorem can be straightforwardly extended without any problems. Several issues arise when we come to the formalization of the extra case in the fundamental theorem, though. In this proof, we obtain by induction a proof of $\hat{\gamma}(e_0) \searrow \overline{n}$ for some number n which is bound at an existential quantifier in the definition of the logical relation. This means that in the Twelf formalization, this number is a hypothetical number that is quantified over inside the assertion logic. This is problematic, as the proof continues by subcases over the possible forms of n, which we have no way of doing in assertion logic proofs.

It turns out that we need to extend the assertion logic to make it more expressive. Specifically, we need to be able to "unfold" the structure of natural numbers and continue reasoning from the possible outcomes. The extended logic can be seen in Figure 3.12.

The extension of the Twelf encoding can be seen in Figure 3.13. We add Nat as a new sort, as well as a unary predicate Nat^+ on natural numbers; we will get back to this in a moment. We also add the auxiliary judgment $\boxed{e \doteq e'}$, having only a single rule:

$$\mathsf{q\_id}: \frac{}{e \doteq e}$$

The judgment is encoded by the following type family:

```
eq-exp : exp -> exp -> type.
eq-exp/id : eq-exp E E.
```

The judgment is used to encode equality as explicit proofs, which will be needed when we add case analysis. Explicit equality proofs are necessary to be able reason about equality in the assertion logic, in contrast to meta-level proofs where equality is often implicit via unification of meta-variables.

But how do we apply such a proof in the assertion logic? We cannot pattern match on `eq-exp/id` and rely on unification, but must intead add an explicit conversion axiom to the encoding of the $\boxed{e \searrow v}$ judgment:

$$\mathsf{ev\_conv}: \frac{e \doteq e' \quad v \doteq v' \quad e \searrow v}{e' \searrow v'}$$

```
eval~/conv : eq-exp E E' -> eq-exp V V' -> eval~ E V -> eval~ E' V'.
```

The rule is trivially admissible, and the extra case needed in the proof of Lemma 3.16 is immediate. But now the admissibility proof has been postponed until *after* we have extracted an evaluation derivation from a cut-free proof, like we did with the `eval~/ctx` rule in the last section, admitting equality conversions of iterated evaluation derivations to be used inside assertion logic proofs.

It remains to find a way to reason about the possible forms of a number n, obtaining equality proofs for each possible case. Specifically, we would like to have a principle in the assertion logic that says that for any n, we can conclude C if either

1. Given $\overline{n} \doteq \overline{\mathsf{z}}$, we can prove C, or
2. Given some n' and $\overline{n} \doteq \overline{\mathsf{s}(n')}$, we can prove C.

It is a bit tricky to add such reasoning principles to the assertion logic while preserving cut admissibility. A starting point for our solution is [MM00], which describes the more general problem of adding induction and definitions to a sequent calculus.

It turns out that we need to add an atomic formula, Nat^+, which is a unary predicate for some natural number n. For each constructor for the sort Nat, we add a corresponding right-rule to the assertion logic, reflected by the addition of the rules natR_z and natR_s. $\mathsf{Nat}^+(n)$ can thus be viewed as the statement that "n is well-formed", i.e., it is constructed

Allowance of cut:	c	::	Allow	::=	$\mathsf{cut} \mid \mathsf{cf}$
Metavariables:	α				
Formulas:	$A, B, \ldots$	::	Form	::=	$\top \mid \forall\alpha : \mathsf{Exp}.\, A \mid \exists\alpha : \mathsf{Exp}.\, A$
					$\mid \exists\alpha : e \searrow v.\, A$
					$\mid A \vee B \mid A \wedge B \mid A \supset B$
					$\mid \exists\alpha : \mathsf{Nat} \mid \mathsf{Nat}^+(n)$
Parameters:	Ξ	::	Parms	::=	$\cdot \mid \Xi, \alpha : \mathsf{Exp} \mid \Xi, \alpha : e \searrow v$
					$\mid \Xi, \alpha : e \doteq e' \mid \Xi, \alpha : \mathsf{Nat}$
Assumptions:	Δ	::	Assm	::=	$\cdot \mid \Delta, A$
Proof judgment:	$\mathcal{S}$	::	$\boxed{\Xi \mid \Delta \vdash^c_\Sigma A}$		

(All rules present in the system in Figure 3.4 are present in this system as well.)

Right rules:

$$\mathsf{exinR}: \frac{\Xi \mid \Delta \vdash^c_\Sigma A[n/\alpha] \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_\Sigma \ulcorner n \urcorner \Leftarrow \ulcorner \mathsf{Nat} \urcorner}{\Xi \mid \Delta \vdash^c_\Sigma \exists\alpha : \mathsf{Nat}.\, A}$$

$$\mathsf{natR\_z}: \frac{}{\Xi \mid \Delta \vdash^c_\Sigma \mathsf{Nat}^+(\mathsf{z})} \qquad \mathsf{natR\_s}: \frac{\Xi \mid \Delta \vdash^c_\Sigma \mathsf{Nat}^+(n) \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_\Sigma \ulcorner n \urcorner \Leftarrow \ulcorner \mathsf{Nat} \urcorner}{\Xi \mid \Delta \vdash^c_\Sigma \mathsf{Nat}^+(\mathsf{s}(n))}$$

Left rules:

$$\mathsf{exinL}: \frac{\Xi, \alpha' : \mathsf{Nat} \mid \Delta, \exists\alpha : \mathsf{Nat}.\, A, A[\alpha'/\alpha] \vdash^c_\Sigma C}{\Xi \mid \Delta, \exists\alpha : \mathsf{Nat}.\, A \vdash^c_\Sigma C}$$

$$\mathsf{natL}: \frac{\Xi, \overline{n} \doteq \overline{\mathsf{z}} \mid \Delta \vdash^c_\Sigma C \quad \Xi, \alpha : \mathsf{Nat}, \overline{n} \doteq \overline{\mathsf{s}(\alpha)} \mid \Delta, \mathsf{Nat}^+(\alpha) \vdash^c_\Sigma C}{\Xi \mid \Delta, \mathsf{Nat}^+(n) \vdash^c_\Sigma C}$$

Parameter encoding:

$$\begin{array}{lcl}
\ulcorner \alpha \urcorner & = & \mathtt{x}_\alpha \\
 & & \\
\ulcorner \cdot \urcorner & = & \cdot \\
\ulcorner \Xi, \alpha : \mathsf{Exp} \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_\alpha : \mathtt{exp} \\
\ulcorner \Xi, \alpha : e \searrow v \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_\alpha : \mathtt{eval} \sim \ulcorner e \urcorner \ulcorner v \urcorner \\
\ulcorner \Xi, \alpha : e \doteq e' \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_\alpha : \mathtt{eq\text{-}exp}\ \ulcorner e \urcorner \ulcorner e' \urcorner
\end{array}$$

Figure 3.12: *Extended assertion logic with case analysis over numbers.*

```
% Added formulas:
nat+ : nat -> form.
existsn : (nat -> form) -> form.

% Rules for quantification over numbers
existsnr : {x:nat}
           conc V (F x)
        -> conc V (existsn F).
existsnl : ({x:nat} hyp (F x)
             -> conc V C)
        -> (hyp (existsn F)
            -> conc V C).
```

```
% Right-rules for structural predicate
nat+/z : conc V (nat+ z).
nat+/s : conc V (nat+ N)
      -> conc V (nat+ (s N)).

% Left-rule for structural predicate
nat+/l : (eq-exp (num N) (num z)
           -> conc V C)
      -> ({n'}
          eq-exp (num N) (num (s n'))
          -> hyp (nat+ n')
          -> conc V C)
      -> (hyp (nat+ N)
          -> conc V C).
```

Figure 3.13: *Twelf signature for the extended assertion logic (extends Figure 3.5).*

using only the constructors z and s. This may seem a bit like a tautology at first, but since the grammar of natural numbers in principle could contain additional constructors (e.g., a syntactic plus operator), this is the way of saying that we only consider this particular definition of natural numbers as well-formed.

We also add a left-rule which allows us to consider all the possible ways a number n was constructed, adding hypothetical equalities to the parameters of the proofs for each case. This is reflected by the addition of the rule natL, which has two premises; one for each right-rule that we added. Note that the proof in one of the premises may assume both the existence of some previously unknown number n', *and* that this number is also well-formed. Thus, we can express any fixed number of unfoldings of a given number. Note how the right-rules are effectively used to build up assertion-logic "witnesses" of the structure of numbers, which is what ensures that the left-rule indeed covers all possible cases—namely those that can be used to construct valid witnesses.

To keep things simple, the rule is specialized for the purposes of our termination proof, by using equality of numeral expressions instead of using a separate equality judgment on natural numbers.

To be able to apply the rule on some number n, we need to have a proof of $\mathsf{Nat}^+(n)$. The proof of this predicate must "travel" with n, so we need to include it explicitly in the definition of the logical relation, which is extended with the following in the Twelf formalization:

```
lr/nat' : lr nat' ([e] existsn [n]
                        (existsev [ep: eval e (num n)] top)
                        /\ nat+ n).
```

This also means that the case for t_num in the fundamental theorem must produce a proof that the given number is well-formed. We can easily prove that such a proof exists

for *any* number, formulated as the following theorem:

```
struct-nat : {N} conc cutful (nat+ N) -> type.
%mode struct-nat +N -SP.
%block bstructnat : block {n:nat}{sp:conc cutful (nat+ n)}{_:struct-nat n sp}.
%worlds (bfund | bstructnat) (struct-nat _ _).
%total (N) (struct-nat N _).
```

The block ensures that we always add a structural proof together with any natural number that is added to the LF context. The block is also added to the possible worlds of all meta-theorems using `struct-nat`. The remaining case for t_if can now be straightforwardly formalized using the added left-rule for structural predicates.

3.5.1 Cut admissibility

Adding structural predicates to the assertion logic has non-trivial implications for the cut-admissibility proof, as the induction hypothesis turns out to be too weak to prove the new essential cases. The proof of Theorem 3.9 proceeds by induction on the cut formula, relying on it either getting smaller og staying the same while one of the cut derivations gets smaller. This is normally ensured to be the case by the subformula property of cut-free derivations, but one needs only observe the natR_s rule to see that this property no longer holds: The rule proves $\mathsf{Nat}^+(\mathsf{s}(n))$ from a proof of $\mathsf{Nat}^+(n)$, but the second formula is not a subformula of the first. However, the *subject* n is a subexpression of $\mathsf{s}(n)$; we will exploit this in the following to show how the induction hypothesis of the cut admissibility proof can be strengthened.

We define a formula measure judgment whose definition can be seen in Figure 3.14. Given a formula A, the judgment calculates a measure $\langle K, n\rangle$, where K reflects the structure of A, and n is almost always z, except when $A = \mathsf{Nat}^+(n')$ (for some n'), in which case $n = n'$. It can easily be shown that for any formula A, there exists a derivation of $\|A\| = \langle K, n\rangle$ for some K and n, justifying the use of functional notation.

We will use $\prec$ to denote strict subterm ordering, and $\preccurlyeq$ for its reflexive closure. We can also easily show that if $\|A_1\| = \langle K_1, n_1\rangle$ and $\|A_2\| = \langle K_2, n_2\rangle$, then $A_1 \preccurlyeq A_2$ implies $K_1 \preccurlyeq K_2$, justifying the use of formula metrics as a drop-in replacement for formulas as a termination metric. The difference lies in the case where $A_1 = \mathsf{Nat}^+(n)$ and $A_2 = \mathsf{Nat}^+(n')$: We have $A_1 \neq A_2$, but $K_1 = K_2$. We can thus continue by lexicographic induction over natural numbers n.

The strengthened lemma is formulated as follows. We will show an example of a proof an essential case, namely the cut between natR_s and natL:

Theorem 3.22 (Cut admissibility, strengthened)**.**

If $\mathcal{R} :: \|A\| = \langle K, n\rangle$ *and we have* $\mathcal{S}_1 :: \Xi|\Delta \vdash^{\mathsf{cf}}_{\Sigma} A$ *and* $\mathcal{S}_2 :: \Xi|\Delta, A \vdash^{\mathsf{cf}}_{\Sigma} C$*, then also* $\Xi|\Delta \vdash^{\mathsf{cf}}_{\Sigma} C$*.*

$$\begin{array}{lcll} \text{Formula metrics:} & K & ::= & K_1 \circ K_2 \mid \triangleright K_0 \mid \bullet \\ \text{“Dummy” individuals:} & n_0 & = & \mathsf{z} \\ \text{Formula measure:} & \mathcal{R} & :: & \boxed{\|A\| = \langle K, n\rangle} \end{array}$$

$$\mathsf{r\_top}\colon \frac{}{\|\top\| = \langle \bullet, n_0\rangle} \qquad \mathsf{r\_alle}^{\alpha}\colon \frac{\|A_0\| = \langle K, n'\rangle}{\|\forall \alpha : \mathsf{Exp}.\, A_0\| = \langle \triangleright K, n_0\rangle}$$

$$\mathsf{r\_exie}^{\alpha}\colon \frac{\|A_0\| = \langle K, n'\rangle}{\|\exists \alpha : \mathsf{Exp}.\, A_0\| = \langle \triangleright K, n_0\rangle} \quad \mathsf{r\_exiev}^{\alpha}\colon \frac{\|A_0\| = \langle K, n'\rangle}{\|\exists \alpha : e \searrow v.\, A_0\| = \langle \triangleright K, n_0\rangle}$$

$$\mathsf{r\_exin}^{\alpha}\colon \frac{\|A_0\| = \langle K, n'\rangle}{\|\exists \alpha : \mathsf{Nat}.\, A_0\| = \langle \triangleright K, n_0\rangle}$$

$$\mathsf{r\_bin}\colon \frac{\|A_1\| = \langle K_1, n'\rangle \quad \|A_2\| = \langle K_2, n''\rangle}{\|A_1 \oplus A_2\| = \langle K_1 \circ K_2, n_0\rangle}\ (\oplus \in \{\wedge, \vee, \supset\})$$

$$\mathsf{r\_nat}\colon \frac{}{\|\mathsf{nat}^+(n)\| = \langle \bullet, n\rangle}$$

Figure 3.14: *Measure of formulas.*

Proof sketch. We proceed by lexicographic induction on K, followed by n, followed by mutual induction on $\mathcal{S}_1, \mathcal{S}_2$.

We will prove a single essential case, namely the case where:

$$\mathcal{S}_1 \quad = \quad \mathsf{natR\_s}\colon \frac{\overset{\mathcal{S}_{11}}{\Xi|\Delta \vdash^{\mathsf{cf}}_{\Sigma} \mathsf{Nat}^+(n')} \quad \overset{\mathcal{L}}{\ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner n' \urcorner \Leftarrow \ulcorner \mathsf{Nat} \urcorner}}{\Xi|\Delta \vdash^{\mathsf{cf}}_{\Sigma} \mathsf{Nat}^+(\mathsf{s}(n'))}$$

and,

$$\mathcal{S}_2 \quad = \quad \mathsf{natL}\colon \frac{\overset{\mathcal{S}_{21}}{\Xi, t : \overline{\mathsf{s}(n'')} \doteq \overline{\mathsf{z}} \vdash^{\mathsf{cf}}_{\Sigma} C} \quad \overset{\mathcal{S}_{22}}{\Xi, n'' : \mathsf{Nat}, t : \overline{\mathsf{s}(n')} \doteq \overline{\mathsf{s}(n'')}|\Delta, \mathsf{Nat}^+(n''), \mathsf{Nat}^+(\mathsf{s}(n')) \vdash C}}{\Xi|\Delta, \mathsf{Nat}^+(\mathsf{Nat}^+(\mathsf{s}(n'))) \vdash^{\mathsf{cf}}_{\Sigma} C}$$

So, $\mathcal{R}$ must end in r_nat, implying that we have $K = \bullet$ and $n = \mathsf{s}(n')$. By weakening and IH on $\mathcal{S}_1$ and $\mathcal{S}_{22}$ with $\mathcal{R}$, we obtain

$$\mathcal{S}'_{22} :: \Xi, n'' : \mathsf{Nat}, t : \overline{\mathsf{s}(n')} \doteq \overline{\mathsf{s}(n'')}|\Delta, \mathsf{Nat}^+(n'')) \vdash C.$$

By Proposition 3.8 on $\mathcal{L}$ and $\mathcal{S}'_{22}$, we obtain

$$\mathcal{S}''_{22} :: \Xi, t : \overline{\mathsf{s}(n')} \doteq \overline{\mathsf{s}(n')}|\Delta, \mathsf{Nat}^+(n') \vdash C.$$

By q_id, we obtain a proof of $\overline{\mathsf{s}(n')} \doteq \overline{\mathsf{s}(n')}$. By $\mathcal{L}$, we must then have

$$\ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \mathtt{eq\text{-}exp/id} \Leftarrow \mathtt{eq\text{-}exp}\ \ulcorner \overline{n'} \urcorner\ \ulcorner \overline{n'} \urcorner.$$

So, by Proposition 3.8 on $\mathcal{S}''_{22}$, we obtain

$$\mathcal{S}^{(3)}_{22} :: \Xi|\Delta, \mathsf{Nat}^+(n') \vdash C.$$

By r_nat, we construct $\mathcal{R}' :: \|\mathsf{Nat}^+(n')\| = \langle \bullet, n' \rangle$. Since $n' \prec \mathsf{s}(n')$, we may apply IH on $\mathcal{S}_{11}$ and $\mathcal{S}^{(3)}_{22}$ with $\mathcal{R}'$, and we obtain

$$\mathcal{S}^{(4)}_{22} :: \Xi|\Delta \vdash C,$$

and we are done. □

We formalize formula metrics and formula measures as the following type families. We only show a few representative measure rules, as they are straightforward:

```
% Formula metrics
metric : type.
metric/bin : metric -> metric
                    -> metric.
metric/una : metric -> metric.
metric/nul : metric.

% Formula measure
metric-red : form
          -> metric -> nat -> type.
metric-red/imp :
    metric-red (F1 ==> F2)
               (metric/bin M1 M2) z
               <- metric-red F1 M1 N
               <- metric-red F2 M2 N'.
```

```
% Formula measure, cont'd
metric-red/and :
    metric-red (F1 /\ F2)
               (metric/bin M1 M2) z
               <- metric-red F1 M1 N
               <- metric-red F2 M2 N'.
metric-red/top :
    metric-red top metric/nul z.

metric-red/foralle :
    metric-red (foralle F)
               (metric/una M1) z
  <- ({x} metric-red (F x) M1 N1).

%{ ... remaining rules elided ... }%
```

Due to technical limitations of Twelf, we cannot show totality of the formula measure judgment itself, but must show a separate "effectiveness lemma", by straightforward induction on formulas:

```
metric-red-tot : {F} metric-red F M N -> type.
%mode metric-red-tot +F -RP.
%{ ... proof elided ... }%
%worlds (bhyp | bexp | beval~ | bnat | beq) (metric-red-tot _ _).
%total (F) (metric-red-tot F _).
```

The cut elimination proof must invoke the effectiveness lemma each time it appeals to the cut-admissibility lemma, in order to obtain a measure derivation. The cut-admissibility lemma is reformulated as follows:

```
ca : {M}{N}{RP:metric-red A M N}
     conc cutfree A -> (hyp A -> conc cutfree C) -> conc cutfree C -> type.
%mode ca +M +N +RP +SP1 +SP2 -SP'.
%{ ... proof elided ... }%
%worlds (bhyp | bexp | beval~ | bnat | beq) (ca _ _ _ _ _ _).
% Lexicographic induction on metrics, followed by naturals:
%total {M N [SP1 SP2]} (ca M N _ SP1 SP2 _).
```

3.5.2 Nested data

The metric can be further generalized to work for the case where we have more than one sort that we would like to be able to do case analysis on, by trivially extending the formula measure to larger tuples. If we need to do case analysis on multiple objects which may be defined in terms of each other (e.g., expressions which may also contain natural numbers), we will need to define a common metric for both sorts. This is also the case for mutually defined data. We will see an example of this when we add case analysis on derivations in Section 4.4.

3.5.3 The limits of Twelf

It would be very convenient to be able to do structural induction in the assertion logic, as this would provide case analysis as a special case and at the same time remove many limitations on what we can prove. As shown in [Sar10], Twelf is unfortunately unable to prove cut elimination for such a logic, as the lexicographic subterm ordering used to justify that induction is well-founded in meta-theorems is not proof-theoretically strong enough. The argument involves ordinal analysis, showing that the proof-theoretical strength of Twelf corresponds to transfinite induction up to the ordinal ω^{ω^ω}, while at least ε_0 (the limit of the sequence $\omega, \omega^\omega, \omega^{\omega^\omega}, \ldots$) is required to prove consistency of a first-order logic with induction. An experimental extension of Twelf implements a much much stronger induction principle, known as *lexicographic path induction*, which enables Twelf to, for example, verify the normalization of Gödel's T [SS09]. We consider both ordinal analysis and experimental Twelf extensions as out scope for this thesis though, and will focus our efforts on standard Twelf.

This also defines the limit of the expressive power of the object language. If we add features such that the power of the language matches or exceeds Gödel's T, then we cannot verify its termination in Twelf. This limitation would show up as an inability to verify termination of the cut-admissibility proof for the assertion logic. We could, however, still prove the existence of an *assertion logic* proof of termination, but we would then have to believe in the consistency of the assertion logic, or, the termination of its cut-admissibility procedure, which we could still formulate, but just not prove to be terminating. Whether this is satisfactory is a matter of personal preference and philosophy—in the end we also have to believe in the consistency of Twelf anyway.

In the following chapters, we will not study the termination proof and its formalization any further, but will instead shift our attention to *binary* logical relations, which can be used to prove some interesting relationships between programs.

4 Equational reasoning for CBN simply typed λ-calculus

In this chapter, we will set up a binary logical relation for proving observational equivalence of expressions in a variant of the language $\lambda_{\mathsf{cbn}}^{\rightarrow,\mathsf{bool},\mathsf{nat}}$, but extended with the possibility of divergence. We will refer to the extended language by the name $\lambda_{\mathsf{cbn}}^{\rightarrow,\mathsf{nat}}$. Note that we have left out booleans as they can be considered a special case of natural numbers.

Our language does not feature unrestricted or even primitive recursion, and hence does not fully represent a prototypical function language. It does, however, still give useful insight on the difficult aspects of the formalization of proofs using binary logical relations. We have chosen to include the possibility of divergence to make sure that our developments do not rely on termination, which may not be present in a "real" language.

The formalization will generally follow the same structure as in the last chapter, but does get a bit more complicated due to the need of the assertion logic to be able to inspect derivations. The developments in this section are inspired by the chapter on equational reasoning in [Har13], but adapted to a big-step semantics.

The chapter is structured as follows. In Section 4.1, we introduce our object language, and discuss what it means for two programs to be observationally equivalent. In Section 4.2, we set up a binary logical relation on expressions which implies observational equivalence, and we then prove that the relation is a congruent equivalence relation. In Section 4.3, we demonstrate how the logical relation can be used to prove soundness of a syntactic reasoning system for deriving program equivalences. In Section 4.4, we describe the assertion logic and introduce a representation logic for enabling the combination of equality proofs with case analysis on derivations. In Section 4.5, we discuss a challenge related to the representation of our notion of equivalence in LF, and how this complicates formalization of the soundness proof for axiomatic equivalence. We briefly summarize the formalization in Section 4.6.

Natural numbers: $n :: \mathsf{Nat} ::= \mathsf{z} \mid \mathsf{s}(n)$
Types: $\tau :: \mathsf{Tp} ::= \mathsf{nat} \mid \tau_2 \rightharpoonup \tau_0$
Contexts: $\Gamma :: \mathsf{Ctx} ::= \cdot \mid \Gamma, x : \tau$
Expressions: $e, v :: \mathsf{Exp} ::= \overline{n} \mid x \mid \lambda x.\, e_0 \mid e_1\ e_2 \mid \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \mid \mathsf{diverge}$
Values: $\mathcal{V} :: \boxed{v\ \mathsf{value}}$:

$$\mathsf{v\_lam}: \frac{}{\lambda x.\, e_0\ \mathsf{value}} \qquad \mathsf{v\_num}: \frac{}{\overline{n}\ \mathsf{value}}$$

Dynamic semantics: $\mathcal{E} :: \boxed{e \Downarrow v}$:

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

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

Static semantics: $\mathcal{T} :: \boxed{\Gamma \vdash e : \tau}$:

$$\mathsf{t\_var}: \frac{}{\Gamma \vdash x : \tau}\ (\Gamma(x) = \tau) \qquad \mathsf{t\_num}: \frac{}{\Gamma \vdash \overline{n} : \mathsf{nat}}$$

$$\mathsf{t\_lam}: \frac{\Gamma, x : \tau_2 \vdash e_0 : \tau_0}{\Gamma \vdash \lambda x.\, e_0 : \tau_2 \rightharpoonup \tau_0} \qquad \mathsf{t\_app}: \frac{\Gamma \vdash e_1 : \tau_2 \rightharpoonup \tau_0 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_0}$$

$$\mathsf{t\_case}: \frac{\Gamma \vdash e_0 : \mathsf{nat} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_2 : \tau}{\Gamma \vdash \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) : \tau} \qquad \mathsf{t\_diverge}: \frac{}{\Gamma \vdash \mathsf{diverge} : \tau}$$

Figure 4.1: *Syntax and semantics of* $\lambda^{\rightharpoonup,\mathsf{nat}}_{\mathsf{cbn}}$.

4.1 Language definition

The syntax and semantics of our language can be seen in Figure 4.1. Again, whenever we write $\Gamma, x : \tau$, we implicitly include the condition $x \notin \mathrm{dom}(\Gamma)$. The Twelf formalization is straightforward, and is shown in 4.2.

What do we mean when we say that two expressions are equivalent? Certainly, identical expressions are equivalent in the most trivial sense, but this is a very boring notion of equivalence. Intuitively, we can say that two expressions are equivalent whenever we cannot conduct a test on them that yields different results. In other words; that we cannot tell them apart by executing them. To give a more precise definition, we define the notion of a *context*, written as $\mathcal{C}\{\circ\}$, as an expression with a single "hole". *Replacement* is the process of filling the hole with an expression, e, written $\mathcal{C}\{e\}$. Note that any free variables exposed in the context are captured by the replacement, making it different from substitution. A *program context* is a context which leaves no variables uncaptured,

```
% types
tp : type.
nat' : tp.
=> : tp -> tp -> tp.
% infix right 1 =>.

% expressions
exp : type.
num : nat -> exp.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
case : exp -> exp
    -> (exp -> exp) -> exp.
diverge : exp.

% typing
of : exp -> tp -> type.
of/num : of (num N) nat'.
of/lam : of (lam E0) (T2 => T0)
       <- ({x} of x T2
           -> of (E0 x) T0).
of/app : of (app E1 E2) T0
       <- of E1 (T2 => T0)
       <- of E2 T2.
of/diverge : of diverge T.
```

```
% typing, cont'd
of/case : of (case E0 E1 E2) T
        <- of E0 nat'
        <- of E1 T
        <- ({x} of x nat'
            -> of (E2 x) T).

% evaluation (lazy, big-step)
eval : exp -> exp -> type.
eval/lam : eval (lam E0) (lam E0).
eval/num : eval (num N) (num N).
eval/app : eval (app E1 E2) V
         <- eval (E0 E2) V
         <- eval E1 (lam E0).
eval/case/z : eval (case E0 E1 E2) V
            <- eval E1 V
            <- eval E0 (num z).
eval/case/s : eval (case E0 E1 E2) V
            <- eval (E2 (num N)) V
            <- eval E0 (num (s N)).

% values
val : exp -> type.
val/lam : val (lam E0).
val/num : val (num N).
```

Figure 4.2: *Twelf signature for* $\lambda^{\rightarrow,\mathsf{nat}}_{\mathsf{cbn}}$

and thus results in a program that we can attempt to evaluate.

In this development, we will only consider the question of proving equivalence between *well-typed* expressions of the same type. Given a context $\mathcal{C}$, we will write $\mathcal{C} : (\Gamma, \tau) \rightsquigarrow (\Gamma', \tau')$ if $\Gamma' \vdash \mathcal{C}\{e\} : \tau'$ for any e where $\Gamma \vdash e : \tau$. In order to tell two well-typed expressions $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ apart, we must thus find a program context $\mathcal{C} : (\Gamma, \tau) \rightsquigarrow (\cdot, \mathsf{nat})$ such that $\mathcal{C}\{e\} \Downarrow \overline{n}$ and $\mathcal{C}\{e'\} \not\Downarrow \overline{n}$, or vice-versa. That is, we must find a test returning a natural number which either diverges for one expression and succeeds for the other, or which yields different results for each expression. This captures the notion of equivalence very well: If it is impossible to write a program that can tell e and e' apart, then the two expressions must indeed implement exactly the same functionality. In languages with possible divergence (such as this one), checking whether both expressions *terminate* succesfully in all contexts would also be sufficient.

It is, however, a daunting task to prove that two expressions yields the same results in *every conceivable context*. Instead, we approach the problem by introducing a binary logical relation which essentially captures the above concept. We show this by proving that it is a congruence relation on expressions which implies equality of results at base types. Thus, if we can prove that two expressions e and e' are related, we can apply

congruence to "build" any conceivable context up around them, including program contexts, while preserving the property that all programs we can build this way will yield equal results at base types.

4.2 Logical equivalence

Definition 4.1. Logical equivalence is a family of relations $e \sim_\tau e'$ between closed expressions. It is inductively defined on types as follows:

$$\begin{aligned} e \sim_{\mathsf{nat}} e' \quad &\text{iff} \quad \forall n.\ e \Downarrow \overline{n} \Leftrightarrow e' \Downarrow \overline{n}, \\ e \sim_{\tau_2 \rightharpoonup \tau_0} e' \quad &\text{iff} \quad \forall e_2, e_2'.\ e_2 \sim_{\tau_2} e_2' \Rightarrow e\ e_2 \sim_{\tau_0} e'\ e_2' \end{aligned} \qquad \diamondsuit$$

The relation at base types asserts that related expressions are *Kleene equivalent* (they either both evaluate to identical numerical values, or they both diverge), whereas the relation at function types asserts that related expressions takes related arguments to related results.

Remark 4.2. Note that related expressions need not necessarily be well-typed. For instance, we can easily verify that $(\lambda x.\, \overline{z})\ (\lambda x.\, x\ x) \sim_{\mathsf{nat}} (\lambda x.\, \overline{z})\ (\lambda x.\, x\ x)$ (both sides always evaluate to $\overline{z}$), but $[] \not\vdash (\lambda x.\, \overline{z})\ (\lambda x.\, x\ x) : \mathsf{nat}$, as the expression contains the untypable expression $\lambda x.\, x\ x$.

On the other hand, identical expressions are not necessarily related. For instance, consider the example where $e = \lambda x.\, x\ \overline{z}$. We trivially have $e \sim_{\mathsf{nat}} e$ (e never evaluates to a numeral), but if $\tau = ((\mathsf{nat} \rightharpoonup \mathsf{nat}) \rightharpoonup \mathsf{nat} \rightharpoonup \mathsf{nat}) \rightharpoonup \mathsf{nat}$, we have $e \not\sim_\tau e$, since e fails to take related arguments to related results at type τ: Consider the arguments $e_2 = \lambda f.\, f$ and $e_2' = \lambda f.\, \lambda x.\, f\ x$, for which it can be proved that $e_2 \sim_{(\mathsf{nat} \rightharpoonup \mathsf{nat}) \rightharpoonup \mathsf{nat} \rightharpoonup \mathsf{nat}} e_2'$. But for $e \sim_\tau e$ to hold, we must then have $(\lambda x.\, x\ \overline{z})\ \lambda f.\, f \sim_{\mathsf{nat}} (\lambda x.\, x\ \overline{z})\ \lambda f.\, \lambda x.\, f\ x$, which is clearly *not* true: The left expression always evaluates to $\overline{z}$, while the right expression always evaluates to $\lambda x.\, \overline{z}\ x$, and hence they are not Kleene equivalent. $*$

Lemma 4.3. *Logical equivalence is symmetric: If $e \sim_\tau e'$ then $e' \sim_\tau e$.*

Proof. By induction on the structure of τ.

- Case $\tau = \mathsf{nat}$. By assumption we have $e \sim_{\mathsf{nat}} e'$. It suffices to show $e' \Downarrow \overline{n} \Leftrightarrow e \Downarrow \overline{n}$ for any n, which follows directly from the assumption.
- Case $\tau = \tau_2 \rightharpoonup \tau_0$. By assumption we have $h_1 :: e \sim_{\tau_2 \rightharpoonup \tau_0} e'$, and by implication introduction we have $h_2 :: e_2' \sim_{\tau_2} e_2$ for some e_2, e_2'. It suffices to show $e'\ e_2' \sim_{\tau_0} e\ e_2$. By IH on h_2 we have $e_2 \sim_{\tau_2} e_2'$, and hence by h_1 assumption we get $h_3 :: e\ e_2 \sim_{\tau_0} e'\ e_2'$. By induction on h_3 we get the desired result. $\square$

Lemma 4.4. *Logical equivalence is transitive: If $e \sim_\tau e'$ and $e' \sim_\tau e''$ then $e \sim_\tau e''$.*

Proof. By induction on the structure of τ.

- Case $\tau = \mathsf{nat}$. By assumption we have $h_1 :: e \sim_{\mathsf{nat}} e'$ and $h_2 :: e' \sim_{\mathsf{nat}} e''$. It suffices to show that for any n, we have $e \Downarrow \overline{n} \Leftrightarrow e'' \Downarrow \overline{n}$. By h_1 we have $e \Downarrow \overline{n} \Leftrightarrow e' \Downarrow \overline{n}$, and by h_2 we have $e' \Downarrow \overline{n} \Leftrightarrow e'' \Downarrow \overline{n}$. Hence the desired result follows immediately.

- Case $\tau = \tau_2 \rightharpoonup \tau_0$. By assumption we have $h_1 :: e \sim_{\tau_2 \rightharpoonup \tau_0} e'$ and $h_2 :: e' \sim_{\tau_2 \rightharpoonup \tau_0} e''$. It suffices to show that for any $e_2 : \tau_2$ and $e_2'' : \tau_2$, if $h_3 :: e_2 \sim_{\tau_2} e_2''$, then $e\ e_2 \sim_{\tau_0} e''\ e_2''$. By Lemma 4.3 on τ_2 with h_3, we have $h_4 :: e_2'' \sim_{\tau_2} e_2$, and by induction on τ_2 with h_3, h_4, we get $h_5 :: e_2 \sim_{\tau_2} e_2$. By h_5 on h_1, we get $h_6 :: e\ e_2 \sim_{\tau_0} e'\ e_2$. By h_3 on h_2 we get $h_7 :: e'\ e_2 \sim_{\tau_2} e''\ e_2''$, and hence by induction on τ_2 with h_6, h_7, we are done. □

The above establishes that logical equivalence is a partial equivalence relation at any type. A general property of partial equivalences is conditional reflexivity, i.e., expressions are related to themselves if they are related to any expression:

Lemma 4.5. *Logical equivalence is conditionally reflexive: If $e \sim_\tau e'$ then also $e \sim_\tau e$ and $e' \sim_\tau e'$.*

Proof. Assume $h_1 :: e \sim_\tau e'$. By Lemma 4.3 on h_1, we have $h_2 :: e' \sim_\tau e$. By Lemma 4.4 on h_1, h_2, respectively h_2, h_1, we get $e \sim_\tau e$, respectively $e' \sim_\tau e'$, and we are done. □

To be useful for our purposes, we need to extend the notion of logical equivalence to open terms.

Definition 4.6 (Closing substitution)**.** A *closing substitution*, γ, for a context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a finite function assigning closed expressions $e_1, \ldots, e_n$ to each x_i in the domain of Γ.

We write $\hat{\gamma}(e)$ for the substitution $e[\gamma(x_1), \ldots, \gamma(x_n)/x_1, \ldots, x_n]$. ♢

Definition 4.7 (Pointwise equivalence)**.** Given two closing substitutions γ, γ', we write $\gamma \sim_\Gamma \gamma'$ to mean that we have $\mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma) = \mathrm{dom}(\gamma')$, and that for for every $x \in \mathrm{dom}(\Gamma)$, we have $\gamma(x) \sim_{\Gamma(x)} \gamma'(x)$. ♢

By pointwise application of Lemma 4.3 and Lemma 4.4, we also get symmetry and transitivity of the relation $\cdot \sim_\Gamma \cdot$.

Given a closing substitution γ, a closed expression e and some variable $x \notin \mathrm{dom}(\gamma)$ we define $\gamma[x \mapsto e]$ as the extension of γ such that

$$(\gamma[x \mapsto e])(x') = \begin{cases} \gamma(x') & \text{if } x \neq x' \\ e & \text{if } x = x' \end{cases}$$

Definition 4.8 (Open logical equivalence)**.** Suppose e, e' are (open) expressions. *Open logical equivalence*, written $\Gamma \vdash e \sim e' : \tau$, means that for any Γ-closing substitutions γ, γ', if $\gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim_\tau \hat{\gamma}'(e')$. ♢

Lemma 4.9 (Symmetry)**.** *If* $\Gamma \vdash e \sim e' : \tau$ *then* $\Gamma \vdash e' \sim e : \tau$.

Proof. Assume $h_1 :: \Gamma \vdash e \sim e' : \tau$. It suffices to show that for any γ, γ', if $h_2 :: \gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e') \sim_\tau \hat{\gamma}'(e)$. By Lemma 4.3 (pointwise) on h_2, we get $h_2' :: \hat{\gamma}' \sim_\Gamma \hat{\gamma}$. By h_2' and h_1, we get $h_3 :: \hat{\gamma}'(e) \sim_\tau \hat{\gamma}(e')$. By Lemma 4.3 on h_3, we are done. □

Lemma 4.10 (Transitivity)**.** *If* $\Gamma \vdash e \sim e' : \tau$ *and* $\Gamma \vdash e' \sim e'' : \tau$*, then we have* $\Gamma \vdash e \sim e'' : \tau$.

Proof. Assume $h_1 :: \Gamma \vdash e \sim e' : \tau$ and $h_2 :: \Gamma \vdash e' \sim e'' : \tau$. It suffices to show that for any γ, γ', if $h_3 :: \gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim_\tau \hat{\gamma}'(e'')$. By pointwise application of Lemma 4.5 on h_3 we have $h_4 :: \gamma \sim_\Gamma \gamma$. By h_1 and h_4, we thus have $h_5 :: \hat{\gamma}(e) \sim_\tau \hat{\gamma}(e')$. By h_3 and h_2 we have $h_6 :: \hat{\gamma}(e') \sim_\tau \hat{\gamma}'(e'')$, and then by Lemma 4.4 on h_5, h_6, we are done. □

We have now proved that open logical equivalence is a partial equivalence relation. It remains to show that it is also a congruence, and that it is reflexive at atomic (leaf) expressions.

To prove congruence, we first need to define a degenerate variant of observational equivalence:

Definition 4.11 (Weak equivalence)**.** We will write $e \approx e'$ and say that e and e' are *weakly equivalent* whenever

$$\forall v.\, e \Downarrow v \Leftrightarrow e' \Downarrow v.$$ ♢

Weak equivalence obviously implies observational equivalence, since the expressions are required to evaluate to identical results.

Logical equivalence is closed under weak equivalence:

Lemma 4.12 (Closure under weak equivalence)**.** *Suppose* $e \sim_\tau e$*. If* $d \approx e$*, then* $d \sim_\tau e$.

Proof. Assume $h_1 :: e \sim_\tau e$ and $h_2 :: \forall v.\, d \Downarrow v \Leftrightarrow e \Downarrow v$. We proceed by induction on the structure of τ:

- Case $\tau = \mathsf{nat}$. Then the result follows immediately from Definition 4.1 and h_2.
- Case $\tau = \tau_2 \rightharpoonup \tau_0$. It suffices to show that for any e_2, e_2', if $h_3 :: e_2 \sim_{\tau_2} e_2'$ then $d\ e_2 \sim_{\tau_0} e\ e_2'$.

  By h_3 and h_1 we obtain $h_4 :: e\ e_2 \sim_{\tau_0} e\ e_2'$, and by Lemma 4.5 on h_4, we get $h_5 :: e\ e_2 \sim_{\tau_0} e\ e_2$. By Lemma 4.4 on h_4, it thus suffices to show $d\ e_2 \sim_{\tau_0} e\ e_2$. But then by IH on τ_0 and by h_5, it suffices to show $h_6 :: \forall v'.\, d\ e_2 \Downarrow v' \Leftrightarrow e\ e_2 \Downarrow v'$.

  For the forward direction, assume $\mathcal{E}_d :: d\ e_2 \Downarrow v'$. $\mathcal{E}_d$ must end in an application of e_app, implying that we have $\mathcal{E}_{d1} :: d \Downarrow \lambda x.\, e_0$ and $\mathcal{E}_{d2} :: e_0[e_2/x] \Downarrow v'$. By h_2 and

$\mathcal{E}_{d1}$, we obtain a derivation $\mathcal{E}_{e1} :: e \Downarrow \lambda x.\, e_0$. By e_app on $\mathcal{E}_{e1}$ and $\mathcal{E}_{d2}$, we obtain the desired derivation of $e\ e_2 \Downarrow v'$. The argument for the other direction is analogous, establishing h_6, and we are done. □

In the above, the first premise $e \sim_\tau e$ may seem redundant. After all, the second premise implies that d and e have identical results (a slight generalization of Kleene equivalence), which would seem to be sufficient for proving $d \sim_\tau e$. It is indeed sufficient when $\tau = \mathsf{nat}$, however, logical equivalence is a stronger notion than that, as it also requires expressions related at *function types* to take related arguments to related results. At function types, this does not necessarily follow just because e and d evaluates to identical results, as identical expressions are not necessarily related at function types (see Remark 4.2).

By conditional reflexivity and transitivity, the above lemma can be strengthened slightly:

Corollary 4.13. *Suppose $a_1 :: e \sim_\tau e'$ and $a_2 :: e \approx d$ and $a_3 :: e' \approx d'$, then also $d \sim_\tau d'$.*

Proof. By Lemma 4.5 on a_1, we get $r_1 :: e \sim_\tau e$ and $r_2 :: e' \sim_\tau e'$. Thus by Lemma 4.12 on r_1 and a_2, we get $r_1' :: d \sim_\tau e$. Similarly, by 4.12 on r_2 and a_3, we get $d' \sim_\tau e'$, and by Lemma 4.3 we get $r_2' :: e' \sim_\tau d'$.

But then by Lemma 4.4 (twice) on r_1', a_1 and r_2', we are done. □

We will need the following result about commutativity of application over case branches when proving congruence at case:

Lemma 4.14 (Application commutes over case)**.** *For any expressions e_0, e_1, $x.\, e_2$ and e', where $x \notin \mathrm{FV}(e')$, we have*

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

Proof. For the "only if" direction: Assume that we have

$$\mathcal{E} :: \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2)\ e' \Downarrow v,$$

which must end in e_app, implying that there exists an expression $x'.\, e_0'$, and that we have derivations

$$\mathcal{E}_1 :: \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \Downarrow \lambda x'.\, e_0'$$
$$\mathcal{E}_2 :: e_0'[e'/x] \Downarrow v.$$

There are only two possible rules that $\mathcal{E}_1$ can end in, namely:

- $\mathcal{E}_1$ ends in e_case0. Then we have derivations $\mathcal{E}_{10} :: e_0 \Downarrow \overline{z}$ and $\mathcal{E}_{11} :: e_1 \Downarrow \lambda x'.\, e_0'$. But then by e_app on $\mathcal{E}_{11}$ and $\mathcal{E}_2$, we get $\mathcal{E}_{11}' :: e_1\ e' \Downarrow v$. Thus by e_case0 on $\mathcal{E}_{10}$ and $\mathcal{E}_{11}'$, we are done.

- $\mathcal{E}_1$ ends in e_case1. Then there exists an n' and derivations $\mathcal{E}_{10} :: e_0 \Downarrow \overline{\mathsf{s}(n')}$ and $\mathcal{E}_{12} :: e_2[\overline{n'}/x] \Downarrow \lambda x'.\, e_0'$. By e_app on $\mathcal{E}_{12}$ and $\mathcal{E}_2$, we get $\mathcal{E}_{12}' :: (e_2\ e')[\overline{n'}/x] \Downarrow v$, and thus by e_case1 on $\mathcal{E}_{10}$ and $\mathcal{E}_{12}'$, we are done.

For the "if" direction: Assume that we have

$$\mathcal{E} :: \mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1)\ e,\ x.\, \hat{\gamma}(e_2)\ e) \Downarrow v.$$

We see that $\mathcal{E}$ can only end in e_case0 and e_case1, and that in each case we obtain derivations from which we can directly construct the evaluation on the left hand side using e_app and one of e_case0 and e_case1, concluding the proof. □

We have now set up the necessary machinery for proving that logical equivalence is a congruence. This property will be proved as a series of separate lemmas in the following.

Lemma 4.15 (Congruence at application)**.**
Suppose $\Gamma \vdash e_1 \sim e_1' : \tau_2 \rightharpoonup \tau_0$ *and* $\Gamma \vdash e_2 \sim e_2' : \tau_2$. *Then also* $\Gamma \vdash e_1\ e_2 \sim e_1'\ e_2' : \tau_0$.

Proof. Assume $h_1 :: \Gamma \vdash e_1 \sim e_1' : \tau_2 \rightharpoonup \tau_0$ and $h_2 :: \Gamma \vdash e_2 \sim e_2' : \tau_2$. It suffices to show that for any γ, γ', if $h_3 :: \gamma \sim_\Gamma \gamma'$, then also $\hat{\gamma}(e)\ \hat{\gamma}(e_2) \sim_{\tau_2 \rightharpoonup \tau_0} \hat{\gamma}'(e')\ \hat{\gamma}'(e_2)$.

By h_2 and h_3, we have $h_4 :: \hat{\gamma}(e_2) \sim_{\tau_2} \hat{\gamma}'(e_2')$. But then by Definition 4.1 on h_1 with h_4, we get the desired result. □

Lemma 4.16 (Congruence at abstraction)**.**
Suppose $\Gamma, x : \tau_2 \vdash e \sim e' : \tau_0$. *Then also* $\Gamma \vdash \lambda x.\, e \sim \lambda x.\, e' : \tau_2 \rightharpoonup \tau_0$.

Proof. Assume $h_1 :: \Gamma, x : \tau_2 \vdash e \sim e' : \tau_0$. We need to show that if $h_2 :: \gamma \sim_\Gamma \gamma'$, then $\lambda x.\, \hat{\gamma}(e) \sim_{\tau_2 \rightharpoonup \tau_0} \lambda x.\, \hat{\gamma}'(e') : \tau_2 \rightharpoonup \tau_0$.

By Definition 4.1, assuming $h_3 :: e_2 \sim_{\tau_2} e_2'$ for some e_2, e_2', it suffices to show $(\lambda x.\, \hat{\gamma}(e))\ e_2 \sim_{\tau_0} (\lambda x.\, \hat{\gamma}'(e'))\ e_2'$.

Let $\gamma_0 = \gamma[x \mapsto e_2]$, and $\gamma_0' = \gamma'[x \mapsto e_2']$. By h_2, we then have $h_4 :: \gamma_0 \sim_{\Gamma, x:\tau_2} \gamma_0'$. But then by Definition 4.1 and h_1, h_4, we have $h_5 :: \hat{\gamma_0}(e) \sim_{\tau_0} \hat{\gamma_0}'(e')$. Since closing substitutions substitute closed expressions for variables, this is equivalent to $h_6 :: \hat{\gamma}(e)[e_2/x] \sim_{\tau_0} \hat{\gamma}'(e')[e_2'/x]$.

By e_app and e_lam, it follows that

$$h_7 :: (\lambda x.\, \hat{\gamma}(e))\ e_2 \approx \hat{\gamma}(e)[e_2/x], \qquad \text{and,}$$
$$h_7' :: (\lambda x.\, \hat{\gamma}'(e'))\ e_2' \approx \hat{\gamma}'(e')[e_2'/x].$$

But then by Corollary 4.13 on h_6, h_7 and h_7', we get $(\lambda x.\, \hat{\gamma}(e))\ e_2 \sim_{\tau_0} (\lambda x.\, \hat{\gamma}'(e'))\ e_2'$, and we are done. □

Lemma 4.17 (Congruence at case). *Suppose* $a_1 :: \Gamma \vdash e_0 \sim e_0' : \mathsf{nat}$ *and* $a_2 :: \Gamma \vdash e_1 \sim e_1' : \tau$ *and* $a_3 :: \Gamma, x : \mathsf{nat} \vdash e_2 \sim e_2' : \tau$.

Then also $\Gamma \vdash \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \sim \mathsf{ncase}(e_0',\ e_1',\ x.\, e_2') : \tau$.

Proof. It suffices to show that if $h_1 :: \gamma \sim_\Gamma \gamma'$, then

$$\mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2)) \sim_\tau \mathsf{ncase}(\hat{\gamma}'(e_0'),\ \hat{\gamma}'(e_1'),\ x.\, \hat{\gamma}'(e_2')).$$

We proceed by induction on τ:

- Case $\tau = \mathsf{nat}$. Then it suffices to show that for any natural number n, we have

  $$\mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2)) \Downarrow \overline{n} \Leftrightarrow \mathsf{ncase}(\hat{\gamma}'(e_0'),\ \hat{\gamma}'(e_1'),\ x.\, \hat{\gamma}'(e_2')) \Downarrow \overline{n}.$$

  We will only show the forward direction, as the other is analogous. Assume that we have a derivation $\mathcal{E}$ of the left hand side. We have two possible subcases:

  - $\mathcal{E}$ ends in e_case0. Then we have derivations $\mathcal{E}_0 :: \hat{\gamma}(e_0) \Downarrow \overline{z}$ and $\mathcal{E}_1 :: \hat{\gamma}(e_1) \Downarrow \overline{n}$. But then by a_1, a_2 and h_1, we also have $\mathcal{E}_0' :: \hat{\gamma}'(e_0') \Downarrow \overline{z}$ and $\mathcal{E}_1' :: \hat{\gamma}'(e_1') \Downarrow \overline{n}$. Thus by e_case0 on $\mathcal{E}_0', \mathcal{E}_1'$, we are done.
  - $\mathcal{E}$ ends in e_case1. Then there exists an n' and derivations $\mathcal{E}_0 :: \hat{\gamma}(e_0) \Downarrow \overline{\mathsf{s}(n')}$ and $\mathcal{E}_2 :: \hat{\gamma}(e_2)[\overline{n'}/x] \Downarrow \overline{n}$. By a_1 and h_1 on $\mathcal{E}_0$, we obtain $\mathcal{E}_0' :: \hat{\gamma}'(e_0') \Downarrow \overline{\mathsf{s}(n')}$. Thus by e_case1, it suffices to show

    $$\hat{\gamma}'(e_2)[\overline{n'}/x] \Downarrow \overline{n}.$$

    We define $\gamma_0 = \gamma[x \mapsto \overline{n'}]$ and $\gamma_0' = \gamma'[x \mapsto \overline{n'}]$. Since trivially $\overline{n'} \sim_{\mathsf{nat}} \overline{n'}$, then by h_1 we have $h_1' :: \gamma_0 \sim_{\Gamma, x:\mathsf{nat}} \gamma_0'$. Thus we get the desired derivation by a_3 on h_1' and $\mathcal{E}_2$, and we are done.

- Case $\tau = \tau_2 \rightharpoonup \tau_0$. It suffices to show that for any e, e', if $h_2 :: e \sim_{\tau_2} e'$, then also

  $$\mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2))\ e \sim_{\tau_0} \mathsf{ncase}(\hat{\gamma}'(e_0'),\ \hat{\gamma}'(e_1'),\ x.\, \hat{\gamma}'(e_2'))\ e'. \tag{4.1}$$

  Since e, e' are closed, we trivially have $h_2' :: \Gamma \vdash e \sim e' : \tau_2$ and $h_2'' :: \Gamma, x : \mathsf{nat} \vdash e \sim e' : \tau_2$. But then by Lemma 4.15 on a_2, h_2' and a_3, h_2'', respectively, we get $r_2 :: \Gamma \vdash e_1\ e \sim e_1'\ e' : \tau_0$ and $r_3 :: \Gamma, x : \mathsf{nat} \vdash e_2\ e \sim e_2'\ e' : \tau_0$. By the induction hypothesis on a_1, r_2 and r_3, followed by h_1, we then get

  $$\mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1)\ e,\ x.\, \hat{\gamma}(e_2)\ e) \sim_{\tau_0} \mathsf{ncase}(\hat{\gamma}'(e_0'),\ \hat{\gamma}'(e_1')\ e',\ x.\, \hat{\gamma}'(e_2')\ e').$$

  This is not quite (4.1), as the application has been moved into the branches. By Corollary 4.13 on the above, it suffices to show the following two subgoals:

  $$\mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2))\ e \approx \mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1)\ e,\ x.\, \hat{\gamma}(e_2)\ e) \tag{4.2}$$

  $$\mathsf{ncase}(\hat{\gamma}'(e_0'),\ \hat{\gamma}'(e_1'),\ x.\, \hat{\gamma}'(e_2'))\ e' \approx \mathsf{ncase}(\hat{\gamma}'(e_0'),\ \hat{\gamma}'(e_1')\ e,\ x.\, \hat{\gamma}'(e_2')\ e') \tag{4.3}$$

  Both follows directly from Lemma 4.14, and we are done. □

Additionally, logical equivalence is reflexive at variable, numeral, and diverge expressions:

Lemma 4.18 (Reflexivity at variables)**.** *If* $\Gamma(x) = \tau$*, then also* $\Gamma \vdash x \sim x : \tau$*.*

Proof. It suffices to show that for any γ, γ', if $h_1 :: \gamma \sim_\Gamma \gamma'$, then $\gamma(x) \sim_\tau \gamma'(x)$.

From h_1 we get by definition that $\gamma(x) \sim_{\Gamma(x)} \gamma'(x)$. But then the desired result follows directly from the assumption that $\Gamma(x) = \tau$, and we are done. □

Lemma 4.19 (Reflexivity at numerals)**.** *For any* $n ::$ Nat *and context* Γ*, we have that* $\Gamma \vdash \overline{n} \sim \overline{n} :$ nat*.*

Proof. Numerals are closed, and hence it suffices to show that $\overline{n} \sim_{\mathsf{nat}} \overline{n}$, which by Definition 4.1 is trivially true. □

It is not possible to show reflexivity at diverge directly, but it follows from this more general lemma that says that expressions that do not evaluate to anything are related:

Lemma 4.20 (Strictness)**.** *For any expressions* e, e'*, if* $e \not\Downarrow$ *and* $e' \not\Downarrow$*, then* $e \sim_\tau e'$*.*

Proof. By induction on the structure of τ. Assume $h_1 :: e \not\Downarrow$ and $h_2 :: e \not\Downarrow$.

- Case $\tau =$ nat. By Definition 4.1 and h_1, h_2, the result follows immediately.
- Case $\tau = \tau_2 \rightharpoonup \tau_0$. It suffices to show that for any $e_2 : \tau_2$ and $e_2' : \tau_2$, if $h_3 :: e_2 \sim_{\tau_2} e_2'$, then we have $e\ e_2 \sim_{\tau_0} e'\ e_2'$.

  Now, assume $e\ e_2 \Downarrow v$ for some v. The only applicable evaluation rule is e_app, which entails that we have $\mathcal{E} :: e \Downarrow \lambda x^{\tau_2}.\, e_0$ for some e_0. But by $\mathcal{E}$ and h_1 we have a contradiction, and thus we have $h_4 :: e\ e_2 \not\Downarrow$. An analogous argument gives us $h_5 :: e'\ e_2' \not\Downarrow$.

  But then we can get the desired result by induction on τ_0 with h_4, h_5, and we are done. □

Lemma 4.21 (Reflexivity at diverge)**.** *We have* $\Gamma \vdash$ diverge $\sim$ diverge $: \tau$ *for any type* τ *and context* Γ*.*

Proof. Since diverge is closed, it suffices to show diverge $\sim_\tau$ diverge. Since there is no evaluation rule matching diverge, we get $h_1 ::$ diverge $\not\Downarrow$. But then by Lemma 4.20 on h_1 (twice), we get the desired result. □

We could continue to prove the *fundamental theorem* for logical equivalence, i.e., that all well-typed expressions are related to themselves. The proof would work by trivial induction over typing derivations, using the appropriate congruence and reflexivity lemmas where needed. The fundamental theorem for this relation is not particularly interesting though, as it only says that all well-typed programs behaves as themselves, i.e.: if a given well-typed program evaluates to a result, then it evaluates to a result.

In the following section, we use our logical relation to prove a much more interesting result, namely soundness of an equational reasoning system.

Judgment $\boxed{\Gamma \vdash e \doteq e' : \tau}$:

$$\textsf{q\_sym}: \frac{\Gamma \vdash e \doteq e' : \tau}{\Gamma \vdash e' \doteq e : \tau} \qquad \textsf{q\_trans}: \frac{\Gamma \vdash e \doteq e' : \tau \quad \Gamma \vdash e' \doteq e'' : \tau}{\Gamma \vdash e \doteq e'' : \tau}$$

$$\textsf{q\_var}: \frac{}{\Gamma \vdash x \doteq x : \tau}\ (\Gamma(x) = \tau)$$

$$\textsf{q\_num}: \frac{}{\Gamma \vdash \overline{n} \doteq \overline{n} : \mathsf{nat}} \qquad \textsf{q\_diverge}: \frac{}{\Gamma \vdash \mathsf{diverge} \doteq \mathsf{diverge} : \tau}$$

$$\textsf{q\_lam}: \frac{\Gamma, x : \tau_2 \vdash e_0 \doteq e_0' : \tau_0}{\Gamma \vdash \lambda x.\, e_0 \doteq \lambda x.\, e_0' : \tau_2 \rightharpoonup \tau_0}$$

$$\textsf{q\_app}: \frac{\Gamma \vdash e_1 \doteq e_1' : \tau_2 \rightharpoonup \tau_0 \quad \Gamma \vdash e_2 \doteq e_2' : \tau_2}{\Gamma \vdash e_1\ e_2 \doteq e_1'\ e_2' : \tau_0}$$

$$\textsf{q\_case}: \frac{\Gamma \vdash e_0 \doteq e_0' : \mathsf{nat} \quad \Gamma \vdash e_1 \doteq e_1' : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_2 \doteq e_2' : \tau}{\Gamma \vdash \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \doteq \mathsf{ncase}(e_0',\ e_1',\ x.\, e_2') : \tau}$$

$$\textsf{q\_case0}: \frac{\Gamma \vdash e_1 \doteq e_1 : \tau}{\Gamma \vdash \mathsf{ncase}(\overline{\mathsf{z}},\ e_1,\ x.\, e_2) \doteq e_1 : \tau} \qquad \textsf{q\_case1}: \frac{\Gamma, x : \mathsf{nat} \vdash e_2 \doteq e_2 : \tau}{\Gamma \vdash \mathsf{ncase}(\overline{\mathsf{s}(n)},\ e_1,\ x.\, e_2) \doteq e_2[\overline{n}/x] : \tau}$$

$$\textsf{q\_eta}: \frac{\Gamma \vdash e \doteq e : \tau_2 \rightharpoonup \tau_0}{\Gamma \vdash e \doteq \lambda x.\, e\ x : \tau_2 \rightharpoonup \tau_0}\ (x \notin \mathrm{dom}(\Gamma))$$

$$\textsf{q\_beta}: \frac{\Gamma, x : \tau_2 \vdash e_0 \doteq e_0 : \tau_0 \quad \Gamma \vdash e_2 \doteq e_2 : \tau_2}{\Gamma \vdash (\lambda x.\, e_0)\ e_2 \doteq e_0[e_2/x] : \tau_0}$$

$$\textsf{q\_subst}: \frac{\Gamma \vdash e_2 \doteq e_2 : \tau \quad \Gamma, x : \tau \vdash e \doteq e' : \tau'}{\Gamma \vdash e[e_2/x] \doteq e'[e_2/x] : \tau'}$$

Figure 4.3: *Axioms for reasoning about program equivalence.*

4.3 Derivable equivalence axioms

We now give a definition of a syntactic reasoning system for deriving program equivalence, and then show that a derivation in this system implies logical equivalence, and hence observational equivalence. The rules are defined in Figure 4.3. As the rules are a syntactic axiomatization of observational equivalence, they are not necessarily complete. We could add several other rules to the system, e.g, we could have added rules for reasoning about commutativity of application over case branches. For simplicity, we have chosen to keep the system relatively small; an example of a larger equational reasoning system is presented in Chapter 5. The Twelf representation of this system can be seen in Figure 4.4.

We will prove soundness of the system by simple induction over derivations. Soundness of the rules q_var to q_case will hence follow directly from the results in the previous

```
sim : exp -> exp -> tp -> type.
sim/num : sim (num N) (num N) nat'.
sim/diverge : sim diverge diverge T.
sim/sym : sim E' E T
        <- sim E E' T.
simm/trans : sim E E'' T
           <- sim E E' T
           <- sim E' E'' T.
sim/cong/lam : sim (lam E0) (lam E0')
                   (T2 => T0)
             <- ({x} sim x x T2
                -> sim (E0 x)
                       (E0' x) T0).
sim/cong/app : sim (app E1 E2)
                   (app E1' E2') T0
             <- sim E1 E1' (T2 => T0)
             <- sim E2 E2' T2.
sim/cong/case :
   sim (case E0 E1 E2)
       (case E0' E1' E2') T
        <- sim E0 E0' nat'
        <- sim E1 E1' T
        <- ({x} sim x x nat'
            -> sim (E2 x) (E2' x) T).
sim/case/z :
   sim (case (num z) E1 E2) E1 T
           <- sim E1 E1 T.
sim/case/s :
   sim (case (num (s N)) E1 E2)
       (E2 (num N)) T
           <- ({x} sim x x nat'
               -> sim (E2 x) (E2 x) T).
sim/eta : sim E (lam [x] app E x)
                (T2 => T0)
        <- sim E E (T2 => T0).
sim/beta : sim (app (lam E0) E2)
               (E0 E2) T0
         <- ({x} sim x x T2
            -> sim (E0 x) (E0 x) T0)
         <- sim E2 E2 T2.
sim/subst : sim (E E2) (E' E2) T0
          <- ({x} sim x x T2
             -> sim (E x) (E' x) T0)
          <- sim E2 E2 T2.
```

Figure 4.4: *Twelf signature for equivalence axiomatization.*

section, and it remains to justify the soundness of the remaining rules.

Lemma 4.22 (Case reduction at zero numeral (logical))**.** *If* $a_1 :: \Gamma \vdash e_1 \sim e_1' : \tau$*, then also* $\Gamma \vdash \mathsf{ncase}(\overline{\mathsf{z}},\ e_1,\ x.\, e_2) \sim e_1 : \tau$*.*

Proof. It suffices to show that for any γ, γ' if $h_1 :: \gamma \sim_\Gamma \gamma'$, then

$$\mathsf{ncase}(\overline{\mathsf{z}},\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2)) \sim_\tau \hat{\gamma}'(e_1).$$

By a_1 and h_1, we get $r_1 :: \hat{\gamma}(e_1) \sim_\tau \hat{\gamma}'(e_1')$. But then by Corollary 4.13 on r_1, it suffices to show

$$\mathsf{ncase}(\overline{\mathsf{z}},\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2)) \approx \hat{\gamma}(e_1).$$

For the "only if" direction, we see that the given derivation can only end in e_case0, implying the existence of a derivation of the right hand side. For the "if" direction, we construct the desired derivation directly by e_case0 on e_num and the given derivation, and we are done. □

Lemma 4.23 (Case reduction at non-zero numeral (logical))**.** *If* $a_1 :: \Gamma, x : \mathsf{nat} \vdash e_2 \sim e_2' : \tau$*, then also* $\Gamma \vdash \mathsf{ncase}(\overline{\mathsf{s}(n')},\ e_1,\ x.\, e_2) \sim e_2[\overline{n'}/x] : \tau$*.*

Proof. It suffices to show that for any γ, γ' if $h_1 :: \gamma \sim_\Gamma \gamma'$, then

$$\mathsf{ncase}(\overline{\mathsf{s}(n')},\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2)) \sim_\tau \hat{\gamma}'(e_2)[\overline{n'}/x].$$

Since we trivially have $\overline{n'} \sim_{\mathsf{nat}} \overline{n'}$, then by h_1 we can construct substitutions $\gamma_0 = \gamma[x \mapsto \overline{n'}]$ and $\gamma_0' = \gamma'[x \mapsto \overline{n'}]$ such that $h_1' :: \gamma_0 \sim_{\Gamma, x:\mathsf{nat}} \gamma_0'$. Then by a_1 and h_1', we get $r_1 :: \hat{\gamma}(e_2)[\overline{n'}/x] \sim_\tau \hat{\gamma}'(e_2')[\overline{n'}/x]$.

But then by Corollary 4.13 on r_1, it suffices to show

$$\mathsf{ncase}(\overline{\mathsf{s}(n')},\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2)) \approx \hat{\gamma}(e_2)[\overline{n'}/x].$$

This follows directly by the same argument as in Lemma 4.22, and we are done. □

Lemma 4.24 (Substitution (logical))**.** *If* $a_1 :: \Gamma \vdash e_2 \sim e_2 : \tau$ *and* $a_2 :: \Gamma, x : \tau \vdash e \sim e' : \tau'$*, then* $\Gamma \vdash e[e_2/x] \sim e'[e_2/x] : \tau'$*.*

Proof. It suffices to show that for any γ, γ' where $h_1 :: \gamma \sim_\Gamma \gamma'$, we have $\hat{\gamma}(e)[e_2/x] \sim \hat{\gamma}'(e')[e_2/x] : \tau'$.

By a_1 and h_1, we can construct substitutions $\gamma_0 = \gamma[x \mapsto e_2]$ and $\gamma_0' = \gamma'[x \mapsto e_2]$ such that $h_1' :: \gamma_0 \sim_{\Gamma, x:\tau} \gamma_0'$. But then by a_2 and h_1', we are done. □

Lemma 4.25 (η-expansion (logical))**.** *If* $\Gamma \vdash e \sim e : \tau_2 \rightharpoonup \tau_0$ *and* $x \notin \mathrm{dom}(\Gamma)$*, then* $\Gamma \vdash e \sim \lambda x.\, e\ x : \tau_2 \rightharpoonup \tau_0$*.*

Proof. Assume $h_1 :: \Gamma \vdash e \sim e : \tau_2 \rightharpoonup \tau_0$. It suffices to show that for any γ, γ', if $h_2 :: \gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim_{\tau_2 \rightharpoonup \tau_0} (\lambda x.\, \hat{\gamma}'(e)\ x)$. To show this, it suffices to show that for any e_2, e_2', if $h_3 :: e_2 \sim_{\tau_2} e_2'$, then $\hat{\gamma}(e)\ e_2 \sim_{\tau_0} (\lambda x.\, \hat{\gamma}'(e)\ x)\ e_2'$. Note that x does not get replaced by $\hat{\gamma}'$, exactly because $x \notin \Gamma$.

By Lemma 4.16 on h_1, we get $h_4 :: \Gamma \vdash \lambda x.\, e\ x \sim \lambda x.\, e\ x : \tau_2 \rightharpoonup \tau_0$. By h_2 and h_4, we then have $h_5 :: \lambda x.\, \hat{\gamma}(e)\ x \sim_{\tau_2 \rightharpoonup \tau_0} \lambda x.\, \hat{\gamma}'(e)\ x$. By Definition 4.1 and h_3, this gives us $h_6 :: (\lambda x.\, \hat{\gamma}(e)\ x)\ e_2 \sim_{\tau_0} (\lambda x.\, \hat{\gamma}'(e)\ x)\ e_2'$.

The right hand side of h_6 is on the desired form, but the left is not. By Corollary 4.13 on h_6, it thus suffices to show $\forall v.\, \hat{\gamma}(e)\ e_2 \Downarrow v \Leftrightarrow (\lambda x.\, \hat{\gamma}(e)\ x)\ e_2 \Downarrow v$:

For the forward direction, assume $\mathcal{E}_l :: \hat{\gamma}(e)\ e_2 \Downarrow v$. But then by e_app on e_lam and $\mathcal{E}_l$, we get $(\lambda x.\, \hat{\gamma}(e)\ x)\ e_2 \Downarrow v$.

For the reverse direction, assume $\mathcal{E}_r :: (\lambda x.\, \hat{\gamma}(e)\ x)\ e_2 \Downarrow v$. This must end in an application of e_app, implying that we have $\mathcal{E}_{r1} :: \lambda x.\, \hat{\gamma}(e)\ x \Downarrow \lambda x.\, \hat{\gamma}(e)\ x$ and $\mathcal{E}_{r2} :: \hat{\gamma}(e)\ e_2 \Downarrow v$. But $\mathcal{E}_{r2}$ is exactly our goal, and hence we are done. □

Lemma 4.26 (β-value-reduction (logical))**.** *If we have* $\Gamma \vdash \lambda x.\, e_0 \sim \lambda x.\, e_0' : \tau_2 \rightharpoonup \tau_0$ *and* $\Gamma \vdash e_2 \sim e_2' : \tau_2$*, where* $x \notin \Gamma$*, then* $\Gamma \vdash (\lambda x.\, e_0)\ e_2 \sim e_0'[e_2'/x] : \tau_0$.

Proof. Assume $h_1 :: \Gamma \vdash \lambda x.\, e_0 \sim \lambda x.\, e_0' : \tau_2 \rightharpoonup \tau_0$ and $h_2 :: \Gamma \vdash e_2 \sim e_2' : \tau_2$. It suffices to show that for any γ, γ' if $h_3 :: \gamma \sim_\Gamma \gamma'$, then $(\lambda x.\, \hat{\gamma}(e_0))\ \hat{\gamma}(e_2) \sim_{\tau_0} \hat{\gamma}'(e_0')[\hat{\gamma}'(e_2')/x]$ (substitutions commute as $x \notin \Gamma$).

By Lemma 4.15 on h_1 and h_2, we get $h_4 :: \Gamma \vdash (\lambda x.\, e_0)\ e_2 \sim (\lambda x.\, e_0')\ e_2' : \tau_0$. By h_4 and h_3, we then get $h_5 :: (\lambda x.\, \hat{\gamma}(e_0))\ \hat{\gamma}(e_2) \sim_{\tau_0} (\lambda x.\, \hat{\gamma}'(e_0'))\ \hat{\gamma}'(e_2')$.

The left expression of h_5 is on the correct form, but the right is not. By Corollary 4.13, it thus suffices to show $\hat{\gamma}'(e_0')[\hat{\gamma}'(e_2')/x] \Downarrow v \Leftrightarrow \forall v.\, (\lambda x.\, \hat{\gamma}'(e_0'))\ \hat{\gamma}'(e_2') \Downarrow v$: The forward direction follows by an application of e_app on e_lam and the assumption derivation. The other direction follows from the fact that the assumption derivation must end in e_app, implying the right hand side, and we are done. □

We can now prove the main theorem, namely that our equational reasoning system is sound, since it implies logical (and hence also observational) equivalence:

Theorem 4.27 (Soundness)**.** *If* $\mathcal{Q} :: \Gamma \vdash e \doteq e' : \tau$*, then* $\Gamma \vdash e \sim e' : \tau$.

Proof. By induction on the derivation $\mathcal{Q}$:

The cases for the rules q_var, q_num, q_diverge, q_sym, q_trans, q_lam, q_app and q_case are handled by applying one of Lemmas 4.18, 4.19, 4.9, 4.10, 4.16, 4.15 and 4.17, respectively, to the result of the induction hypothesis applied to all subderivations.

The cases for q_case0, q_case1, q_eta, q_beta and q_subst are handled by applying one of Lemmas 4.22, 4.23, 4.25, 4.26 and 4.24, respectively, to the result of the induction hypothesis applied to all subderivations. □

4.4 Formalization

The Twelf formalization of the binary logical relation will follow the same structure as in Chapter 3, and will proceed by representing and reasoning about the logical relation in an auxiliary assertion logic. The formalization is, however, complicated by the combination of several factors. We will give an overview of these in the following, and will then show how to solve the associated problems.

First of all, a number of lemmas make use of case analysis on evaluation derivations in their proofs, notably Lemma 4.12 (Closure under weak equivalence) and Lemma 4.17 (Congruence at case). Lemma 4.12 is very similar to Lemma 3.5 (Closure under weak head expansion) from Chapter 3, and we could in principle "postpone" the critical parts of the proof in a similar way by extending the embedded evaluation judgment with suitable axioms, thus avoiding the need to add case analysis on derivations to the assertion logic. To do this, we would essentially need to define a syntactic axiomatization

of the property $e \approx e'$, for example:

$$\text{Weak equivalence (syntactic):} \quad \mathcal{W} \quad :: \quad \boxed{e \mathrel{\dot{\approx}} e'}$$

$$\text{w\_id:}\ \frac{}{e \mathrel{\dot{\approx}} e} \qquad \text{w\_sym:}\ \frac{e \mathrel{\dot{\approx}} e'}{e' \mathrel{\dot{\approx}} e} \qquad \text{w\_trans:}\ \frac{e \mathrel{\dot{\approx}} e' \quad e' \mathrel{\dot{\approx}} e''}{e \mathrel{\dot{\approx}} e''}$$

$$\text{w\_beta:}\ \frac{}{(\lambda x.\, e_0)\ e_2 \mathrel{\dot{\approx}} e_2[e_2/x]} \qquad \text{w\_congapp:}\ \frac{e \mathrel{\dot{\approx}} e'}{e\ e_2 \mathrel{\dot{\approx}} e'\ e_2}$$

$$\text{w\_congcase:}\ \frac{e \mathrel{\dot{\approx}} e'}{\mathsf{ncase}(e,\ e_1,\ x.\, e_2) \mathrel{\dot{\approx}} \mathsf{ncase}(e',\ e_1,\ x.\, e_2)}$$

$$\text{w\_case0:}\ \frac{}{\mathsf{ncase}(\overline{\mathsf{z}},\ e_1,\ x.\, e_2) \mathrel{\dot{\approx}} e_1} \qquad \text{w\_case1:}\ \frac{}{\mathsf{ncase}(\overline{\mathsf{s}(n)},\ e_1,\ x.\, e_2) \mathrel{\dot{\approx}} e_2[\overline{n}/x]}$$

We could then easily justify that the following rules are admissible for the evaluation judgment:

$$\text{e\_weq1:}\ \frac{e \mathrel{\dot{\approx}} e' \quad e \Downarrow v}{e' \Downarrow v} \qquad \text{e\_weq2:}\ \frac{e \mathrel{\dot{\approx}} e' \quad e' \Downarrow v}{e \Downarrow v}$$

This would be sufficient for formalizing Lemma 4.12, but changing the premise to $e \mathrel{\dot{\approx}} e'$ instead of $e \approx e'$. However, the approach is not general enough for the formalization of Lemma 4.17. In the proof for the case $\tau = \mathsf{nat}$, we need to show that if $\mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \Downarrow \overline{n}$ then we have an evaluation of $\mathsf{ncase}(e_0',\ e_1',\ x.\, e_2') \Downarrow \overline{n}$, given that the subexpressions of the two case constructs are pairwise logically equivalent. This is proved by reasoning about the possible ways the given derivation could have been constructed, obtaining evaluation derivations for the subexpressions in each case, which we then convert by the logical equivalence of the subexpressions. Since proofs of logical equivalence must inherently live in the assertion logic, we cannot express this reasoning axiomatically as an extension of the rules for the evaluation judgment. If we tried to do this, we would find that the rules needed would have cutful assertion logic proofs as premises, which would be unaffected by cut elimination on the "outer" assertion logic proof.

It therefore seems that there is no way around adding case analysis on derivations to the assertion logic, very much like the situation we had when we added a case construct to the object language in Section 3.4, although case analysis on natural numbers was sufficient for that particular development. We will come back to how this is accomplished later, but first we need to identify another challenge of the formalization.

When doing case analysis on derivations, each proof case will be provided with a set of equalities between expressions, corresponding to the specific case. These equality proofs are then used to derive the goal we want to prove, e.g. by converting some other evaluation derivation, or by proving that the case is actually impossible. The assertion logic will therefore need to provide a notion of equality which also supports identifying absurd (i.e., impossible) equalities. For example, suppose we have a derivation $\mathcal{E}$ ::

$e_1\ e_2 \Downarrow v$ for some e_1, e_2 and v, and that we want to prove a goal C by using the fact that only e_app could have been used to derive $\mathcal{E}$. When doing case analysis within the assertion logic, we have to cover *all* cases, including those that are impossible. This is normally avoidable in meta-level proofs, as the coverage checker can rule out most impossible cases by unification. Returning to the example, we should therefore be able to derive C from, e.g., the absurd case where we have $e_1\ e_2 = \lambda x.\, e_0$. This suggests that the assertion logic should also provide some notion of *falsehood*, i.e., from an absurd equality, anything follows.

In Section 3.4, we added an equality conversion axiom to the embedded evaluation judgment in order to enable conversion of evaluation derivations. This was needed since we cannot pattern match on equality proofs within the assertion logic, and hence cannot rely on unification on meta-variables when "applying" equalities. This approach is problematic when combined with case analysis, since an extra derivation rule will also add an extra case to all proofs that work by case analysis on derivations. The only way to cover such a case is effectively by proving admissibility of the equality conversion rule, but our inability to do so within the assertion logic was the motivation to add the rule to begin with. How do we solve this apparent circularity?

It is possible to solve all the problems mentioned above, but we need quite a bit of extra machinery. In the following, we will introduce a new logic to be used exclusively for embedding judgments and for reasoning about equality and falsehood.

4.4.1 Data representation logic

We now introduce a new auxiliary logic, henceforth referred to as the *data representation logic*, or *representation logic* for short. The logic will be very restricted, and is essentially a logic for representing judgments with explicit equality proofs. We will use it to represent all the embedded judgments that we need in assertion logic proofs, to represent equalities between objects, and to represent falsehood. A subset of the logic for the embedding of the evaluation judgment for $\lambda_{\mathsf{cbn}}^{\rightarrow,\mathsf{nat}}$ is defined in Figure 4.5. We follow the convention of writing all data formulas in banana brackets $(\!|\cdot|\!)$. We have only presented a representative subset of the rules concerned with equality, as the remaining rules are very similar in structure. For each equality formula, we have standard reflexivity, symmetry and transitivity rules, as well as a long range of rules for proving $(\!|\,\mathsf{void}\,|\!)$ from absurd equalities, like for example the rule dqe_app_num. Since we cannot give a short definition of what it means for two objects to be distinct, we have to define a rule for each possible combination of distinct head constructors. We will assume that all possible combinations are defined; by symmetry we only need 15 rules, e.g., by dqe_sym we can derive the symmetric version of dqe_app_lam, and we thus do not need a rule dqe_lam_app.

The logic defines an alternative formulation of the evaluation judgment; it is equivalent to the original, in the sense that we can show the following:

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

Data derivations: $\mathcal{D} :: \boxed{\Vdash D}$:

Evaluation embedding:

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

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

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

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

Equality and falsehood (representative subset):

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

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

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

$$\mathsf{dqe\_cvrs\_lam}: \frac{\Vdash (\!|\,\lambda x.\,e_0 \stackrel{\star}{=} \lambda x'.\,e_0'\,|\!)}{\Vdash (\!|\,x.\,e_0 \stackrel{\star}{=}_2 x'.\,e_0'\,|\!)}$$

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

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

(dqe_cvrs_case1 and dqe_cvrs_case2 defined similarly.)

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

$$\mathsf{dqe2\_id}: \frac{}{\Vdash (\!|\,x.\,e_0 \stackrel{\star}{=}_2 x.\,e_0\,|\!)} \qquad \cdots$$

(Similar rules as the above for $\stackrel{\star}{=}_2$ and $\stackrel{\star}{=}_{\mathsf{nat}}$.)

Figure 4.5: *Data representation logic.*

Lemma 4.28 (Soundness)**.** *The following holds:*

- *If* $\Vdash (\!| \mathsf{void} |\!)$*, then false; i.e.,* $\not\Vdash (\!| \mathsf{void} |\!)$.
- *If* $\Vdash (\!| e \stackrel{\star}{=} e' |\!)$*, then* $e = e'$.
- *If* $\Vdash (\!| n \stackrel{\star}{=}_{\mathsf{Nat}} n' |\!)$*, then* $n = n'$.
- *If* $\Vdash (\!| x.\, e \stackrel{\star}{=}_2 x'.\, e' |\!)$*, then* $x.\, e = x'.\, e'$.
- *If* $\Vdash (\!| e \Downarrow v |\!)$*, then* $e \Downarrow v$.

Proof sketch. By mutual induction on derivations. □

Lemma 4.29 (Completeness)**.** *The following holds:*

- *For any* e, $\Vdash (\!| e \stackrel{\star}{=} e |\!)$.
- *For any* n, $\Vdash (\!| n \stackrel{\star}{=}_{\mathsf{Nat}} n |\!)$.
- *For any* $x.\, e$, $\Vdash (\!| x.\, e \stackrel{\star}{=}_2 x.\, e |\!)$.
- *If* $e \Downarrow v$*, then* $\Vdash (\!| e \Downarrow v |\!)$.

Proof sketch. The first three are trivial by the identity rules. The last follows by induction on the derivation, and by the first three lemmas. □

It may not be immediately clear what we gain by this alternative formulation of the evaluation judgment. The central difference between the two representations is that no introduction rules impose any restrictions on the involved expressions. Instead, equality restrictions are ensured entirely through syntactic proofs. This enables full equality reasoning, as long as we can do case analysis on representation logic derivations. For example, we can now show the following, without relying on meta-level equality or induction:

Lemma 4.30. *If* $\mathcal{D}_1 :: \Vdash (\!| e_1 \stackrel{\star}{=} e_1' |\!)$ *and* $\mathcal{D}_2 :: \Vdash (\!| e_2 \stackrel{\star}{=} e_2' |\!)$ *and* $\mathcal{D}_3 :: \Vdash (\!| e_1 \Downarrow e_2 |\!)$*, then also* $\Vdash (\!| e_1' \Downarrow e_2' |\!)$.

Proof sketch. By case analysis on $\mathcal{D}_3$. In each case, we apply dqe_sym and dqe_trans on the obtained equality proofs and reapply the original rule. □

Importantly, the proof does not depend on a separate equality conversion axiom for the evaluation judgment—we only need to cover the same number of cases as in the original definition.

We can also show that falsehood implies the existence of any evaluation derivation, again without relying on meta-level unification to refute impossible cases:

Lemma 4.31. *For any e, e', if $\Vdash (\!| \mathsf{void} |\!)$, then $\Vdash (\!| e \Downarrow e' |\!)$.*

Proof. By dqe_void (twice), we obtain $\Vdash (\!| e \stackrel{\star}{=} \overline{\mathsf{z}} |\!)$ and $\Vdash (\!| e' \stackrel{\star}{=} \overline{\mathsf{z}} |\!)$. But then by de_num, we are done. □

This means that we can do reasoning like the following:

Example 4.32. *If $\mathcal{D} :: \Vdash (\!| e_1\ e_2 \Downarrow v |\!)$, then there exists an expression $x.\, e_0$ and derivations $\mathcal{D}_1 :: \Vdash (\!| e_1 \Downarrow \lambda x.\, e_0 |\!)$ and $\mathcal{D}_2 :: \Vdash (\!| e_2[e_2/x] \Downarrow v |\!)$.*

Proof sketch. By case analysis on $\mathcal{D}$. We show the proof for the only possible case, and the proof for one of the absurd cases:

- Case $\mathcal{D}$ ends in de_app. Then there exists expressions $x.\, e_0$, e_1', e_2' and v', and we have derivations

  $$\begin{aligned}
  &\mathcal{D}_{11} :: \Vdash (\!| e_1' \Downarrow \lambda x.\, e_0 |\!),\\
  &\mathcal{D}_{12} :: \Vdash (\!| e_0[e_2'/x] \Downarrow v' |\!),\\
  &\mathcal{D}_{13} :: \Vdash (\!| e_1\ e_2 \stackrel{\star}{=} e_1'\ e_2' |\!),\\
  &\mathcal{D}_{14} :: \Vdash (\!| v \stackrel{\star}{=} v' |\!).
  \end{aligned}$$

  By the equality rules dqe_id, dqe2_id, dqe_cvrs_app1, dqe_cvrs_app2, dqe_subst, dqe_trans and dqe_sym, we obtain

  $$\begin{aligned}
  &\mathcal{D}_1' :: \Vdash (\!| e_1' \stackrel{\star}{=} e_1 |\!),\\
  &\mathcal{D}_2' :: \Vdash (\!| \lambda x.\, e_0 \stackrel{\star}{=} \lambda x.\, e_0 |\!),\\
  &\mathcal{D}_3' :: \Vdash (\!| e_0[e_2'/x] \stackrel{\star}{=} e_0[e_2/x] |\!),\\
  &\mathcal{D}_4' :: \Vdash (\!| v' \stackrel{\star}{=} v |\!).
  \end{aligned}$$

  So, by Lemma 4.30 on $\mathcal{D}_{11}$, $\mathcal{D}_{12}$ and the above, we are done.

- Case $\mathcal{D}$ ends in de_num. Then we have a derivation of $\Vdash (\!| e_1\ e_2 \stackrel{\star}{=} \overline{n} |\!)$ for some n. By dqe_app_num, we can derive $\Vdash (\!| \mathsf{void} |\!)$, and hence by Lemma 4.31, we are done. □

The Twelf signature for the representation logic can be seen in Figure 4.6. Only some representative examples are shown, as the definitions can be mechanically derived from the definition of the original judgment (in fact, most of the representation logic *was* generated by ad-hoc automation when we developed the formalization, including soundness and completeness proofs). The full definition can be seen in Appendix B.4.

It remains to enable case analysis on representation logic derivations in assertion logic proofs. We will demonstrate how that is accomplished in the following.

```
dform : type. %name dform D.
data : dform -> type. %name data DP.

% Formulas
@void : dform.
@eval : exp -> exp -> dform.
@eq-exp : exp -> exp -> dform.
@eq-exp2 : (exp -> exp) -> (exp -> exp)
                        -> dform.
@eq-nat : nat -> nat -> dform.

% Evaluation judgment
@eval/lam :
   data (@eq-exp X1 (lam E0))
   -> data (@eq-exp X2 (lam E0))
   -> data (@eval X1 X2).
@eval/num :
   data (@eq-exp X1 (num N0))
   -> data (@eq-exp X2 (num N0))
   -> data (@eval X1 X2).
@eval/app :
   data (@eval E1 (lam E0))
   -> data (@eval (E0 E2) Ev)
   -> data (@eq-exp X1 (app E1 E2))
   -> data (@eq-exp X2 Ev)
   -> data (@eval X1 X2).
%{ ... remaining rules elided ... }%
```

```
% Equality rules, expressions
@eq-exp/void : data @void
            -> data (@eq-exp X1 X2).
@eq-exp2/id : data (@eq-exp2 X X).
@eq-exp/id : data (@eq-exp X X).
@eq-exp/sym : data (@eq-exp X1 X2)
           -> data (@eq-exp X2 X1).
@eq-exp/trans : data (@eq-exp X1 X2)
             -> data (@eq-exp X2 X3)
             -> data (@eq-exp X1 X3).
@eq-exp/subst :
   data (@eq-exp2 E E')
   -> data (@eq-exp E2 E2')
   -> data (@eq-exp (E E2) (E' E2')).
@eq-exp/cvrs-lam :
   data (@eq-exp (lam E0) (lam E1))
   -> data (@eq-exp2 (E0) (E1)).
@eq-exp/cvrs-app1 :
   data (@eq-exp (app E1 E2)
                 (app E3 E4))
   -> data (@eq-exp (E1) (E3)).
@eq-exp/cvrs-app2 :
   data (@eq-exp (app E1 E2)
                 (app E3 E4))
   -> data (@eq-exp (E2) (E4)).

%{ ... remaining rules elided ... }%
```

Figure 4.6: *Twelf signature for the representation logic*

4.4.2 Assertion logic

The assertion logic that we need for formalizing the proofs for logical equivalence follows the same structure as earlier formulations. We will need to quantify over expressions, natural numbers and derivations of representation logic proofs. Additionally, we need to be able to do case analysis on the latter, although not in full generality. We will never need to do case analysis on equality proofs, since their proof structure do not carry any useful information—the existence of an equality proof is the only thing we care about. We can therefore restrict the case analysis rules such that they are only concerned with derivations of proofs of evaluation formulas (i.e., derivations of judgments of the form $\Vdash (\!| e_1 \Downarrow e_2 |\!)$).

We will add case analysis to the assertion logic in a similar way as we did in Section 3.5, by adding a structural predicate on representation logic derivations. The structural predicate has the form $\mathsf{Data}^+(\mathcal{D} : D)$, and is actually *binary*; it is a predicate on a derivation $\mathcal{D}$, but annotated with a data formula D denoting what $\mathcal{D}$ derives. We need this since the representation logic judgment is essentially a *family* of sorts, indexed by

$$
\begin{array}{llcll}
\text{Allowance of cut:} & c & :: & \mathsf{Allow} & ::= \quad \mathsf{cut} \mid \mathsf{cf} \\
\text{Formulas:} & A, B, \ldots & :: & \mathsf{Form} & ::= \quad \top \mid \forall x : \mathsf{Exp}.\, A \mid \forall y : \mathsf{Nat}.\, A \mid \exists d : (\Vdash D).\, A \\
 & & & & \quad \mid A \vee B \mid A \wedge B \mid A \supset B \mid \mathsf{Data}^{+}(d : D) \\
\text{Parameters:} & \Xi & :: & \mathsf{Parms} & ::= \quad \cdot \mid \Xi, x : \mathsf{Exp} \mid \Xi, b : (\mathsf{Exp})\mathsf{Exp} \\
 & & & & \quad \mid \Xi, u : \mathsf{Nat} \mid \Xi, \mathcal{D} : (\Vdash D) \\
\text{Assumptions:} & \Delta & :: & \mathsf{Assm} & ::= \quad \cdot \mid \Delta, A \\
\text{Proof judgment:} & \mathcal{S} & :: & \boxed{\Xi | \Delta \vdash^{c}_{\Sigma} A} &
\end{array}
$$

$$
\begin{pmatrix} \text{Rules } \mathsf{ax}, \mathsf{cut}, \mathsf{topR}, \mathsf{impR}, \mathsf{andR}, \mathsf{orR1}, \mathsf{orR2}, \mathsf{alleR}, \mathsf{andL1}, \\ \mathsf{andL2}, \mathsf{orL}, \mathsf{impL} \text{ and } \mathsf{alleL} \text{ are defined as in Figure 3.4} \end{pmatrix}
$$

$$
\mathsf{exidR}^{\mathcal{D}}\colon \frac{\Xi | \Delta \vdash^{c}_{\Sigma} A[\mathcal{D}/d] \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner \mathcal{D} \urcorner \Leftarrow \ulcorner \Vdash D \urcorner}{\Xi | \Delta \vdash^{c}_{\Sigma} \exists d : (\Vdash D).\, A} \qquad \mathsf{allnR}\colon \frac{\Xi, y : \mathsf{Nat} | \Delta \vdash^{c}_{\Sigma} A}{\Xi | \Delta \vdash^{c}_{\Sigma} \forall y : \mathsf{Nat}.\, A}
$$

$$
\mathsf{exidL}\colon \frac{\Xi, d' : (\Vdash D) | \Delta, \exists d : (\Vdash D).\, A, A[d'/d] \vdash^{c}_{\Sigma} C}{\Xi | \Delta, \exists d : (\Vdash D).\, A \vdash^{c}_{\Sigma} C}
$$

$$
\mathsf{allnL}^{n}\colon \frac{\Xi | \Delta, \forall y : \mathsf{Nat}.\, A, A[n/y] \vdash^{c}_{\Sigma} C \quad \ulcorner \Xi \urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner n \urcorner \Leftarrow \ulcorner \mathsf{Nat} \urcorner}{\Xi | \Delta, \forall y : \mathsf{Nat}.\, A \vdash^{c}_{\Sigma} C}
$$

$$
\begin{pmatrix} \text{Omitted for brevity: Right-rules for each of } \mathsf{de\_lam}, \mathsf{de\_num}, \\ \mathsf{de\_app}, \mathsf{de\_case0} \text{ and } \mathsf{de\_case1} \text{ and the left-rule } \mathsf{dataL\_ev}. \text{ See} \\ \text{Figure 4.8 for example.} \end{pmatrix}
$$

Parameter encoding:

$$
\begin{array}{lcl}
\ulcorner . \urcorner & = & \cdot \\
\ulcorner \Xi, x : \mathsf{Exp} \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{x}_x : \mathtt{exp} \\
\ulcorner \Xi, b : (\mathsf{Exp})\mathsf{Exp} \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{b}_b : (\mathtt{exp} \to \mathtt{exp}) \\
\ulcorner \Xi, y : \mathsf{Nat} \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{y}_y : \mathtt{nat} \\
\ulcorner \Xi, d : (\Vdash D) \urcorner & = & \ulcorner \Xi \urcorner, \mathtt{d}_d : \mathtt{data}\, \ulcorner D \urcorner
\end{array}
$$

Figure 4.7: *A representative subset of the assertion logic for the formalization of* $\lambda^{\rightarrow,\mathsf{nat}}_{\mathsf{cbn}}$.

formulas: The formula D uniquely specifies this index, which we need in the left-rule to discriminate evaluation derivations from equality derivations.

We have attempted to present a subset of the assertion logic in Figure 4.7 and Figure 4.8, although the presence of derivations-in-judgments combined with large parameter and context extensions results in a somewhat cluttered presentation.

The Twelf signature for the assertion logic can be seen in Figure 4.9. We have elided the definitions related to the standard logical connectives, and show only some representative cases of the new rules related to case analysis. The full definition can be seen in Appendix B.5.

To prove cut admissibility, we need to strengthen the induction hypothesis in the

$$\textsf{dataR\_eapp:}\ \frac{\Xi|\Delta \vdash^{c}_{\Sigma} \mathsf{Data}^{+}(\mathcal{D}_1 : (\!| e_1' \Downarrow \lambda x.\, e_0 |\!)) \quad \Xi|\Delta \vdash^{c}_{\Sigma} \mathsf{Data}^{+}(\mathcal{D}_2 : (\!| e_0[e_2'/x] \Downarrow v |\!)) \quad \ulcorner\Xi\urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner\mathcal{D}_1\urcorner \Leftarrow \ulcorner\Vdash (\!| e_1' \Downarrow \lambda x.\, e_0 |\!)\urcorner \quad \ulcorner\Xi\urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner\mathcal{D}_3\urcorner \Leftarrow \ulcorner\Vdash (\!| e_1 \stackrel{\star}{=} e_1'\ e_2' |\!)\urcorner \quad \ulcorner\Xi\urcorner \vdash^{\mathsf{LF}}_{\Sigma} \ulcorner\mathcal{D}_4\urcorner \Leftarrow \ulcorner\Vdash (\!| e_2 \stackrel{\star}{=} v |\!)\urcorner}{\Xi|\Delta \vdash^{c}_{\Sigma} \mathsf{Data}^{+}\left(\begin{array}{l}\textsf{de\_app:}\ \dfrac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \mathcal{D}_3 \quad \mathcal{D}_4}{\Vdash (\!| e_1' \Downarrow \lambda x.\, e_0 |\!) \quad \Vdash (\!| e_0[e_2'/x] \Downarrow v |\!) \quad \Vdash (\!| e_1 \stackrel{\star}{=} e_1'\ e_2' |\!) \quad \Vdash (\!| e_2 \stackrel{\star}{=} v |\!)}{\Vdash (\!| e_1 \Downarrow e_2 |\!)} : (\!| e_1 \Downarrow e_2 |\!)\end{array}\right)}$$

$$\textsf{dataL\_ev:}\ \frac{\begin{array}{l}\overset{\cdots}{\Xi},\ x_v{:}\mathsf{Exp},\ x_1{:}\mathsf{Exp},\ x_2{:}\mathsf{Exp},\ b_0{:}(\mathsf{Exp})\mathsf{Exp},\ d_1{:}(\Vdash (\!| x_1 \Downarrow \lambda x.\, b_0[x] |\!)),\ d_2{:}(\Vdash (\!| b_0[x_2] \Downarrow x_v |\!)) \\ \quad d_3{:}(\Vdash (\!| e_1 \stackrel{\star}{=} x_1\ x_2 |\!)),\ d_4{:}(\Vdash (\!| e_2 \stackrel{\star}{=} x_v |\!))|\Delta,\ \mathsf{Data}^{+}(d_1 : (\!| x_1 \Downarrow \lambda b_0 |\!)),\ \mathsf{Data}^{+}(d_2 : (\!| b_0[x_2] \Downarrow x_v |\!)) \vdash^{c}_{\Sigma} C\end{array}}{\Xi|\Delta, \mathsf{Data}^{+}(\mathcal{D} : (\!| e_1 \Downarrow e_2 |\!)) \vdash^{c}_{\Sigma} C}$$

Figure 4.8: *Structural predicate rules: A representative example of a right-rule and the left-rule with one premise shown.*

```
% Formulas
form : type.
%{...}%
data+ : data D -> form.

% Judgments and assumptions
allow : type.
cutful : allow.
cutfree : allow.
hyp : form -> type.
conc : allow -> form -> type.

%{ ... standard rules ... }%

% Right rules for case analysis
data+/@eval/lam : conc V (data+ (@eval/lam Q1 Q2)).
data+/@eval/num : conc V (data+ (@eval/num Q1 Q2)).
data+/@eval/app : conc V (data+ DP1)
                -> conc V (data+ DP2)
                -> conc V (data+ (@eval/app DP1 DP2 Q1 Q2)).

% The left rule; only the two premises for the lambda and
% application cases are shown.
data+/@eval/l :
     (%{lam case}%
     {E0}
     {q1:data (@eq-exp X1 (lam E0))}{q2:data (@eq-exp X2 (lam E0))}
       conc V C)
  -> (%{num ...}%)
  -> (%{app}%
      {Ev}{E1}{E2}{E0}
      {dp1:data (@eval E1 (lam E0))}{dp2:data (@eval (E0 E2) Ev)}
      {h1:hyp (data+ dp1)}{h2:hyp (data+ dp2)}
      {q1:data (@eq-exp X1 (app E1 E2))}{q2:data (@eq-exp X2 Ev)}
         conc V C)
  -> (%{case/0 ...}%) -> (%{case/1 ...}%)
  -> hyp (data+ (DP : data (@eval X1 X2))) -> conc V C.
```

Figure 4.9: *Twelf signature for the assertion logic.*

same way as we did in Section 3.5.1, replacing natural numbers by representation logic derivations (i.e., derivations $\mathcal{D} :: \Vdash D$ for some D) in the formula measure.

4.4.3 Formalizing the logical relation

To keep the formalization uncluttered, we introduce some convenient abbreviations as follows:

```
% Bi-implication
<==> : form -> form -> form
  = [f1][f2] (f1 ==> f2) /\ (f2 ==> f1). %infix left 1 <==>.

% Evaluations with structure
#eval : exp -> exp -> form
  = [e1][e2] existsd [dp:data (@eval e1 e2)] data+ dp.
```

The first is just a shorthand for writing bi-implications. The second is the canonical way that we are going to represent derivations, namely by an existence proof of a representation logic derivation, together with a proof of the well-formedness of said derivation.

We can now formalize the logical relation as a relation between types and formulas with two bound expressions as follows:

```
lr : tp -> (exp -> exp -> form) -> type.
lr/nat' : lr nat' ([e1][e2]
                     foralln [n] (#eval e1 (num n) <==> #eval e2 (num n))).
lr/=> : lr (T2 => T0) ([e][e']
                         foralle [e2] foralle [e2']
                           R2 e2 e2' ==> R1 (app e e2) (app e' e2'))
      <- lr T2 R2
      <- lr T0 R0.
```

The formalization of the congruence lemmas is relatively straightforward, and will therefore not be covered in detail here. We will give a brief overview of the formalization in Section 4.6. Since we have the ability to do case analysis on derivations within assertion logic proofs, the formalization of the congruence proofs actually follows the paper proofs closely in structure, although they are a lot more explicit with regards to equality reasoning.

We do, however, run into problems with regards to how open logical equivalence is represented in Twelf, which we will cover in the following.

4.5 Context separation

A surprisingly complicated aspect of the formalization turns out to be the representation of open logical equivalence. The preferred way of representing contexts in Twelf is by

reusing the LF context, so this ought to be the natural approach for the representation of open logical equivalence as well. Unfortunately, Twelf is unable to represent the explicit closing substitutions (γ and γ') in a way that is in direct correspondence to the original definition. We will illustrate the problem by an example: assume that e and e' are expressions with free variables among $x_1, \ldots, x_n$. In the LF encodings $\ulcorner e \urcorner$ and $\ulcorner e' \urcorner$, an occurrence of a variable x is represented by an LF variable $\mathtt{x}_x$. Importantly, if some variable x occurs free in both e and e', it is represented by *the same* LF variable in both $\ulcorner e \urcorner$ and $\ulcorner e' \urcorner$. Assuming $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, how do we then represent the open logical equivalence $\Gamma \vdash e \sim e' : \tau$? The central question is how we represent the context Γ as an LF context. Assuming that $\ulcorner \sim_\tau \urcorner$ is the LF representation of the formula that characterizes the logical relation at type τ, a first attempt at defining a representation of open logical equivalence is as follows:

$$\ulcorner . \urcorner = .$$
$$\ulcorner \Gamma, x : \tau \urcorner = \ulcorner \Gamma \urcorner, \mathtt{x}_x : \mathtt{exp}, \mathtt{s}_x : \mathtt{conc}\ \mathtt{cutful}\ (\ulcorner \sim_\tau \urcorner\ \mathtt{x}_x\ \mathtt{x}_x)$$

Unfortunately, this is not adequate at all, since we only introduce a single hypothetical expression, $\mathtt{x}_x$, for each assumption $x : \tau$. This means that we can only substitute the same expression for the different occurrences of x in e and e', thus failing to represent the distinct substitutions γ, γ'.

Our notion of variables in the definition of open logical equivalence does not seem to correspond to the notion of variables in LF. Specifically, in our paper definition, equal variables stand for possibly distinct expressions depending on the context in which they occur, whereas in LF, equal variables stand for *identical* expressions. We cannot work around this in the definition of the translation function, i.e., we cannot define that $\ulcorner e \sim_\tau e' \urcorner = \ulcorner \sim_\tau \urcorner \ulcorner e \urcorner^{\mathtt{L}} \ulcorner e \urcorner^{\mathtt{R}}$, where $\ulcorner \cdot \urcorner^{\mathtt{L}}$ and $\ulcorner \cdot \urcorner^{\mathtt{R}}$ are defined as $\ulcorner \cdot \urcorner$, except that $\ulcorner x \urcorner^{\mathtt{L}} = \mathtt{x}_x^{\mathtt{L}}$ and $\ulcorner x \urcorner^{\mathtt{R}} = \mathtt{x}_x^{\mathtt{R}}$ for variables x. This would not be a *compositional* encoding, however, as illustrated by the following counter-example: If the encoding was compositional, then we should have $\ulcorner x \sim_\tau e' \urcorner[\ulcorner e \urcorner / \ulcorner x \urcorner] = \ulcorner e \sim_\tau e' \urcorner$. However, since $\mathtt{x}_x \neq \mathtt{x}_x^{\mathtt{L}}$, then

$$\begin{aligned}
\ulcorner x \sim_\tau e' \urcorner[\ulcorner e \urcorner / \ulcorner x \urcorner] &= (\ulcorner \sim_\tau \urcorner\ \mathtt{x}_x^{\mathtt{L}}\ \ulcorner e' \urcorner)[\ulcorner e \urcorner / \mathtt{x}_x] \\
&= \ulcorner \sim_\tau \urcorner\ \mathtt{x}_x^{\mathtt{L}}\ \ulcorner e' \urcorner \\
&\neq \ulcorner \sim_\tau \urcorner \ulcorner e \urcorner \ulcorner e' \urcorner \\
&= \ulcorner e \sim_\tau e' \urcorner.
\end{aligned}$$

It turns out that we can formulate open logical equivalence in another way, which has a compositional representation in Twelf:

Definition 4.33 (Open logical equivalence, separated context)**.** For any expressions e, e', type τ and context $\Gamma^\star = (x_1, x_1') : \tau_1, \ldots, (x_n, x_n') : \tau_n$, we write $\Gamma^\star \vdash^\star e \sim e' : \tau$ iff

1. For every i, $x_i \neq x_i'$, and
2. $\mathrm{FV}(e) \subseteq \{x_i\}_i$, $\mathrm{FV}(e') \subseteq \{x_i'\}_i$, and

3. For any substitution γ with $\mathrm{dom}(\gamma) = \{x_i\}_i \cup \{x_i'\}_i$, if $\gamma(x_i) \sim_{\tau_i} \gamma(x_i')$ for every i, then $\hat{\gamma}(e) \sim_\tau \hat{\gamma}(e')$. ♢

This formulation has a compositional representation in LF: The translation of separated contexts $\Gamma^\star$ can be defined as follows:

$$\ulcorner . \urcorner = .$$
$$\ulcorner \Gamma^\star, (x, x') : \tau \urcorner = \ulcorner \Gamma^\star \urcorner, \mathtt{x}_x : \mathtt{exp}, \mathtt{x}_{x'} : \mathtt{exp}, \mathtt{s}_{x,x'} : \mathtt{conc}\ \mathtt{cutful}\ \ulcorner x_x \sim_\tau x_{x'} \urcorner$$

The restriction that variables must be distinct is crucial, as we cannot add the same variable to the LF context twice.

The alternative formulation trivially coincides with the original in empty contexts. For non-empty contexts, it is equivalent to the original up to separation of variable names:

Lemma 4.34. *For any expressions e, e' and types τ, if*

$$\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$$

and

$$\Gamma^\star = (x_1^L, x_1^R) : \tau_1, \ldots, (x_n^L, x_n^R) : \tau_n$$

for distinct variables, i.e., $x_i^L \neq x_i^R$, then we have

$$\Gamma \vdash e \sim e' : \tau \quad \Leftrightarrow \quad \Gamma^\star \vdash^\star e[x_1^L/x_1, \ldots, x_n^L/x_1] \sim e'[x_1^R/x_1, \ldots, x_n^R/x_n] : \tau.$$

Proof sketch. In the forward direction, we assume the LHS, and are given a substitution γ where we have $\gamma(x_i^L) \sim_{\tau_i} \gamma(x_i^R)$ for every i. We must show

$$\hat{\gamma}(e[x_1^L/x_1, \ldots, x_n^L/x_1]) \sim_\tau \hat{\gamma}(e'[x_1^R/x_1, \ldots, x_n^R/x_n]).$$

We can construct substitutions $\gamma^L = [x_i \mapsto \gamma(x_i^R)]_i$ and $\gamma^R = [x_i \mapsto \gamma(x_i^R)]_i$. It can easily be shown that $\gamma^L \sim_\Gamma \gamma^R$, so by LHS we get $\hat{\gamma}^L(e) \sim_\tau \hat{\gamma}^R(e')$. But we can easily see that $\hat{\gamma}^L(e) = \hat{\gamma}(e)$ and $\hat{\gamma}^R(e') = \hat{\gamma}(e')$, and we are done.

In the other direction, we assume the RHS, and are given substitutions γ^L and γ^R where $\gamma^L \sim_\Gamma \gamma^R$. We construct $\gamma = [x_i^L \mapsto \gamma^L(x_i)]_i \cup [x_i^R \mapsto \gamma^R(x_i)]_i$, from which it is easy to show that $\gamma(x_i^L) \sim_{\tau_i} \gamma(x_i^R)$ (but only since $x_i^L \neq x_i^R$!). By RHS we therefore have $\hat{\gamma}(e) \sim_\tau \hat{\gamma}(e')$. We can easily verify that $\hat{\gamma}(e) = \hat{\gamma}^L(e)$ and $\hat{\gamma}(e') = \hat{\gamma}^R(e')$, and we are done. □

So, how should we formulate soundness of axiomatic equivalence using this alternative representation? A first attempt might be to try to show that a derivation of $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e \doteq e' : \tau$ implies $(x_1, x_1) : \tau_1, \ldots, (x_n, x_n) : \tau_n \vdash^\star e \sim e' : \tau$. This only holds for the empty context though, since separated variables are required to be distinct. Instead, we could try to show that

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

implies

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

This could work, but unfortunately results in a very complicated induction hypothesis in the Twelf formalization, since we have to introduce two fresh variables for each assumption, *and* associate these with the original. It can be done, but requires a tertiary equality relation on expressions: an example for a simple system can be found in [Ras13].

A much simpler solution consists of defining an alternative formulation of axiomatic equivalence that coincides with the representable formulation of open logical equivalence. We can then show that a derivation in the original system can be converted to one in the alternative one, which in turn has a much more straightforward soundness proof via logical relations in Twelf.

An excerpt of the alternative system can be seen in Figure 4.10. We have only shown some representative rules, as the definitions for the remaining ones should be self-evident. Importantly, rules that extend the context are defined such that only distinct variables are added, by renaming bound variables. However, we put no restrictions distinctness of variables in the context; assumptions of the form $(x, x) : \tau$ are still allowed. We will implicitly allow weakening and exchange. The Twelf representation of the shown rules is defined as follows (see Appendix B.2 for the full definition):

```
sim* : exp -> exp -> tp -> type.
sim*/sym : sim* E' E T
            <- sim* E E' T.
sim*/trans : sim* E E'' T
              <- sim* E E' T
              <- sim* E' E'' T.
```

```
sim*/cong/app :
   sim* (app E1 E2) (app E1' E2') T0
      <- sim* E1 E1' (T2 => T0)
      <- sim* E2 E2' T2.
sim*/cong/lam :
   sim* (lam E0) (lam E0') (T2 => T0)
      <- ({x}{x'} sim* x x' T2
           -> sim* (E0 x) (E0' x') T0).
%{...}%
```

We will now prove that a derivation in the original equational reasoning system implies a derivation in the alternative one. We first show that we can rewrite an assumption of the form $(x, x) : \tau$ to an assumption of the form $(x^L, x^R) : \tau$:

Lemma 4.35 (Doubling)**.**
If $\mathcal{Q} :: \Gamma^\star, (x, x) : \tau' \vdash^\star e \doteq e' : \tau$*, then also*

$$\Gamma^\star, (x^L, x^R) : \tau' \vdash^\star e[x^L/x] \doteq e[x^R/x] : \tau.$$

Proof sketch. By induction on $\mathcal{Q}$. We show some representative cases.

- The case where $\mathcal{Q}$ ends in `qa_assm` is immediate.
- The case for `qa_app` follows by IH on each subderivation, followed by `qa_app` on the results.

Assumptions: $\Gamma^\star :: \mathsf{QAssm} ::= \cdot \mid \Gamma^\star, (x, x') : \tau$

Alternative equivalence: $\mathcal{Q} :: \boxed{\Gamma^\star \vdash^\star e \doteq e' : \tau}$:

$$\textsf{qa\_assm}: \frac{}{\Gamma^\star, (x, x') : \tau \vdash^\star x \doteq x' : \tau} \qquad \textsf{qa\_sym}: \frac{\Gamma^\star \vdash^\star e \doteq e' : \tau}{\Gamma^\star \vdash^\star e' \doteq e : \tau}$$

$$\textsf{qa\_trans}: \frac{\Gamma^\star \vdash^\star e \doteq e' : \tau \quad \Gamma^\star \vdash^\star e' \doteq e'' : \tau}{\Gamma^\star \vdash^\star e \doteq e'' : \tau}$$

$$\textsf{q\_lam}: \frac{\Gamma^\star, (x^l, x^r) : \tau_2 \vdash^\star e_0[x^l/x] \doteq e_0'[x^r/x'] : \tau_0}{\Gamma^\star \vdash^\star \lambda x.\, e_0 \doteq \lambda x'.\, e_0' : \tau_2 \rightharpoonup \tau_0} \; (x^l \neq x^r)$$

$$\textsf{q\_app}: \frac{\Gamma^\star \vdash^\star e_1 \doteq e_1' : \tau_2 \rightharpoonup \tau_0 \quad \Gamma^\star \vdash^\star e_2 \doteq e_2' : \tau_2}{\Gamma^\star \vdash^\star e_1\ e_2 \doteq e_1'\ e_2' : \tau_0}$$

$$\vdots$$

(Alternative versions of the remaining equivalence rules are defined similarly.)

Figure 4.10: *Alternative axiomatic equivalence.*

- Case $\mathcal{Q} = \textsf{qa\_sym}: \dfrac{\overset{\mathcal{Q}'}{\Gamma^\star, (x, x) : \tau' \vdash^\star e' \doteq e : \tau}}{\Gamma^\star, (x, x) : \tau' \vdash^\star e \doteq e' : \tau}$.

  By IH on $\mathcal{Q}'$, we obtain $\mathcal{Q}'' :: \Gamma^\star, (x^L, x^R) : \tau' \vdash^\star e'[x^L/x] \doteq e[x^R/x] : \tau$. By qa_sym, it suffices to prove

  $$\Gamma^\star, (x^L, x^R) : \tau' \vdash^\star e'[x^R/x] \doteq e[x^L/x] : \tau.$$

  This can be obtained by applying qa_sym in all places where the assumption $(x^L, x^R) : \tau'$ is used, and thus follows by inner induction on $\mathcal{Q}''$.

- Case $\mathcal{Q} = \textsf{qa\_trans}: \dfrac{\overset{\mathcal{Q}_1}{\Gamma^\star, (x, x) : \tau' \vdash^\star e \doteq e' : \tau} \quad \overset{\mathcal{Q}_2}{\Gamma^\star, (x, x) : \tau' \vdash^\star e' \doteq e'' : \tau}}{\Gamma^\star, (x, x) : \tau' \vdash^\star e \doteq e'' : \tau}$.

  By IH on $\mathcal{Q}_1$ and $\mathcal{Q}_2$, we obtain $\mathcal{Q}_1' :: \Gamma^\star, (x^L, x^R) : \tau' \vdash^\star e[x^L/x] \doteq e'[x^R/x] : \tau$ and $\mathcal{Q}_2' :: \Gamma^\star, (x^L, x^R) : \tau' \vdash^\star e'[x^L/x] \doteq e''[x^R/x] : \tau$. By qa_trans on $\mathcal{Q}_1'$, it suffices to show

  $$\Gamma^\star, (x^L, x^R) : \tau' \vdash^\star e'[x^R/x] \doteq e''[x^R/x] : \tau.$$

  This can be obtained by inner induction on $\mathcal{Q}_2'$, by applying rules qa_sym and qa_trans in all places where the assumption $(x^L, x^R) : \tau'$ is used.

- Case

  $$\mathcal{Q} = \textsf{qa\_cong\_lam}: \frac{\overset{\mathcal{Q}_1}{\Gamma^\star, (x, x) : \tau', (x_2^L, x_2^R) : \tau_2 \vdash^\star e_0[x_2^L/x_2] \doteq e_0'[x_2^R/x_2'] : \tau_0}}{\Gamma^\star, (x, x) : \tau' \vdash^\star \lambda x_2.\, e_0 \doteq \lambda x_2'.\, e_0' : \tau_2 \rightharpoonup \tau_0}.$$

By exchange, we may proceed by IH on $\mathcal{Q}_1$, obtaining

$$\mathcal{Q}_1' :: \Gamma^\star, (x_2^L, x_2^R) : \tau_2, (x^L, x^R) : \tau' \vdash^\star e_0[x_2^L/x_2][x^L/x] \doteq e_0'[x_2^R/x_2'][x^R/x] : \tau_0.$$

But then by exchange again, we can apply qa_cong_lam, and we are done.

- Remaining cases elided.

□

Using this, we can show the main conversion lemma which, interestingly enough, works in a context where variables are *identical*:

Lemma 4.36 (Conversion)**.**
Suppose $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ *and* $\Gamma^\star = (x_1, x_1) : \tau_1, \ldots, (x_n, x_n) : \tau_n$.
If $\mathcal{Q} :: \Gamma \vdash e \doteq e' : \tau$, *then also* $\Gamma^\star \vdash^\star e \doteq e' : \tau$.

Proof sketch. By induction on $\mathcal{Q}$. The only interesting cases are those involving binders:

- Case $\mathcal{Q}$ ends in q_assm: The result is immediate.
- Case $\mathcal{Q}$ ends in q_app: We proceed by IH on the subderivations and apply qa_app.
- Case $\mathcal{Q}$ ends in q_sym or q_trans: Again, proceed by IH on subderivations, and one of qa_sym or qa_trans.
- Case $\mathcal{Q} = \mathsf{q\_lam}: \dfrac{\mathcal{Q}_1}{\dfrac{\Gamma, x : \tau_2 \vdash e_0 \doteq e_0' : \tau_0}{\Gamma \vdash \lambda x.\, e_0 \doteq \lambda x.\, e_0' : \tau_2 \rightharpoonup \tau_0}}$.

  By IH on $\mathcal{Q}_1$, we obtain $\mathcal{Q}_1' :: \Gamma^\star, (x, x) : \tau_2 \vdash^\star e_0 \doteq e_0' : \tau_0$. By Lemma 4.35 on $\mathcal{Q}_1'$, we rewrite this into $\mathcal{Q}_1'' :: \Gamma^\star, (x^L, x^R) : \tau_2 \vdash^\star e_0[x^L/x] \doteq e_0'[x^R/x] : \tau_0$. This matches the premise of qa_cong_lam which we then apply to obtain our goal, and we are done. □

The above two lemmas are formalized as the following Twelf meta-theorems:

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

It remains to formulate the soundness theorem for the alternative equational reasoning system. Since its context matches that of the alternative formulation of logical equivalence, we do not have to introduce fresh variables. On the other hand, since open logical equivalence requires all variables to distinct, this also becomes a premise of the soundness theorem:

Theorem 4.37 (Soundness of alternative axiomatic equivalence)**.**
Suppose $\Gamma^\star = (x_1^L, x_1^R) : \tau_1, \ldots, (x_n^L, x_n^R) : \tau_n$*, where* $x_i^L \neq x_i^R$ *for every* i*.*
If $\Gamma^\star \vdash^\star e \doteq e' : \tau$*, then also* $\Gamma^\star \vdash^\star e \sim e' : \tau$*.*

Proof sketch. Straightforward, as in the proof of Theorem 4.27. The invariant that all variables in assumptions are distinct is ensured by the fact that all context extensions in the alternative equational reasoning system guarantees distinctness of variables. □

This is proved as the following Twelf type family:

```
sim*-lr : sim* E E' T -> lr T R -> conc* (R E E') -> type.
%mode sim*-lr +SIP -LP -SP.
%{ ... proof elided ... }%
%block bsim* : some
               {T2:tp}{R2}{LP2:lr T2 R2}
               block
               {x:exp}{x':exp}{sip:sim* x x' T2}{sp:conc cutful (R2 x x')}
               {_:sim*-lr sip LP2 sp}.
%worlds (bsim*) (sim*-lr _ _ _) %{ ... }%.
%total (SIP %{ ... }%) (sim*-lr SIP _ _) %{ ... }%.
```

The regular world shows how the context of the alternative equational reasoning system translates to the context of alternative open logical equivalence.

Note that since Lemma 4.36 results in a derivation with a context containing assumptions with identical variables, we can only show soundness of *closed* derivations of the original equational system. That is, a theorem showing that axiomatic equivalence at type nat implies Kleene equivalence can only work on closed contexts:

Theorem 4.38 (Axiomatic equivalence implies Kleene equivalence)**.**
If $\mathcal{Q} :: \cdot \vdash e \doteq e' : \mathsf{nat}$ *and* $e \Downarrow \overline{n}$*, then also* $e' \Downarrow \overline{n}$*.*

Proof. By Lemma 4.36 on $\mathcal{Q}$, obtain $\mathcal{Q}' :: \cdot \vdash^\star e \doteq e' : \mathsf{nat}$. By Theorem 4.37 on $\mathcal{Q}'$, we obtain $e \sim_{\mathsf{nat}} e'$. By the assumption of $e \Downarrow \overline{n}$ and the definition of the logical relation at type nat, we obtain $e' \Downarrow \overline{n}$. □

```
lr-ext : eval E (num N) -> lr nat' R -> conc* (R E E') -> eval E' (num N) -> type.
%mode lr-ext +EP +LP +SP -EP'.
%{... proof elided ...}%
%worlds () (lr-ext _ _ _ _).
%total {} (lr-ext _ _ _ _).

sim-ext : eval E (num N) -> sim E E' nat' -> eval E' (num N) -> type.
%mode sim-ext +EP +SIP -EP'.
- : sim-ext EP (SIP : sim E E' nat') EP'
     <- sim=>sim* SIP SIP'
     <- sim*-lr SIP' LP SP
     <- lr-ext EP LP SP EP'.
%worlds () (sim-ext _ _ _).
%total {} (sim-ext _ _ _).
```

4.6 Summary of the formalization

In this section, we will briefly summarize the formalization of the core congruence lemmas. Due to the code size of the complete formalization, we have not included the source code in the appendix. In can instead be found in the electronic appendix [Ras13]. To conserve space, we will use `conc*` as an abbreviation for `conc cutful`. Also, we use `flr/=>` as an abbreviation for the definition of the logical relation at arrow types, and `keq+` as an abbreviation for the definition at natural numbers.

Lemmas 4.3, 4.4 and 4.5 are formalized as the following type families:

```
sym-lr : {T}{E1 : exp}{E2 : exp}{R : exp -> exp -> form} lr T R
  -> conc* (R E1 E2) -> conc* (R E2 E1) -> type.
%mode sym-lr +T +E1 +E2 +R +LP +RP -RP'.

trans-lr : {T}{E}{E'}{E''}{R : exp -> exp -> form} lr T R
          -> conc* (R E E') -> conc* (R E' E'') -> conc* (R E E'') -> type.
%mode trans-lr +T +E +E' +E'' +R +LP +RP1 +RP2 -RP'.

cond-refl-lr : {T}{E}{E'}{R : exp -> exp -> form} lr T R
             -> conc* (R E E') -> conc* (R E E) -> type.
%mode cond-refl-lr +T +E +E' +R +LP +SP1 -SP'.
```

We do not formalize the lemmas for open logical equivalence. As contexts are implicitly represented by the LF context, the open variants are also covered by the above.

Closure under weak equivalence is also straightforward:

```
cvrs-lr : {T}{D}{E}{R : exp -> exp -> form} lr T R
        -> conc* (R E E) -> conc* (foralle [v] #eval D v <==> #eval E v)
        -> conc* (R D E) -> type.
%mode cvrs-lr +T +E +D +R +LP +SP1 +SP2 -SP'.
```

Lemma 4.14 does not do any induction or appeal to other lemmas, and can actually be proved as an abbreviation:

```
% Application commutes over case
bieq-app-case : (conc* (foralle [v]
                         #eval (app (case E0 E1 E2) E) v
                         <==> #eval (case E0 (app E1 E) ([x] app (E2 x) E)) v))
  = %{ ... }%
```

The formalization of Lemma 4.15 is even shorter than the original proof:

```
cong-app-lr : lr T2 R2 -> lr T1 R1
            -> conc* (flr/=> R2 R1 E1 E1') -> conc* (R2 E2 E2')
            -> conc* (R1 (app E1 E2) (app E1' E2')) -> type.
%mode cong-app-lr +LP2 +LP1 +SP1 +SP2 -SP'.
- : cong-app-lr (LP2 : lr T2 R2) LP1 SP1 (SP2  : conc* (R2 E2 E2'))
     (cut SP1 (forallel E2 (forallel E2' (impl (cut SP2 ax) ax)))).
```

The formulation of Lemma 4.16 involves a context extension for one of the premises. Thus, one of the inputs to the formalization is a function:

```
cong-lam-lr : lr T2 R2 -> lr T0 R0
            -> ({e2}{e2'} conc* (R2 e2 e2') -> conc* (R0 (E e2) (E' e2')))
            -> conc* (flr/=> R2 R0 (lam E) (lam E')) -> type.
%mode cong-lam-lr +LP2 +LP0 +SP -SP'.
```

Likewise for Lemma 4.17:

```
cong-case-lr : {T} 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 cong-case-lr +T +LP +SP0 +SP1 +SP2 -SP'.
```

Reflexivity at variables is implicitly represented by the encoding, and does therefore not need to be formalized. Reflexively at numerals is a one-liner:

```
refl-num-lr : {N} conc* (keq+ (num N) (num N)) -> type.
%mode refl-num-lr +N -SP.
```

The formalization of the strictness lemma is interesting, as the representation of $e \not\Downarrow$ (i.e., that e does *not* evaluate) lives entirely on the meta-level. It turns out that the following characterization suffices for our purposes:

```
noeval : exp -> type. %name noeval NP.
noeval/diverge : noeval diverge.
noeval/app : noeval E1 -> noeval (app E1 E2).
```

Using this, we can prove that expressions satisfying the above cannot evaluate, i.e., an evaluation implies absurdity:

```
noeval-void : noeval E
               -> conc* (foralle [v] #eval E v ==> existsd [q:data @void] top)
               -> type.
%mode noeval-void +NP -SP.
```

The strictness lemma is then formulated as follows:

```
strict-lr : {T} lr T R -> noeval E -> noeval E' -> conc* (R E E') -> type.
%mode strict-lr +T +LP +NP +NP' -SP.
```

Reflexivity for diverge follows directly as a special case:

```
refl-diverge-lr : {T} lr T R -> conc* (R diverge diverge) -> type.
%mode refl-diverge-lr +T +LP -SP.
- : refl-diverge-lr T LP SP
     <- strict-lr T LP noeval/diverge noeval/diverge SP.
```

5 Equational reasoning for CBV simply typed λ-calculus

In the previous chapter, we developed a methodology for formalizing proofs of observational equivalence via a binary logical relation in Twelf. We will now test the scalability of that methodology, by applying it in the formalization of an equational reasoning system for a more complex language. This language, which we call $\lambda^{\rightarrow,\mathsf{nat}}_{\mathsf{cbv},\mathsf{nd}}$, is a simply typed lambda calculus with a call-by-value operational semantics and the possibility of failure. Furthermore, we implement natural numbers via expression constructors, which should serve as a minimal example of algebraic data without embedded functions, and finally, the language is equipped with a simple non-deterministic *choice* operator.

The developments will follow the same structure as Chapter 4, but with a few changes. The primary difference is due to the call-by-value semantics, which complicates the definition of our logical relation a bit, requiring the definition of a monadic extension to the logical relation to capture the notion of equivalence of computations.

It turns out that the basic methodology developed in the preceding chapters need not be extended further to formalize these developments in Twelf. That is, we use a data representation logic to represent our embedded judgments, and then do case analysis on derivations in this logic inside assertion logic proofs. We have to make a few extra efforts in order to avoid needing induction on the assertion logic level, but these can be regarded as orthogonal to the general technique which remains unchanged.

The chapter is structured as follows. In Section 5.1, we will introduce our object language. We will skip the description of what it means for programs to be observationally equivalent, and refer to Section 4.1 for a discussion. In Section 5.2 we define our logical relation and introduce the *computation extension*. In Section 5.2.1, we show that the computation extension has some properties akin to monadic bind and return. In Section 5.2.2 we show that logical equivalence is indeed a partial equivalence relation, followed by Section 5.3 where we show that it is also a congruence. These results are then used in Section 5.4, where we give a definition of an axiomatic reasoning system for proving observational equivalence, which we then prove sound via our logical relation. In Section 5.5, we describe the extra efforts that were required in order to make all

proofs go through in the formalization. We conclude the chapter in Section 5.6, where we briefly summarize the formalization.

5.1 Language definition

The syntax and semantics of $\lambda^{\rightharpoonup,\mathsf{nat}}_{\mathsf{cbv},\mathsf{nd}}$ are given in Figure 5.1. Note that we represent numerals using applicative constructors instead of embedding the syntax of natural numbers. This concept may be generalized to other algebraic data types such as lists or trees, although we will stick to numbers in this particular development. In order to be able to distinguish the subset of values that are well-formed numerals, we introduce a new judgment $v \overset{\mathrm{num}}{\leftrightarrow} n$ that characterizes when v is a numeral representing the natural number n.

We have chosen to replace diverge from the language in Chapter 4 with fail. The two constructs are in a sense similar, since neither has an evaluation derivation. However, due to the addition of non-determinism in this language, fail does not really model true divergence: consider for example the expression $\mathsf{fail} \,[\!]\, \mathsf{zero}$, for which we have only a single derivation $\mathsf{fail} \,[\!]\, \mathsf{zero} \Downarrow \mathsf{zero}$. This is due to the choice-operator being rather simplistic, in that it represents a form of *angelic choice*; if there is just a single possible way to evaluate a choice-expression, we are guaranteed to find it. A more faithful representation of non-determinism would involve some sort of state parameter on the evaluation judgment, which would model an oracle committing to a particular choice. However, to keep things simple, we have chosen to go with the "angelic choice" instead.

In our developments, we will be needing the following general results about values:

Lemma 5.1 (Values evaluate)**.** *For any expression v where $\mathcal{V} :: v$ value, we have $v \Downarrow v$.*

Proof. By induction on $\mathcal{V}$.

- Case $\mathcal{V}$ ends in v_lam, so $v = \lambda x.\, e_0$. Then we get the desired result by e_lam.
- Case $\mathcal{V}$ ends in v_zero, so $v = \mathsf{zero}$. Then we get the desired result by e_zero.
- Case $\mathcal{V}$ ends in v_succ, so $v = \mathsf{succ}(v')$ and we have $\mathcal{V}' :: v'$ value. By induction on $\mathcal{V}'$, using e_succ on the result, we are done. □

Additionally, the result of an evaluation is always a value:

Lemma 5.2 (Value completeness)**.** *For any expressions e, v, if $\mathcal{E} :: e \Downarrow v$, then v* value*.*

Proof sketch. By straightforward induction on $\mathcal{E}$. □

Lemma 5.3 (Value determinism)**.** *For any expressions v, v' where $\mathcal{V} :: v$* value*, if $\mathcal{E} :: v \Downarrow v'$, then $v = v'$.*

Proof. By induction on $\mathcal{E}$.

Natural numbers: $n :: \mathsf{Nat} ::= \mathsf{z} \mid \mathsf{s}(n)$
Types: $\tau :: \mathsf{Tp} ::= \mathsf{nat} \mid \tau_2 \rightharpoonup \tau_0$
Contexts: $\Gamma :: \mathsf{Ctx} ::= \cdot \mid \Gamma, x : \tau$
Expressions: $e, v :: \mathsf{Exp} ::= \mathsf{zero} \mid \mathsf{succ}(e) \mid \lambda x.\, e_0 \mid e_1\ e_2 \mid \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \mid e_1 \,[\!]\, e_2 \mid \mathsf{fail}$

Numerals: $\mathcal{N} :: \boxed{v \overset{\mathsf{num}}{\leftrightarrow} n}$:

$$\mathsf{n\_zero}: \frac{}{\mathsf{zero} \overset{\mathsf{num}}{\leftrightarrow} \mathsf{z}} \qquad \mathsf{n\_succ}: \frac{v \overset{\mathsf{num}}{\leftrightarrow} n}{\mathsf{succ}(v) \overset{\mathsf{num}}{\leftrightarrow} \mathsf{s}(n)}$$

Values: $\mathcal{V} :: \boxed{v\ \mathsf{value}}$:

$$\mathsf{v\_lam}: \frac{}{\lambda x.\, e_0\ \mathsf{value}} \qquad \mathsf{v\_zero}: \frac{}{\mathsf{zero}\ \mathsf{value}} \qquad \mathsf{v\_succ}: \frac{v\ \mathsf{value}}{\mathsf{succ}(v)\ \mathsf{value}}$$

Dynamic semantics: $\mathcal{E} :: \boxed{e \Downarrow v}$:

$$\mathsf{e\_zero}: \frac{}{\mathsf{zero} \Downarrow \mathsf{zero}} \qquad \mathsf{e\_succ}: \frac{e \Downarrow v}{\mathsf{succ}(e) \Downarrow \mathsf{succ}(v)} \qquad \mathsf{e\_lam}: \frac{}{\lambda x.\, e_0 \Downarrow \lambda x.\, e_0}$$

$$\mathsf{e\_app}: \frac{e_1 \Downarrow \lambda x.\, e_0 \quad e_2 \Downarrow v_2 \quad e_0[v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\mathsf{e\_choice1}: \frac{e_1 \Downarrow v}{e_1 \,[\!]\, e_2 \Downarrow v} \qquad \mathsf{e\_choice2}: \frac{e_2 \Downarrow v}{e_1 \,[\!]\, e_2 \Downarrow v}$$

$$\mathsf{e\_case0}: \frac{e_0 \Downarrow \mathsf{zero} \quad e_1 \Downarrow v}{\mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \Downarrow v} \qquad \mathsf{e\_case1}: \frac{e_0 \Downarrow \mathsf{succ}(v_0) \quad e_2[v_0/x] \Downarrow v}{\mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \Downarrow v}$$

Static semantics: $\mathcal{T} :: \boxed{\Gamma \vdash e : \tau}$:

$$\mathsf{t\_var}: \frac{}{\Gamma \vdash x : \tau}\ (\Gamma(x) = \tau) \qquad \mathsf{t\_zero}: \frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}} \qquad \mathsf{t\_succ}: \frac{\Gamma \vdash e_0 : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e_0) : \mathsf{nat}}$$

$$\mathsf{t\_lam}: \frac{\Gamma, x : \tau_2 \vdash e_0 : \tau_0}{\Gamma \vdash \lambda x.\, e_0 : \tau_2 \rightharpoonup \tau_0} \qquad \mathsf{t\_app}: \frac{\Gamma \vdash e_1 : \tau_2 \rightharpoonup \tau_0 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_0}$$

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

$$\mathsf{t\_choice}: \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \,[\!]\, e_2 : \tau}$$

Figure 5.1: *Syntax and semantics for* $\lambda^{\rightharpoonup,\mathsf{nat}}_{\mathsf{cbv},\mathsf{nd}}$.

```
% types
tp : type.
nat' : tp.
=> : tp -> tp -> tp.
%infix right 1 =>.

% expressions
exp : type.
zero : exp.
succ : exp -> exp.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
case : exp -> exp
    -> (exp -> exp) -> exp.
choice : exp -> exp -> exp.
fail : exp.

% numerals
num : nat -> exp -> type.
num/z : num z zero.
num/s : num (s N) (succ E)
         <- num N E.
```

```
% value judgment
value : exp -> type.
value/zero : value zero.
value/succ : value (succ E0)
            <- value E0.
value/lam : value (lam E0).

% evaluation
eval : exp -> exp -> type.

eval/zero : eval zero zero.
eval/succ : eval (succ E) (succ V)
             <- eval E V.
eval/lam : eval (lam E0) (lam E0).
eval/app : eval (app E1 E2) V
            <- eval E1 (lam E0)
            <- eval E2 V2
            <- eval (E0 V2) V.
eval/choice1 : eval (choice E1 E2) V
                <- eval E1 V.
eval/choice2 : eval (choice E1 E2) V
                <- eval E2 V.
eval/case0 : eval (case E0 E1 E2) V
              <- eval E0 zero
              <- eval E1 V.
eval/case1 : eval (case E0 E1 E2) V
              <- eval E0 (succ V0)
              <- eval (E2 V0) V.
```

Figure 5.2: *Twelf signature for* $\lambda^{\rightarrow,\mathsf{nat}}_{\mathsf{cbv},\mathsf{nd}}$.

- If $\mathcal{E}$ ends in e_lam or e_zero. Then $v = v'$ follows directly.
- If $\mathcal{E}$ ends in e_succ, then $v = \mathsf{succ}(v_0)$ and $v' = \mathsf{succ}(v'_0)$, and we have a derivation $\mathcal{E}' :: v_0 \Downarrow v'_0$. Then $\mathcal{V}$ must end in v_succ, and we thus have $\mathcal{V}_0 :: v_0$ value. By the induction hypothesis on $\mathcal{E}'$ with $\mathcal{V}_0$, we obtain $v_0 = v'_0$, and we are done.
- The cases for e_app, e_choice1,e_choice2, e_case0 and e_case1 are impossible by $\mathcal{V}$. □

Lemma 5.4 (Numerals are values)**.** *For any value v, if $\mathcal{N} :: v \overset{\text{num}}{\leftrightarrow} n$, then v* value*.*

Proof sketch. By straightforward induction on $\mathcal{N}$. □

Lemma 5.5 (Numeral determinism)**.** *If $\mathcal{V} :: v \overset{\text{num}}{\leftrightarrow} n$ and $\mathcal{V}' :: v' \overset{\text{num}}{\leftrightarrow} n$ for some n, then $v = v'$.*

Proof. By straightforward induction on $\mathcal{V}$. □

Lemma 5.6 (Uniqueness of numerals)**.** *If* $\mathcal{V} :: v \overset{\text{num}}{\leftrightarrow} n$ *and* $\mathcal{V}' :: v \overset{\text{num}}{\leftrightarrow} n'$*, then* $n = n'$*.*

Proof. By straightforward induction on $\mathcal{V}$. □

5.2 Logical equivalence

In this section we will define logical equivalence as a type-indexed relation between values. We define value relations as follows:

Definition 5.7 (Relations on values)**.** Given a binary relation R, we say that R is a relation between values iff for any expressions v,v', if $v \ R \ v'$, then also v value and v' value. ♢

We also have to define what it means for expressions to be equivalent when they are not necessarily values. We have to take into account that expressions might have side effects, in that they may fail or be non-deterministic. To capture this, we define the following extension of binary relations:

Definition 5.8 (Computation extension)**.** Any binary relation R between values gives rise to a relation $R^\dagger$ between expressions, called the *computation extension* of R, where for any expressions e, e', we have

$$\begin{aligned} e \ R^\dagger \ e' \Leftrightarrow (\forall v.\, e \Downarrow v \Rightarrow \exists v'.\, e' \Downarrow v' \wedge v \ R \ v') \\ \wedge \ (\forall v'.\, e' \Downarrow v' \Rightarrow \exists v.\, e \Downarrow v \wedge v \ R \ v'). \end{aligned}$$ ♢

Intuitively, this says that two expressions are related at the computation extension if they evaluate to related values *and have the same observable side-effects*, where in this case possible side-effects are failure and non-determinism. The computation extension captures both by requiring the existence of a bijection between all possible evaluation derivations of the related expressions.

As we have shown in Lemma 5.1, values are "pure" in that they never have any side-effects. They might, however, represent suspended computations (i.e., functions) that, when applied, might have observable effects. We capture this in our logical relation by using the computation extension in the definition at function types, arriving at the following:

Definition 5.9 (Logical equivalence)**.** Logical equivalence $v \sim_\tau v'$ is a type-indexed family of relations between closed values. It is inductively defined on types as follows:

$$\begin{aligned} v \sim_{\mathsf{nat}} v' &\Leftrightarrow \exists n.\ v \overset{\text{num}}{\leftrightarrow} n \wedge v' \overset{\text{num}}{\leftrightarrow} n \\ v \sim_{\tau_2 \to \tau_0} v' &\Leftrightarrow \exists x.\, e_0,\ x'.\, e_0'.\ v = \lambda x.\, e_0 \wedge v' = \lambda x'.\, e_0' \\ &\qquad \wedge \forall v_2, v_2'.\ v_2 \sim_{\tau_2} v_2' \Rightarrow e_0[v_2/x] \sim_{\tau_0}^\dagger e_0'[v_2'/x']. \end{aligned}$$ ♢

We can see that the relation $\cdot \sim^{\dagger}_{\mathsf{nat}} \cdot$ captures precisely what we mean by observational equivalence, i.e., that if $e \sim^{\dagger}_{\mathsf{nat}} e'$, then e evaluates iff e' does, yielding identical results. What remains to show is that the computation extension of logical equivalence is also a congruence relation, from which it follows that it implies observational equivalence.

5.2.1 Properties of the computation extension

In the following, we will derive a number of abstract results about the computation extension, which do not depend on any specific underlying relation.

Lemma 5.10. *Suppose R is a symmetric relation on values. Then $R^{\dagger}$ is also symmetric.*

Proof. Assume that R is symmetric, that is, we have $a_1 :: \forall v, v'.\, v\ R\ v' \Rightarrow v'\ R\ v$. It suffices to show that for any e, e', if $h_1 :: e\ R^{\dagger}\ e'$, then also $g_1 :: e'\ R^{\dagger}\ e$.

By Definition 5.8 it suffices to show the following subgoals:

- For any v, if $\mathcal{E} :: e' \Downarrow v$, then there is a v' and a derivation $\mathcal{E}' :: e \Downarrow v'$ where $g_1 :: v\ R\ v'$.

  By h_1 and $\mathcal{E}$, there is a v_0 and a derivation $\mathcal{E}_0 :: e \Downarrow v_0$ and $r_1 :: v_0\ R\ v$. By a_1 on r_1, we get $r_1' :: v\ R\ v_0$. But then we can pick $v' = v_0$, $\mathcal{E}' = \mathcal{E}_0$ and $g_1 = r_1'$, and we are done.

- For any v', if $\mathcal{E} :: e \Downarrow v'$, then there is a v and a derivation $\mathcal{E}' :: e' \Downarrow v$ where $g_1 :: v\ R\ v'$.

  Analogous to the above. □

Lemma 5.11. *Suppose R is a transitive relation on values. Then $R^{\dagger}$ is also transitive.*

Proof. Assume that R is transitive, that is, we have $a_1 :: \forall v, v', v''.\, v\ R\ v' \Rightarrow v'\ R\ v'' \Rightarrow v\ R\ v''$. It suffices to show that for any e, e', e'', if $h_1 :: e\ R^{\dagger}\ e'$ and $h_2 :: e'\ R^{\dagger}\ e''$, then also $g_1 :: e\ R^{\dagger}\ e''$.

By Definition 5.8 it suffices to show the following subgoals:

- For any v, if $\mathcal{E} :: e \Downarrow v$, then there is a v'' and a derivation $\mathcal{E}'' :: e'' \Downarrow v''$ where $g_1 :: v\ R\ v''$.

  By h_1 and $\mathcal{E}$, there is a v_0' such that $\mathcal{E}_0' :: e' \Downarrow v_0'$ and $r_0 :: v\ R\ v_0'$. By h_2 and $\mathcal{E}_0'$, there is a v_0'' such that $\mathcal{E}_0'' :: e'' \Downarrow v_0''$ and $r_0' :: v_0'\ R\ v_0''$.

  We pick $v'' = v_0''$ and thus get $\mathcal{E}''$ by $\mathcal{E}_0''$. It remains to establish g_1, which follows by a_1 on r_0, r_0'.

- For any v'', if $\mathcal{E}'' :: e'' \Downarrow v''$, then there is a v and a derivation $\mathcal{E} :: e \Downarrow v$ where $g_1 :: v\ R\ v''$.

  Analogous to the above. □

We have now established that the computation extension inherits symmetry and transitivity of any relation that it extends, including logical equivalence. In the following, we will show another general result, namely that the computation extension supports operations akin to the monadic operations bind and return.

The bind operation is expressed in terms of the following notion of reduction contexts:

Definition 5.12. A *reduction frame* $\mathcal{F}$ is an expression with a single hole, generated by the following grammar:

$$\mathcal{F} ::= \mathsf{succ}(\circ) \mid \circ\ e \mid v\ \circ \mid \mathsf{ncase}(\circ,\ e_1,\ x.\, e_2)$$

where every occurrence of expressions e and values v are closed.

A *reduction context* $\mathcal{R}$ is an expression with a single hole, generated by the grammar

$$\mathcal{R} ::= \circ \mid \mathcal{F}\{\mathcal{R}\},$$

that is, zero or more successive frame replacements. Likewise, we use $\mathcal{R}\{e\}$ to denote the result of substituting a closed expression e for the single hole in $\mathcal{R}$. $\diamondsuit$

A reduction context $\mathcal{R}$ is constructed such that for any expression e, the expression $\mathcal{R}\{e\}$ will have the same side-effect as e if e has a side effect. In our specific setting of call-by-value PCF, this means that $\mathcal{R}\{e\}$ fails and/or is non-deterministic whenever e fails and/or is non-deterministic.

To prove the bind lemma, we will need the following two technical lemmas, which deal with evaluation inside reduction frames:

Lemma 5.13 (Converse frame evaluation)**.** *If $\mathcal{E} :: e \Downarrow v$ and $\mathcal{E}' :: \mathcal{F}\{v\} \Downarrow v'$, then there is a $\mathcal{E}'' :: \mathcal{F}\{e\} \Downarrow v'$.*

Proof. By case analysis on $\mathcal{F}$.

- Case $\mathcal{F} = \mathsf{succ}(\circ)$. Then $\mathcal{E}'$ must end in an application of e_succ, implying that there exists a derivation $\mathcal{E}'_0 :: v \Downarrow v_0$ for some v_0, and that $v' = \mathsf{succ}(v_0)$. By Lemma 5.3 on $\mathcal{E}'_0$, we have $v = v_0$, and hence $\mathcal{E}$ is a derivation of $e \Downarrow v_0$. But then we get the desired derivation by e_succ on $\mathcal{E}$.

- Case $\mathcal{F} = \circ\ e_2$ for some e_2. But then $\mathcal{E}'$ is a derivation of $v\ e_2 \Downarrow v'$, and must end in an application of e_app, implying that there exists an expression e_0, a value v_2, and derivations $\mathcal{E}'_1 :: v \Downarrow \lambda x.\, e_0$ and $\mathcal{E}'_2 :: e_2 \Downarrow v_2$ and $\mathcal{E}'_3 :: e_0[v_2/x] \Downarrow v'$. By Lemma 5.3 on $\mathcal{E}'_1$, we have $v = \lambda x.\, e_0$, and hence $\mathcal{E}$ is a derivation of $e \Downarrow \lambda x.\, e_0$. But then we can construct $\mathcal{E}''$ by e_app on $\mathcal{E}$, $\mathcal{E}'_2$ and $\mathcal{E}'_3$.

- Case $\mathcal{F} = v_1\ \circ$ for some value v_1. But then $\mathcal{E}'$ is a derivation of $v_1\ v \Downarrow v'$, and must end in an application of e_app, implying that there exists an expression e_0, a

value v_2 and derivations $\mathcal{E}_1' :: v_1 \Downarrow \lambda x.\, e_0$ and $\mathcal{E}_2' :: v \Downarrow v_2$ and $\mathcal{E}_3' :: e_0[v_2/x] \Downarrow v'$. By Lemma 5.3 on $\mathcal{E}_2'$, we have $v = v_2$, and hence $\mathcal{E}$ is a derivation of $e \Downarrow v_2$. But then we can construct $\mathcal{E}''$ by e_app on $\mathcal{E}_1'$, $\mathcal{E}$ and $\mathcal{E}_3'$.

- Case $\mathcal{F} = \mathsf{ncase}(\circ,\ e_1,\ x.\, e_2)$ for some e_1 and e_2. But then $\mathcal{E}'$ is a derivation of $\mathsf{ncase}(v,\ e_1,\ x.\, e_2) \Downarrow v'$, and must end in either e_case0 or e_case1, where in each case we have two subderivations. In each case, we apply Lemma 5.3 on the first subderivation, which justifies reapplying one of e_case0 or e_case1 on $\mathcal{E}$ and the second subderivation. □

Lemma 5.14 (Frame evaluation extraction)**.** *If $\mathcal{E} :: \mathcal{F}\{e\} \Downarrow v$, there exists a v' such that $\mathcal{E}' :: e \Downarrow v'$ and $\mathcal{E}'' :: \mathcal{F}\{v'\} \Downarrow v$.*

Proof. By case analysis on $\mathcal{F}$.

- Case $\mathcal{F} = \mathsf{succ}(\circ)$. Then $\mathcal{E}$ must end in e_succ, implying that we have $\mathcal{E}_0 :: e \Downarrow v_0$ for some v_0. We pick $v' = v_0$ and obtain $\mathcal{E}'$ by $\mathcal{E}_0$. $\mathcal{E}''$ is constructed by e_succ on the result of Lemma 5.1 on v_0.

- Case $\mathcal{F} = \circ\ e_2$. Then $\mathcal{E}$ must end in e_app, implying that we have derivations $\mathcal{E}_1 :: e \Downarrow \lambda x.\, e_0$ and $\mathcal{E}_2 :: e_2 \Downarrow v_2$ and $\mathcal{E}_3 :: e_0[v_2/x] \Downarrow v$, for some v_2, e_0. We choose $v' = \lambda x.\, e_0$ and get $\mathcal{E}'$ by $\mathcal{E}_1$. We construct $\mathcal{E}''$ by e_app on $\mathcal{E}_2, \mathcal{E}_3$ and e_lam.

- Case $\mathcal{F} = v_1\ \circ$. Then $\mathcal{E}$ must end in e_app, implying that we have derivations $\mathcal{E}_1 :: v_1 \Downarrow \lambda x.\, e_0$ and $\mathcal{E}_2 :: e \Downarrow v_2$ and $\mathcal{E}_3 :: e_0[v_2/x] \Downarrow v$, for some v_2, e_0. We choose $v' = v_2$, and obtain $\mathcal{E}'$ by $\mathcal{E}_2$. By Lemma 5.1 on v_2, we get $\mathcal{E}_2' :: v_2 \Downarrow v_2$. But then we can construct $\mathcal{E}''$ by e_app on $\mathcal{E}_1$, $\mathcal{E}_2'$ and $\mathcal{E}_3$.

- Case $\mathcal{F} = \mathsf{ncase}(\circ,\ e_1,\ x.\, e_2)$. Then $\mathcal{E}$ must end in either e_case0 or e_case1. In each case, we get two subderivations where one is a derivation for e, giving us both v' and $\mathcal{E}'$ directly. $\mathcal{E}''$ is then constructed as in the previous cases, by reappling one of e_case0 or e_case1 to the result of an application of Lemma 5.1 on v'.

□

We are now ready to prove our monad lemmas, which are presented together in the following:

Lemma 5.15 (Monadic return)**.** *Any value relation is contained in its computation extension: For any binary relation R on values, if $v\ R\ v'$, then $v\ R^{\dagger}\ v'$.*

Proof. Assume $a_1 :: v\ R\ v'$. By Definition 5.8, it suffices to show the following:

1. For any v_0, if $\mathcal{E}_1 :: v \Downarrow v_0$, then there is a v_0' such that $g_1 :: v' \Downarrow v_0'$ and $g_2 :: v_0\ R\ v_0'$:
   By Lemma 5.3 on $\mathcal{E}_1$, we have $q_1 :: v = v_0$. By Lemma 5.1 on v', we establish $g_1 :: v' \Downarrow v'$, picking $v_0' = v'$. But then by q_1, we establish g_2 by a_1, and we are done.

2. For any v_0', if $\mathcal{E}_1 :: v' \Downarrow v_0'$, then there is a v_0 such that $g_1 :: v \Downarrow v_0$ and $g_2 :: v_0 \; R \; v_0'$:

   Analogous to the above. □

Lemma 5.16 (Monadic bind)**.** *Suppose R and S are relations on values and $\mathcal{F}, \mathcal{F}'$ are reduction frames. If $a_1 :: e \; R^\dagger \; e'$ and $a_2 :: \forall v, v'. \, v \; R \; v' \Rightarrow \mathcal{F}\{v\} \; S^\dagger \; \mathcal{F}'\{v'\}$, then $g :: \mathcal{F}\{e\} \; S^\dagger \; \mathcal{F}'\{e'\}$.*

Proof. Assume a_1, a_2 as introduced above. To establish g, it suffices to show the following two subgoals:

Subgoal 1. *For any v, if $\mathcal{E} :: \mathcal{F}\{e\} \Downarrow v$ then there is a v' and a derivation $\mathcal{E}' :: \mathcal{F}'\{e'\} \Downarrow v'$ such that $g' :: v \; S \; v'$.*

By Lemma 5.14 on $\mathcal{E}$, there is a v_0 and derivations $\mathcal{E}_0 :: e \Downarrow v_0$ and $\mathcal{E}_f :: \mathcal{F}\{v_0\} \Downarrow v$. By a_1 and $\mathcal{E}_0$, there is a v_0' and derivation $\mathcal{E}_0' :: e' \Downarrow v_0'$, and we have $r :: v_0 \; R \; v_0'$. Thus, by a_2 on r, we get $s :: \mathcal{F}\{v_0\} \; S^\dagger \; \mathcal{F}'\{v_0'\}$.

But then by s on $\mathcal{E}_f$, there is a v'' and a derivation $\mathcal{E}'' :: \mathcal{F}\{v_0'\} \Downarrow v''$ such that $g'' :: v \; S \; v''$. But then we pick $v' = v''$, and construct $\mathcal{E}'$ by Lemma 5.13 on $\mathcal{E}_0'$ and $\mathcal{E}''$.

Subgoal 2. *For any v', if $\mathcal{E}' :: \mathcal{F}'\{e'\} \Downarrow v'$, then there is a v and a derivation $\mathcal{E} :: \mathcal{F}\{e\} \Downarrow v$ such that $g' :: v \; S \; v'$.*

Analogous to the above. □

The bind lemma can be generalized to reduction contexts by generalizing Lemma 5.13 and Lemma 5.14:

Lemma 5.17 (Converse context evaluation)**.** *If $\mathcal{E} :: e \Downarrow v$ and $\mathcal{E}' :: \mathcal{R}\{v\} \Downarrow v'$, then there is a $\mathcal{E}'' :: \mathcal{R}\{e\} \Downarrow v'$.*

Proof. By induction on the reduction context $\mathcal{R}$.

- Case $\mathcal{R} = \circ$. Then $\mathcal{E}'$ is a derivation of $v \Downarrow v'$, and thus by Lemma 5.3 we have $v = v'$. But then we can just obtain $\mathcal{E}''$ by $\mathcal{E}$, and we are done.

- Case $\mathcal{R} = \mathcal{F}\{\mathcal{R}'\}$. Then $\mathcal{E}'$ is a derivation of $\mathcal{F}\{\mathcal{R}'\{v\}\} \Downarrow v'$. Thus, we can apply Lemma 5.14 on $\mathcal{E}'$, to obtain a v_0' and derivations $\mathcal{E}_0' :: \mathcal{R}'\{v\} \Downarrow v_0'$ and $\mathcal{E}_0'' :: \mathcal{F}\{v_0'\} \Downarrow v'$.

  By the induction hypothesis on $\mathcal{R}'$ with $\mathcal{E}$ and $\mathcal{E}_0'$, we obtain a derivation $\mathcal{E}_1' :: \mathcal{R}'\{e\} \Downarrow v_0'$.

  Finally, by Lemma 5.13 on $\mathcal{E}_1'$ and $\mathcal{E}_0''$, we obtain a derivation of $\mathcal{F}\{\mathcal{R}'\{e\}\} \Downarrow v'$, and we are done.

□

Lemma 5.18 (Context evaluation extraction)**.** *If $\mathcal{E} :: \mathcal{R}\{e\} \Downarrow v$, there exists a v' such that $\mathcal{E}' :: e \Downarrow v'$ and $\mathcal{E}'' :: \mathcal{R}\{v'\} \Downarrow v$.*

Proof. By induction on the reduction context $\mathcal{R}$.

- Case $\mathcal{R} = \circ$. Then $\mathcal{E}$ is a derivation of $e \Downarrow v$. Thus we can pick $v' = v$, $\mathcal{E}' = \mathcal{E}$ and construct $\mathcal{E}''$ by Lemma 5.1 on v.
- Case $\mathcal{R} = \mathcal{F}\{\mathcal{R}'\}$. Then $\mathcal{E}$ is a derivation of $\mathcal{F}\{\mathcal{R}'\{e\}\} \Downarrow v$. By Lemma 5.14 on $\mathcal{E}$, there is a v'_0 such that $\mathcal{E}'_0 :: \mathcal{R}'\{e\} \Downarrow v'_0$ and $\mathcal{E}''_0 :: \mathcal{F}\{v'_0\} \Downarrow v$.

  By the induction hypothesis on $\mathcal{R}'$ with $\mathcal{E}'_0$, there is a v'_1 such that $\mathcal{E}'_1 :: e \Downarrow v'_1$ and $\mathcal{E}''_1 :: \mathcal{R}'\{v'_1\} \Downarrow v'_0$.

  We now pick $v' = v'_1$ and can therefore obtain $\mathcal{E}'$ by $\mathcal{E}'_1$. It thus remains to construct a derivation of
  $$\mathcal{F}\{\mathcal{R}'\{v'_1\}\} \Downarrow v,$$
  which is obtained by Lemma 5.13 on $\mathcal{E}''_1$ and $\mathcal{E}''_0$, and we are done.

□

Lemma 5.19 (Monadic bind, contexts)**.** *Suppose R and S are relations on values and $\mathcal{R}, \mathcal{R}'$ are reduction contxts. If $a_1 :: e\ R^\dagger e'$ and $a_2 :: \forall v, v'.\, v\ R\ v' \Rightarrow \mathcal{R}\{v\}\ S^\dagger\ \mathcal{R}'\{v'\}$, then $g :: \mathcal{R}\{e\}\ S^\dagger\ \mathcal{R}'\{e'\}$.*

Proof. Analogous to the proof of Lemma 5.16, but by using Lemma 5.17 and Lemma 5.18 in place of Lemma 5.13 and Lemma 5.14, respectively. □

Another useful property is the converse of monadic return, which says that the computation extension implies the underlying relation at values:

Lemma 5.20 (Extract)**.** *Suppose R is a relation on values, and v, v' are values. If $c :: v\ R^\dagger\ v'$, then also $v\ R\ v'$.*

Proof. Assume c. By Lemma 5.1 on v, we get $\mathcal{E} :: v \Downarrow v$. By c on $\mathcal{E}$, there exists a v'_0 such that $\mathcal{E}'_0 :: v' \Downarrow v'_0$ and $v\ R\ v'_0$. By Lemma 5.3 on $\mathcal{E}'_0$, we have $v'_0 = v'$, and we are done. □

5.2.2 Properties of logical equivalence

We begin by verifying that logical equivalence is actually a relation on values:

Lemma 5.21. *Logical equivalence is a relation on values. I.e., if $v \sim_\tau v'$, then also v* value *and v'* value*.*

Proof. By cases on τ.

For $\tau = \mathsf{nat}$, the result follows by inner induction on the definition of numerals, using v_zero in the base case and v_succ in the inductive case.

For $\tau = \tau_2 \to \tau_0$, the result is immediate by v_lam. □

This justifies the implicit assumption that whenever two expressions are logically equivalent, they are also values.

Like in Chapter 4, logical equivalence is a partial equivalence relation. Together with the earlier results, the computation extension of logical equivalence is a partial equivalence too:

Lemma 5.22 (Symmetry)**.** *For any type τ and values v, v', if $v \sim_\tau v'$ then $v' \sim_\tau v$.*

Proof. Assume $h_1 :: v \sim_\tau v'$. We proceed by induction over τ.

- Case $\tau = \mathsf{nat}$. The first result follows directly from Definition 5.9.

- Case $\tau = \tau_2 \rightharpoonup \tau_0$. Then by Definition 5.9, we have expressions e_0, e_0' such that $h_2 :: v = \lambda x.\, e_0$ and $h_3 :: v' = \lambda x'.\, e_0'$ and $h_4 :: \forall v_2, v_2'.\, v_2 \sim_{\tau_2} v_2' \Rightarrow e_0[v_2/x] \sim^\dagger_{\tau_0} e_0'[v_2'/x']$.

  It suffices to show that for any v_2, v_2', if $h_5 :: v_2' \sim_{\tau_2} v_2$, then $e_0'[v_2'/x'] \sim^\dagger_{\tau_0} e_0[v_2/x]$. By IH on τ_2 with h_5, we obtain $h_6 :: v_2 \sim_{\tau_2} v_2'$. By h_6 and h_4, we get $h_7 :: e_0[v_2/x] \sim^\dagger_{\tau_0} e_0'[v_2'/x']$. But then by IH on τ_0 and Lemma 5.3, we are done. □

Lemma 5.23 (Transitivity)**.** *For any values v, v', v'', if $v \sim_\tau v'$ and $v' \sim_\tau v''$, then $v \sim_\tau v''$.*

Proof. Assume $h_1 :: v \sim_\tau v'$ and $h_2 :: v' \sim_\tau v''$. We proceed by induction on τ:

- Case $\tau = \mathsf{nat}$. By assumption there exists some n and n' such that $\mathcal{V}_1 :: v \overset{\mathrm{num}}{\leftrightarrow} n$, $\mathcal{V}_1' :: v' \overset{\mathrm{num}}{\leftrightarrow} n$, $\mathcal{V}_2 :: v' \overset{\mathrm{num}}{\leftrightarrow} n'$ and $\mathcal{V}_2' :: v'' \overset{\mathrm{num}}{\leftrightarrow} n'$. By Lemma 5.6 on $\mathcal{V}_1'$ and $\mathcal{V}_2$, we have $n = n'$. Thus the desired result follows directly by h_1, h_2.

- Case $\tau_2 \rightharpoonup \tau_0$. It suffices to show that there exists e, e'' such that $v = \lambda x.\, e$ and $v'' = \lambda x''.\, e''$ and $\forall v_2, v_2''.\, v_2 \sim_{\tau_2} v_2'' \Rightarrow e[v_2/x] \sim^\dagger_{\tau_0} e''[v_2''/x'']$. By h_1, there are expressions e_0, e_0' such that $h_3 :: v = \lambda x.\, e_0$ and $h_4 :: v' = \lambda x'.\, e_0'$ and $h_5 :: \forall v_2, v_2'.\, v_2 \sim_{\tau_2} v_2' \Rightarrow e_0[v_2/x] \sim^\dagger_{\tau_0} e_0'[v_2'/x']$. By h_2 together with h_4, there is an expression e_0'' such that $h_6 :: v'' = \lambda x''.\, e_0''$ and $h_7 :: \forall v_2', v_2''.\, v_2' \sim_{\tau_2} v_2'' \Rightarrow e_0'[v_2'/x'] \sim^\dagger_{\tau_0} e_0''[v_2''/x'']$.

  Picking $e = e_0$ and $e'' = e_0''$, assume $h_8 :: v_2 \sim_{\tau_2} v_2''$. It suffices to show $e_0[v_2/x] \sim^\dagger_{\tau_0} e_0''[v_2''/x'']$. By Lemma 5.22 on h_8, we get $h_8' :: v_2'' \sim_{\tau_2} v_2$. By IH on h_8, h_8', we get $h_9 :: v_2 \sim_{\tau_2} v_2$. By h_1 and h_9, we get $h_{10} :: e_0[v_2/x] \sim^\dagger_{\tau_0} e_0'[v_2/x']$. By h_2 and h_8, we get $h_{11} :: e_0'[v_2/x'] \sim^\dagger_{\tau_0} e_0''[v_2''/x'']$. But then by Lemma 5.11 and IH on τ_0 with h_{10} and h_{11}, we are done. □

To show congruence, we need to reason about open expressions. We therefore reintroduce the concept of open logical equivalence from Chapter 4, but with the difference that closing substitutions are restricted, such that they onle substitute closed values for variables, instead of expressions:

Definition 5.24 (Closing substitution)**.** A *closing substitution* γ is a finite function from variables $x_1, \ldots, x_n$ to closed values $v_1, \ldots, v_n$.

We write $\hat{\gamma}(e)$ for the substitution $e[\gamma(x_1), \ldots, \gamma(x_n)/x_1, \ldots, x_n]$. ♢

Definition 5.25 (Pointwise equivalence)**.** Given two closing substitutions γ, γ', we write $\gamma \sim_\Gamma \gamma'$ if $\mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma) = \mathrm{dom}(\gamma')$, and for any $x \in \mathrm{dom}(\Gamma)$, we have $\gamma(x) \sim_{\Gamma(x)} \gamma'(x)$. ♢

Definition 5.26 (Open logical equivalence)**.** Suppose e, e' are open expressions. *Open logical equivalence*, written $\Gamma \vdash e \sim e' : \tau$ means that for any closing substitutions γ, γ', if $\gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim^\dagger_\tau \hat{\gamma}'(e')$. ♢

Symmetry and transitivity is preserved for open logical equivalence:

Lemma 5.27 (Symmetry, open expressions)**.** *If* $\Gamma \vdash e \sim e' : \tau$*, then* $\Gamma \vdash e' \sim e : \tau$*.*

Proof. Assume $h_1 :: \Gamma \vdash e \sim e' : \tau$. It suffices to show that for any γ, γ', if $h_2 :: \gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e') \sim^\dagger_\tau \hat{\gamma}'(e)$. By pointwise application of Lemma 5.22 on h_2, we get $h_2' :: \gamma' \sim_\Gamma \gamma$. By h_2' and h_1, we get $h_3 :: \hat{\gamma}'(e) \sim^\dagger_\tau \hat{\gamma}(e')$. But then by Lemma 5.22 on h_3, we are done. □

Lemma 5.28 (Transitivity, open expressions)**.** *If* $\Gamma \vdash e \sim e' : \tau$ *and* $\Gamma \vdash e' \sim e'' : \tau$*, then* $\Gamma \vdash e \sim e'' : \tau$*.*

Proof. Assume $h_1 :: \Gamma \vdash e \sim e' : \tau$ and $h_2 \Gamma \vdash e' \sim e'' : \tau$. It suffices to show that for any γ, γ', if $h_3 :: \gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim^\dagger_\tau \hat{\gamma}'(e'')$.

By h_1 and h_3, we get $h_4 :: \hat{\gamma}(e) \sim^\dagger_\tau \hat{\gamma}'(e')$. By pointwise application of Lemma 5.22 on h_3, we get $h_3' :: \gamma' \sim_\Gamma \gamma$. By pointwise application of Lemma 5.23 on h_3', h_3, we get $h_5 :: \gamma' \sim_\Gamma \gamma'$. Then by h_2 and h_5, we get $h_6 :: \hat{\gamma}'(e') \sim^\dagger_\tau \hat{\gamma}'(e'')$. But then by Lemma 5.23 on h_4 and h_6, we are done. □

5.3 Logical equivalence is a congruence relation

We now proceed with showing that open logical equivalence is a congruent equivalence relation. Like we did in Chapter 4, we take the approach of showing that the relation is congruent at each recursive expression constructor, and that the relation is reflexive at atomic (leaf) expressions.

Lemma 5.29 (Congruence at abstraction, open expressions)**.** *If* $\Gamma, x : \tau_2 \vdash e_0 \sim e_0' : \tau_0$*, then* $\Gamma \vdash \lambda x.\, e_0 \sim \lambda x.\, e_0' : \tau_2 \rightharpoonup \tau_0$*.*

Proof. Assume $h_1 : \Gamma, x : \tau_2 \vdash e_0 \sim e_0' : \tau_0$. It suffices to show that for any γ, γ', if $h_2 :: \gamma \sim_\Gamma \gamma'$, then $\lambda x.\, \hat{\gamma}(e_0) \sim^\dagger_{\tau_2 \rightharpoonup \tau_0} \lambda x.\, \hat{\gamma}'(e_0')$. By Lemma 5.15, it suffices to show $\lambda x.\, \hat{\gamma}(e_0) \sim_{\tau_2 \rightharpoonup \tau_0} \lambda x.\, \hat{\gamma}'(e_0')$.

By Definition 5.9, we need to show that for any v_2, v_2', if $h_3 :: v_2 \sim_{\tau_2} v_2'$, then $e_0[v_2/x] \sim^\dagger_{\tau_0} e_0'[v_2'/x]$.

By constructing $\gamma_0 = \gamma[x \mapsto v_2]$ and $\gamma_0' = \gamma'[x \mapsto v_2']$, we get by h_2 and h_3 that $h_4 :: \gamma_0 \sim_{\Gamma,x:\tau_2} \gamma_0'$. But then by h_4 and h_1, we get $h_5 :: \hat{\gamma_0}(e_0) \sim^{\dagger}_{\tau_0} \hat{\gamma_0}'(e_0')$, which by our definition of γ_0, γ_0' is equivalent to $\hat{\gamma}(e_0)[v_2/x] \sim^{\dagger}_{\tau_0} \hat{\gamma}'(e_0')[v_2'/x]$, and we are done. □

Congruence at application is shown in two steps. First we verify that logically equivalent values at function type takes related arguments to related results. We then generalize the result to open logical equivalence using the bind lemma we proved earlier.

Lemma 5.30 (Logical application)**.** *If $a_1 :: v_1 \sim_{\tau_2 \to \tau_0} v_1'$ and $a_2 :: v_2 \sim_{\tau_2} v_2'$, then $v_1\ v_2 \sim^{\dagger}_{\tau_0} v_1'\ v_2'$.*

Proof. It suffices to show both of the following subgoals. We will show the first, since the other is analogous.

Subgoal 1. *For any value v, if $\mathcal{E} :: v_1\ v_2 \Downarrow v$, then there is a v' such that $\mathcal{E}' :: v_1'\ v_2' \Downarrow v'$ and $r' :: v \sim_{\tau_0} v'$.*

By a_1, there must exist open expressions $x.\, e_0$ and $x'.\, e_0'$ such that

$$\begin{array}{rcl} q_1 & :: & v_1 = \lambda x.\, e_0, \\ q_1' & :: & v_1' = \lambda x'.\, e_0', \\ f & :: & \forall v_2, v_2'.\, v_2 \sim_{\tau_2} v_2' \Rightarrow e_0[v_2/x] \sim^{\dagger}_{\tau_0} e_0'[v_2'/x]. \end{array}$$

By f on a_2, we have

$$r_0 \quad :: \quad e_0[v_2/x] \sim^{\dagger}_{\tau_0} e_0'[v_2'/x].$$

The derivation $\mathcal{E}$ must end in an application of e_app, implying that there exists derivations

$$\begin{array}{rcl} \mathcal{E}_{01} & :: & v_1 \Downarrow \lambda x_0.\, e_{00}, \\ \mathcal{E}_{02} & :: & v_2 \Downarrow v_{02}, \\ \mathcal{E}_{03} & :: & e_{00}[v_{02}/x_0] \Downarrow v. \end{array}$$

But then by Lemma 5.3 and q_1 on $\mathcal{E}_{01}$, we have $q_{00} :: x_0.\, e_{00} = x.\, e_0$. By Lemma 5.3 on $\mathcal{E}_{02}$, we have $q_{02} :: v_{02} = v_2$. By q_{00} and q_{02} on $\mathcal{E}_{03}$, we have

$$\mathcal{E}_3 \quad :: \quad e_0[v_2/x] \Downarrow v.$$

Now, by r_0 on $\mathcal{E}_3$, there exists a v_0' such that $\mathcal{E}_3' :: e_0'[v_2'/x] \Downarrow v_0'$ and $r_0' :: v \sim_{\tau_0} v_0'$. We pick $v' = v_0'$ and $r' = r_0'$, and it suffices to show that there is a derivation

$$\mathcal{E}' \quad :: \quad v_1'\ v_2' \Downarrow v'.$$

By Lemma 5.1 on v_1' (with q_1'), we have a derivation $\mathcal{E}_1' :: v_1' \Downarrow \lambda x'.\, e_0'$. By Lemma 5.1 on v_2', we have a derivation $\mathcal{E}_2' :: v_2' \Downarrow v_2'$. But then we construct the desired $\mathcal{E}'$ by e_app on $\mathcal{E}_1'$, $\mathcal{E}_2'$ and $\mathcal{E}_3'$, and we are done.

Subgoal 2. *For any value v', if $\mathcal{E}' :: v_1'\ v_2' \Downarrow v'$, then there is a v such that $\mathcal{E} :: v_1\ v_2 \Downarrow v$ and $r :: v \sim_{\tau_0} v'$.*

The argument is analogous to the one above. □

Lemma 5.31 (Congruence at application, open expressions)**.** *If* $a_1 :: \Gamma \vdash e_1 \sim e_1' : \tau_2 \rightharpoonup \tau_0$ *and* $a_2 :: \Gamma \vdash e_2 \sim e_2' : \tau_2$*, then* $\Gamma \vdash e_1\ e_2 \sim e_1'\ e_2' : \tau_0$.

Proof. It suffices to show that for any γ, γ', if $h_1 :: \gamma \sim_\Gamma \gamma'$, then

$$\hat{\gamma}(e_1)\ \hat{\gamma}(e_2) \sim^\dagger_{\tau_0} \hat{\gamma}'(e_1')\ \hat{\gamma}'(e_2').$$

By a_1 and h_1, we have $a_1' :: \hat{\gamma}(e_1) \sim_{\tau_2 \rightharpoonup \tau_0} \hat{\gamma}'(e_1')$. By constructing reduction frames $\mathcal{F}_1 = \circ\ \hat{\gamma}(e_2)$ and $\mathcal{F}_1' = \circ\ \hat{\gamma}'(e_2')$, then by Lemma 5.16 on a_1', $\mathcal{F}_1$ and $\mathcal{F}_1'$, it suffices to show that for any values v_1, v_1', if $h_2 :: v_1 \sim_{\tau_2 \rightharpoonup \tau_0} v_1'$, then

$$v_1\ \hat{\gamma}(e_2) \sim^\dagger_{\tau_0} v_1'\ \hat{\gamma}'(e_2').$$

By a_2 and h_1, we have $a_2' :: \hat{\gamma}(e_2) \sim^\dagger_{\tau_2} \hat{\gamma}'(e_2')$. By constructing reduction frames $\mathcal{F}_2 = v_1 \circ$ and $\mathcal{F}_2' = v_1' \circ$, then again by Lemma 5.16 on a_2', $\mathcal{F}_2$ and $\mathcal{F}_2'$, it suffices to show that for any values v_2, v_2' if $h_3 :: v_2 \sim_{\tau_2} v_2'$, then

$$v_1\ v_2 \sim^\dagger_{\tau_0} v_1'\ v_2'.$$

But this follows by Lemma 5.30 on h_2, h_3, and we are done. □

Lemma 5.32 (Congruence at successor)**.** *If* $a_1 :: \Gamma \vdash e \sim e' : \mathsf{nat}$*, then* $\Gamma \vdash \mathsf{succ}(e) \sim \mathsf{succ}(e') : \mathsf{nat}$.

Proof. By constructing reduction frames $\mathcal{F} = \mathcal{F}' = \mathsf{succ}(\circ)$, then by Lemma 5.16 on a_1, $\mathcal{F}$ and $\mathcal{F}'$, it suffices to show that for any values v, v', if $h_1 :: v \sim_{\mathsf{nat}} v'$, then also

$$\mathsf{succ}(v) \sim^\dagger_{\mathsf{nat}} \mathsf{succ}(v').$$

But then by Lemma 5.15, it suffices to show

$$\mathsf{succ}(v) \sim_{\mathsf{nat}} \mathsf{succ}(v').$$

That is, that there exists an n such that $\mathsf{succ}(v) \overset{\mathrm{num}}{\leftrightarrow} n$ and $\mathsf{succ}(v') \overset{\mathrm{num}}{\leftrightarrow} n$.

By Definition 5.9 and h_1, there exists an n_0 such that $v \overset{\mathrm{num}}{\leftrightarrow} n_0$ and $v' \overset{\mathrm{num}}{\leftrightarrow} n_0$. But then we just choose $n = \mathsf{succ}(n_0)$, and we are done. □

Lemma 5.33 (Congruence at case)**.** *If* $a_0 :: \Gamma \vdash e_0 \sim e_0' : \mathsf{nat}$ *and* $a_1 :: \Gamma \vdash e_1 \sim e_1' : \tau$ *and* $a_2 :: \Gamma, x : \mathsf{nat} \vdash e_2 \sim e_2' : \tau$*, then also* $g :: \Gamma \vdash \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \sim \mathsf{ncase}(e_0',\ e_1',\ x.\, e_2') : \tau$.

Proof. It suffices to show that for any γ, γ', if $h_1 :: \gamma \sim_\Gamma \gamma'$, then

$$\begin{aligned}&\mathsf{ncase}(\hat{\gamma}(e_0),\ \hat{\gamma}(e_1),\ x.\, \hat{\gamma}(e_2))\\ &\qquad \sim^\dagger_\tau \mathsf{ncase}(\hat{\gamma}'(e_0'),\ \hat{\gamma}'(e_1'),\ x.\, \hat{\gamma}'(e_2')).\end{aligned}$$

By h_1 on a_0, a_1, we get

$$\begin{aligned} a_0' &:: \hat{\gamma}(e_0) \sim_{\mathsf{nat}} \hat{\gamma}'(e_0'),\\ a_1' &:: \hat{\gamma}(e_1) \sim_\tau \hat{\gamma}'(e_1').\end{aligned}$$

By constructing reduction frames

$$\begin{array}{rcl} \mathcal{F} & = & \mathsf{ncase}(\circ,\ \hat{\gamma}(e_1),\ x.\,\hat{\gamma}(e_2)), \\ \mathcal{F}' & = & \mathsf{ncase}(\circ,\ \hat{\gamma}'(e_1'),\ x.\,\hat{\gamma}'(e_2')), \end{array}$$

then by Lemma 5.16 on a_1' and $\mathcal{F}$, $\mathcal{F}'$, it suffices to show that for any values v_0, v_0', if $h_2 :: v_0 \sim_{\mathsf{nat}} v_0'$, then

$$\begin{array}{l} \mathsf{ncase}(v_0,\ \hat{\gamma}(e_1),\ x.\,\hat{\gamma}(e_2)) \\ \qquad \sim_\tau^\dagger \mathsf{ncase}(v_0',\ \hat{\gamma}'(e_1'),\ x.\,\hat{\gamma}'(e_2')). \end{array}$$

It suffices to show the following subgoals.

Subgoal 1. *For any v, if $\mathcal{E} :: \mathcal{F}\{v_0\} \Downarrow v$, then there exists a v' such that $\mathcal{E}' :: \mathcal{F}'\{v_0'\}$ and $r :: v \sim_\tau v'$.*

By Definition 5.9 and h_2, there must exist some n such that $\mathcal{V} :: v_0 \overset{\mathrm{num}}{\leftrightarrow} n$ and $\mathcal{V}' :: v_0' \overset{\mathrm{num}}{\leftrightarrow} n$. By cases on derivations and Lemma 5.5, we then either have $v_0 = v_0' = \mathsf{zero}$; or $v_0 = \mathsf{succ}(v_{01})$ and $v_0' = \mathsf{succ}(v_{01})$ for some n_{01}, v_{01} where $\mathcal{V}_{01} :: v_{01} \overset{\mathrm{num}}{\leftrightarrow} n_{01}$:

- Subcase $v_0 = v_0' = \mathsf{zero}$. Then $\mathcal{E}$ must end in e_case0: Since $v_0 \Downarrow \mathsf{zero}$ by e_zero, then e_case1 is impossible by Lemma 5.3. This implies that we have a derivation $\mathcal{E}_1 :: \hat{\gamma}(e_1) \Downarrow v$. But then by a_2' on $\mathcal{E}_1$, there exists a v'' such that $\mathcal{E}_1'' :: \hat{\gamma}'(e_1') \Downarrow v''$ and $r'' :: v \sim_\tau v''$. We pick $v' = v''$ and get r' by r'', and construct $\mathcal{E}'$ by e_case0 on $\mathcal{E}_1''$ and e_zero on v_0' (justified since we know from before that $v_0' = v_0 = \mathsf{zero}$).
- Subcase $v_0 = v_0' = \mathsf{succ}\,(v_{01})$. Then $\mathcal{E}$ must end in e_case1, by ruling out the other possibility using Lemma 5.3 and Lemma 5.1 on v_0. This implies that we have a derivation $\mathcal{E}_2 :: \hat{\gamma}(e_2)[v_{01}/x] \Downarrow v$.

  We construct substitutions $\gamma_0 = \gamma[x \mapsto v_{01}]$ and $\gamma_0' = \gamma'[x \mapsto v_{01}]$, and observe that since $v_{01} \sim_{\mathsf{nat}} v_{01}$ (directly by definition and $\mathcal{V}_{01}$), then by h_1 we have $h_1' :: \gamma_0 \sim_{\Gamma, x:\mathsf{nat}} \gamma_0'$. By h_1' and a_2, we then obtain $a_2'' :: \hat{\gamma_0}(e_2) \sim_\tau^\dagger \hat{\gamma_0}'(e_2')$, and since the codomain of γ, γ' is closed expressions, we also have that $a_2' :: \hat{\gamma}(e_2)[v_{01}/x] \sim_\tau^\dagger \hat{\gamma}'(e_2')[v_{01}/x]$.

  Now, by a_2' on $\mathcal{E}_2$, there is a v'' such that $\mathcal{E}_2'' :: \hat{\gamma}'(e_2')[v_{01}/x] \Downarrow v''$ and $r'' :: v \sim_\tau v''$. By picking $v' = v''$, we obtain r by r'' and construct $\mathcal{E}'$ by e_case1 on $\mathcal{E}_2''$ and the result of Lemma 5.1 on v_0' (since we know from before that $v_0' = v_0 = \mathsf{succ}\,(v_{01})$).

Subgoal 2. *For any v', if $\mathcal{E}' :: \mathcal{F}'\{v_0'\} \Downarrow v'$, then there exists a v such that $\mathcal{E} :: \mathcal{F}\{v_0\} \Downarrow v$ and $r :: v \sim_\tau v'$.*

The argument is analogous to the one above. □

Lemma 5.34 (Congruence at choice)**.** *If* $a_1 :: \Gamma \vdash e_1 \sim e_1' : \tau$ *and* $a_2 :: \Gamma \vdash e_2 \sim e_2' : \tau$*, then* $g :: \Gamma \vdash e_1 \,[\!]\, e_2 \sim e_1' \,[\!]\, e_2' : \tau$*.*

Proof. It suffices to show that for any γ, γ', if $h_1 :: \gamma \sim_\Gamma \gamma'$, then

$$\hat{\gamma}(e_1) \,[\!]\, \hat{\gamma}(e_2) \sim_\tau^\dagger \hat{\gamma}'(e_1') \,[\!]\, \hat{\gamma}'(e_2').$$

We need to show the following subgoals.

Subgoal 1. *For any* v*, if* $\mathcal{E} :: \hat{\gamma}(e_1) \,[\!]\, \hat{\gamma}(e_2) \Downarrow v$*, then there is a* v' *such that* $\mathcal{E}' :: \hat{\gamma}'(e_1') \,[\!]\, \hat{\gamma}'(e_2')$ *and* $r :: v \sim_\tau v'$*.*

We must have that $\mathcal{E}$ ends in either e_choice1 or e_choice2; we only cover the first case here, since the other one is analogous. Thus, we have a derivation $\mathcal{E}_1 :: \hat{\gamma}(e_1) \Downarrow v$, and by e_choice1, it suffices to show that there exists a v' such that $\hat{\gamma}'(e_1') \Downarrow v'$ and $v \sim_\tau v'$. But by h_1 and a_1 on $\mathcal{E}_1'$ we get both, and we are done.

Subgoal 2. *For any* v'*, if* $\mathcal{E}' :: \hat{\gamma}'(e_1') \,[\!]\, \hat{\gamma}'(e_2') \Downarrow v'$*, then there is a* v *such that* $\mathcal{E} :: \hat{\gamma}(e_1) \,[\!]\, \hat{\gamma}(e_2)$ *and* $r :: v \sim_\tau v'$*.*

The argument is analogous to the one above. □

Lemma 5.35 (Reflexivity at zero)**.** *For any* Γ*, we have* $\Gamma \vdash \mathsf{zero} \sim \mathsf{zero} : \mathsf{nat}$*.*

Proof. It suffices to show that for any γ, γ', if $h_1 :: \gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(\mathsf{zero}) \sim_{\mathsf{nat}}^\dagger \hat{\gamma}'(\mathsf{zero})$, which is equivalent to showing $\mathsf{zero} \sim_{\mathsf{nat}}^\dagger \mathsf{zero}$.

We will show one direction of Definition 5.8 - the other is equivalent. Assume for some v that $\mathcal{E} :: \mathsf{zero} \Downarrow v$. We need to show that there is a v' such that $\mathsf{zero} \Downarrow v'$ and $v \sim_{\mathsf{nat}} v'$. The first follows by picking $v' = v$.

It remains to find an n such that $v \overset{\text{num}}{\leftrightarrow} n$ and $v' \overset{\text{num}}{\leftrightarrow} n$. The derivation $\mathcal{E}$ must end in e_zero, meaning that $v = \mathsf{zero}$. By definition, we can pick $n = 0$, and we are done. □

5.4 Axiomatic equational reasoning

In this section, we will present an axiomatization of observational equivalence. We will then show soundness of the system via our previously defined logical relation.

Our axiomatization will, first of all, need to include the usual rules of symmetry, reflexivity, transitivity and congruence which makes it into a congruence relation. Having only such rules will not enable us to show anything but the most trivial equivalences though, so we need to add additional rules that capture equivalence between certain *computations*. For example, we would like to be able to derive that the expression $(\lambda x.\, \mathsf{ncase}(x,\ e_1,\ y.\, e_2))\ \mathsf{zero}$ is equivalent to e_1. Thus, we will need rules for reasoning about β-reduction and case analysis as well.

In a pure call-by-name language, β-reduction will always result in an equivalent expression. In the case of a call-by-value language with side effects, things are not so

Open values: $\mathcal{O} :: \boxed{v\downarrow}$:

$$\textsf{o\_lam}: \frac{}{\lambda x.\, e_0 \downarrow} \qquad \textsf{o\_zero}: \frac{}{\textsf{zero} \downarrow} \qquad \textsf{o\_succ}: \frac{v\downarrow}{\textsf{succ}(v)\downarrow} \qquad \textsf{o\_var}: \frac{}{x\downarrow}$$

Figure 5.3: *Open values.*

simple though. For example, we cannot in general say that an expression of the form $(\lambda x.\, \textsf{zero})\ e_2$ is equivalent to zero, since e_2 might fail. As another example, consider the expression $e \equiv (\lambda x.\, f\ x\ x)\ e_2$, where f is a function returning succ(zero) if the arguments are either both zero or non-zero at the same time, and zero otherwise. Even if e_2 cannot fail, this is *not* necessarily equivalent to $e' \equiv f\ e_2\ e_2$, since e_2 could be the non-deterministic expression zero $[]$ succ(zero). The expression e requires e_2 to fully evaluate to a single value *before* applying the f function, while e' allows each argument to f to evaluate to possibly different values. Thus, we can derive $e' \Downarrow \textsf{zero}$, while $e \not\Downarrow \textsf{zero}$.

To remedy this, we can restrict our β-reduction rule such that it can only show equivalence of applications where the subexpression occuring as argument cannot have observable side-effects. This is exactly the expressions that are equivalent to values, hence we only consider a more restriced β-value-reduction. As only values are substituted for variables during evaluation, we are going to treat variables as values as well, which allows reasoning about β-value-reduction under lambdas. We express the fact that variables stand for values with a judgment characterizing open values, which can be seen in Figure 5.3.

We can easily show that open values are sound with respect to closing substitutions:

Lemma 5.36 (Open value soundness)**.** *Suppose* $\mathcal{O} :: e\downarrow$*. Then for any closing substitution* γ *where* $\mathrm{dom}(\gamma) \supseteq \mathrm{FV}(e)$*, we have* $\hat{\gamma}(e) = v$ *and* v value *for some* v*.*

Proof. By straightforward induction on the derivation $\mathcal{O}$. In the case for o_var, we pick v from the codomain of γ and get v value by assumption. In the case v_succ we proceed by induction on the single subderivation, and for v_lam we observe that $\hat{\gamma}(e)$ must be a lambda abstraction and hence a value. □

We define axiomatic equivalence in Figure 5.4 and Figure 5.5. The Twelf encoding is straightforward, and is encoding as the following type family:

```
sim : exp -> exp -> tp -> type.
```

The encoding is very similar to the one in Section 4.4, and is hence omitted. We refer to Appendix C.3 for the full definition.

Judgment $\boxed{\Gamma \vdash e \doteq e' : \tau}$:

$$\textsf{q\_var}: \frac{}{\Gamma \vdash x \doteq x : \tau}\,(\Gamma(x) = \tau)$$

$$\textsf{q\_sym}: \frac{\Gamma \vdash e \doteq e' : \tau}{\Gamma \vdash e' \doteq e : \tau} \qquad \textsf{q\_trans}: \frac{\Gamma \vdash e \doteq e' : \tau \quad \Gamma \vdash e' \doteq e'' : \tau}{\Gamma \vdash e \doteq e'' : \tau}$$

$$\textsf{q\_diverge}: \frac{}{\mathsf{diverge} \doteq \mathsf{diverge} : \tau}$$

$$\textsf{q\_zero}: \frac{}{\Gamma \vdash \mathsf{zero} \doteq \mathsf{zero} : \mathsf{nat}} \qquad \textsf{q\_succ}: \frac{\Gamma \vdash e \doteq e' : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e) \doteq \mathsf{succ}(e') : \mathsf{nat}}$$

$$\textsf{q\_lam}: \frac{\Gamma, x : \tau_2 \vdash e_0 \doteq e_0' : \tau_0}{\Gamma \vdash \lambda x.\, e_0 \doteq \lambda x.\, e_0' : \tau_2 \rightharpoonup \tau_0}$$

$$\textsf{q\_app}: \frac{\Gamma \vdash e_1 \doteq e_1' : \tau_2 \rightharpoonup \tau_0 \quad \Gamma \vdash e_2 \doteq e_2' : \tau_2}{\Gamma \vdash e_1\ e_2 \doteq e_1'\ e_2' : \tau_0}$$

$$\textsf{q\_case}: \frac{\Gamma \vdash e_0 \doteq e_0' : \mathsf{nat} \quad \Gamma \vdash e_1 \doteq e_1' : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_2 \doteq e_2' : \tau}{\Gamma \vdash \mathsf{ncase}(e_0,\ e_1,\ x.\, e_2) \doteq \mathsf{ncase}(e_0',\ e_1',\ x.\, e_2') : \tau}$$

$$\textsf{q\_choice}: \frac{\Gamma \vdash e_1 \doteq e_1' : \tau \quad \Gamma \vdash e_2 \doteq e_2' : \tau}{\Gamma \vdash e_1 \,[\!]\, e_2 \doteq e_1' \,[\!]\, e_2' : \tau}$$

Figure 5.4: *Axiomatization of observational equivalence (1/2).*

We can prove soundness by showing that axiomatic equivalence implies logical equivalence. We will show this by induction on derivations, relying mostly on the results in the previous sections for showing soundness of the rules concerned with the basic properties that makes axiomatic equivalence a congruence relation. For the remaining axiomatic equivalence rules concerned with β-value-reduction, context replacement, etc., we will need to show additonal lemmas that establishes the underlying logical equivalence. To conserve space, we only show the proofs for β-value-reduction and η-expansion in detail, and state the remaining lemmas as propositions. They have all been proven in the Twelf formalization.

Lemma 5.37 (β-reduction)**.** *If* $a_1 :: \Gamma, x : \tau_2 \vdash e_0 \sim e_0 : \tau_0$ *and* $a_2 :: \Gamma \vdash e_2 \sim e_2 : \tau_2$ *and* $a_3 :: e_2 \downarrow$*, then also* $\Gamma \vdash (\lambda x.\, e_0)\ e_2 \sim e_0[e_2/x] : \tau_0$.

Proof. Assume a_1, a_2 and a_3 as introduced above. It suffices to show that for any γ, γ', if $h_1 :: \gamma \sim_\Gamma \gamma'$, then also $(\lambda x.\, \hat{\gamma}(e_0))\ \hat{\gamma}(e_2) \sim^{\dagger}_{\tau_0} \hat{\gamma}'(e_0)[\hat{\gamma}'(e_2)/x]$. By h_1, we have that γ and γ' covers all free variables of e_2, and hence by Lemma 5.36 on a_3, there are values v_2, v_2' such that $q_1 :: \hat{\gamma}(e_2) = v_2$ and $q_2 :: \hat{\gamma}'(e_2) = v_2'$.

Hence, it suffices to show the following two subgoals.

Judgment $\boxed{\Gamma \vdash e \doteq e' : \tau}$ (continued) :

$$\textsf{q\_cmerge}: \frac{\Gamma \vdash e_1 \doteq e : \tau \quad \Gamma \vdash e_2 \doteq e : \tau}{\Gamma \vdash e_1 \parallel e_2 \doteq e : \tau} \qquad \textsf{q\_csym}: \frac{\Gamma \vdash e_1 \doteq e_1 : \tau \quad \Gamma \vdash e_2 \doteq e_2 : \tau}{\Gamma \vdash e_1 \parallel e_2 \doteq e_2 \parallel e_1 : \tau}$$

$$\textsf{q\_cassoc}: \frac{\Gamma \vdash e_1 \doteq e_1 : \tau \quad \Gamma \vdash e_2 \doteq e_2 : \tau \quad \Gamma \vdash e_3 \doteq e_3 : \tau}{\Gamma \vdash (e_1 \parallel e_2) \parallel e_3 \doteq e_1 \parallel (e_2 \parallel e_3) : \tau}$$

$$\textsf{q\_rchoice}: \frac{\Gamma \vdash e \doteq e : \tau' \quad \Gamma \vdash e' \doteq e' : \tau' \quad \Gamma, x : \tau' \vdash \mathcal{R}\{x\} \doteq \mathcal{R}\{x\} : \tau}{\Gamma \vdash \mathcal{R}\{e \parallel e'\} \doteq \mathcal{R}\{e\} \parallel \mathcal{R}\{e'\} : \tau}$$

$$\textsf{q\_r}: \frac{\Gamma \vdash e \doteq e' : \tau' \quad \Gamma, x : \tau' \vdash \mathcal{R}\{x\} \doteq \mathcal{R}'\{x\} : \tau}{\Gamma \vdash \mathcal{R}\{e\} \doteq \mathcal{R}'\{e'\} : \tau}$$

$$\textsf{q\_rdiverge}: \frac{}{\Gamma \vdash \mathcal{R}\{\textsf{diverge}\} \doteq \textsf{diverge} : \tau}$$

$$\textsf{q\_rcase}: \frac{\Gamma \vdash e_0 \doteq e_0 : \textsf{nat} \quad \Gamma \vdash e_1 \doteq e_1 : \tau' \quad \Gamma, x' : \textsf{nat} \vdash e_2 \doteq e_2 : \tau' \quad \Gamma, x : \tau' \vdash \mathcal{R}\{x\} \doteq \mathcal{R}\{x\} : \tau}{\Gamma \vdash \mathcal{R}\{\textsf{ncase}(e_0,\ e_1,\ x'.\, e_2)\} \doteq \textsf{ncase}(e_0,\ \mathcal{R}\{e_1\},\ x'.\, \mathcal{R}\{e_2\}) : \tau}$$

$$\textsf{q\_case1}: \frac{\Gamma \vdash e_1 \doteq e_1 : \tau}{\Gamma \vdash \textsf{ncase}(\textsf{zero},\ e_1,\ x.\, e_2) \doteq e_1 : \tau}$$

$$\textsf{q\_case2}: \frac{\Gamma, x : \textsf{nat} \vdash e_2 \doteq e_2 : \tau \quad \Gamma \vdash e_0 \doteq e_0 : \textsf{nat} \quad e_0 \downarrow}{\Gamma \vdash \textsf{ncase}(\textsf{succ}(e_0),\ e_1,\ x.\, e_2) \doteq e_2[e_0/x] : \tau}$$

$$\textsf{q\_eta}: \frac{\Gamma \vdash e \doteq e : \tau_2 \rightharpoonup \tau_0 \quad e \downarrow}{\Gamma \vdash e \doteq \lambda x.\, e(x) : \tau_2 \rightharpoonup \tau_0}\ (x \notin \mathrm{dom}(\Gamma))$$

$$\textsf{q\_beta}: \frac{\Gamma, x : \tau_2 \vdash e_0 \doteq e_0 : \tau_0 \quad \Gamma \vdash e_2 \doteq e_2 : \tau_2 \quad e_2 \downarrow}{\Gamma \vdash (\lambda x.\, e_0)\ e_2 \doteq e_0[e_2/x] : \tau_0}\ (x \notin \mathrm{dom}(\Gamma))$$

Figure 5.5: *Axiomatization of observational equivalence (2/2).*

Subgoal 1. *For any v, if $\mathcal{E} :: (\lambda x.\, \hat{\gamma}(e_0))\ v_2 \Downarrow v$, then there exists a v' and a derivation $\mathcal{E}' :: \hat{\gamma}'(e_0)[v_2'/x] \Downarrow v'$ such that $v \sim_{\tau_0} v'$.*

The derivation $\mathcal{E}$ must end in e_app, implying that there exists a value v_{02}, an open expression $x_0.\, e_{00}$ and derivations

$$\mathcal{E}_1 :: \lambda x.\, \hat{\gamma}(e_0) \Downarrow \lambda x_0.\, e_{00},$$
$$\mathcal{E}_2 :: v_2 \Downarrow v_{02},$$
$$\mathcal{E}_3 :: e_{00}[v_{02}/x_0] \Downarrow v.$$

By Lemma 5.3 on $\mathcal{E}_1$, we have $x.\, e_0 = x_0.\, e_{00}$, and hence $\mathcal{E}_3$ is a derivation of $\hat{\gamma}(e_0)[v_{02}/x] \Downarrow v$.

By a_2, h_1 and $\mathcal{E}_2$, there exists a v_{02}' such that $\mathcal{E}_2' :: \hat{\gamma}'(e_2) \Downarrow v_{02}'$ and $r_2 :: v_{02} \sim_{\tau_2} v_{02}'$. By Lemma 5.3 on $\mathcal{E}_2'$, we get $q_2' :: v_2' = v_{02}'$.

We now define new closing substitutions

$$\gamma_2 = \gamma[x \mapsto v_{02}],$$
$$\gamma_2' = \gamma'[x \mapsto v_{02}'],$$

and note that by h_1 and r_2, it follows that $h_1' :: \gamma_2 \sim_{\Gamma, x:\tau_2} \gamma_2'$. Furthermore, we note that since γ_2, γ_2' are closing substitutions, we have $\hat{\gamma_2}(e_0) = \hat{\gamma}(e_0)[v_{02}/x]$ and

$$\begin{aligned}\hat{\gamma_2}'(e_0) &= \hat{\gamma}'(e_0)[v_{02}'/x] \\ &= \hat{\gamma}'(e_0)[v_2'/x] && \text{(by } q_2'\text{)}.\end{aligned}$$

Hence we can construct v', $\mathcal{E}'$ and $v \sim_{\tau_0} v'$ by a_1 on h_1' and $\mathcal{E}_3$, and we are done.

Subgoal 2. *For any v', if $\mathcal{E}' :: \hat{\gamma}'(e_0)[v_2'/x] \Downarrow v'$, then there exists a v and a derivation $\mathcal{E} :: (\lambda x.\, \hat{\gamma}(e_0))\ v_2 \Downarrow v$ such that $v \sim_{\tau_0} v'$.*

By Lemma 5.1 on v_2', there is a derivation $\mathcal{E}_2' :: v_2' \Downarrow v_2'$. By a_2 on $\mathcal{E}_2'$, there is a v_{02} such that $v_2 \Downarrow v_{02}$ and $v_{02} \sim_{\tau_2} v_2'$. By Lemma 5.3 on $\mathcal{E}_{02}$, we have $v_{02} = v_2$, implying that we also have $\mathcal{E}_2 :: v_2 \Downarrow v_2$ and $r_2 :: v_2 \sim_{\tau_2} v_2'$.

By constructing closing substitutions $\gamma_2 = \gamma[x \mapsto v_2]$ and $\gamma_2' = \gamma'[x \mapsto v_2']$ as in the previous subgoal, and see that by h_1 and r_2, we have $h_1' :: \gamma_2 \sim_{\Gamma, x:\tau_2} \gamma_2'$. Thus, by a_1 on h_1' and $\mathcal{E}'$, there is a v_0 such that $\mathcal{E}_3 :: \hat{\gamma}(e_0)[v_2/x] \Downarrow v_0$ where $v_0 \sim_{\tau_0} v'$. We pick $v = v_0$, and thus it suffices to construct a derivation of $(\lambda x.\, \hat{\gamma}(e_0))\ v_2 \Downarrow v$. By e_lam we have $\mathcal{E}_1 :: \lambda x.\, \hat{\gamma}(e_0) \Downarrow \lambda x.\, \hat{\gamma}(e_0)$, so by e_app on $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$, we are done. □

Lemma 5.38 (η-expansion)**.** *If $a_1 :: \Gamma \vdash e \sim e : \tau_2 \rightharpoonup \tau_0$ and $a_2 :: e \downarrow$ and $x \notin \mathrm{dom}(\Gamma)$, then $\Gamma \vdash e \sim \lambda x.\, e\ x : \tau_2 \rightharpoonup \tau_0$.*

Proof. Assume a_1, a_2 as introduced above. It suffices to show that for any γ, γ', if $h_1 :: \gamma \sim_\Gamma \gamma'$, then $\hat{\gamma}(e) \sim^\dagger_{\tau_2 \rightharpoonup \tau_0} \lambda x.\, \hat{\gamma}'(e)\ x$.

By h_1, it follows that γ, γ' covers all free variables of e, and thus by Lemma 5.36 on a_2, there are values v_0, v_0' such that $q_1 :: \hat{\gamma}(e) = v_0$ and $q_2 :: \hat{\gamma}'(e) = v_0'$.

Addtionally, by a_1 on h_1, we have $c_{20} :: v_0 \sim^\dagger_{\tau_2 \rightharpoonup \tau_0} v_0'$. By Lemma 5.20 on c_{20}, we get $v_0 \sim_{\tau_2 \rightharpoonup \tau_0} v_0'$. But this implies that there exists open expressions $x_0.\, e_0$ and $x_0'.\, e_0'$ such that $v_0 = \lambda x_0.\, e_0$ and $v_0' = \lambda x_0'.\, e_0'$ and that we have

$$f :: \forall v_2, v_2'.\, v_2 \sim_{\tau_2} v_2' \Rightarrow e_0[v_2/x_0] \sim^\dagger_{\tau_0} e_0'[v_2'/x_0'].$$

We thus need to show $\lambda x_0.\, e_0 \sim^\dagger_{\tau_2 \rightharpoonup \tau_0} \lambda x.\, (\lambda x_0'.\, e_0')\ x$. By Lemma 5.15, it suffices to show

$$\lambda x_0.\, e_0 \sim_{\tau_2 \rightharpoonup \tau_0} \lambda x.\, (\lambda x_0'.\, e_0')\ x.$$

Since both expressions are lambda abstractions, it suffices to show that for any v_2, v_2', if $h_2 :: v_2 \sim_{\tau_2} v_2'$, then

$$e_0[v_2/x_0] \sim^\dagger_{\tau_0} (\lambda x_0'.\, e_0')\ v_2'.$$

By f on h_2, we get $c_0 :: e_0[v_2/x_0] \sim^{\dagger}_{\tau_0} e'_0[v'_2/x'_0]$.
To show our goal, we need to show the following two subgoals:

Subgoal 1. *For any value v, if $\mathcal{E} :: e_0[v_2/x_0] \Downarrow v$, then there exists a v' and a derivation $\mathcal{E}' :: (\lambda x'_0.\, e'_0)\ v'_2 \Downarrow v'$ where $v \sim_{\tau_0} v'$.*

By c_0 on $\mathcal{E}$, there exists a v'' such that $\mathcal{E}'_3 :: e'_0[v'_2/x'_0] \Downarrow v''$ and $v \sim_{\tau_2} v''$. By picking $v' = v''$, it suffices to construct the desired derivation $\mathcal{E}'$. By e_lam we get a derivation $\mathcal{E}'_1 :: \lambda x'_0.\, e'_0 \Downarrow \lambda x'_0.\, e'_0$, and by Lemma 5.1 on v'_2, we get a derivation $\mathcal{E}'_2 :: v'_2 \Downarrow v'_2$. But then we construct $\mathcal{E}'$ by e_app on $\mathcal{E}'_1$, $\mathcal{E}'_2$ and $\mathcal{E}'_3$, and we are done.

Subgoal 2. *For any value v', if $\mathcal{E}' :: (\lambda x'_0.\, e'_0)\ v'_2 \Downarrow v'$, then there exists a v and a derivation $\mathcal{E} :: e_0[v_2/x_0] \Downarrow v$ where $v \sim_{\tau_0} v'$.*

The derivation $\mathcal{E}'$ must end in e_app, implying that we have derivations

$$\begin{aligned}
&\mathcal{E}'_1 :: \lambda x'_0.\, e'_0 \Downarrow \lambda x''_0.\, e''_0,\\
&\mathcal{E}'_2 :: v'_2 \Downarrow v''_2,\\
&\mathcal{E}'_3 :: e''_0[v''_2/x''_0] \Downarrow v'.
\end{aligned}$$

By Lemma 5.3 on $\mathcal{E}'_1$, respectively $\mathcal{E}'_2$, we get that $v'_2 = v''_2$ and $x'_0.\, e'_0 = x''_0.\, e''_0$. $\mathcal{E}'_3$ is therefore a derivation of $e'_0[v'_2/x'_0] \Downarrow v'$.

By c_0 on $\mathcal{E}'_3$, there exists a v'' such that $\mathcal{E}'' :: e_0[v_2/x_0] \Downarrow v''$ and $v'' \sim_{\tau_0} v'$. But then by picking $v = v''$ and $\mathcal{E} = \mathcal{E}''$, we are done. □

The cases for the remaining rules are just stated as propositions in the following, but are formally proved in the Twelf formalization:

Proposition 5.39 (Choice merge)**.** *If $\Gamma \vdash e_1 \sim e : \tau$ and $\Gamma \vdash e_2 \sim e : \tau$, then also $\Gamma \vdash e_1 \parallel e_2 \sim e : \tau$.*

Proposition 5.40 (Choice symmetry)**.** *If $\Gamma \vdash e_1 \sim e_1 : \tau$ and $\Gamma \vdash e_2 \sim e_2 : \tau$, then also $\Gamma \vdash e_1 \parallel e_2 \sim e_2 \parallel e_2 : \tau$.*

Proposition 5.41 (Choice associativity)**.** *If $\Gamma \vdash e_1 \sim e_1 : \tau$ and $\Gamma \vdash e_2 \sim e_2 : \tau$ and $\Gamma \vdash e_3 \sim e_3 : \tau$, then also $\Gamma \vdash e_1 \parallel e_2 \parallel e_3 \sim e_1 \parallel e_2 \parallel e_3 : \tau$.*

Proposition 5.42 (Context commutes over choice)**.** *If $\Gamma \vdash e \sim e : \tau'$ and $\Gamma \vdash e' \sim e' : \tau'$ and $\Gamma, x : \tau' \vdash \mathcal{R}\{x\} \sim \mathcal{R}\{x\} : \tau$, then also $\Gamma \vdash \mathcal{R}\{e \parallel e'\} \sim \mathcal{R}\{e\} \parallel \mathcal{R}\{e'\} : \tau$.*

Proposition 5.43 (Context substitution)**.** *If $\Gamma \vdash e \sim e' : \tau'$ and*

$$\Gamma, x : \tau' \vdash \mathcal{R}\{x\} \sim \mathcal{R}'\{x\} : \tau,$$

then also $\Gamma \vdash \mathcal{R}\{e\} \sim \mathcal{R}'\{e'\} : \tau$

Proposition 5.44 (Context failure)**.** *For any $\mathcal{R}$ and τ, we have $\Gamma \vdash \mathcal{R}\{\mathsf{fail}\} \sim \mathsf{fail} : \tau$.*

Proposition 5.45 (Context commutes over case)**.**
If $\Gamma \vdash e_0 \sim e_0 : \mathsf{nat}$, *and* $\Gamma \vdash e_1 \sim e_1 : \tau'$, *and* $\Gamma, x' : \mathsf{nat} \vdash e_2 \sim e_2 : \tau'$, *and*

$$\Gamma, x : \tau' \vdash \mathcal{R}\{x\} \sim \mathcal{R}\{x\} : \tau,$$

then also

$$\Gamma \vdash \mathcal{R}\{\mathsf{ncase}(e_0,\ e_1,\ x'.\, e_2)\} \sim \mathsf{ncase}(e_0,\ \mathcal{R}\{e_1\},\ x'.\, \mathcal{R}\{e_2\}) : \tau.$$

Proposition 5.46 (Case reduction 1)**.** *If* $\Gamma \vdash e_1 \sim e_1 : \tau$, *then*

$$\Gamma \vdash \mathsf{ncase}(\mathsf{zero},\ e_1,\ x.\, e_2) \sim e_1 : \tau.$$

Proposition 5.47 (Case reduction 2)**.** *if* $\Gamma, x : \mathsf{nat} \vdash e_2 \sim e_2 : \tau$ *and* $\Gamma \vdash e_0 \sim e_0 : \mathsf{nat}$ *and* $e_0 \downarrow$, *then also*

$$\Gamma \vdash \mathsf{ncase}(\mathsf{succ}(e_0),\ e_1,\ x.\, e_2) \sim e_2[e_0/x] : \tau.$$

We can now prove our main theorem, which says that our axiomatization of observational equivalence is sound:

Theorem 5.48. *If* $\mathcal{Q} :: \Gamma \vdash e \doteq e' : \tau$, *then also* $\Gamma \vdash e \sim e' : \tau$.

Proof. The proof proceeds by induction on the derivation of $\mathcal{Q}$.

In the case for q_var, the proof is immediate by assumption. The case q_sym follows by IH and Lemma 5.27; q_trans by IH and Lemma 5.28; q_zero by Lemma 5.35; q_succ by IH and Lemma 5.32; q_lam by IH and Lemma 5.29; q_app by IH and Lemma 5.31; q_case by IH and Lemma 5.33; q_choice by IH and Lemma 5.34.

The case for q_cmerge follows by IH and Proposition 5.39; q_csym by IH and Proposition 5.40; q_cassoc by IH and Proposition 5.41; q_rchoice by IH and Proposition 5.42; q_r by IH and Proposition 5.43; q_rfail by Proposition 5.44; q_rcase by IH and Proposition 5.45; q_case1 by IH and Proposition 5.46; q_case2 by IH and Proposition 5.47. The cases for q_eta and q_beta follow by IH and Lemma 5.38 or 5.37, respectively. □

5.5 Formalization

We will be using the same technique as introduced in Chapter 4, i.e., we will use a representation logic for embedding judgments with explicit equality proofs, and add rules to the assertion logic to provide a notion of case analysis.

The definitions of both the representation logic and assertion logic are mechanical, and will not be covered in detail. However, another challenge appears as a consequence of the increased expressivity of numbers in the object language, which we will describe in the following.

5.5.1 Encoding judgment invariants

We have moved to a definition of object language numerals as value constructors instead of embedding the syntax of natural numbers. This means that the definition of the value judgment is now recursive in the case for $\mathsf{succ}(v')$, and that proving v value from a proof of $v \overset{\text{num}}{\leftrightarrow} n$ requires induction over the number n (this is how we prove Lemma 5.4). Additionally, our meta-theory depends crucially on Lemma 5.1 (Values evaluate); Lemma 5.2 (Value completeness); Lemma 5.3 (Value determinism); Lemma 5.5 (Numeral determinism); and Lemma 5.6 (Uniqueness of numerals), which are all proven by induction on either value or evaluation derivations. We cannot prove these lemmas inside the assertion logic due to the lack of an induction principle, but we can get around this in another way. It turns out that Lemma 5.2 can be represented as an invariant of the evaluation judgment, that Lemmas 5.3, 5.5 and 5.6 can be axiomatized in the equality reasoning system, and that Lemma 5.1 can be axiomatized as an extra evaluation rule without interfering with case analysis. Last, the fact that all numerals are values (Lemma 5.4) can also be represented as an axiom in the value judgment.

The solution is to define alternative value and evaluation judgments. The definitions can be seen in Figure 5.7 and Figure 5.8. The Twelf encoding can be seen in Figure 5.6; the type families are named as the original ones, but with a star suffix. The value judgment has an extra rule that allows us to conclude that numerals are values, and the evaluation judgment has an extra rule that allows us to conclude that values evaluate to themselves. Additionally, we add value proofs to the premises of each evaluation rule.

We can easily prove that the alternative system is equivalent to the original:

```
eval=>eval* : eval E V -> eval* E V -> type.
%mode eval=>eval* +EP -EP'.
eval*=>eval : eval* E V -> eval E V -> type.
%mode eval*=>eval +EP -EP'.
%{ ... proofs elided ... }%
%worlds () (eval=>eval* _ _).
%total (EP) (eval=>eval* EP _).
%worlds () (eval*=>eval _ _).
%total (EP) (eval*=>eval EP _).
```

We can now prove most of the desired properties by case analysis only:

Lemma 5.49 (Values evaluate, alt.)**.** *For any expression v where $\mathcal{V}^\star :: v$ value*, we have $v \Downarrow^\star v$.*

Proof. Immediate, by ea_val □

Lemma 5.50 (Value completeness, alt.)**.** *For any expressions e, v, if $\mathcal{E}^\star :: e \Downarrow^\star v$, then v value*.*

Proof sketch. By case analysis on $\mathcal{E}^\star$. In each case we get a derivation v value* as an immediate subderivation. □

```
% Alternative value judgment
value* : exp -> type.

% Extra rule
value*/num : value* E
           <- num N E.

value*/zero : value* zero.
value*/succ : value* (succ E0)
            <- value* E0.
value*/lam : value* (lam E0).


% Alternative evaluation judgment
eval* : exp -> exp -> type.

% Extra rule
eval*/val : eval* V V
             <- value* V.

eval*/zero : eval* zero zero.
eval*/succ : eval* (succ E) (succ V)
             <- eval* E V
             <- value* V.
eval*/lam : eval* (lam E0) (lam E0).
```

```
% cont'd
eval*/app : eval* (app E1 E2) V
            <- eval* E1 (lam E0)
            <- eval* E2 V2
            <- eval* (E0 V2) V
            <- value* V2
            <- value* V.
eval*/choice1 : eval* (choice E1 E2) V
                 <- eval* E1 V
                 <- value* V.
eval*/choice2 : eval* (choice E1 E2) V
                 <- eval* E2 V
                 <- value* V.
eval*/case0 : eval* (case E0 E1 E2) V
              <- eval* E0 zero
              <- eval* E1 V
              <- value* V.
eval*/case1 : eval* (case E0 E1 E2) V
              <- eval* E0 (succ V0)
              <- eval* (E2 V0) V
              <- value* V0
              <- value* V.
```

Figure 5.6: *Alternative value and evaluation judgments.*

Values, alt.: $\mathcal{V}^\star \quad :: \quad \boxed{v \text{ value}^\star} :$

$$\text{va\_num}: \frac{e \overset{\text{num}}{\leftrightarrow} n}{e \text{ value}^\star}$$

$$\text{va\_lam}: \frac{}{\lambda x.\, e_0 \text{ value}^\star} \qquad \text{va\_zero}: \frac{}{\text{zero value}^\star} \qquad \text{va\_succ}: \frac{v \text{ value}^\star}{\text{succ}(v) \text{ value}^\star}$$

Figure 5.7: *Alternative value judgment.*

Dynamic semantics, alt.: $\mathcal{E}^\star :: \boxed{e \Downarrow^\star v}$:

$$\textsf{ea\_val}: \frac{v \text{ value}^\star}{v \Downarrow^\star v}$$

$$\textsf{ea\_zero}: \frac{}{\textsf{zero} \Downarrow^\star \textsf{zero}} \qquad \textsf{ea\_succ}: \frac{e \Downarrow^\star v \quad v \text{ value}^\star}{\textsf{succ}(e) \Downarrow^\star \textsf{succ}(v)} \qquad \textsf{ea\_lam}: \frac{}{\lambda x.\, e_0 \Downarrow^\star \lambda x.\, e_0}$$

$$\textsf{ea\_app}: \frac{e_1 \Downarrow^\star \lambda x.\, e_0 \quad e_2 \Downarrow^\star v_2 \quad e_0[v_2/x] \Downarrow^\star v \quad v_2 \text{ value}^\star \quad v \text{ value}^\star}{e_1\ e_2 \Downarrow^\star v}$$

$$\textsf{ea\_choice1}: \frac{e_1 \Downarrow^\star v \quad v \text{ value}^\star}{e_1 \,[\!]\, e_2 \Downarrow^\star v} \qquad \textsf{ea\_choice2}: \frac{e_2 \Downarrow^\star v \quad v \text{ value}^\star}{e_1 \,[\!]\, e_2 \Downarrow^\star v}$$

$$\textsf{ea\_case0}: \frac{e_0 \Downarrow^\star \textsf{zero} \quad e_1 \Downarrow^\star v \quad v \text{ value}^\star}{\textsf{ncase}(e_0,\ e_1,\ x.\, e_2) \Downarrow^\star v}$$

$$\textsf{ea\_case1}: \frac{e_0 \Downarrow^\star \textsf{succ}(v_0) \quad e_2[v_0/x] \Downarrow^\star v \quad v_2 \text{ value}^\star \quad v \text{ value}^\star}{\textsf{ncase}(e_0,\ e_1,\ x.\, e_2) \Downarrow^\star v}$$

Figure 5.8: *Alternative evaluation judgment*

Lemma 5.51 (Numerals are values, alt.)**.** *For any value v, if $\mathcal{N} :: v \overset{\text{num}}{\leftrightarrow} n$, then v value*.*

Proof. Immediate, by va_num. □

Now that we have added an extra rule to each judgment, we might expect that all proofs depending on case analysis on evaluation and value judgments might fail, as the new value evaluation rule also has to be taken into consideration. It turns out that any proof that worked by case analysis on the original judgments can be converted to a proof using the alternative judgments, by effectively "unrolling" a single step of the admissibility proofs for the new rules:

Lemma 5.52 (Single-step admissibility)**.** *The following holds:*

1. *For any derivation $\mathcal{V}^\star :: v$ value*, there is a derivation $\mathcal{V}^{\star\prime} :: v$ value* where $\mathcal{V}^{\star\prime}$ only ends in one of* va_zero, va_succ *or* va_lam.
2. *For any derivation $\mathcal{E}^\star :: e \Downarrow^\star v$, there is a derivation $\mathcal{E}^{\star\prime} :: e \Downarrow^\star v$, where $\mathcal{E}^{\star\prime}$ only ends in one of* ea_zero, ea_succ, ea_lam, ea_app, ea_choice1, ea_choice2, ea_case0 *or* ea_case1.

Proof sketch.

1. By case analysis on $\mathcal{V}^\star$. If $\mathcal{V}^\star$ ends in va_zero, va_succ or va_lam, we are done. In the case of va_num, we have a derivation $\mathcal{N} :: v \overset{\text{num}}{\leftrightarrow} n$ for some n, and proceed by subcases on $\mathcal{N}$.

   In the subcase where $\mathcal{N}$ ends in n_zero, we get $\mathcal{V}^{\star\prime}$ by va_zero. In the subcase where $\mathcal{N}$ ends in n_succ, we have $n = \mathsf{s}(n')$, $v = \textsf{succ}(v')$ and $\mathcal{N}' :: v' \overset{\text{num}}{\leftrightarrow} n'$. By va_num

on $\mathcal{N}'$, we get a derivation of $\mathcal{V}^{\star\prime\prime} :: v'$ value*. But then by va_succ, on $\mathcal{V}^{\star\prime\prime}$, we are done.

2. By case analysis on $\mathcal{E}^\star$. Similar to above, proceeding by case analysis on the inner value derivation in the case of ea_val. Due to the above result, we only have to consider the cases va_zero, va_succ and va_lam.

   In the subcase of va_zero or va_lam, we are done by ea_zero or ea_lam, respectively. In the subcase of va_succ, we apply ea_val to the subderivation, and are done by ea_succ on the result. □

It remains to show how we obtain Lemma 5.3 (Value determinism) without induction. This will be demonstrated in the following section.

5.5.2 The assertion logic

We will be using the same approach as in the previous chapter and define a representation logic in which we represent the judgments that assertion logic proofs will work with. We then extend the basic assertion logic with appropriate quantifiers and rules for doing case analysis on derivations. As there are no "architectural" differences, we will just summarize the representation logic formulas and the added assertion logic quantifiers as Twelf code:

```
%% Data formulas
% Equality
@void : dform.
@eq-nat : nat -> nat -> dform.
@eq-exp2 : (exp -> exp)
         -> (exp -> exp) -> dform.
@eq-exp : exp -> exp -> dform.

% Judgments
@eval* : exp -> exp -> dform.
@value* : exp -> dform.
@num : nat -> exp -> dform.
```

```
% Quantifiers
%{ ... standard defs elided ... }%
existsn : (nat -> form) -> form.
existsd : (data D -> form) -> form.
foralle : (exp -> form) -> form.
existse : (exp -> form) -> form.
% We also quantify over binders:
existse2 : ((exp -> exp) -> form)
         -> form.

% Case analysis for data derivations.
data+ : data D -> form.
```

That is, our assertion logic will quantify existentially over natural numbers, representation logic derivations, expressions and binders; additionally, it will quantify universally over expressions. It will provide case analysis over data derivations. The representation logic supports equality proofs for natural numbers, binders and expressions. Additionally, it embeds the alternative evaluation and value judgments, as well as the judgment characterizing numerals.

We add the standard rules for reasoning about equality, i.e., reflexivity, symmetry, transitivity, congruence, converse congruence and rules for proving absurd equalities. Additionally, we add the following extra axioms:

```
@eq-exp/val-det : data (@value* E) -> data (@eval* E V) -> data (@eq-exp E V).
@eq-nat/num-uniq : data (@num N V) -> data (@num N' V) -> data (@eq-nat N N').
@eq-exp/num-det : data (@num N V) -> data (@num N V') -> data (@eq-exp V V').
```

These axioms gives us Lemma 5.3, Lemma 5.6 and Lemma 5.5 as equality rules. Like the rest of the representation logic, the soundness proof of the extra axioms only need to be proved on the meta-level—the proofs are standard.

5.6 Summary of the formalization

In this section, we will brifly summarize the Twelf formalization. The overall structure is the same as for the formalization in Chapter 4, but involves a larger number of lemmas due to the increased size and complexity of the system. The total size of the code is just below 300KiB, and about 5000 lines of code.

The actual soundness proof is formalized in exactly the same way as in Chapter 4, and hence also requires a translation of the adequate representation of the axiomatic equational reasoning system into one that uses separated contexts, as described in Section 4.5. Like in Chapter 4, we will present the formulation of the lemmas proving congruence. We will not present the body of the proofs, but refer to the electronic appendix [Ras13] for the full code.

We will again use `conc*` as an abbreviation for `conc cutful`. We also abbreviate derivations which "travel" with proofs of their well-formedness as follows:

```
#eval : exp -> exp -> form =
  [e1][e2] existsd [dp:data (@eval* e1 e2)] data+ dp.
#num : nat -> exp -> form =
  [n][v] existsd [dp:data (@num n v)] data+ dp.
#eq-exp : exp -> exp -> form =
  [e][e'] existsd [dp:data (@eq-exp e e')] top.
#val : exp -> form = [v] existsd [dp:data (@value* v)] data+ dp.
```

where equality proofs do not actually have a proof of well-formedness, since we never need to perform case analysis on them.

To keep things manageable, we define the following abbreviates to characterize relations and their properties. Thus, the following defines what a binary relation is, what it means for a relation to be symmetric and transitive, and what a relation on values must satisfy:

```
%abbrev
rel : type = exp -> exp -> form.

sym : rel -> form
 = [R] foralle [x] foralle [x'] R x x' ==> R x' x.

trans : rel -> form
```

```
 = [R] foralle [x] foralle [x'] foralle [x'']
    R x x' ==> R x' x'' ==> R x x''.

val' : rel -> form
 = [R] foralle [v] foralle [v'] R v v' ==> #val v /\ #val v'.
```

The definition of the logical relation is also split over several abbreviations. We also split the definition of the computation extension into its forwards and converse parts; we will come back to the reason for doing this later:

```
% Computation extension, "left" and "right" parts:
%abbrev
compl : rel -> rel
 = [R][E][E'] (foralle [v] #eval E v ==> existse [v'] #eval E' v' /\ R v v').
%abbrev
compr : rel -> rel
 = [R][E][E'] (foralle [v'] #eval E' v' ==> existse [v] #eval E v /\ R v v').
comp : rel -> rel
  = [R][E][E'] (compl R E E') /\ (compr R E E').

% Logical relation at nat and function types, respectively:
flr/nat' : rel
  = [v1][v2] existsn [n] #num n v1 /\ #num n v2.
flr/=> : rel -> rel -> rel
  = [R2][R0][v][v'] existse2 [e0] existse2 [e0']
     #eq-exp v (lam e0) /\ #eq-exp v' (lam e0')
     /\ foralle [v2] foralle [v2'] R2 v2 v2' ==> comp R0 (e0 v2) (e0' v2').

% Logical relation as a meta-level relation between types and
% open formulas:
lr : tp -> rel -> type. %name lr LP.
lr/nat' : lr nat' flr/nat'.
lr/=> : lr (T2 => T0) (flr/=> R2 R0)
         <- lr T0 R0
         <- lr T2 R2.
```

Since we do not have a general equality conversion principle for assertion logic formulas, but only equality conversion for the derivations that we quantify over, we need to characterize when a proof of two expressions being related supports equality conversions:

```
conv : rel -> form
 = [R] foralle [v1] foralle [v1'] foralle [v2] foralle [v2']
    #eq-exp v1 v2 ==> #eq-exp v1' v2' ==> R v1 v1' ==> R v2 v2'.
```

As all derivations that we quantify over supports equality conversions, we could in principle prove that a general equality conversion principle is admissible for any formula. We do not need that much generality, so in practice it suffices to show that equality conversion is supported at the computation extension and at any logical relation.

Since the computation extension is agnostic to the type of the underlying logical relation, we can actually prove equality conversion support directly as an abbreviation:

```
% Computation extension supports conversion
comp-conv : conc* (conv (comp R))
  = %{ ... }%

% Logical relation supports conversion
lr-conv : lr T R -> conc* (conv R) -> type.
%mode lr-conv +LP -SP.
```

Finally, we can also prove that if we can prove false (i.e., a representation logic derivation of $\Vdash (\!| \mathsf{void} |\!)$), then any two expressions are logically related:

```
% Void implies any logical relation
lr-void : lr T R -> {E}{E'} (data @void -> conc* (R E E')) -> type.
%mode lr-void +LP +E +E' -SP.
```

5.6.1 Properties of the computation extension

Lemma 5.10 (Symmetry) and Lemma 5.11 (Transitivity) are straightforward, and formalized as follows:

```
% Computation extension preserves symmetry
comp-sym : conc* (sym R) -> conc* (sym (comp R))
 = %{ ... }%
% Computation extension preserves transitivity
comp-trans : conc* (trans R) -> conc* (trans (comp R))
 = %{ ... }%
```

We need a slightly different notion of reduction frames and reduction contexts. As one of the possible reduction frames for application is restricted such that only a value is allowed in function position, we need it to carry an assertion-logic proof. This is not a problem, as frame and context proofs otherwise live entirely on the meta-level:

```
frame' : (exp -> exp) -> type.
frame'/succ : frame' succ.
frame'/app1 : frame' ([x] app V1 x)
              <- conc* (#val V1).
frame'/app2 : frame' ([x] app x E2).
frame'/case : frame' ([x] case x E1 E2).
```

```
ctx' : (exp -> exp) -> type.
ctx'/id : ctx' [x] x.
ctx'/frame : ctx' ([x] F(R(x)))
              <- frame' F
              <- ctx' R.
```

Lemma 5.13 (Converse frame evaluation), Lemma 5.14 (Frame evaluation extraction), Lemma 5.15 (Monadic return) and Lemma 5.16 (Monadic bind) can then be represented as follows:

```
% Converse frame evaluation
frame-cvrs : frame' F
          -> conc* (#eval E V)
          -> conc* (#eval (F V) V')
          -> conc* (#eval (F E) V')
          -> type.
%mode frame-cvrs +FP +SP1 +SP2 -SP'.
```

```
% Frame evaluation extraction
frame-ext : frame' F
         -> conc* (#eval (F E) V)
         -> conc* (existse [v'] #eval E v'
                  /\ #eval (F v') V)
         -> type.
%mode frame-ext +FP +SP -SP'.
```

```
% Return lemma
return : conc* (val' R)
      -> conc* (conv R)
      -> conc* (R V V')
      -> conc* (comp R V V')
 = %{ ... }%
```

```
% Bind lemma, frames
frame-bind :
  {R:rel}{S:rel}
  frame' F
  -> frame' F'
  -> conc* (comp R E E')
  -> ({v}{v'}
      conc* (R v v'
             ==> comp S (F v) (F' v')))
  -> conc* (comp S (F E) (F' E'))
  -> type.
%mode frame-bind +R +S +FP1 +FP2
                 +SP1 +SP2 -SP'.
```

The context variants are similar:

```
ctx-cvrs : ctx' RX
        -> conc* (#eval* E V)
        -> conc* (#eval* (RX V) V')
        -> conc* (#eval* (RX E) V')
        -> type.
%mode ctx-cvrs +CP +SP +SP' -SP''.

ctx-ext : ctx' RX
       -> conc* (#eval* (RX E) V)
       -> conc* (existse [v'] #eval* E v'
                /\ #eval* (RX v') V)
       -> type.
%mode ctx-ext +RP +SP -SP'.
```

```
ctx-bind :
  {R:rel}{S:rel}
  ctx' RX
  -> ctx' RX'
  -> conc* (comp R E E')
  -> ({v}{v'}
      conc* (R v v'
          ==> comp S (RX v) (RX' v')))
  -> conc* (comp S (RX E) (RX' E'))
  -> type.
%mode ctx-bind +R +S +FP1 +FP2
               +SP1 +SP2 -SP'.

comp-ext : conc* (conv R)
            -> conc* (#val V)
            -> conc* (#val V')
            -> conc* (comp R V V')
            -> conc* (R V V')
 = %{ ... }%
```

5.6.2 Properties of logical equivalence and congruence

We represent Lemma 5.21 (Logical equivalence is a value relation), Lemma 5.22 (Symmetry) and Lemma 5.23 (Transitivity) as follows:

```
lr-val : lr T R
      -> conc* (val' R)
      -> type.
%mode lr-val +LP -SP.
```

```
lr-sym : {T} lr T R -> conc* (sym R)
                    -> type.
%mode lr-sym +T +LP -SP.
lr-trans : {T} lr T R -> conc* (trans R)
                      -> type.
%mode lr-trans +T +LP -SP.
```

Again, we do not explicitly prove congruence for open logical equivalence, as contexts are implicitly represented using the LF context. Thus, Lemma 5.29 (Congruence at abstraction), Lemma 5.30 (Logical application), Lemma 5.31 (Congruence at application) and Lemma 5.32 (Congruence at successor), are represented as follows:

```
lr-cong-lam :
  lr T2 R2
  -> lr T0 R0
  -> ({x}{x'}
      conc* (R2 x x')
      -> conc* (comp R0 (E0 x) (E0' x')))
  ->  conc* (comp (flr/=> R2 R0)
                  (lam E0) (lam E0'))
  -> type.
%mode lr-cong-lam +LP2 +LP0 +SP0 -SP'.

lr-app :
  lr T2 R2
  -> lr T0 R0
  -> conc* (flr/=> R2 R0 V1 V1')
  -> conc* (R2 V2 V2')
  -> conc* (comp R0 (app V1 V2)
                    (app V1' V2'))
  -> type.
%mode lr-app +LP2 +LP0 +SP1 +SP2 -SP'.
```

```
lr-cong-app :
  lr T2 R2
  -> lr T0 R0
  -> conc* (comp (flr/=> R2 R0) V1 V1')
  -> conc* (comp R2 V2 V2')
  -> conc* (comp R0 (app V1 V2)
                    (app V1' V2'))
  -> type.
%mode lr-cong-app +LP2 +LP0 +SP1 +SP2 -SP'.

lr-cong-succ :
  conc* (comp flr/nat' E E')
  -> conc* (comp flr/nat' (succ E)
                          (succ E'))
  -> type.
%mode lr-cong-succ +SP -SP'.
```

The formalization of the proof for congruence at case turns out to be very long. Since the proof involves two analogous directions, we avoid having two large identical proofs by first proving only the forward direction. This is where the abbreviation `compl` of the forward direction of the computation extensions comes to use:

```
lr-cong-case' :
  lr T R
  -> conc* (flr/nat' V0 V0')
  -> conc* (comp R E1 E1')
  -> ({x}{x'}
      conc* (flr/nat' x x')
```

```
      -> conc* (comp R (E2 x) (E2' x')))
  -> conc* (compl R (case V0 E1 E2)
                    (case V0' E1' E2'))
  -> type.
%mode lr-cong-case' +LP +SP0 +SP1 +SP2 -SP'.
```

It turns out that we can, in general, prove that if the underlying relation is symmetric, then the other direction of the computation extension follows:

```
comp-flip : conc* (sym R) -> conc* (compl R E E') -> conc* (compr R E' E)
 = %{ ... }%
```

Thus, we avoid proving both directions explicitly, and can prove Lemma 5.33 using a much shorter proof:

```
% Congruence at case
lr-cong-case : lr T R
                -> conc* (comp flr/nat' E0 E0')
                -> conc* (comp R E1 E1')
                -> ({x}{x'} conc* (flr/nat' x x') -> conc* (comp R (E2 x) (E2' x')))
                -> conc* (comp R (case E0 E1 E2) (case E0' E1' E2'))
                -> type.
%mode lr-cong-case +LP +SP0 +SP1 +SP2 -SP'.
```

The exact same strategy is used in the proof of Lemma 5.34 (Congruence at choice):

```
 % Only forward direction
 lr-cong-choice' :
   lr T R
   -> conc* (comp R E1 E1')
   -> conc* (comp R E2 E2')
   -> conc* (compl R (choice E1 E2)
                     (choice E1' E2'))
   -> type.
 %mode lr-cong-choice' +LP +SP1 +SP2 -SP'.
```

```
% Full lemma
lr-cong-choice :
  lr T R
  -> conc* (comp R E1 E1')
  -> conc* (comp R E2 E2')
  -> conc* (comp R (choice E1 E2)
                   (choice E1' E2'))
  -> type.
%mode lr-cong-choice +LP +SP1 +SP2 -SP'.
```

Finally, Lemma 5.35 (Reflexivity at zero) is represented as follows:

```
lr-refl-zero : conc* (comp flr/nat' zero zero) -> type.
%mode lr-refl-zero -SP.
```

5.6.3 Semantic equivalence lemmas

We show only the representation of Lemma 5.37 (β-value-reduction), as the remaining semantic equivalence lemmas are similar:

```
lr-beta :
  lr T2 R2
  -> lr T0 R0
  -> ({x}{x'} conc* (R2 x x') -> conc* (comp R0 (E0 x) (E0' x')))
  -> conc* (comp R2 E2 E2')
  -> conc* (#val E2')
  -> conc* (comp R0 (app (lam E0) E2) (E0' E2'))
  -> type.
%mode lr-beta +LP2 +LP0 +SP0 +SP2 +SPv -SP'.
```

6 Conclusion

In the previous chapters, we have presented applications of the method of structural logical relations of Schürmann and Sarnat, proving both termination and observational equivalence in Twelf. We have extended the method to provide stronger reasoning principles on the level of the assertion logic, which were not present in the original method. Specifically, our methodology supports assertion logic proofs by case analysis on derivations as well as equality reasoning. Additionally, we have demonstrated how we in some cases can work around the lack of an induction principle in the assertion logic, by extending the embedded judgments and the equality theory with appropriate axioms.

The methodology we have developed seems to enable a broader range of proofs by logical relations to be formalized in Twelf. The object languages that we have worked with differ from those in the original presentation of structural logical relations in that they come closer to actual programming languages. By defining an operational semantics and a richer feature set for the object languages, we hoped to approach, if just a little bit, some of the features that may be found in a programming language that a language researcher would want to study in Twelf. We realize that there are still some unanswered questions. First of all, we have not been studying whether we could formalize proofs of observational equivalence for a language with recursion. Unfortunately, we have not been able to find the time to investigate this further.

In the last sections of this thesis, we will compare our work with that of other authors, and finally, we will point out some further directions that this work could be taken in.

6.1 Related work

Besides the first demonstration of structural logical relations by Schürmann and Sarnat [SS08], the following work is also somewhat related to the subject of this thesis.

6.1.1 Twelf modules

Twelf has been extended with a module system [RS09], which supports the definition of isolated Twelf signatures and *signature morphisms*, which are total translations of types

and terms from one signature into another. The module system somewhat makes up for the lack of polymorphism in Twelf, as it allows some generic data structures such as lists to be reused by defining proper views. The module system can basically be considered a type-safe preprocessor, as a Twelf signature defined using modules and views can be *elaborated* into a type-correct signature where modules and views are not mentioned in the definitions. We have briefly experimented with using the module system for building a composable framework of building blocks from which an assertion logic and its cut-admissibility proof can be built, and also for defining translations between adequate object language representations and their embedded versions. We did not end up using the module system in the final work, due to several reasons:

- The module system is not part of the official Twelf implementation, but is implemented in an experimental branch. The implementation also has several bugs, and is poorly documented, which makes it somewhat difficult to work with.

- The views (i.e., signature morphisms) that one can define are somewhat limited in what they have to offer. One limiting factor is the fact that there is no support for doing case analysis on terms, which is an obstacle when trying to define certain translations. More importantly, since views live on the level of declarations, they cannot be operationalized. This means that even if we could use views to prove that two different representations are equivalent, we cannot use this fact in a Twelf meta-theorem.

- Modules and views do not transfer Twelf meta-directives such as totality assertions, worlds declarations, blocks, etc. These can still be written for the elaborated signatures, but no facilities are present to make this task more automatic.

The module system has been further extended with support for defining n-ary logical relations over signature morphisms [RS13]. The extensions enable for compact *representations* of logical relations and the *fundamental theorem*, but are otherwise orthogonal to our goals. Like the situation for views, the logical relations defined this way cannot be operationalized, and are hence unavailable to Twelf meta-theorems. In the context of our work, this means that we could, e.g., use a module-level logical relation to prove that there exists a translation of well-typed expressions into termination proofs in the cutful assertion logic. However, since this translation cannot be operationalized, we cannot apply cut elimination to this result.

6.1.2 Delphin

A system which may be somewhat related to our work is Delphin [Pos08], which is a functional programming language for manipulating terms in the LF type theory. Delphin is based on the idea of a two-level system for separating computation and representation. This idea seems to be somewhat similar to the idea of "executing" an assertion logic

proof, where the assertion logic can be viewed as a functional programming language manipulating LF terms.

We have unfortunately not been able to find the time to study Delphin in more detail, but it would be interesting to investigate this connection further.

6.2 Future work

In this section, we will point out some of the directions that the work of this thesis can be extended in.

6.2.1 Code generation

The amount of boilerplate that has to be written in order to enable proofs by case analysis is substantial. We thus ended up developing specialized automation for auto-generating large parts of the code. The implementation of this automation was written ad-hoc, and is hence not in a state that is ready to be documented.

A subject for future work would therefore be to properly formalize a framework for a generic assertion logic, such that everything up the actual logical relations proofs can be generated from a signature representing the object language that we want to study. It may be possible that parts of the Twelf module system [RS09] could be used for this.

6.2.2 Embedding of meta-theorems

As the expressiveness of the assertion logic was increased, Twelf meta-theorems could often be used as prototypes for assertion logic proofs. For example, if we restricted ourselves to writing meta-theorems that only used case analysis and no induction, the proofs could often be translated by hand into a more verbose assertion logic proof.

It would be interesting to investigate whether a subset of Twelf meta-theorems could be extracted as explicit assertion logic certificates. This would also be an interesting exercise in the more general case where we allow induction, since it would reduce our belief in the consistency in Twelf to believing in the consistency of the assertion logic and type-checking of LF terms.

6.2.3 Increasing the expressiveness of the assertion logic

We have only worked with assertion logics that could be proved consistent within Twelf. However, if we choose to believe in the consistency of a logic with an induction principle for natural numbers, we will be able to formalize a much larger body of proofs by logical relations.

A cut elimination procedure could still be formulated for such a logic, but not verified to be total, and as such would still have an operational interpretation which allows extraction to meta-level proofs. A subject for future work would be to make the part

that has to be trusted as small as possible, for example by only adding induction on natural numbers and deriving assertion-level proofs of complete induction and structural induction.

Bibliography

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. II. *Mathematische Zeitschrift*, 39(1):405–431, 1935.

[Gen65] Gerhard Gentzen. Investigations into logical deduction: II. *American Philosophical Quarterly*, 2(3):pp. 204–218, 1965.

[Har13] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, February 2013.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, July 2007.

[HP05] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Logic*, 6(1):61–101, January 2005.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ML. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 173–184, New York, NY, USA, 2007. ACM Press.

[MM00] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232(1–2):91–119, 2000.

[Pfe00] Frank Pfenning. Structural cut elimination: I. Intuitionistic and classical logic. *Information and Computation*, 157(1–2):84–141, 2000.

[Pfe01] Frank Pfenning. Computation and deduction. Unpublished lecture notes; see `http://www.cs.cmu.edu/~twelf/notes/cd.pdf`, 2001.

[Pos08] Adam Brett Poswolsky. *Functional Programming with Logical Frameworks*. PhD thesis, Yale University, December 2008.

[PS99] 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, July 1999.

[PS02] Frank Pfenning and Carsten Schürmann. Twelf User's Guide, Version 1.4. Available electronically at `http://www.cs.cmu.edu/~twelf/guide-1-4/`, December 2002.

[Ras13] Ulrik Rasmussen. Formalization of proofs by logical relations in a logical framework, technical appendix. Available electronically at `http://utr.dk/slr.tar.gz`, June 2013.

[RS09] 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.

[RS13] 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`.

[Sar10] Jeffrey Sarnat. *Syntactic Finitism in the Metatheory of Programming Languages*. PhD thesis, Yale University, May 2010.

[SS08] 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, Washington, DC, USA, 2008. IEEE Computer Society.

[SS09] Jeffrey Sarnat and Carsten Schürmann. Lexicographic path induction. In *Proceedings of the 9th Internationl Conference on Typed Lambda Calculi and Applications*, pages 279–293, July 2009.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[TTA+13] Aaron J. Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343–356. ACM, 2013.

Appendices

A Twelf: Termination with numerals and case

The following is a listing of the full formalization of the termination proof from Section 3.4.

A.1 `sources.cfg`

```
nat.elf
nat-blocks.elf
lc.elf
lc-blocks.elf
eq.elf
eq-blocks.elf
lc-ax.elf
lc-ax-blocks.elf
form.elf
assert.elf
assert-blocks.elf
admit.elf
cutelim.elf
assert-theorems.elf
lr.elf
ext.elf
```

A.2 `nat.elf`, `nat-blocks.elf`

```
nat : type. %name nat N.
z : nat.
s : nat -> nat. %prefix 1 s.

%block bnat : block {_:nat}.
%worlds (bnat) (nat).
```

A.3 `lc.elf`, `lc-blocks.elf`

```
tp : type. %name tp T.
bool : tp.
nat' : tp.
=> : tp -> tp -> tp. %infix right 1 =>.
exp : type. %name exp E.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
true : exp.
false : exp.
```

```
if : exp -> exp -> exp -> exp.
num : nat -> exp.
case : exp -> exp -> (exp -> exp) -> exp.

eval : exp -> exp -> type. %name eval EP.
eval/lam : eval (lam E0) (lam E0).
eval/app : eval E1 (lam E0) -> eval (E0 E2) V -> eval (app E1 E2) V.
eval/true : eval true true.
eval/false : eval false false.
eval/num : eval (num N) (num N).
eval/ift : eval E0 true -> eval E1 V -> eval (if E0 E1 E2) V.
eval/iff : eval E0 false -> eval E2 V -> eval (if E0 E1 E2) V.
eval/case0 : eval E0 (num z) -> eval E1 V -> eval (case E0 E1 E2) V.
eval/case1 : eval E0 (num (s N)) -> eval (E2 (num N)) V -> eval (case E0 E1 E2) V.

of : exp -> tp -> type. %name of OP.
of/lam : ({x} of x T2 -> of (E0 x) T0) -> of (lam E0) (T2 => T0).
of/app : of E1 (T2 => T0) -> of E2 T2 -> of (app E1 E2) T0.
of/true : of true bool.
of/false : of false bool.
of/if : of E0 bool -> of E1 T -> of E2 T -> of (if E0 E1 E2) T.
of/num : of (num N) nat'.
of/case : of E0 nat'
       -> of E1 T
       -> ({x} of x nat' -> of (E2 x) T)
       -> of (case E0 E1 E2) T.


%block bexp : block {_:exp}.
%block beval : some {E:exp}{V:exp} block {_:eval E V}.

%worlds (bnat | bexp) (exp).
```

A.4 `eq.elf`, `eq-blocks.elf`

```
void : type.

eq-exp : exp -> exp -> type. %name eq-exp QP.
eq-exp/id : eq-exp E E.

void-eq-exp : void -> eq-exp E1 E2 -> type.
%mode +{E1:exp} +{E2:exp} +{VP:void} -{QP:eq-exp E1 E2} void-eq-exp VP QP.
%worlds () (void-eq-exp _ _).
%total {} (void-eq-exp _ _).

eq-exp-sym : eq-exp E E' -> eq-exp E' E -> type.
%mode eq-exp-sym +QP -QP'.
- : eq-exp-sym eq-exp/id eq-exp/id.
%worlds () (eq-exp-sym _ _).
%total {} (eq-exp-sym _ _).

eq-exp-true-false : eq-exp true false -> void -> type.
%mode eq-exp-true-false +QP -VP.
%worlds () (eq-exp-true-false _ _).
%total {} (eq-exp-true-false _ _).

eq-exp-cong1 : {F} eq-exp E E' -> eq-exp (F E) (F E') -> type.
%mode eq-exp-cong1 +F +QP -QP'.
- : eq-exp-cong1 F eq-exp/id eq-exp/id.
%worlds () (eq-exp-cong1 _ _ _).
%total {} (eq-exp-cong1 _ _ _).


%block beq : some {E:exp}{E':exp} block {_:eq-exp E E'}.
```

A.5 `lc-ax.elf`, `lc-blocks.elf`

```
ctx : (exp -> exp) -> type. %name ctx RP.
ctx/id : ctx [x] x.
ctx/if : ctx R -> ctx ([x] if (R x) E1 E2).
ctx/case : ctx R -> ctx ([x] case (R x) E1 E2).
ctx/app : ctx R -> ctx ([x] app (R x) E2).

whr : exp -> exp -> type. %name whr WP.
whr/beta : whr (app (lam E0) E2) (E0 E2).
whr/ift : whr (if true E1 E2) E1.
whr/iff : whr (if false E1 E2) E2.
```

```
whr/case0 : whr (case (num z) E1 E2) E1.
whr/case1 : whr (case (num (s N)) E1 E2) (E2 (num N)).

val : exp -> type.
val/lam : val (lam E0).
val/true : val true.
val/false : val false.
val/num : val (num N).

eval-val : eval E V -> val V -> type.
%mode eval-val +EP -VP.
- : eval-val eval/true val/true.
- : eval-val eval/false val/false.
- : eval-val eval/num val/num.
- : eval-val eval/lam val/lam.
- : eval-val (eval/app EP1 EP2) VP
     <- eval-val EP2 VP.
- : eval-val (eval/ift EP1 EP2) VP
     <- eval-val EP2 VP.
- : eval-val (eval/iff EP1 EP2) VP
     <- eval-val EP2 VP.
- : eval-val (eval/case0 EP0 EP1) VP
     <- eval-val EP1 VP.
- : eval-val (eval/case1 EP0 EP2) VP
     <- eval-val EP2 VP.
%worlds () (eval-val _ _).
%total (EP) (eval-val EP _).

val-eval : val V -> eval V V -> type.
%mode val-eval +VP -EP.
- : val-eval val/num eval/num.
- : val-eval val/false eval/false.
- : val-eval val/true eval/true.
- : val-eval val/lam eval/lam.
%worlds () (val-eval _ _).
%total (VP) (val-eval VP _).

val-det : val E -> eval E V -> eq-exp E V -> type.
%mode val-det +VP +EP -QP.
- : val-det val/true eval/true eq-exp/id.
- : val-det val/false eval/false eq-exp/id.
- : val-det val/lam eval/lam eq-exp/id.
- : val-det val/num eval/num eq-exp/id.
%worlds () (val-det _ _ _).
%total (VP) (val-det VP _ _).

eval-conv : eq-exp E1 E1' -> eq-exp E2 E2' -> eval E1 E2 -> eval E1' E2' -> type.
%mode eval-conv +QP1 +QP2 +EP -EP'.
- : eval-conv eq-exp/id eq-exp/id EP EP.
%worlds () (eval-conv _ _ _ _).
%total {} (eval-conv _ _ _ _).

eval~ : exp -> exp -> type. %name eval~ EP.
eval~/val : val V -> eval~ V V.
eval~/whr :  ctx RX -> whr E E' -> eval~ (RX E') V -> eval~ (RX E) V.
eval~/ctx : ctx RX -> eval~ E0 V0 -> eval~ (RX V0) V -> eval~ (RX E0) V.
eval~/conv : eq-exp E1 E1' -> eq-exp E2 E2' -> eval~ E1 E2 -> eval~ E1' E2'.

eval-ctx : ctx RX -> eval E0 V0 -> eval (RX V0) V -> eval (RX E0) V -> type.
%mode eval-ctx +RP +EP +EP' -EP''.
- : eval-ctx ctx/id (EP : eval E0 V0) EP' EP''
     <- eval-val EP VP
     <- val-det VP EP' QP
     <- eval-conv eq-exp/id QP EP EP''.
- : eval-ctx (ctx/if RP) EP (eval/ift EP0 EP1) (eval/ift EP0' EP1)
     <- eval-ctx RP EP EP0 EP0'.
- : eval-ctx (ctx/if RP) EP (eval/iff EP0 EP1) (eval/iff EP0' EP1)
     <- eval-ctx RP EP EP0 EP0'.
- : eval-ctx (ctx/case RP) EP (eval/case0 EP0 EP1) (eval/case0 EP0' EP1)
     <- eval-ctx RP EP EP0 EP0'.
- : eval-ctx (ctx/case RP) EP (eval/case1 EP0 EP1) (eval/case1 EP0' EP1)
     <- eval-ctx RP EP EP0 EP0'.
- : eval-ctx (ctx/app RP) EP (eval/app EP1 EP2) (eval/app EP1' EP2)
     <- eval-ctx RP EP EP1 EP1'.
%worlds () (eval-ctx _ _ _ _).
%total (RP) (eval-ctx RP _ _ _).

eval-cvrs : ctx RX -> eval (RX E') V -> whr E E' -> eval (RX E) V -> type.
%mode eval-cvrs +RP +EP +WP -EP'.
- : eval-cvrs ctx/id EP whr/beta (eval/app eval/lam EP).
- : eval-cvrs ctx/id EP whr/ift (eval/ift eval/true EP).
- : eval-cvrs ctx/id EP whr/iff (eval/iff eval/false EP).
```

```
- : eval-cvrs ctx/id EP whr/case0 (eval/case0 eval/num EP).
- : eval-cvrs ctx/id EP whr/case1 (eval/case1 eval/num EP).
- : eval-cvrs (ctx/app RP) (eval/app EP1' EP2) WP (eval/app EP1 EP2)
     <- eval-cvrs RP EP1' WP EP1.
- : eval-cvrs (ctx/if RP) (eval/ift EP1' EP2) WP (eval/ift EP1 EP2)
     <- eval-cvrs RP EP1' WP EP1.
- : eval-cvrs (ctx/if RP) (eval/iff EP1' EP2) WP (eval/iff EP1 EP2)
     <- eval-cvrs RP EP1' WP EP1.
- : eval-cvrs (ctx/case RP) (eval/case0 EP1' EP2) WP (eval/case0 EP1 EP2)
     <- eval-cvrs RP EP1' WP EP1.
- : eval-cvrs (ctx/case RP) (eval/case1 EP1' EP2) WP (eval/case1 EP1 EP2)
     <- eval-cvrs RP EP1' WP EP1.
%worlds () (eval-cvrs _ _ _ _).
%total (RP) (eval-cvrs RP _ _ _).

eval~=>eval : eval~ E V -> eval E V -> type.
%mode eval~=>eval +EP -EP'.
- : eval~=>eval (eval~/val VP) EP
     <- val-eval VP EP.
- : eval~=>eval (eval~/whr RP WP EP) EP''
     <- eval~=>eval EP EP'
     <- eval-cvrs RP EP' WP EP''.
- : eval~=>eval (eval~/ctx RP EP1 EP2) EP''
     <- eval~=>eval EP1 EP1'
     <- eval~=>eval EP2 EP2'
        <- eval-ctx RP EP1' EP2' EP''.
- : eval~=>eval (eval~/conv QP1 QP2 EP) EP''
     <- eval~=>eval EP EP'
     <- eval-conv QP1 QP2 EP' EP''.

%worlds () (eval~=>eval _ _).
%total (EP) (eval~=>eval EP _).


%block beval~ : some {E:exp}{V:exp} block {_:eval~ E V}.
```

A.6 form.elf

```
form : type. %name form F.

top : form.
/\ : form -> form -> form. %infix left 4 /\.
\/ : form -> form -> form. %infix left 3 \/.
==> : form -> form -> form. %infix right 2 ==>.

existse : (exp -> form) -> form.
foralle : (exp -> form) -> form.
existsn : (nat -> form) -> form.
foralln : (nat -> form) -> form.
existsev : (eval~ E V -> form) -> form.
forallev : (eval~ E V -> form) -> form.

nat+ : nat -> form.

% Useful abbreviations

#eval : exp -> exp -> form = [E][V] existsev [ep:eval~ E V] top.


% Formula metrics

metric : type. %name metric M.
metric/bin : metric -> metric -> metric.
metric/una : metric -> metric.
metric/nul : metric.

metric-red : form
              -> metric   % Form metric
              -> nat      % Auxiliary metric 2
              -> type. %name metric-red RP.

metric-red/imp : metric-red (F1 ==> F2) (metric/bin M1 M2) z
                  <- metric-red F1 M1 N
                  <- metric-red F2 M2 N'.
metric-red/or : metric-red (F1 \/ F2) (metric/bin M1 M2) z
                 <- metric-red F1 M1 N
                 <- metric-red F2 M2 N'.
metric-red/and : metric-red (F1 /\ F2) (metric/bin M1 M2) z
                  <- metric-red F1 M1 N
                  <- metric-red F2 M2 N'.
metric-red/top : metric-red top metric/nul z.

metric-red/foralle : metric-red (foralle F) (metric/una M1) z
  <- ({x} metric-red (F x) M1 N1).
```

```
metric-red/existse : metric-red (existse F) (metric/una M1) z
  <- ({x} metric-red (F x) M1 N1).

metric-red/forallev : metric-red (forallev F) (metric/una M1) z
  <- ({x} metric-red (F x) M1 N1).
metric-red/existsev : metric-red (existsev F) (metric/una M1) z
  <- ({x} metric-red (F x) M1 N1).

metric-red/foralln : metric-red (foralln F) (metric/una M1) z
  <- ({x} metric-red (F x) M1 (N1 x)).
metric-red/existsn : metric-red (existsn F) (metric/una M1) z
  <- ({x} metric-red (F x) M1 (N1 x)).
metric-red/nat+ : metric-red (nat+ N) metric/nul N.

metric-red-tot : {F} metric-red F M N -> type.
%mode metric-red-tot +F -RP.

- : metric-red-tot (F1 ==> F2) (metric-red/imp RP2 RP1)
     <- metric-red-tot F1 RP1
     <- metric-red-tot F2 RP2.
- : metric-red-tot (F1 \/ F2) (metric-red/or RP2 RP1)
     <- metric-red-tot F1 RP1
     <- metric-red-tot F2 RP2.
- : metric-red-tot (F1 /\ F2) (metric-red/and RP2 RP1)
     <- metric-red-tot F1 RP1
     <- metric-red-tot F2 RP2.
- : metric-red-tot top metric-red/top.
- : metric-red-tot (foralle F) (metric-red/foralle RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (existse F) (metric-red/existse RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (foralln F) (metric-red/foralln RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (existsn F) (metric-red/existsn RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (forallev F) (metric-red/forallev RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (existsev F) (metric-red/existsev RP)
     <- ({x} metric-red-tot (F x) (RP x)).

- : metric-red-tot (nat+ N) metric-red/nat+.
```

A.7 assert.elf, assert-blocks.elf

```
variant : type. %name variant V.
cutful : variant.
cutfree : variant.

conc : variant -> form -> type. %name conc SP.
hyp : form -> type. %name hyp H.

%abbrev conc' = conc cutfree.
%abbrev conc* = conc cutful.

ax : hyp A -> conc V A.
cut : conc V A -> (hyp A -> conc V C) -> conc cutful C.

topr : conc V top.
andr : conc V F -> conc V G
        -> conc V (F /\ G).
andl1 : (hyp F -> conc V C)
         -> (hyp (F /\ G) -> conc V C).
andl2 : (hyp G -> conc V C)
         -> (hyp (F /\ G) -> conc V C).
impr : (hyp F -> conc V G)
        -> conc V (F ==> G).
impl : conc V F -> (hyp G -> conc V C)
        -> (hyp (F ==> G) -> conc V C).
orr1 : conc V F
        -> conc V (F \/ G).
orr2 : conc V G
        -> conc V (F \/ G).
orl : (hyp F -> conc V C) -> (hyp G -> conc V C)
        -> (hyp (F \/ G) -> conc V C).

nat+/z : conc V (nat+ z).
nat+/s : conc V (nat+ N) -> conc V (nat+ (s N)).
nat+/l : (eq-exp (num N) (num z) -> conc V C)
          -> ({n'} eq-exp (num N) (num (s n')) -> hyp (nat+ n') -> conc V C)
          -> (hyp (nat+ N) -> conc V C).
```

```
% Quantifiers, expressions
foraller : ({x:exp} conc V (C x))
           -> conc V (foralle C).
forallel : {x:exp} (hyp (F x) -> conc V C)
           -> (hyp (foralle F) -> conc V C).
existser : {x:exp} conc V (F x) -> conc V (existse F).
existsel : ({x:exp} hyp (F x) -> conc V C)
            -> (hyp (existse F) -> conc V C).

% Quantifiers, naturals
forallnr : ({x:nat} conc V (C x))
           -> conc V (foralln C).
forallnl : {x:nat} (hyp (F x) -> conc V C)
           -> (hyp (foralln F) -> conc V C).
existsnr : {x:nat} conc V (F x) -> conc V (existsn F).
existsnl : ({x:nat} hyp (F x) -> conc V C)
            -> (hyp (existsn F) -> conc V C).

% Quantifiers, evaluations
forallevr : ({x:eval~ E V'} conc V (C x))
           -> conc V (forallev C).
forallevl : {x:eval~ E V'} (hyp (F x) -> conc V C)
           -> (hyp (forallev F) -> conc V C).
existsevr : {x:eval~ E V'} conc V (F x) -> conc V (existsev F).
existsevl : ({x:eval~ E V'} hyp (F x) -> conc V C)
            -> (hyp (existsev F) -> conc V C).


%block bhyp : some {F:form} block {_:hyp F}.
%block bconc : some {F:form} block {_:conc* F}.
```

A.8 admit.elf

```
ca : {M}{N}{RP:metric-red A M N} conc' A -> (hyp A -> conc' C) -> conc' C -> type.
%mode ca +M +N +RP +SP1 +SP2 -SP'.

% Twelf requires that all terms that we do induction over are
% explicit, even if they are never required in proof cases. We define
% the following abbreviation to keep the admissibility proof
% uncluttered in the case where we add extra argument to the type
% family.
%abbrev ca* = ca _ _.

- : ca* _ (ax H) E (E H).
- : ca* _ D ax D.

% Essential cases

- : ca* (metric-red/and RP2 RP1)  (andr D1 D2) ([h] andl1 (E1 h) h) SP
     <- ({h1} ca* (metric-red/and RP2 RP1) (andr D1 D2) ([h] E1 h h1) (E1' h1))
     <- ca*  RP1 D1 E1' SP.
- : ca* (metric-red/and RP2 RP1) (andr D1 D2) ([h] andl2 (E2 h) h) SP
     <- ({h2} ca* (metric-red/and RP2 RP1) (andr D1 D2) ([h] E2 h h2) (E2' h2))
     <- ca* RP2 D2 E2' SP.
-  : ca* (metric-red/or FP2 FP1) (orr1 D1)
      ([h:hyp (A1 \/ A2)] orl (E1 h) (E2 h) h) F
      <- ({h1:hyp A1}
            ca* (metric-red/or FP2 FP1) (orr1 D1)
            ([h:hyp (A1 \/ A2)] E1 h h1) (E1' h1))
      <- ca* FP1 D1 E1' F.
-  : ca* (metric-red/or FP2 FP1) (orr2 D2)
      ([h:hyp (A1 \/ A2)] orl (E1 h) (E2 h) h) F
      <- ({h2:hyp A2}
            ca* (metric-red/or FP2 FP1) (orr2 D2)
            ([h:hyp (A1 \/ A2)] E2 h h2) (E2' h2))
      <- ca* FP2 D2 E2' F.
- : ca* (metric-red/imp RP2 RP1) (impr D2) ([h] impl (E1 h) (E2 h) h) SP
     <- ca* (metric-red/imp RP2 RP1) (impr D2) E1 E1'
     <- ({h2} ca* (metric-red/imp RP2 RP1) (impr D2) ([h] E2 h h2) (E2' h2))
     <- ca* RP1 E1' D2 D2'
     <- ca* RP2 D2' E2' SP.
- : ca* metric-red/nat+ nat+/z ([h] nat+/l (D1 h) (D2 h) h) SP
     <- ca* metric-red/nat+ nat+/z ([h] D1 h eq-exp/id) SP.
- : ca* metric-red/nat+ (nat+/s SP+) ([h] nat+/l (D1 h) (D2 h) h) SP
     <- ({n'+} ca* metric-red/nat+ (nat+/s SP+) ([h] D2 h _ eq-exp/id n'+) (D2' n'+))
     <- ca* metric-red/nat+ SP+ D2' SP.

% Left commutative cases

- : ca* RP (andl1 D1 H) E (andl1 D1' H)
```

```
     <- ({h1} ca* RP (D1 h1) E (D1' h1)).
- : ca* RP (andl2 D2 H) E (andl2 D2' H)
     <- ({h1} ca* RP (D2 h1) E (D2' h1)).
- : ca* RP (impl D1 D2 H) E (impl D1 D2' H)
     <- ({h2} ca* RP (D2 h2) E (D2' h2)).
- : ca* RP (orl D1 D2 H) E (orl D1' D2' H)
     <- ({h1} ca* RP (D1 h1) E (D1' h1))
     <- ({h2} ca* RP (D2 h2) E (D2' h2)).
- : ca* RP (nat+/l D1 D2 H) E (nat+/l D1' D2' H)
     <- ({h1} ca* RP (D1 h1) E (D1' h1))
     <- ({h1}{h2}{h3} ca* RP (D2 h1 h2 h3) E (D2' h1 h2 h3)).

% Right commutative cases

- : ca* _ D ([h] E) E.
- : ca* RP D ([h] andr (E1 h) (E2 h)) (andr E1' E2')
     <- ca* RP D E1 E1'
     <- ca* RP D E2 E2'.
- : ca* RP D ([h] andl1 (E1 h) H) (andl1 E1' H)
     <- ({h1} ca* RP D ([h] E1 h h1) (E1' h1)).
- : ca* RP D ([h] andl2 (E2 h) H) (andl2 E2' H)
     <- ({h1} ca* RP D ([h] E2 h h1) (E2' h1)).
- : ca* RP D ([h] impr (E2 h)) (impr E2')
     <- ({h1} ca* RP D ([h] E2 h h1) (E2' h1)).
- : ca* RP D ([h] impl (E1 h) (E2 h) H) (impl E1' E2' H)
     <- ca* RP D E1 E1'
     <- ({h2} ca* RP D ([h] E2 h h2) (E2' h2)).
- : ca* RP D ([h] orr1 (E1 h)) (orr1 E1')
     <- ca* RP D E1 E1'.
- : ca* RP D ([h] orr2 (E2 h)) (orr2 E2')
     <- ca* RP D E2 E2'.
- : ca* RP D ([h] orl (E1 h) (E2 h) H) (orl E1' E2' H)
     <- ({h1} ca* RP D ([h:hyp A] E1 h h1) (E1' h1))
     <- ({h2} ca* RP D ([h:hyp A] E2 h h2) (E2' h2)).
- : ca* RP D ([h] nat+/s (E h)) (nat+/s E')
     <- ca* RP D E E'.
- : ca* RP D ([h] nat+/l (E1 h) (E2 h) H) (nat+/l E1' E2' H)
     <- ({h1} ca* RP D ([h] E1 h h1) (E1' h1))
     <- ({h1}{h2}{h3} ca* RP D ([h] E2 h h1 h2 h3) (E2' h1 h2 h3)).

% Cases for quantification of expressions
- : ca* (metric-red/foralle RP) (foraller D1) ([h] forallel X (E1 h) h) SP
     <- ({h2} ca* (metric-red/foralle RP) (foraller D1) ([h] E1 h h2) (E11 h2))
     <- ca* (RP X) (D1 X) E11 SP.
- : ca* RP (forallel X D1 H) E (forallel X D11 H)
   <- ({h} ca* RP (D1 h) E (D11 h)).
- : ca* RP D ([h] foraller (E1 h)) (foraller E11)
   <- ({x} ca* RP D ([h] E1 h x) (E11 x)).
- : ca* RP D ([h] forallel X (E1 h) H) (forallel X E11 H)
   <- ({h1} ca* RP D ([h] E1 h h1) (E11 h1)).
- : ca* (metric-red/existse RP) (existser X D1) ([h] existsel (E1 h) h) SP
   <- ({x}{h1} ca* (metric-red/existse RP) (existser X D1) ([h] E1 h x h1) (E11 x h1))
   <- ca* (RP X) D1 (E11 X) SP.
- : ca* RP (existsel D1 H) E (existsel D11 H)
   <- ({x}{h} ca* RP (D1 x h) E (D11 x h)).
- : ca* RP D ([h] existser X (E1 h)) (existser X E11)
   <- ca* RP D E1 E11.
- : ca* RP D ([h] existsel (E1 h) H) (existsel E11 H)
   <- ({x}{h1} ca* RP D ([h] E1 h x h1) (E11 x h1)).

% Cases for quantification of evaluation derivations
- : ca* (metric-red/forallev RP) (forallevr D1) ([h] forallevl X (E1 h) h) SP
     <- ({h2} ca* (metric-red/forallev RP) (forallevr D1) ([h] E1 h h2) (E11 h2))
     <- ca* (RP X) (D1 X) E11 SP.
- : ca* RP (forallevl X D1 H) E (forallevl X D11 H)
   <- ({h} ca* RP (D1 h) E (D11 h)).
- : ca* RP D ([h] forallevr (E1 h)) (forallevr E11)
   <- ({x} ca* RP D ([h] E1 h x) (E11 x)).
- : ca* RP D ([h] forallevl X (E1 h) H) (forallevl X E11 H)
   <- ({h1} ca* RP D ([h] E1 h h1) (E11 h1)).
- : ca* (metric-red/existsev RP) (existsevr X D1) ([h] existsevl (E1 h) h) SP
   <- ({x}{h1} ca* (metric-red/existsev RP) (existsevr X D1) ([h] E1 h x h1) (E11 x h1))
   <- ca* (RP X) D1 (E11 X) SP.
- : ca* RP (existsevl D1 H) E (existsevl D11 H)
   <- ({x}{h} ca* RP (D1 x h) E (D11 x h)).
- : ca* RP D ([h] existsevr X (E1 h)) (existsevr X E11)
   <- ca* RP D E1 E11.
- : ca* RP D ([h] existsevl (E1 h) H) (existsevl E11 H)
   <- ({x}{h1} ca* RP D ([h] E1 h x h1) (E11 x h1)).

% Cases for quantification of nats
```

```
- : ca* (metric-red/foralln RP) (forallnr D1) ([h] forallnl X (E1 h) h) SP
     <- ({h2} ca* (metric-red/foralln RP) (forallnr D1) ([h] E1 h h2) (E11 h2))
     <- ca* (RP X) (D1 X) E11 SP.
- : ca* RP (forallnl X D1 H) E (forallnl X D11 H)
   <- ({h} ca* RP (D1 h) E (D11 h)).
- : ca* RP D ([h] forallnr (E1 h)) (forallnr E11)
   <- ({x} ca* RP D ([h] E1 h x) (E11 x)).
- : ca* RP D ([h] forallnl X (E1 h) H) (forallnl X E11 H)
   <- ({h1} ca* RP D ([h] E1 h h1) (E11 h1)).
- : ca* (metric-red/existsn RP) (existsnr X D1) ([h] existsnl (E1 h) h) SP
   <- ({x}{h1} ca* (metric-red/existsn RP) (existsnr X D1) ([h] E1 h x h1) (E11 x h1))
   <- ca* (RP X) D1 (E11 X) SP.
- : ca* RP (existsnl D1 H) E (existsnl D11 H)
   <- ({x}{h} ca* RP (D1 x h) E (D11 x h)).
- : ca* RP D ([h] existsnr X (E1 h)) (existsnr X E11)
   <- ca* RP D E1 E11.
- : ca* RP D ([h] existsnl (E1 h) H) (existsnl E11 H)
   <- ({x}{h1} ca* RP D ([h] E1 h x h1) (E11 x h1)).
```

A.9 cutelim.elf

```
ce : conc* A -> conc' A -> type.
%mode ce +SP* -SP'.
- : ce (cut SP1 SP2) SP
     <- ce SP1 SP1'
     <- ({h} ce (SP2 h) (SP2' h))
     <- metric-red-tot _ RP
     <- ca* RP SP1' SP2' SP.
- : ce (cut SP1 SP2) SP
     <- metric-red-tot _ RP
     <- ca* RP SP1 SP2 SP.
- : ce topr topr.
- : ce (ax H) (ax H).
- : ce (andr SP1 SP2) (andr SP1' SP2')
     <- ce SP1 SP1'
     <- ce SP2 SP2'.
- : ce (andl1 SP1 H) (andl1 SP1' H)
     <- ({h} ce (SP1 h) (SP1' h)).
- : ce (andl2 SP1 H) (andl2 SP1' H)
     <- ({h} ce (SP1 h) (SP1' h)).
- : ce (orr1 SP) (orr1 SP')
     <- ce SP SP'.
- : ce (orr2 SP) (orr2 SP')
     <- ce SP SP'.
- : ce (orl SP1 SP2 H) (orl SP1' SP2' H)
     <- ({x} ce (SP1 x) (SP1' x))
     <- ({x} ce (SP2 x) (SP2' x)).
- : ce (impr SP1) (impr SP1')
     <- ({h} ce (SP1 h) (SP1' h)).
- : ce (impl SP1 SP2 H) (impl SP1' SP2' H)
     <- ce SP1 SP1'
     <- ({h1} ce (SP2 h1) (SP2' h1)).
- : ce nat+/z nat+/z.
- : ce (nat+/s SP) (nat+/s SP')
     <- ce SP SP'.
- : ce (nat+/l SP1 SP2 H) (nat+/l SP1' SP2' H)
     <- ({h1} ce (SP1 h1) (SP1' h1))
     <- ({h1}{h2}{h3} ce (SP2 h1 h2 h3) (SP2' h1 h2 h3)).
% Cut elimination cases for quantifiers over expressions.
- : ce (forallel X SP H) (forallel X SP1 H)
     <- ({h} ce (SP h) (SP1 h)).
- : ce (foraller SP) (foraller SP1)
     <- ({h} ce (SP h) (SP1 h)).
- : ce (existsel SP H) (existsel SP1 H)
     <- ({x}{h} ce (SP x h) (SP1 x h)).
- : ce (existser X SP) (existser X SP1)
     <- ce SP SP1.
% Cut elimination cases for quantifiers over evaluations.
- : ce (forallevl X SP H) (forallevl X SP1 H)
     <- ({h} ce (SP h) (SP1 h)).
- : ce (forallevr SP) (forallevr SP1)
     <- ({h} ce (SP h) (SP1 h)).
- : ce (existsevl SP H) (existsevl SP1 H)
     <- ({x}{h} ce (SP x h) (SP1 x h)).
- : ce (existsevr X SP) (existsevr X SP1)
```

```
     <- ce SP SP1.
% Cut elimination cases for quantifiers over num judgment.
- : ce (forallnl X SP H) (forallnl X SP1 H)
     <- ({h} ce (SP h) (SP1 h)).
- : ce (forallnr SP) (forallnr SP1)
     <- ({h} ce (SP h) (SP1 h)).
- : ce (existsnl SP H) (existsnl SP1 H)
     <- ({x}{h} ce (SP x h) (SP1 x h)).
- : ce (existsnr X SP) (existsnr X SP1)
     <- ce SP SP1.
```

A.10 assert-theorems.elf

```
%worlds (bhyp | bexp | beval~ | bnat | beq)
           (ca _ _ _ _ _ _) (ce _ _ ) (metric-red-tot _ _).
%total (F) (metric-red-tot F _).
%total {M N [SP1 SP2]} (ca M N _ SP1 SP2 _).
%total (SP) (ce SP _).
```

A.11 lr.elf

```
%abbrev pred : type = exp -> form.
lr : tp -> pred -> type.
%abbrev flr/bool : pred = ([e] #eval e true \/ #eval e false).
%abbrev flr/nat' : pred = ([e] existsn [n] #eval e (num n) /\ nat+ n).
%abbrev flr/=> : pred -> pred -> pred
  = [R2][R0][e] (existse [v] #eval e v)
                  /\ foralle [e2] R2 e2 ==> R0 (app e e2).
lr/bool : lr bool flr/bool.
lr/nat' : lr nat' flr/nat'.
lr/=> : lr (T2 => T0) (flr/=> R2 R0)
         <- lr T0 R0
         <- lr T2 R2.
lr-tot : {T} lr T R -> type.
%mode lr-tot +T -LP.
- : lr-tot bool lr/bool.
- : lr-tot nat' lr/nat'.
- : lr-tot (T2 => T0) (lr/=> LP2 LP0)
     <- lr-tot T2 LP2
     <- lr-tot T0 LP0.
eq-pred : pred -> pred -> type. %name eq-pred QP.
eq-pred/id : eq-pred F F.
eq-pred-cong2 : {F} eq-pred R1 R1'
                 -> eq-pred R2 R2'
                 -> eq-pred (F R1 R2) (F R1' R2') -> type.
%mode eq-pred-cong2 +F +QP1 +QP2 -QP'.
- : eq-pred-cong2 _ eq-pred/id eq-pred/id eq-pred/id.
lr-det : lr T R -> lr T R' -> eq-pred R R' -> type.
%mode lr-det +LP +LP' -QP.
- : lr-det lr/bool lr/bool eq-pred/id.
- : lr-det lr/nat' lr/nat' eq-pred/id.
- : lr-det (lr/=> LP2 LP0) (lr/=> LP2' LP0') QP
     <- lr-det LP2 LP2' QP2
     <- lr-det LP0 LP0' QP0
     <- eq-pred-cong2 flr/=> QP2 QP0 QP.
pred-conv : conc* (R E) -> eq-pred R R' -> conc* (R' E) -> type.
%mode pred-conv +SP +QP -SP'.
- : pred-conv SP eq-pred/id SP.
cwhe : lr T R -> ctx RX -> whr E E'
    -> conc* (R (RX E')) -> conc* (R (RX E)) -> type.
%mode cwhe +LP +RP +SP +SPR -SP'.
- : cwhe lr/nat' RP WP SP
     (cut SP (existsnl [n][h] existsnr n
                (andr
                   (andl1 (existsevl [E'=>num-n][_]
                             (existsevr (eval~/whr RP WP E'=>num-n)
```

```
                                topr)) h)
                  (andl2 ax h))))).
- : cwhe lr/bool RP WP SP (cut SP (orl
                                     (existsevl [E'=>true][_]
                                        (orr1
                                           (existsevr
                                              (eval~/whr RP WP E'=>true)
                                              topr)))
                                     (existsevl [E'=>false][_]
                                        (orr2
                                           (existsevr
                                              (eval~/whr RP WP E'=>false)
                                              topr))))).
- : cwhe (lr/=> (LP2:lr T2 R2) (LP0:lr T0 R0)) RP
     WP SP (andr
              (cut SP (andl1 (existsel [v] existsevl [E1'=>v][_]
                                existser v (existsevr
                                              (eval~/whr RP WP E1'=>v)
                                              topr))))
              (foraller [e2] impr [R2-e2]
                 (SP0 e2
                    (cut SP (andl2 (forallel e2 (impl (ax R2-e2) ax)))))))
     <- ({e2}{sp} cwhe LP0 (ctx/app RP) WP sp (SP0 e2 sp)).

lr-ctxred : lr T R -> ctx RX -> eval~ E0 V0 -> conc* (R (RX V0))
                                             -> conc* (R (RX E0)) -> type.
%mode lr-ctxred +LP +RP +EP +SP -SP'.
- : lr-ctxred lr/bool RP EP SP
     (cut SP (orl
                (existsevl [E'=>true][_]
                   orr1 (existsevr (eval~/ctx RP EP E'=>true) topr))
                (existsevl [E'=>false][_]
                   orr2 (existsevr (eval~/ctx RP EP E'=>false) topr)))).
- : lr-ctxred lr/nat' (RP : ctx RX) (EP : eval~ E0 V0) SP
     (cut SP (existsnl [n][h] existsnr n
                (andr
                   (andl1 (existsevl [RX-V0=>num-n][_]
                             existsevr (eval~/ctx RP EP RX-V0=>num-n) topr) h)
                   (andl2 ax h)))).
- : lr-ctxred (lr/=> LP2 (LP0 : lr T0 R0)) (RP : ctx RX) EP SP
     (cut SP [h]
        (andr
           (andl1
              (existsel [v] existsevl [E=>v][_]
                 existser v (existsevr (eval~/ctx RP EP E=>v) topr)) h)
           (foraller [e2] impr [R2-e2]
              andl2
              (forallel e2
                 (impl (ax R2-e2) [R0-app-RX-E1-e2]
                    IH e2 (ax R0-app-RX-E1-e2)))
              h)))
     <- ({e2}{sp}
           lr-ctxred LP0 (ctx/app RP : ctx [x] app (RX x) e2) EP sp (IH e2 sp)).

struct-nat : {N} conc* (nat+ N) -> type.
%mode struct-nat +N -SP.
- : struct-nat z nat+/z.
- : struct-nat (s N) (nat+/s SP)
     <- struct-nat N SP.

fund : of E T -> lr T R -> conc* (R E) -> type.
fund+ : of E T -> lr T R -> conc* (R E) -> type.
%mode (fund +OP -LP -SP)
      (fund+ +OP +LP -SP).

- : fund+ OP (LP : lr T R) SP
     <- fund OP LP' SP'
     <- lr-det LP' LP QP
     <- pred-conv SP' QP SP.

- : fund (of/lam OP : of (lam E0) (T2 => T0)) (lr/=> LP2 LP0)
     (andr
        (existser (lam E0) (existsevr (eval~/val val/lam) topr))
        (foraller [e2] impr [R2-e2]
           CWHE e2 (ax R2-e2)))
     <- lr-tot T2 (LP2 : lr T2 R2)
     <- lr-tot T0 (LP0 : lr T0 R0)
     <- ({x}{op:of x T2}{sp:conc* (R2 x)}
           fund op LP2 sp
           -> fund+ (OP x op) LP0 (SP' x sp))
```

```
     <- ({e2}{sp}
           cwhe LP0 ctx/id (whr/beta : whr (app (lam E0) e2) (E0 e2)) (SP' e2 sp)
                (CWHE e2 sp)).

- : fund (of/app OP1 OP2 : of (app E1 E2) T0) LP0
     (cut SP20 (andl2 (forallel E2 (impl SP2 ax))))
     <- fund OP1 (lr/=> LP2 LP0) SP20
     <- fund+ OP2 LP2 SP2.

- : fund of/true lr/bool (orr1 (existsevr (eval~/val val/true) topr)).

- : fund of/false lr/bool (orr2 (existsevr (eval~/val val/false) topr)).

- : fund of/num lr/nat' (existsnr _ (andr (existsevr (eval~/val val/num) topr) SP+))
     <- struct-nat _ SP+.

- : fund (of/if OP0 OP1 OP2 : of (if E0 E1 E2) T) (LP : lr T R)
     (cut SP0 (orl
                 (existsevl [E0=>true][_] CTXRED1 E0=>true)
                 (existsevl [E0=>false][_] CTXRED2 E0=>false)))
     <- fund+ OP0 lr/bool SP0
     <- fund OP1 LP SP1
     <- fund+ OP2 LP SP2
     <- cwhe LP ctx/id whr/ift SP1 CWHE1
     <- cwhe LP ctx/id whr/iff SP2 CWHE2
     <- ({ep} lr-ctxred LP (ctx/if ctx/id) ep CWHE1 (CTXRED1 ep))
     <- ({ep} lr-ctxred LP (ctx/if ctx/id) ep CWHE2 (CTXRED2 ep)).

- : fund (of/case OP0 OP1 OP2 : of (case E0 E1 E2) T) (LP : lr T R)
     (cut SP0 (existsnl [n][h]
                 (andl1
                    (existsevl [E0=>num-n][_]
                       andl2
                       (nat+/l
                          ([num-n=num-z]
                             CTXRED1 (eval~/conv eq-exp/id num-n=num-z E0=>num-n))
                          ([n'][num-n=num-s-n'][n'+]
                             (CTXRED2 n'
                                (existsnr n'
                                   (andr
                                      (existsevr (eval~/val val/num) topr)
                                      (ax n'+)))
                                (ax n'+)
                                (eval~/conv eq-exp/id num-n=num-s-n' E0=>num-n))))
                          h)
                       h)))
     <- fund+ OP0 lr/nat' SP0
     <- fund OP1 LP SP1
     <- ({x}{op:of x nat'}{sp:conc* (flr/nat' x)}
           fund op lr/nat' sp
           -> fund+ (OP2 x op) LP (SP2 x sp))
     <- cwhe LP ctx/id whr/case0 SP1 CWHE1
     <- ({n'}{n'+ : conc* (nat+ n')}
           {_:struct-nat n' n'+}{sp:conc* (flr/nat' (num n'))}
           cwhe LP ctx/id whr/case1 (SP2 (num n') sp)
                (CWHE2 n' sp n'+ : conc* (R (case (num (s n')) E1 E2))))
     <- ({ep} lr-ctxred LP (ctx/case ctx/id) ep CWHE1 (CTXRED1 ep))
     <- ({n'}{sp}{n'+}{ep:eval~ E0 (num (s n'))}
           lr-ctxred LP (ctx/case ctx/id) ep (CWHE2 n' sp n'+)
                     (CTXRED2 n' sp n'+ ep)).

%block bstructnat : block {n:nat}{sp:conc* (nat+ n)}{_:struct-nat n sp}.

%block bfund : some {T':tp}{R':pred}{LP':lr T' R'}
                block {x:exp}{op:of x T'}{sp:conc* (R' x)}
                {_:fund op LP' sp}.

%worlds (bnat | bexp | bconc | beval~) (lr-ctxred _ _ _ _ _).
%total (LP) (lr-ctxred LP _ _ _ _).

%worlds (bfund | bstructnat) (struct-nat _ _).
%total (N) (struct-nat N _).

%worlds (bfund | bstructnat) (lr-tot _ _).
%total (T) (lr-tot T _).

%worlds (bfund | bstructnat) (eq-pred-cong2 _ _ _ _).
%total {} (eq-pred-cong2 _ _ _ _).

%worlds (bfund | bstructnat) (lr-det _ _ _).
%total (LP) (lr-det LP _ _).

%worlds (bfund | bstructnat) (pred-conv _ _ _).
%total {} (pred-conv _ _ _).
```

```
%worlds (bnat | bexp | bconc | bstructnat) (cwhe _ _ _ _ _).
%total (LP) (cwhe LP _ _ _ _).

%worlds (bfund | bstructnat) (fund _ _ _) (fund+ _ _ _).
%total (OP OP') (fund OP _ _) (fund+ OP' _ _).
```

A.12 ext.elf

```
ext : lr T R -> conc' (R E) -> eval E V -> type.
%mode ext +LP +SP -EP.
- : ext lr/bool (orr1 (existsevr EP _)) EP'
     <- eval~=>eval EP EP'.
- : ext lr/bool (orr2 (existsevr EP _)) EP'
     <- eval~=>eval EP EP'.
- : ext lr/nat' (existsnr N (andr (existsevr EP _) _)) EP'
     <- eval~=>eval EP EP'.
- : ext (lr/=> _ _) (andr (existser V (existsevr EP _)) _) EP'
     <- eval~=>eval EP EP'.
%worlds () (ext _ _ _).
%total (EP) (ext EP _ _).

term : of E T -> eval E V -> type.
%mode term +OP -EP.
- : term OP EP
     <- fund OP LP SP
     <- ce SP SP'
     <- ext LP SP' EP.
%worlds () (term _ _).
%total {} (term _ _).

%solve op1 : of (if (app (lam [x] true) false) false true) T.
%query 1 1 fund op1 LP SP.
```

B Twelf: Equational reasoning for CBN STLC

The following is a listing of selected parts of the formalization of logical equivalence for call-by-name simply typed lambda calculus. For the full formalization, we refer to the electronic appendix [Ras13].

Note that this listing includes an additional `flip` construct, which models a non-determistic "coin-flip". The construct was removed from the report for simplicity reasons, but does not affect the overall structure of the formalization.

B.1 `sources.cfg`

```
void.elf
% Definitions
nat.elf
lc.elf
sim.elf
% Equality and meta-theorems
eq.elf
eq-lemmas.elf
% Data representation logic
data.elf
% Soundness of the representation logic
norm.elf
% Assertion logic and its consistency
form.elf
assert.elf
admit.elf
cutelim.elf
assert-theorems.elf
% Completeness of representation logic
data-emb.elf
% Useful assertion logic definitions
assert-abbrev.elf
% Definition and proofs of logical equivalence
lr.elf
% Extraction and soundness proof of axiomatic equivalence
ext.elf
sim-lemmas.elf
```

B.2 `nat.elf`, `lc.elf`, `sim.elf`

```
% Natural numbers
nat : type. %name nat N.

z : nat.
s : nat -> nat. %prefix 1 s.


% types
tp : type. %name tp T.
nat' : tp.
=> : tp -> tp -> tp.
%infix right 1 =>.

% expressions
exp : type. %name exp E.
num : nat -> exp.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
case : exp -> exp -> (exp -> exp) -> exp.
flip : exp.
diverge : exp.

% typing
of : exp -> tp -> type. %name of OP.
of/num : of (num N) nat'.
of/lam : of (lam E0) (T2 => T0)
          <- ({x} of x T2 -> of (E0 x) T0).
of/app : of (app E1 E2) T0
          <- of E1 (T2 => T0)
          <- of E2 T2.
of/diverge : of diverge T.
of/case : of (case E0 E1 E2) T
           <- of E0 nat'
           <- of E1 T
           <- ({x} of x nat' -> of (E2 x) T).
of/flip : of flip nat'.

% evaluation (lazy, big-step)
eval : exp -> exp -> type. %name eval EP.

eval/lam : eval (lam E0) (lam E0).
eval/num : eval (num N) (num N).
eval/app : eval (app E1 E2) V
            <- eval (E0 E2) V
            <- eval E1 (lam E0).
eval/case/z : eval (case E0 E1 E2) V
               <- eval E1 V
               <- eval E0 (num z).
eval/case/s : eval (case E0 E1 E2) V
               <- eval (E2 (num N)) V
               <- eval E0 (num (s N)).
eval/flip/z : eval flip (num z).
eval/flip/s : eval flip (num (s z)).

val : exp -> type. %name val VP.
val/lam : val (lam E0).
val/num : val (num N).

%block bnat : block {x:nat}.
%block bexp : block {x:exp}.
%block bexp2 : block {x:exp -> exp}.

%worlds (bnat) (nat).
%worlds (bexp | bexp2 | bnat) (exp).


% Adequate encoding of axiomatization
sim : exp -> exp -> tp -> type. %name sim SIP.
sim/num : sim (num N) (num N) nat'.
sim/flip : sim flip flip nat'.
sim/diverge : sim diverge diverge T.
sim/sym : sim E' E T
           <- sim E E' T.
sim/trans : sim E E'' T
             <- sim E E' T
             <- sim E' E'' T.
sim/cong/lam : sim (lam E0) (lam E0') (T2 => T0)
                <- ({x} sim x x T2
                      -> sim (E0 x) (E0' x) T0).
sim/cong/app : sim (app E1 E2) (app E1' E2') T0
                <- sim E1 E1' (T2 => T0)
                <- sim E2 E2' T2.
```

```
sim/cong/case : sim (case E0 E1 E2) (case E0' E1' E2') T
                <- sim E0 E0' nat'
                <- sim E1 E1' T
                <- ({x} sim x x nat'
                      -> sim (E2 x) (E2' x) T).
sim/case/z : sim (case (num z) E1 E2) E1 T
             <- sim E1 E1 T.
sim/case/s : sim (case (num (s N)) E1 E2) (E2 (num N)) T
             <- ({x} sim x x nat' -> sim (E2 x) (E2 x) T).
sim/case/flip : sim (case flip E1 E2) E' T
                <- sim E1 E' T
                <- sim (E2 (num z)) E' T.
sim/eta : sim E (lam [x] app E x) (T2 => T0)
           <- sim E E (T2 => T0).
sim/beta : sim (app (lam E0) E2) (E0 E2) T0
            <- ({x} sim x x T2 -> sim (E0 x) (E0 x) T0)
            <- sim E2 E2 T2.
sim/subst : sim (E E2) (E' E2) T0
            <- ({x} sim x x T2 -> sim (E x) (E' x) T0)
            <- sim E2 E2 T2.

% A variant of the above that is more useful in proofs. We will show
% that the adequate encoding implies this encoding.  This encoding is
% a little more general, in that it allows rules to be parametric in
% two expressions that are assumed sim-related, but not necessarily
% identical.
sim* : exp -> exp -> tp -> type. %name sim* SIP.

sim*/sym : sim* E' E T
           <- sim* E E' T.
sim*/num : sim* (num N) (num N) nat'.
sim*/flip : sim* flip flip nat'.
sim*/diverge : sim* diverge diverge T.
sim*/trans : sim* E E'' T
             <- sim* E E' T
             <- sim* E' E'' T.
sim*/cong/app : sim* (app E1 E2) (app E1' E2') T0
                <- sim* E1 E1' (T2 => T0)
                <- sim* E2 E2' T2.
sim*/cong/case : sim* (case E0 E1 E2) (case E0' E1' E2') T
                 <- sim* E0 E0' nat'
                 <- sim* E1 E1' T
                 <- ({x}{x'} sim* x x' nat'
                       -> sim* (E2 x) (E2' x') T).
sim*/case/z : sim* (case (num z) E1 E2) E1' T
              <- sim* E1 E1' T.
sim*/case/s : sim* (case (num (s N)) E1 E2) (E2' (num N)) T
              <- ({x}{x'} sim* x x' nat' -> sim* (E2 x) (E2' x') T).
sim*/case/flip : sim* (case flip E1 E2) E' T
                 <- sim* E1 E' T
                 <- sim* (E2 (num z)) E' T.
sim*/beta : sim* (app (lam E0) E2) (E0' E2') T0
            <- ({l}{r} sim* l r T2 -> sim* (E0 l) (E0' r) T0)
            <- sim* E2 E2' T2.
sim*/eta : sim* E (lam [x] app E' x) (T2 => T0)
            <- sim* E E' (T2 => T0).

sim*/subst : sim* (E0 E2) (E0' E2') T0
             <- ({l}{r} sim* l r T2 -> sim* (E0 l) (E0' r) T0)
             <- sim* E2 E2' T2.
sim*/cong/lam : sim* (lam E0) (lam E0') (T2 => T0)
                 <- ({x}{x'} sim* x x' T2
                       -> sim* (E0 x) (E0' x') T0).
```

B.3 eq.elf

```
%%
%% Equality relations
%%
eq-nat : nat -> nat -> type.
eq-nat/id : eq-nat X X.

eq-exp2 : (exp -> exp) -> (exp -> exp) -> type.
eq-exp2/id : eq-exp2 X X.

eq-exp : (exp) -> (exp) -> type.
eq-exp/id : eq-exp X X.
```

B.4 data.elf

```
%%
%% Data representation logic
%%
dform : type. %name dform D.
data : dform -> type. %name data DP.

%%
%% Axiomatization of equality
%%
@void : dform.

@eq-nat : nat -> nat -> dform.
@eq-nat/id : data (@eq-nat X X).
@eq-nat/cong : {X} data (@eq-nat X1 X2) -> data (@eq-nat (X X1) (X X2)).
@eq-nat/trans : data (@eq-nat X1 X2) -> data (@eq-nat X2 X3)
                                     -> data (@eq-nat X1 X3).
@eq-nat/sym : data (@eq-nat X1 X2) -> data (@eq-nat X2 X1).
@eq-nat/void : data @void -> data (@eq-nat X1 X2).
@eq-nat/zs : data (@eq-nat (z) (s N)) -> data @void.
@eq-nat/sz : data (@eq-nat (s N) (z)) -> data @void.

@eq-exp2 : (exp -> exp) -> (exp -> exp) -> dform.
@eq-exp2/id : data (@eq-exp2 X X).
@eq-exp2/cong : {X} data (@eq-exp2 X1 X2) -> data (@eq-exp2 (X X1) (X X2)).
@eq-exp2/trans : data (@eq-exp2 X1 X2) -> data (@eq-exp2 X2 X3)
                                       -> data (@eq-exp2 X1 X3).
@eq-exp2/sym : data (@eq-exp2 X1 X2) -> data (@eq-exp2 X2 X1).
@eq-exp2/void : data @void -> data (@eq-exp2 X1 X2).

@eq-exp : exp -> exp -> dform.
@eq-exp/id : data (@eq-exp X X).
@eq-exp/cong : {X} data (@eq-exp X1 X2) -> data (@eq-exp (X X1) (X X2)).
@eq-exp/trans : data (@eq-exp X1 X2) -> data (@eq-exp X2 X3)
                                     -> data (@eq-exp X1 X3).
@eq-exp/sym : data (@eq-exp X1 X2) -> data (@eq-exp X2 X1).
@eq-exp/void : data @void -> data (@eq-exp X1 X2).
@eq-exp/num-lam : data (@eq-exp (num N) (lam E0)) -> data @void.
@eq-exp/num-app : data (@eq-exp (num N) (app E1 E2)) -> data @void.
@eq-exp/num-case : data (@eq-exp (num N) (case E0 E1 E2)) -> data @void.
@eq-exp/num-flip : data (@eq-exp (num N) (flip)) -> data @void.
@eq-exp/lam-num : data (@eq-exp (lam E0) (num N)) -> data @void.
@eq-exp/lam-app : data (@eq-exp (lam E0) (app E1 E2)) -> data @void.
@eq-exp/app-lam : data (@eq-exp (app E1 E2) (lam E0)) -> data @void.
@eq-exp/app-num : data (@eq-exp (app E1 E2) (num N0)) -> data @void.
@eq-exp/app-flip : data (@eq-exp (app E1 E2) (flip)) -> data @void.
@eq-exp/diverge-lam : data (@eq-exp (diverge) (lam E0)) -> data @void.
@eq-exp/diverge-num : data (@eq-exp (diverge) (num N)) -> data @void.
@eq-exp/diverge-app : data (@eq-exp (diverge) (app E1 E2)) -> data @void.
@eq-exp/diverge-case : data (@eq-exp (diverge) (case E0 E1 E2)) -> data @void.
@eq-exp/diverge-flip : data (@eq-exp (diverge) (flip)) -> data @void.
@eq-exp/flip-lam : data (@eq-exp (flip) (lam E0)) -> data @void.
@eq-exp/flip-num : data (@eq-exp (flip) (num _)) -> data @void.
@eq-exp/flip-app : data (@eq-exp (flip) (app _ _)) -> data @void.
@eq-exp/flip-case : data (@eq-exp (flip) (case _ _ _)) -> data @void.
@eq-exp/app-case : data (@eq-exp (app E1 E2) (case E3 E4 E5)) -> data @void.
@eq-exp/lam-case : data (@eq-exp (lam E0) (case E1 E2 E3)) -> data @void.
@eq-exp/lam-flip : data (@eq-exp (lam E0) (flip)) -> data @void.
@eq-exp/case-lam : data (@eq-exp (case E0 E1 E2) (lam E3)) -> data @void.
@eq-exp/case-num : data (@eq-exp (case E0 E1 E2) (num N)) -> data @void.
@eq-exp/case-app : data (@eq-exp (case E0 E1 E2) (app E3 E4)) -> data @void.
@eq-exp/case-flip : data (@eq-exp (case E0 E1 E2) (flip)) -> data @void.
@eq-exp/cvrs-app1 : data (@eq-exp (app E1 E2) (app E3 E4))
                     -> data (@eq-exp (E1) (E3)).
@eq-exp/cvrs-app2 : data (@eq-exp (app E1 E2) (app E3 E4))
                     -> data (@eq-exp (E2) (E4)).
@eq-exp/cvrs-lam : data (@eq-exp (lam E0) (lam E1))
                    -> data (@eq-exp2 (E0) (E1)).
@eq-exp/cvrs-case0 : data (@eq-exp (case E0 E1 E2) (case E3 E4 E5))
                      -> data (@eq-exp (E0) (E3)).
@eq-exp/cvrs-case1 : data (@eq-exp (case E0 E1 E2) (case E3 E4 E5))
                      -> data (@eq-exp (E1) (E4)).
@eq-exp/cvrs-case2 : data (@eq-exp (case E0 E1 E2) (case E3 E4 E5))
                      -> data (@eq-exp2 (E2) (E5)).
```

```
@eq-exp/cvrs-num : data (@eq-exp (num N1) (num N2)) -> data (@eq-nat (N1) (N2)).
@eq-nat/cvrs-s : data (@eq-nat (s N1) (s N2)) -> data (@eq-nat (N1) (N2)).
@eq-exp/cong-nat : {X} data (@eq-nat X1 X2) -> data (@eq-exp (X X1) (X X2)).

@eq-exp/subst : data (@eq-exp2 E E') -> data (@eq-exp E2 E2')
                                     -> data (@eq-exp (E E2) (E' E2')).

%%
%% Embedding of evaluation judgment
%%
@top : dform.

@top/top : data @top.

@eval : exp -> exp -> dform.

@eval/lam : data (@eq-exp X1 (lam E0))
             -> data (@eq-exp X2 (lam E0)) -> data (@eval X1 X2).
@eval/num : data (@eq-exp X1 (num N0))
             -> data (@eq-exp X2 (num N0)) -> data (@eval X1 X2).
@eval/app : data (@eval E1 (lam E0))
             -> data (@eval (E0 E2) Ev)
             -> data (@eq-exp X1 (app E1 E2))
             -> data (@eq-exp X2 Ev) -> data (@eval X1 X2).
@eval/case/z : data (@eval E0 (num z))
                -> data (@eval E1 Ev)
                -> data (@eq-exp X1 (case E0 E1 E2))
                -> data (@eq-exp X2 Ev) -> data (@eval X1 X2).
@eval/case/s : data (@eval E0 (num (s Nx)))
                -> data (@eval (E2 (num Nx)) Ev)
                -> data (@eq-exp X1 (case E0 E1 E2))
                -> data (@eq-exp X2 Ev) -> data (@eval X1 X2).
@eval/flip/z : data (@eq-exp X1 flip)
                -> data (@eq-exp X2 (num z)) -> data (@eval X1 X2).
@eval/flip/s : data (@eq-exp X1 flip)
                -> data (@eq-exp X2 (num (s z))) -> data (@eval X1 X2).%%
```

B.5 form.elf, assert.elf

```
form : type. %name form F.

% Basic logical connectives.
==> : form -> form -> form. %infix right 1 ==>.
\/ : form -> form -> form. %infix left 2 \/.
/\ : form -> form -> form. %infix left 3 /\.
top : form.

% Auto generated formulas for quantification of objects in the domain nat
foralln : (nat -> form) -> form.
existsn : (nat -> form) -> form.

% Auto generated formulas for quantification of objects in the domain data D
foralld : (data D -> form) -> form.
existsd : (data D -> form) -> form.

% Auto generated formulas for quantification of objects in the domain exp
foralle : (exp -> form) -> form.
existse : (exp -> form) -> form.

data+ : data D -> form.

metric : type. %name metric M.
metric/bin : metric -> metric -> metric.
metric/una : metric -> metric.
metric/nul : metric.

%abbrev
dp-z : data @top = @top/top.

% We can show that case analysis over nested derivations is sound as
% long as we do not have mutually recursive judgments. We use prec to
% define an implicit order on judgments.
prec : type. %name prec P.
prec/zero : prec.
prec/succ : prec -> prec.

metric-red : form
              -> metric   % Form metric
%              -> prec     % Auxiliary metric 1
              -> data D   % Auxiliary metric 2
```

```
%              -> nat      % Auxiliary metric 3
              -> type. %name metric-red RP.
metric-red/imp : metric-red (F1 ==> F2) (metric/bin M1 M2) dp-z
                  <- metric-red F1 M1 _
                  <- metric-red F2 M2 _.
metric-red/or : metric-red (F1 \/ F2) (metric/bin M1 M2) dp-z
                 <- metric-red F1 M1 _
                 <- metric-red F2 M2 _.
metric-red/and : metric-red (F1 /\ F2) (metric/bin M1 M2) dp-z
                  <- metric-red F1 M1 _
                  <- metric-red F2 M2 _.
metric-red/top : metric-red top metric/nul dp-z.

% Auto generated rules for metric reduction of formulas for
% quantification of objects in the domain nat

metric-red/foralln : metric-red (foralln F) (metric/una M1) dp-z
  <- ({x} metric-red (F x) M1 (DP1 x : data (D x))).
metric-red/existsn : metric-red (existsn F) (metric/una M1) dp-z
  <- ({x} metric-red (F x) M1 (DP1 x : data (D x))).

% Auto generated rules for metric reduction of formulas for
% quantification of objects in the domain data D

metric-red/foralld : metric-red (foralld F) (metric/una M1) dp-z
  <- ({x} metric-red (F x) M1 (DP1 x : data (D))).
metric-red/existsd : metric-red (existsd F) (metric/una M1) dp-z
  <- ({x} metric-red (F x) M1 (DP1 x : data (D))).

% Auto generated rules for metric reduction of formulas for
% quantification of objects in the domain exp

metric-red/foralle : metric-red (foralle F) (metric/una M1) dp-z
  <- ({x} metric-red (F x) M1 (DP1 x : data (D x))).
metric-red/existse : metric-red (existse F) (metric/una M1) dp-z
  <- ({x} metric-red (F x) M1 (DP1 x : data (D x))).


metric-red/data+ : metric-red (data+ DP) metric/nul DP.

metric-red-tot : {F} metric-red F M DP -> type.
%mode metric-red-tot +F -RP.

- : metric-red-tot (F1 ==> F2) (metric-red/imp RP2 RP1)
     <- metric-red-tot F1 RP1
     <- metric-red-tot F2 RP2.
- : metric-red-tot (F1 \/ F2) (metric-red/or RP2 RP1)
     <- metric-red-tot F1 RP1
     <- metric-red-tot F2 RP2.
- : metric-red-tot (F1 /\ F2) (metric-red/and RP2 RP1)
     <- metric-red-tot F1 RP1
     <- metric-red-tot F2 RP2.
- : metric-red-tot top metric-red/top.

% Auto generated rules for totality of metric reduction of formulas
% for quantification of objects in the domain nat
- : metric-red-tot (foralln F) (metric-red/foralln RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (existsn F) (metric-red/existsn RP)
     <- ({x} metric-red-tot (F x) (RP x)).

% Auto generated rules for totality of metric reduction of formulas
% for quantification of objects in the domain data D
- : metric-red-tot (foralld F) (metric-red/foralld RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (existsd F) (metric-red/existsd RP)
     <- ({x} metric-red-tot (F x) (RP x)).

% Auto generated rules for totality of metric reduction of formulas
% for quantification of objects in the domain exp
- : metric-red-tot (foralle F) (metric-red/foralle RP)
     <- ({x} metric-red-tot (F x) (RP x)).
- : metric-red-tot (existse F) (metric-red/existse RP)
     <- ({x} metric-red-tot (F x) (RP x)).

- : metric-red-tot (data+ DP) metric-red/data+.

%%
%% Assertion logic
%%
allow : type. %name allow V.
cutful : allow.
```

```
cutfree : allow.

hyp : form -> type. %name hyp H.
conc : allow -> form -> type. %name conc SP.

%abbrev
conc' = conc cutfree.

%abbrev
conc* = conc cutful.

cut : conc V A -> (hyp A -> conc V B)
                -> conc* B.

ax : hyp F -> conc V F.

andr : conc V F1 -> conc V F2 -> conc V (F1 /\ F2).
andl1 : (hyp F1 -> conc V C)
         -> (hyp (F1 /\ F2) -> conc V C).
andl2 : (hyp F2 -> conc V C)
         -> (hyp (F1 /\ F2) -> conc V C).
orr1 : conc V F
        -> conc V (F \/ G).
orr2 : conc V G
        -> conc V (F \/ G).
orl : (hyp F -> conc V C) -> (hyp G -> conc V C)
        -> (hyp (F \/ G) -> conc V C).
impr : (hyp F -> conc V G)
        -> conc V (F ==> G).
impl : conc V F
        -> (hyp G -> conc V C)
        -> (hyp (F ==> G) -> conc V C).
topr : conc V top.

forallnr : ({n : nat} conc V (F n)) -> conc V (foralln F).
forallnl : {n : nat} (hyp (F n) -> conc V C)
    -> (hyp (foralln F) -> conc V C).
existsnr : {x : nat} conc V (F x) -> conc V (existsn F).
existsnl : ({x : nat} hyp (F x) -> conc V C)
              -> (hyp (existsn F) -> conc V C).

foralldr : ({d : data D} conc V (F d)) -> conc V (foralld F).
foralldl : {d : data D} (hyp (F d) -> conc V C)
    -> (hyp (foralld F) -> conc V C).
existsdr : {x : data D} conc V (F x) -> conc V (existsd F).
existsdl : ({x : data D} hyp (F x) -> conc V C)
              -> (hyp (existsd F) -> conc V C).

foraller : ({e : exp} conc V (F e)) -> conc V (foralle F).
forallel : {e : exp} (hyp (F e) -> conc V C)
    -> (hyp (foralle F) -> conc V C).
existser : {x : exp} conc V (F x) -> conc V (existse F).
existsel : ({x : exp} hyp (F x) -> conc V C)
              -> (hyp (existse F) -> conc V C).

% Some useful parameter reorderings:
andl1' : hyp (F1 /\ F2) -> (hyp F1 -> conc V C) -> conc V C
  = [h][p] andl1 p h.
andl2' : hyp (F1 /\ F2) -> (hyp F2 -> conc V C) -> conc V C = [h][p] andl2 p h.

% Rules for case analysis
data+/@eval/lam : conc V (data+ (@eval/lam Q1 Q2)).
data+/@eval/num : conc V (data+ (@eval/num Q1 Q2)).
data+/@eval/app : conc V (data+ DP1)
                  -> conc V (data+ DP2)
                  -> conc V (data+ (@eval/app DP1 DP2 Q1 Q2)).
data+/@eval/case/z : conc V (data+ DP1)
                     -> conc V (data+ DP2)
                        -> conc V (data+ (@eval/case/z DP1 DP2 Q1 Q2)).
data+/@eval/case/s : conc V (data+ DP1)
                     -> conc V (data+ DP2)
                        -> conc V (data+ (@eval/case/s DP1 DP2 Q1 Q2)).
data+/@eval/flip/z : conc V (data+ (@eval/flip/z Q1 Q2)).
data+/@eval/flip/s : conc V (data+ (@eval/flip/s Q1 Q2)).
data+/@eval/l : ({E0}
                 {q1:data (@eq-exp X1 (lam E0))}
                 {q2:data (@eq-exp X2 (lam E0))}
                   conc V C)
             -> ({N0}
                 {q1:data (@eq-exp X1 (num N0))}
                 {q2:data (@eq-exp X2 (num N0))}
                   conc V C)
```

```
             -> ({Ev}{E1}{E2}{E0}
                 {dp1:data (@eval E1 (lam E0))}{dp2:data (@eval (E0 E2) Ev)}
                 {h1:hyp (data+ dp1)}{h2:hyp (data+ dp2)}
                 {q1:data (@eq-exp X1 (app E1 E2))}{q2:data (@eq-exp X2 Ev)}
                   conc V C)
             -> ({E0}{E1}{E2}{Ev}
                 {dp1:data (@eval E0 (num z))}{dp2:data (@eval E1 Ev)}
                 {h1:hyp (data+ dp1)}{h2:hyp (data+ dp2)}
                 {q1:data (@eq-exp X1 (case E0 E1 E2))}
                 {q2:data (@eq-exp X2 Ev)}
                   conc V C)
             -> ({E0}{E1}{E2}{Ev}{Nx}
                 {dp1:data (@eval E0 (num (s Nx)))}
                 {dp2:data (@eval (E2 (num Nx)) Ev)}
                 {h1:hyp (data+ dp1)}{h2:hyp (data+ dp2)}
                 {q1:data (@eq-exp X1 (case E0 E1 E2))}{q2:data (@eq-exp X2 Ev)}
                   conc V C)
             -> ({q1:data (@eq-exp X1 flip)}{q2:data (@eq-exp X2 (num z))}
                   conc V C)
             -> ({q1:data (@eq-exp X1 flip)}{q2:data (@eq-exp X2 (num (s z)))}
                   conc V C)
             -> hyp (data+ (DP : data (@eval X1 X2))) -> conc V C.

%block bconc : some {F:form} block {_:conc* F}.
%block bconc2 : some {R:exp -> exp -> form} {S:exp -> exp -> form}
                block
                {_:{x:exp}{x':exp} conc* (R x x') -> conc* (S x x')}.
%block bhyp : some {F:form} block {h:hyp F}.
%block bdata : some {D:dform} block {x:data D}.
%worlds (bdata | bexp | bexp2 | bnat | bhyp) (hyp _).
%worlds (bdata | bexp | bexp2 | bnat | bhyp | bconc | bconc2) (conc _ _).
```

B.6 ext.elf, sim-lemmas.elf

```
% Extraction of evaluation:
eval-ext : conc' (#eval E V) -> eval E V -> type.
%mode eval-ext +SP -EP.

- : eval-ext (existsdr DP _) EP
   <- eval-unemb DP EP.

%worlds () (eval-ext _ _).
%total {} (eval-ext _ _).

% Extraction, kleene equality
lr-ext : eval E (num N) -> lr nat' R -> conc* (R E E') -> eval E' (num N) -> type.
%mode lr-ext +EP +LP +SP -EP'.

- : lr-ext EP lr/nat' SP EP'
     <- eval-emb EP DP SP+
     <- ce (cut SP (forallnl N (andl1 (impl (existsdr DP SP+) ax)))) SP'
     <- eval-ext SP' EP'.

%worlds () (lr-ext _ _ _ _).
%total {} (lr-ext _ _ _ _).


%%
%% Soundness of axiomatic equational reasoning
%%

% Theorem: sim is reflexive for well-typed expressions.
sim-refl : of E T -> sim E E T -> type.
%mode sim-refl +OP -SIP.

- : sim-refl of/num sim/num.
- : sim-refl of/diverge sim/diverge.
- : sim-refl of/flip sim/flip.
- : sim-refl (of/app OP2 OP1) (sim/cong/app SIP2 SIP1)
     <- sim-refl OP1 SIP1
     <- sim-refl OP2 SIP2.
- : sim-refl (of/lam OP0) (sim/cong/lam SIP0)
     <- ({x}{op:of x T2}{sip:sim x x T2}
           {_:sim-refl op sip}
           sim-refl (OP0 x op) (SIP0 x sip)).
- : sim-refl (of/case OP2 OP1 OP0) (sim/cong/case SIP2 SIP1 SIP0)
     <- sim-refl OP0 SIP0
     <- sim-refl OP1 SIP1
     <- ({x}{op:of x nat'}{sip:sim x x nat'}
```

```
          {_:sim-refl op sip}
          sim-refl (OP2 x op) (SIP2 x sip)).

%block bfundsim : some {T2:tp}
                  block {x}{op:of x T2}{sip:sim x x T2}
                  {_:sim-refl op sip}.

%worlds (bfundsim) (sim-refl _ _).
%total (OP) (sim-refl OP _).

% Doubling lemma, which is necessary in the totality proof of
% sim=>sim*.
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'.

- : sim*-double ([_][_] SIP) ([_][_][_] SIP).
- : sim*-double ([_][sip] sip) ([_][_][sip] sip).
- : sim*-double ([_][_] sim*/num) ([_][_][_] sim*/num).
- : sim*-double ([_][_] sim*/diverge) ([_][_][_] sim*/diverge).
- : sim*-double ([x][sip] sim*/case/flip (SIP2 x sip) (SIP1 x sip))
     ([l][r][sip] sim*/case/flip (SIP2' l r sip) (SIP1' l r sip))
     <- sim*-double SIP1 SIP1'
     <- sim*-double SIP2 SIP2'.
- : sim*-double ([x][sip] sim*/sym (SIP x sip))
     ([l][r][sip] sim*/sym (SIP' r l (sim*/sym sip)))
     <- sim*-double SIP SIP'.
- : sim*-double ([x][sip] sim*/trans (SIP2 x sip) (SIP1 x sip))
     ([l][r][sip]
        sim*/trans (SIP2 r (sim*/trans sip (sim*/sym sip))) (SIP1' l r sip))
     <- sim*-double SIP1 SIP1'.
- : sim*-double ([x][sip] sim*/cong/app (SIP2 x sip) (SIP1 x sip))
     ([l][r][sip] sim*/cong/app (SIP2' l r sip) (SIP1' l r sip))
     <- sim*-double SIP1 SIP1'
     <- sim*-double SIP2 SIP2'.
- : sim*-double ([x][sip] sim*/cong/case (SIP2 x sip) (SIP1 x sip) (SIP0 x sip))
     ([l][r][sip] sim*/cong/case (SIP2' l r sip) (SIP1' l r sip) (SIP0' l r sip))
     <- ({l'}{r'}{sip'}
           {_:sim*-double ([x][sip:sim* x x nat'] sip')
                ([l][r][sip] sip')}
           sim*-double ([x][sip] SIP2 x sip l' r' sip')
           ([l][r][sip] SIP2' l r sip l' r' sip'))
     <- sim*-double SIP1 SIP1'
     <- sim*-double SIP0 SIP0'.
- : sim*-double ([x][sip] sim*/case/z (SIP1 x sip))
     ([l][r][sip] sim*/case/z (SIP1' l r sip))
     <- sim*-double SIP1 SIP1'.
- : sim*-double
     ([x][sip] sim*/case/s (SIP2 x sip) : sim* (case (num (s N)) _ _) _ _)
     ([l][r][sip] sim*/case/s (SIP2' l r sip))
     <- ({l'}{r'}{sip'}
           {_:sim*-double ([x][sip: sim* x x nat'] sip')
                ([l][r][sip:sim* l r nat'] sip')}
           sim*-double ([x][sip] SIP2 x sip l' r' sip')
                       ([l][r][sip] SIP2' l r sip l' r' sip')).
- : sim*-double ([x][sip] sim*/beta (SIP2 x sip) (SIP0 x sip))
     ([l][r][sip] sim*/beta (SIP2' l r sip) (SIP0' l r sip))
     <- sim*-double SIP2 SIP2'
     <- ({l'}{r'}{sip'}
           {_:sim*-double ([x][sip: sim* x x T] sip')
                ([l][r][sip:sim* l r T] sip')}
           sim*-double ([x][sip] SIP0 x sip l' r' sip')
                       ([l][r][sip] SIP0' l r sip l' r' sip')).
- : sim*-double ([x][sip] sim*/eta (SIP x sip))
                ([l][r][sip] sim*/eta (SIP' l r sip))
     <- sim*-double SIP SIP'.
- : sim*-double ([x][sip] sim*/cong/lam (SIP0 x sip))
     ([l][r][sip] sim*/cong/lam (SIP0' l r sip))
     <- ({l'}{r'}{sip'}
           {_:sim*-double ([x][sip:sim* x x T] sip')
                ([l][r][sip] sip')}
           sim*-double ([x][sip] SIP0 x sip l' r' sip')
                       ([l][r][sip] SIP0' l r sip l' r' sip')).
- : sim*-double ([x][sip] sim*/subst (SIP2 x sip) (SIP0 x sip))
     ([l'][r'][sip'] sim*/subst (SIP2' l' r' sip') (SIP0' l' r' sip'))
     <- sim*-double SIP2 SIP2'
     <- ({l'}{r'}{sip'}
```

```
          {_:sim*-double ([x][sip:sim* x x T] sip')
               ([l][r][sip] sip')}
          sim*-double ([x][sip] SIP0 x sip l' r' sip')
                     ([l][r][sip] SIP0' l r sip l' r' sip')).

%block bsimdouble : some {T:tp}{T':tp}
                    block
                    {l}{r}{sip:sim* l r T'}
                    {_:sim*-double ([x][sip': sim* x x T] sip)
                                   ([l][r][sip':sim* l r T] sip)}.

% Translation from adequate encoding to dual variable encoding.
sim=>sim* : sim E E' T -> sim* E E' T -> type.
%mode sim=>sim* +SIP -SIP'.

- : sim=>sim* sim/num sim*/num.
- : sim=>sim* sim/diverge sim*/diverge.
- : sim=>sim* sim/flip sim*/flip.
- : sim=>sim* (sim/case/flip SIP2 SIP1) (sim*/case/flip SIP2' SIP1')
     <- sim=>sim* SIP1 SIP1'
     <- sim=>sim* SIP2 SIP2'.
- : sim=>sim* (sim/sym SIP0) (sim*/sym SIP0')
     <- sim=>sim* SIP0 SIP0'.
- : sim=>sim* (sim/trans SIP2 SIP1) (sim*/trans SIP2' SIP1')
     <- sim=>sim* SIP2 SIP2'
     <- sim=>sim* SIP1 SIP1'.
- : sim=>sim* (sim/cong/app SIP2 SIP1) (sim*/cong/app SIP2' SIP1')
     <- sim=>sim* SIP2 SIP2'
     <- sim=>sim* SIP1 SIP1'.
- : sim=>sim* (sim/cong/case SIP2 SIP1 SIP0) (sim*/cong/case SIP2' SIP1' SIP0')
     <- ({x}{sip:sim x x nat'}{sip':sim* x x nat'}
         {_:sim*-double ([y][sip:sim* y y T'] sip')
              ([l][r][sip] sip')}
         {_:sim=>sim* sip sip'}
           sim=>sim* (SIP2 x sip) (SIP2'' x sip'))
     <- sim*-double SIP2'' SIP2'
     <- sim=>sim* SIP1 SIP1'
     <- sim=>sim* SIP0 SIP0'.
- : sim=>sim* (sim/case/s SIP2 : sim (case (num (s N)) _ _) _ _)
     (sim*/case/s SIP2' : sim* (case (num (s N)) _ _) _ _)
     <- ({x}{sip:sim x x nat'}{sip':sim* x x nat'}
           {_:sim*-double ([y][sip:sim* y y T'] sip')
                ([l][r][sip] sip')}
           {_:sim=>sim* sip sip'}
           sim=>sim* (SIP2 x sip) (SIP2'' x sip'))
     <- sim*-double SIP2'' SIP2'.
- : sim=>sim* (sim/case/z SIP1) (sim*/case/z SIP1')
     <- sim=>sim* SIP1 SIP1'.
- : sim=>sim* (sim/eta SIP) (sim*/eta SIP')
     <- sim=>sim* SIP SIP'.
- : sim=>sim* (sim/beta SIP2 SIP0) (sim*/beta SIP2' SIP0')
     <- sim=>sim* SIP2 SIP2'
     <- ({x}{sip:sim x x T2}{sip':sim* x x T2}
         {_:sim*-double ([y][sip:sim* y y T'] sip')
              ([l][r][sip] sip')}
         {_:sim=>sim* sip sip'}
           sim=>sim* (SIP0 x sip) (SIP0'' x sip'))
     <- sim*-double SIP0'' SIP0'.
- : sim=>sim* (sim/subst SIP2 SIP0) (sim*/subst SIP2' SIP0')
     <- sim=>sim* SIP2 SIP2'
     <- ({x}{sip:sim x x T2}{sip':sim* x x T2}
         {_:sim*-double ([y][sip:sim* y y T'] sip')
              ([l][r][sip] sip')}
         {_:sim=>sim* sip sip'}
           sim=>sim* (SIP0 x sip) (SIP0'' x sip'))
     <- sim*-double SIP0'' SIP0'.
- : sim=>sim* (sim/cong/lam SIP0) (sim*/cong/lam SIP0')
     <- ({x}{sip:sim x x T2}{sip':sim* x x T2}
         {_:sim*-double ([y][sip:sim* y y T'] sip')
              ([l][r][sip] sip')}
         {_:sim=>sim* sip sip'}
           sim=>sim* (SIP0 x sip) (SIP0'' x sip'))
     <- sim*-double SIP0'' SIP0'.

%block bsimconv : some {T2:tp}{T':tp}
                  block
                  {x}{sip:sim x x T2}{sip':sim* x x T2}
                  {_:sim*-double ([y][sip:sim* y y T'] sip')
```

```
                    ([l][r][sip] sip')}
                {_:sim=>sim* sip sip'}.
%worlds (bsimconv | bsimdouble) (sim*-double _ _).
%total (SIP) (sim*-double SIP _).
%worlds (bsimconv) (sim=>sim* _ _).
%total (SIP) (sim=>sim* SIP _).

% Lemma: sim* implies logical relation.
sim*-lr+ : sim* E E' T -> lr T R -> conc* (R E E') -> type.
%mode sim*-lr+ +SIP +LP -SP.
sim*-lr : sim* E E' T -> lr T R -> conc* (R E E') -> type.
%mode sim*-lr +SIP -LP -SP.
- : sim*-lr+ (SIP : sim* E E' T) LP SP
     <- sim*-lr SIP LP' SP'
     <- dlr LP' LP FEQ
     <- conv-conc E E' FEQ SP' SP.
- : sim*-lr (sim*/sym SIP) LP SP
     <- sim*-lr SIP (LP : lr T R) SP'
     <- sym-lr _ _ _ _ LP SP' SP.
- : sim*-lr (sim*/trans (SIP2 : sim* E' E'' T) (SIP1 : sim* E E' T)) LP SP
     <- sim*-lr SIP1 LP SP1
     <- sim*-lr+ SIP2 LP SP2
     <- trans-lr T E E' E'' _ LP SP1 SP2 SP.
- : sim*-lr (sim*/cong/app SIP2 SIP1) LP0 SP
     <- sim*-lr SIP1 (lr/=> (LP0 : lr T0 R0) (LP2 : lr T2 R2)) SP0
     <- sim*-lr+ SIP2 LP2 SP2
     <- cong-app-lr LP2 LP0 SP0 SP2 SP.
- : sim*-lr (sim*/cong/case SIP2 SIP1 SIP0) LP SP'
     <- sim*-lr+ SIP0 lr/nat' SP0
     <- sim*-lr SIP1 (LP : lr T R) SP1
     <- ({x}{x'}{sip:sim* x x' nat'}{sp:conc* (keq+ x x')}
           {_:sim*-lr sip lr/nat' sp}
           sim*-lr+ (SIP2 x x' sip) LP (SP2 x x' sp))
     <- cong-case-lr _ LP SP0 SP1 SP2 SP'.
- : sim*-lr (sim*/case/z SIP1) LP SP
     <- sim*-lr SIP1 (LP : lr T R) SP1
     <- case-z-lr E2 LP SP1 SP.
- : sim*-lr (sim*/case/s SIP2) LP SP
     <- tlr T (LP : lr T R)
     <- ({x}{x'}{sip:sim* x x' nat'}{sp:conc* (keq+ x x')}
           {_:sim*-lr sip lr/nat' sp}
           sim*-lr+ (SIP2 x x' sip) LP (SP2 x x' sp))
     <- case-s-lr _ _ LP SP2 SP.
- : sim*-lr (sim*/case/flip SIP2 SIP1) LP SP
     <- tlr T (LP : lr T R)
     <- sim*-lr+ SIP1 LP SP1
     <- sim*-lr+ SIP2 LP SP2
     <- case-flip-lr T _ _ _ LP SP1 SP2 SP.
- : sim*-lr sim*/num lr/nat' SP
     <- refl-num-lr _ SP.
- : sim*-lr (sim*/diverge : sim* diverge diverge T) LP SP
     <- tlr T LP
     <- refl-diverge-lr T LP SP.
- : sim*-lr (sim*/beta SIP2 SIP0) LP0 SP
     <- sim*-lr SIP2 (LP2 : lr T2 R2) SP2
     <- tlr T0 (LP0 : lr T0 R0)
     <- ({x}{x'}{sip:sim* x x' T2}{sp:conc cutful (R2 x x')}
         {_:sim*-lr sip LP2 sp}
           sim*-lr+ (SIP0 x x' sip) LP0 (SP' x x' sp))
     <- cong-lam-lr LP2 LP0 SP' SP0
     <- beta-lr LP0 LP2 SP0 SP2 SP.
- : sim*-lr (sim*/eta SIP1) LP SP
     <- sim*-lr SIP1 (LP : lr (T2 => T1) R) SP1
     <- eta-lr LP SP1 SP.
- : sim*-lr (sim*/subst (SIP2 : sim* E2 E2' T2)
                  ([x][x'][sip] (SIP x x' sip) : sim* (E x) (E' x') T0))
        (LP : lr T0 R0) (SP' E2 E2' SP2)
     <- sim*-lr SIP2 (LP2 : lr T2 R2) SP2
     <- tlr T0 LP
     <- ({x}{x'}{sip:sim* x x' T2}{sp:conc cutful (R2 x x')}
```

```
        {_:sim*-lr sip LP2 sp}
          sim*-lr+ (SIP x x' sip) LP (SP' x x' sp)).

- : sim*-lr (sim*/cong/lam ([x][x'][sip] (SIP0 x x' sip)
                            : sim* (E0 x) (E0' x') T0))
     (lr/=> LP0 LP2) SP
     <- tlr T2 (LP2 : lr T2 R2)
     <- tlr T0 (LP0 : lr T0 R0)
     <- ({x}{x'}{sip:sim* x x' T2}{sp:conc* (R2 x x')}
           {_:sim*-lr sip LP2 sp}
           sim*-lr+ (SIP0 x x' sip) LP0 (SP' x x' sp))
     <- cong-lam-lr LP2 LP0 SP' SP.

- : sim*-lr sim*/flip lr/nat' refl-flip-lr.

%block bsim* : some
                {T2:tp}{R2}{LP2:lr T2 R2}
                block
                {x:exp}{x':exp}{sip:sim* x x' T2}{sp:conc cutful (R2 x x')}
                {_:sim*-lr sip LP2 sp}.

%worlds (bexp | bexp2 | bconc | bsim*) (beta-lr _ _ _ _ _) (eta-lr _ _ _)
                (case-z-lr _ _ _ _) (case-s-lr _ _ _ _ _)
                (case-flip-lr _ _ _ _ _ _ _ _).
%total {} (eta-lr _ _ _).
%total {} (beta-lr _ _ _ _ _).
%total {} (case-z-lr _ _ _ _).
%total {} (case-s-lr _ _ _ _ _).
%total (T) (case-flip-lr T _ _ _ _ _ _ _).
%worlds (bsim*) (sim*-lr _ _ _) (sim*-lr+ _ _ _).
%total (SIP SIP') (sim*-lr SIP _ _) (sim*-lr+ SIP' _ _).

% Soundness theorem: sim at naturals implies kleene equality.
sim-ext : eval E (num N) -> sim E E' nat' -> eval E' (num N) -> type.
%mode sim-ext +EP +SIP -EP'.

- : sim-ext EP (SIP : sim E E' nat') EP'
     <- sim=>sim* SIP SIP'
     <- sim*-lr SIP' LP SP
     <- lr-ext EP LP SP EP'.

%worlds () (sim-ext _ _ _).
%total {} (sim-ext _ _ _).

% Corollary: equivalence with canonical natural number implies
% evaluation.
sim-eval : sim E (num N) nat' -> eval E (num N) -> type.
%mode sim-eval +SIP -EP.

- : sim-eval SIP EP
     <- sim-ext eval/num (sim/sym SIP) EP.

%worlds () (sim-eval _ _).
%total {} (sim-eval _ _).
```

C Twelf: Equational reasoning for CBV STLC

Due to the code size of the formalization of the proofs from Chapter 5, we only include some representative excerpts. The full formalization can be found in the electronic appendix [Ras13].

C.1 sources.cfg

```
% Preliminary definitions
void.elf
nat.elf
% Language definitions
lc.elf
% Axiomatic equational reasoning
sim.elf
% Equality relations
eq.elf
% Data representation logic
data.elf
% Lemmas required in soundness proofs
% of data representation logic
eq-lemmas.elf
lc-lemmas.elf
% Soundness of representation logic
norm.elf
% Assertion logic definition
form.elf
assert.elf
% Cut admissibility and cut elimination
admit.elf
cutelim.elf
assert-theorems.elf
% Completeness of representation logic
data-emb.elf
% Useful assertion logic abbreviations
assert-abbrev.elf
% Logical relation and proofs
lr.elf
% Soundness of axiomatic equational reasoning
sim-lemmas.elf
% Extraction, i.e., meta-level soundness proof
ext.elf
```

C.2 nat.elf, lc.elf

```
nat : type. %name nat N.
z : nat.
s : nat -> nat.
%block bnat : block {_:nat}.


%%%
%%% Object language representation
%%%
% Types
tp : type. %name tp T.
nat' : tp.
=> : tp -> tp -> tp.
%infix right 1 =>.
% Expressions
exp : type. %name exp E.
zero : exp.
succ : exp -> exp.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
case : exp -> exp -> (exp -> exp) -> exp.
choice : exp -> exp -> exp.
fail : exp.
% Numerals
num : nat -> exp -> type.
num/z : num z zero.
num/s : num (s N) (succ E)
         <- num N E.
eval : exp -> exp -> type. %name eval EP.
eval/zero : eval zero zero.
eval/succ : eval (succ E) (succ V)
             <- eval E V.
eval/lam : eval (lam E0) (lam E0).
eval/app : eval (app E1 E2) V
            <- eval E1 (lam E0)
            <- eval E2 V2
            <- eval (E0 V2) V.
eval/choice1 : eval (choice E1 E2) V
                <- eval E1 V.
eval/choice2 : eval (choice E1 E2) V
                <- eval E2 V.
eval/case0 : eval (case E0 E1 E2) V
              <- eval E0 zero
              <- eval E1 V.
eval/case1 : eval (case E0 E1 E2) V
              <- eval E0 (succ V0)
              <- eval (E2 V0) V.
% value judgment
value : exp -> type. %name value VP.
value/zero : value zero.
value/succ : value (succ E0)
            <- value E0.
value/lam : value (lam E0).
frame : (exp -> exp) -> type. %name frame FP.
frame/succ : frame succ.
frame/app1 : frame ([x] app V1 x)
              <- value V1.
frame/app2 : frame ([x] app x E2).
frame/case : frame ([x] case x E1 E2).
ctx : (exp -> exp) -> type. %name ctx RP.
ctx/id : ctx [x] x.
ctx/frame : ctx ([x] F(R(x)))
             <- frame F
             <- ctx R.
% Alternative value judgment, useful in assertion proofs
value* : exp -> type. %name value* VP.
% Extra rule
value*/num : value* E
           <- num N E.
```

```
value*/zero : value* zero.
value*/succ : value* (succ E0)
            <- value* E0.
value*/lam : value* (lam E0).

% Alternative evaluation judgment, useful in assertion proofs
eval* : exp -> exp -> type. %name eval* EP.

% Extra rule
eval*/val : eval* V V
             <- value* V.

eval*/zero : eval* zero zero.
eval*/succ : eval* (succ E) (succ V)
             <- eval* E V
             <- value* V.
eval*/lam : eval* (lam E0) (lam E0).
eval*/app : eval* (app E1 E2) V
            <- eval* E1 (lam E0)
            <- eval* E2 V2
            <- eval* (E0 V2) V
            <- value* V2
            <- value* V.
eval*/choice1 : eval* (choice E1 E2) V
                <- eval* E1 V
                <- value* V.
eval*/choice2 : eval* (choice E1 E2) V
                <- eval* E2 V
                <- value* V.
eval*/case0 : eval* (case E0 E1 E2) V
              <- eval* E0 zero
              <- eval* E1 V
              <- value* V.
eval*/case1 : eval* (case E0 E1 E2) V
              <- eval* E0 (succ V0)
              <- eval* (E2 V0) V
              <- value* V0
              <- value* V.

%block bexp : block {_:exp}.
%block bexp2 : block {_:exp -> exp}.

%worlds (bexp | bexp2) (exp).
```

C.3 sim.elf

```
sim : exp -> exp -> tp -> type.
sim/sym : sim E E' T -> sim E' E T.
sim/trans : sim E E' T
            -> sim E' E'' T
            -> sim E E'' T.
sim/diverge : sim diverge diverge T.
sim/zero : sim zero zero nat'.
sim/cong/succ :
   sim E E' nat'
   -> sim (succ E) (succ E') nat'.
sim/cong/lam :
   ({x} value x
      -> sim x x T2
      -> sim (E0 x) (E0' x) T0)
   -> sim (lam E0) (lam E0')
          (T2 => T0).
sim/cong/app :
   sim E2 E2' T2
   -> sim E1 E1' (T2 => T0)
   -> sim (app E1 E2) (app E1' E2') T0.
sim/cong/case :
   sim E0 E0' nat'
   -> sim E1 E1' T
   -> ({x} value x -> sim x x nat'
         -> sim (E2 x) (E2' x) T)
   -> sim (case E0 E1 E2)
          (case E0' E1' E2') T.
sim/cong/choice :
   sim E1 E1' T
   -> sim E2 E2' T
   -> sim (choice E1 E2)
          (choice E1' E2') T.
sim/cmerge :
   sim E1 E T
   -> sim E2 E T
   -> sim (choice E1 E2) E T.
sim/csym :
   sim E1 E1 T
   -> sim E2 E2 T
   -> sim (choice E1 E2)
          (choice E2 E1) T.
sim/cassoc :
   sim E1 E1 T
   -> sim E2 E2 T
   -> sim E3 E3 T
   -> sim (choice (choice E1 E2) E3)
          (choice E1 (choice E2 E3)) T.
sim/rchoice :
   ctx R
   -> sim E E T
   -> sim E' E' T
   -> ({x} value x -> sim x x T
          -> sim (R x) (R x) T')
   -> sim (R (choice E E'))
          (choice (R E) (R E')) T'.
sim/r : ctx R
         -> ctx R'
         -> sim E E' T'
         -> ({x} value x -> sim x x T'
               -> sim (R x) (R' x) T)
         -> sim (R E) (R' E') T.
sim/rdiverge :
   ctx R -> sim (R diverge) diverge T.
sim/rcase :
   ctx R
   -> sim E0 E0 nat'
   -> sim E1 E1 T'
   -> ({x} value x -> sim x x nat'
          -> sim (E2 x) (E2 x) T')
   -> ({x} value x -> sim x x T'
          -> sim (R x) (R x) T)
   -> sim (R (case E0 E1 E2))
          (case E0 (R E1)
          ([x] R (E2 x))) T.
sim/case1 :
   sim E1 E1 T
   -> sim (case zero E1 E2) E1 T.
sim/case2 :
   ({x} value x -> sim x x nat'
      -> sim (E2 x) (E2 x) T)
   -> sim E0 E0 nat'
   -> value E0
   -> sim (case (succ E0) E1 E2)
          (E2 E0) T.
sim/eta : sim E E (T2 => T0)
           -> value E
           -> sim E (lam [x] app E x)
                 (T2 => T0).
sim/beta :
   ({x} value x -> sim x x T2
      -> sim (E0 x) (E0 x) T0)
   -> sim E2 E2 T2
   -> value E2
   -> sim (app (lam E0) E2) (E0 E2) T0.
```