A Proof-theoretic Characterization of Independence in Type Theory

Yuting Wang[1] and Kaustuv Chaudhuri[2]

1 **University of Minnesota, USA**
`yuting@cs.umn.edu`
2 **Inria & LIX/École polytechnique, France**
`kaustuv.chaudhuri@inria.fr`

Abstract

For λ-terms constructed freely from a type signature in a type theory such as LF, there is a simple inductive *subordination* relation that is used to control type-formation. There is a related – but not precisely complementary – notion of *independence* that asserts that the inhabitants of the function space $\tau_1 \to \tau_2$ depend vacuously on their arguments. Independence has many practical reasoning applications in logical frameworks, such as pruning variable dependencies or transporting theorems and proofs between type signatures. However, independence is usually not given a formal interpretation. Instead, it is generally implemented in an *ad hoc* and uncertified fashion. We propose a formal definition of independence and give a proof-theoretic characterization of it by: (1) representing the inference rules of a given type theory and a closed type signature as a theory of intuitionistic predicate logic, (2) showing that typing derivations in this signature are adequately represented by a focused sequent calculus for this logic, and (3) defining independence in terms of *strengthening* for intuitionistic sequents. This scheme is then formalized in a meta-logic, called $\mathcal{G}$, that can represent the sequent calculus as an inductive definition, so the relevant strengthening lemmas can be given explicit inductive proofs. We present an algorithm for automatically deriving the strengthening lemmas and their proofs in $\mathcal{G}$.

1998 ACM Subject Classification F.4.2. Mathematical logic: proof theory

Keywords and phrases subordination, independence, sequent calculus, focusing, strengthening

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.332

1 Introduction

In logical frameworks such as LF [6] or $\mathcal{G}$ [4] that are designed to reason about typed λ-terms *qua* data, one notion that appears again and again is *dependency*: when can the structure of one group of λ-terms depend essentially on that of another group of λ-terms? The most widely studied general notion of dependency is *subordination* [15, 17, 7], which is best explained using an example. Consider λ-terms built out of the following *type signature* of constants, where `nat` and `bt` respectively denote natural numbers and binary trees with natural numbers in the leaves.

$$\texttt{z} : \texttt{nat}. \qquad \texttt{s} : \texttt{nat} \to \texttt{nat}. \qquad \texttt{leaf} : \texttt{nat} \to \texttt{bt}. \qquad \texttt{node} : \texttt{bt} \to \texttt{bt} \to \texttt{bt}.$$

From this signature, it is immediately evident that a closed β-normal term of type `bt` can – indeed, *must* – contain a subterm of type `nat`, so we say that `nat` is *subordinate* to `bt`. The subordination relation $\leq$ on types can be derived from a type signature as follows (adapting [7, Definition 2.14]). For any type τ, write $\mathcal{H}(\tau)$ for its *head*, which is the basic type that occurs rightmost in the chain of $\to$s (or dependent products in the case of dependent types) in

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 332–346

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

τ. Then, for every type $\tau_1 \to \cdots \to \tau_n \to a$ (here, a is the head) that occurs anywhere in the signature, set $\mathcal{H}(\tau_i) \leq a$ for every $i \in 1..n$. Finally, define $\tau_1 \leq \tau_2$ generally as $\mathcal{H}(\tau_1) \leq \mathcal{H}(\tau_2)$ and close $\leq$ under reflexivity and transitivity.

With this definition, and the above signature, we have that $\texttt{nat} \leq \texttt{bt}$ but $\texttt{bt} \not\leq \texttt{nat}$. This notion of subordination, when strictly enforced such as in canonical LF [17, 7], enables a kind of modularity of reasoning: inductive theorems about the λ-terms of a given type can be proved in the *smallest* relevant signature and imported into larger signatures that do not contain subordinate types. For instance, meta-theorems about `nat`s, proved in a context of `nat` assumptions, can be transported to contexts of `bt` assumptions since $\texttt{bt} \not\leq \texttt{nat}$. It is indeed this complement of subordination that is most useful in reasoning: intuitively, $\texttt{bt} \not\leq \texttt{nat}$ means that the inhabitants of $\texttt{bt} \to \texttt{nat}$ are functions whose arguments cannot occur in their bodies in the β-normal form. This negative reading of subordination can be used to prune dependencies during unification, which may bring an unsolvable higher-order problem into a solvable pattern problem [9], or to prevent *raising* a variable over non-subordinate arguments, producing more concise proofs [3].

However, this pleasingly simple notion of subordination has a somewhat obscure formal interpretation: the definition is independent of the typing rules and it is unclear how they are related. We set out to formalize such a relation in terms of an inductive characterization of the (β-normal) inhabitants of types, and in the process we discovered a curious aspect of the above definition of subordination that manifests for higher-order constructors. Take, for instance, the alternative type $(\texttt{nat} \to \texttt{bt}) \to \texttt{bt}$ for `leaf`. Nothing changes as far as the $\leq$ relation is concerned: $\texttt{nat} \to \texttt{bt}$ still occurs in the signature, so $\texttt{nat} \leq \texttt{bt}$ is still true.[1] Yet, if we look at all β-normal terms of type `bt` now, there can be no subterms of type `nat` since there does not exist a constructor for injecting them into terms of type `bt` in the base case. The definition of $\leq$ above clearly over-approximates possible dependencies, for reasons that are not at all obvious, so its complement is too conservative.

In this paper we propose an alternative view of dependency that is not based on the subordination relation. We directly characterize when one type is *independent* of another with a *proof-theoretic* view of independence: for every claimed independence, we establish, by means of an induction over all typing derivations based on a given signature, that indeed a certain dependency is not strict. This view has several benefits:

- First, our notion of independence is larger than non-subordination, which means that it can be used for more aggressive pruning of dependencies.
- More important is that we have strong confidence in the correctness of our definition of independence, since it is now a derived property of the type system. Indeed, we propose an algorithm that extracts formal inductive proofs of independence that can be verified without needing a built-in notion of subordination or independence. This changes independence from a trusted framework-level procedure to a certifying procedure.
- Finally, we use only standard and simple proof-theoretic machinery in our definition. We require neither rich type systems nor sophisticated meta-theoretic tools.

Our view of independence has the following outline, which is also the outline of the paper.

1. We start (Section 2) by defining independence as a property of a given type theory.
2. We then (Section 3) describe a *specification language* built around the logic of higher-order hereditary Harrop formulas (HH) [10]. This is a fragment of first-order intuitionistic logic

[1] In canonical LF [7, Definition 2.14], well-formedness of signatures requires subordination of argument types of all dependent products.

with a particularly simple focused proof system [16]. It can also be seen as the minimal logical core of every logical framework.

3. We then (Section 4) use the language to give an *adequate* encoding of the inference system that underlies a given type system. To keep the presentation simple, we have chosen to use the simply typed λ-calculus, but the technique generalizes at least to LF [13, 14]. In terms of this encoding, we characterize independence as a particular kind of *strengthening*.
4. We then (Section 5) take the focused HH sequent calculus as an object logic in the reasoning logic $\mathcal{G}$ that adds the remaining crucial ingredients: a *closed world* reading, induction, $\beta\eta$-equality of λ-terms, and generic reasoning. This is the *two-level logic approach* [5] that underlies the Abella theorem prover [18].
5. Lastly (Section 6) we show how strengthening for typing derivations is formalized in $\mathcal{G}$, and give an algorithm for automatically deriving these lemmas from a given signature. We show an application of the formalization to pruning unnecessary dependence between variables in $\mathcal{G}$.

The Abella development of examples in this paper is available at:

`http://abella-prover.org/independence`.

2 Independence in Type Theory

Intuitively, in a given type theory, τ_2 is independent of τ_1 if and only if the type $\tau_1 \to \tau_2$ is only inhabited by abstractions whose bodies in the β-normal form do not contain the arguments. We can write this as a property on typing derivations:

▶ **Definition 1** (Independence)**.** Let $\Gamma \vdash t : \tau$ be the typing judgment in the given type theory. The type τ_2 is *independent* of τ_1 in Γ if whenever $\Gamma, x{:}\tau_1 \vdash t : \tau_2$ holds for some t, the β-normal form of t does not contain x, *i.e.*, $\Gamma \vdash t : \tau_2$ holds. ◀

A straightforward way to prove such a property is to perform inductive reasoning on the first typing derivation. For this, we need to know not only what are the possible ways to prove a typing judgment, but also that they are the only ways. This is the *closed-world assumption* that underlies reasoning about a fixed type signature. However, even with this assumption, the inductive proofs for independence can be hard to establish: since the target type τ_2 is fixed, the inductive hypothesis will not be applicable to new types encountered during the induction. To see this, suppose we are working with the simply-typed λ-calculus (STLC). Typing rules for STLC derive judgments of the form $\Gamma \vdash t : \tau$ where Γ is a context that assigns unique types to distinct variables, t is a λ-term and τ is its type. The typing rules are standard:

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \texttt{t-bas} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau}\ \texttt{t-app} \qquad \frac{\Gamma, x{:}\tau_1 \vdash t : \tau \quad (x \notin \Gamma)}{\Gamma \vdash \lambda x{:}\tau_1.\, t : \tau_1 \to \tau}\ \texttt{t-abs}$$

A direct induction on $\Gamma \vdash t : \tau$ using the typing rules produces three cases. The case when t is an application results in two premises where the premise for the argument has a new target type τ_1 (as shown in `t-app`). Since τ_1 can be any arbitrary type that is not necessarily related to τ, it is not possible to appeal to the inductive hypothesis.

The key observation is that the signature must be fixed for the dependence between types to be fully characterized. We propose an approach to formalize independence by (1) giving an encoding of a given type theory and a closed signature in a specification logic which finitely characterizes the dependence between types, (2) proving that the encoding is faithful to the

$$\frac{\Sigma, \bar{x}{:}\bar{\tau}; \Gamma, F_1, \ldots, F_n \vdash A}{\Sigma; \Gamma \vdash \Pi\bar{x}{:}\bar{\tau}.\, F_1 \Rightarrow \cdots \Rightarrow F_n \Rightarrow A}\ \textsf{reduce} \qquad \frac{(F \in \Gamma) \quad \Sigma; \Gamma, [F] \vdash A}{\Sigma; \Gamma \vdash A}\ \textsf{focus}$$

$$\frac{\Sigma \vdash \theta : \bar{\tau} \quad A'[\theta] = A \quad \left\{\Sigma; \Gamma \vdash F_i[\theta]\right\} \text{ for } i \in 1..n}{\Sigma; \Gamma, [\Pi x_1{:}\tau_1, \ldots, x_n{:}\tau_n.\, F_1 \Rightarrow \ldots \Rightarrow F_n \Rightarrow A'] \vdash A}\ \textsf{backchain}$$

Figure 1 Inference rules for HH. For reduce, we assume that $\bar{x} \notin \Sigma$. In the backchain rule, θ stands for a *substitution* $[t_1/x_1, \ldots, t_n/x_n]$; we write $\Sigma \vdash \theta : \bar{\tau}$ to mean $\Sigma \vdash t_i : \tau_i$ for each $i \in 1..n$.

original type theory and signature, and (3) formally stating and proving the independence as lemmas in a reasoning logic that gives an inductive reading of the encoding. The first two tasks are covered by Section 3 and 4. The last task is the topic of Section 5 and 6. We use STLC as the example throughout the paper; extending to other type theories such as LF is left for future work.

3 The Specification Language

Let us begin with a sketch of a specification language for encoding rule-based systems. This language will be used to encode type theories in later sections.

3.1 The HH Proof System

The logic of *higher-order hereditary Harrop formulas* (HH) is an intuitionistic and predicative fragment of Church's simple theory of types. Expressions in HH are simply typed λ-terms. Types are freely formed from built-in or user-defined atomic types containing at least the built-in type o (of formulas) and the arrow type constructor $\to$ (right-associative). The *head type* of τ is the atomic type that occurs rightmost in a chain of $\to$s. Terms are constructed from a *signature* (written Σ); we write $\Sigma \vdash t : \tau$ if a λ-term t has type τ in Σ with the usual rules. Logic is built into HH by means of logical constants including $\Rightarrow : \mathsf{o} \to \mathsf{o} \to \mathsf{o}$ (written infix and right-associative) for implications and $\Pi_\tau : (\tau \to \mathsf{o}) \to \mathsf{o}$ for universal quantification over values of a type τ that does not contain o. Predicates are constants in the signature with head type o. For readability, we will often write $t_1 \Leftarrow t_2$ for $t_2 \Rightarrow t_1$, and abbreviate $\Pi_\tau(\lambda x{:}\tau.\, t)$ as $\Pi x{:}\tau.\, t$ and $\Pi x_1{:}\tau_1 \ldots \Pi x_n{:}\tau_n.\, t$ as $\Pi\bar{x}{:}\bar{\tau}.\, t$ where $\bar{x} = x_1, \ldots, x_n$ and $\bar{\tau} = \tau_1, \ldots, \tau_n$. We also omit type subscripts when they are obvious from context.

An *atomic formula* (written A) is a term of type o that head-normalizes to a term that is not an application of one of the logical constants $\Rightarrow$ or Π_τ. Every HH formula is equivalent to a *normalized formula* (written F) of the form $\Pi\bar{x}{:}\bar{\tau}.\, F_1 \Rightarrow \cdots \Rightarrow F_n \Rightarrow A$. In the rest of this paper we will assume that all formulas are normalized unless explicitly excepted. Given a normalized formula $F = \Pi\bar{x}{:}\bar{\tau}.\, F_1 \Rightarrow \cdots \Rightarrow F_n \Rightarrow A$, we write: $\mathcal{H}(F)$, the *head of* F, for A; $\mathcal{L}(F)$, the *body of* F, for the multiset $\{F_1, \ldots, F_n\}$; and $\mathcal{B}(F)$, the *binders of* F, for the typing context $\bar{x}{:}\bar{\tau}$.

The inference system for HH is a negatively polarized focused sequent calculus, also known as a uniform proof system, that contains two kinds of *sequents*: the *goal reduction* sequent $\Sigma; \Gamma \vdash F$ and the *backchaining* sequent $\Sigma; \Gamma, [F] \vdash A$ with F *under focus*. In either case, Γ is a multiset of formulas called *program clauses* and the right hand side formula of $\vdash$ is called the *goal formula*. Figure 1 contains the inference rules.

An HH proof usually starts (reading conclusion upwards) with a goal-reduction sequent $\Sigma; \Gamma \vdash F$. If F is not atomic, then the reduce rule is applied to make it atomic; this rule extends the signature with $\mathcal{B}(F)$ and the context with $\mathcal{L}(F)$. Then the focus rule is applied which selects a formula from the context Γ. This formula is then decomposed in the backchain

rule that produces fresh goal reduction sequents for (instances of the) body, assuming that the instantiated head of the focused formula matches the atomic goal formula. This process repeats unless the focused formula has no antecedents.

The three rules of HH can be combined together into one *synthetic* rule for goal reduction sequents that have an atomic goal formula.

$$\frac{F \in \Gamma \quad \Sigma \vdash \theta : \bar{\tau} \quad \mathcal{H}(F)[\theta] = A \quad \left\{\Sigma, \mathcal{B}(G[\theta]); \Gamma, \mathcal{L}(G[\theta]) \vdash \mathcal{H}(G[\theta])\right\} \text{ for } G \in \mathcal{L}(F)}{\Sigma; \Gamma \vdash A}\ \mathsf{bcred}$$

Every premise of this rule is a goal reduction sequent with an atomic goal formula. In the rest of this paper we will limit our attention to this fragment of HH.

3.2 Encoding Rule-based Systems in HH

Because the expressions of HH are λ-terms, we can use the *λ-tree approach to syntax* (λTS), sometimes known as *higher-order abstract syntax* (HOAS), to represent the rules of deductive systems involving binding using the binding structure of λ-terms. Binding in object-language syntax is represented explicitly by meta-language λ-abstraction, and recursion over such structures is realized by introducing fresh new constants using universal goals and recording auxiliary properties for such constants via hypothetical goals. This kind of encoding is concise and, as we shall see in later sections, has logical properties that we can use in reasoning.

We present the encoding of typing rules for STLC as described in Section 2 as a concrete example of specifying in HH. Two basic types, `ty` and `tm`, are used for classifying types and terms in STLC. We then use the following signature defining the type and term constructors:

$$\texttt{b} : \texttt{ty}. \quad \texttt{arr} : \texttt{ty} \to \texttt{ty} \to \texttt{ty}. \qquad \texttt{app} : \texttt{tm} \to \texttt{tm} \to \texttt{tm}. \quad \texttt{abs} : \texttt{ty} \to (\texttt{tm} \to \texttt{tm}) \to \texttt{tm}.$$

The type of the `abs` uses λTS to encode binding. To illustrate, $(\lambda y{:}b \to b.\, \lambda x{:}b.\, y\ x)$ is encoded as $\texttt{abs}\ (\texttt{arr}\ \texttt{b}\ \texttt{b})\ (\lambda y.\, \texttt{abs}\ \texttt{b}\ (\lambda x.\, \texttt{app}\ y\ x))$.

We define a predicate $\texttt{of} : \texttt{tm} \to \texttt{ty} \to \texttt{o}$ to encode the typing judgments in STLC, with the context implicitly represented by the context of HH sequents. The typing rules are encoded by the following program clauses (where the outermost universal quantifiers are omitted):

$$\texttt{of}\ (\texttt{app}\ M_1\ M_2)\ T \Leftarrow \texttt{of}\ M_1\ (\texttt{arr}\ T_1\ T) \Leftarrow \texttt{of}\ M_2\ T_1$$
$$\texttt{of}\ (\texttt{abs}\ T_1\ R)\ (\texttt{arr}\ T_1\ T) \Leftarrow (\Pi x.\, \texttt{of}\ (R\ x)\ T \Leftarrow \texttt{of}\ x\ T_1)$$

Here, we are following the usual logic programming convention of writing universally quantified variables using upper-case identifiers.

To see that these clauses accurately represent the typing rules of STLC, consider deriving a HH sequent $\Sigma; \Gamma \vdash \texttt{of}\ M\ T$, where Γ contains the clauses above and the possible assignments of types to variables introduced by the second clause corresponding to the abstraction rule `t-abs`. The only way to proceed is by focusing and backchaining on one of the clauses. Backchaining on the first clause unifies the goal with the head formula and produces two premises that corresponds to the premises of `t-app`. Backchaining on the second clause followed by goal reduction results in $\Sigma, x : \texttt{tm}; \Gamma, \texttt{of}\ x\ T_1 \vdash \texttt{of}\ (R\ x)\ T$. Note that x is a fresh variable introduced by reducing the universal goal and $\texttt{of}\ x\ T_1$ is the typing assignment of x from reducing the hypothetical goal, which exactly captures the side condition of `t-abs` that x must be fresh for Γ. The rule `t-bas` is modeled by backchaining on assumptions in Γ that assigns types to variables introduced by `t-abs`.

$\hat{\texttt{ty}}\ \texttt{b}.\qquad \hat{\texttt{ty}}\ (\texttt{arr}\ T_1\ T_2) \Leftarrow \hat{\texttt{ty}}\ T_1 \Leftarrow \hat{\texttt{ty}}\ T_2.\qquad \hat{\texttt{tm}}\ (\texttt{app}\ M_1\ M_2) \Leftarrow \hat{\texttt{tm}}\ M_1 \Leftarrow \hat{\texttt{tm}}\ M_2.$
$\hat{\texttt{tm}}\ (\texttt{abs}\ T\ R) \Leftarrow (\Pi x{:}\texttt{tm}.\, \hat{\texttt{tm}}\ (R\ x) \Leftarrow \hat{\texttt{tm}}\ x) \Leftarrow \hat{\texttt{ty}}\ T.$

Figure 2 An example encoding of an STLC signature into HH clauses

4 Independence via Strengthening

This section presents an encoding of STLC in HH that finitely captures the dependence between types in a closed signature. Then the independence property can be stated as strengthening lemmas and proved by induction.

4.1 An Adequate Encoding of STLC

The encoding is based on the *types-as-predicates* principle: every type is interpreted as a predicate that is true of its inhabitants. The atomic types and constants of STLC are imported directly into the HH signature. For every atomic type b, we define a predicate $\hat{b} : b \to \texttt{o}$. We then define a mapping $[\![-]\!]$ from STLC types τ to a function $\tau \to \texttt{o}$ as follows:

$$[\![b]\!] = \lambda t.\, \hat{b}\ t \quad \text{if } b \text{ is an atomic type.} \qquad [\![\tau_1 \to \tau_2]\!] = \lambda t.\, \Pi x{:}\tau_1.\, [\![\tau_1]\!]\ x \Rightarrow [\![\tau_2]\!]\ (t\ x)$$

The mapping $[\![-]\!]$ is extended to typing contexts: for $\Gamma = x_1{:}\tau_1, \ldots, x_n{:}\tau_n$, we write $[\![\Gamma]\!]$ for the multiset of HH formulas $\{[\![\tau_1]\!]\ x_1, \ldots, [\![\tau_n]\!]\ x_n\}$. A typing judgment $\Gamma \vdash t : \tau$ is encoded as an HH sequent $\Gamma; [\![\Gamma]\!] \vdash [\![\tau]\!]\ t$.

As an example, consider the signature with STLC as an object language as described in Section 3.2 (not to be confused with STLC we are encoding). We have two predicates $\hat{\texttt{tm}} : \texttt{tm} \to \texttt{o}$ and $\hat{\texttt{ty}} : \texttt{ty} \to \texttt{o}$. Let Γ be the STLC signature containing `b`, `arr`, `app` and `abs`. It is translated into $[\![\Gamma]\!]$ containing the program clauses in Figure 2 (in normalized form and with outermost quantifiers elided): The typing judgment $\Gamma, y : \texttt{tm} \to \texttt{tm} \vdash \texttt{abs b}\ y : \texttt{tm}$ which is provable in STLC is encoded as $\Gamma, y{:}\texttt{tm} \to \texttt{tm}; [\![\Gamma]\!], (\Pi x{:}\texttt{tm}.\, \hat{\texttt{tm}}\ x \Rightarrow \hat{\texttt{tm}}\ (y\ x)) \vdash \hat{\texttt{tm}}\ (\texttt{abs b}\ y)$ which is also provable in HH.

This encoding generalizes to richer type systems than STLC. An encoding of LF into HH was presented in [2, 13, 14], which is essentially a superset of the encoding we are doing. The soundness and completeness of the STLC encoding follows easily from the results in [13].

▶ **Theorem 2.** *Let Γ be a well-formed context and τ be a type in* STLC. *If $\Gamma \vdash t : \tau$ has a derivation for a $\beta\eta$-normal term t, then there is a derivation of $\Gamma; [\![\Gamma]\!] \vdash [\![\tau]\!]\ t$ in* HH. *Conversely, if $\Gamma; [\![\Gamma]\!] \vdash [\![\tau]\!]\ t$ for any term t that is well-typed in* HH, *then $\Gamma \vdash t' : \tau$ where t' is the canonical (β-normal η-long) form of t.*

Proof. This theorem is a special case of [13, Theorem 1]. The proof is almost the same. ◀

4.2 Independence as Strengthening Lemmas

By Definition 1 and Theorem 2, τ_2 is independent from τ_1 in Γ iff the following *strengthening* lemma is true: if $\Gamma, x{:}\tau_1; [\![\Gamma]\!], [\![\tau_1]\!]\ x \vdash [\![\tau_2]\!]\ t$ holds for some t, then the β-normal form of t does not contain x and $\Gamma; [\![\Gamma]\!] \vdash [\![\tau_2]\!]\ t$ holds. Because the typing information in formulas generated by $[\![-]\!]$ is statically determined by predicates and the translated program is finite, it is possible to determine the dependence between types finitely. Thus, the independence argument can be proved by induction.

Using the previous encoding example, let's see how to prove that `ty` is independent of `tm` in Γ which is the signature with STLC in Section 3.2. We need to show that $\Gamma, x{:}\texttt{tm} \vdash t : \texttt{ty}$

implies $\Gamma \vdash t : \mathtt{ty}$, which by Theorem 2 is equivalent to $\Gamma, x{:}\mathtt{tm}; [\![\Gamma]\!], \hat{\mathtt{tm}}\ x \vdash \hat{\mathtt{ty}}\ t$ implying $\Gamma; [\![\Gamma]\!] \vdash \hat{\mathtt{ty}}\ t$. The proof proceeds by induction on the first assumption. Apparently, $\hat{\mathtt{ty}}\ t$ can only be proved by backchaining on clauses whose heads start with $\hat{\mathtt{ty}}$. The case when $\hat{\mathtt{ty}}\ t$ is proved by backchaining on $\hat{\mathtt{ty}}\ \mathtt{b}$ is immediate. For the case when $t = \mathtt{arr}\ t_1\ t_2$, backchaining produces two premises $\Gamma, x{:}\mathtt{tm}; [\![\Gamma]\!], \hat{\mathtt{tm}}\ x \vdash \hat{\mathtt{ty}}\ t_i$ $(i \in \{1, 2\})$. Applying the inductive hypothesis we get $\Gamma; [\![\Gamma]\!] \vdash \hat{\mathtt{ty}}\ t_i$, from which the conclusion $\Gamma; [\![\Gamma]\!] \vdash \hat{\mathtt{ty}}\ (\mathtt{arr}\ t_1\ t_2)$ follows easily.

5 The Two-Level Logic Approach

5.1 The Reasoning Logic $\mathcal{G}$

When a relation is described as an inductive inference system, the rules are usually understood as fully characterizing the relation. When interpreting the relation as a computation, we can just give the inference rules a *positive* interpretation. However, when reasoning about the properties of the inference system, the inductive definition must be seen in a *negative* interpretation, *i.e.*, as a prescription of the *only* possible ways to establish the encoded property. Concretely, given the rules for typing λ-terms in STLC, we not only want to identify types with typable terms, but also to argue that a term such as $\lambda x.\, x\ x$ does not have a type. We sketch the logic $\mathcal{G}$ that supports this complete reading of rule-based specifications by means of *fixed-point definitions* [4].

To keep things simple, $\mathcal{G}$ uses the same term language as HH and is also based on Church's simple theory of types. At the type level, the only difference is that $\mathcal{G}$ formulas have type $\mathtt{prop}$ instead of $\mathtt{o}$. The non-atomic formulas of $\mathcal{G}$ include formulas of ordinary first-order intuitionistic logic, built using the constants $\top, \bot : \mathtt{prop}$, $\wedge, \vee, \supset$: $\mathtt{prop} \to \mathtt{prop} \to \mathtt{prop}$ (written infix), and $\forall_\tau, \exists_\tau : (\tau \to \mathtt{prop}) \to \mathtt{prop}$ for types τ that do not contain $\mathtt{prop}$. To this, $\mathcal{G}$ adds intensional equality at all types, $=_\tau$: $\tau \to \tau \to \mathtt{prop}$, which is interpreted as $\beta\eta$-convertibility. Like with HH earlier, we will drop the explicit types for these constants and write quantifiers using more familiar notation. The proof system of $\mathcal{G}$ is a sequent calculus with the standard derivation rules for logical constants [5], which we will not repeat here.

The next crucial ingredient in $\mathcal{G}$ is a mechanism for *defining* predicates by means of *fixed-point definitions*. Such a definition is characterized by a collection of *definitional clauses* each of which has form $\forall \bar{x}.A \triangleq B$ where A is an atomic formula all of whose free variables are bound by $\bar{x}$ and B is a formula whose free variables must occur in A. A is called the head of such a clause and B is called its body. (For a fixed-point definition to be well-formed, it must satisfy certain stratification conditions [8] that we do not elaborate on here.) To illustrate definitions, let $\mathtt{olist}$ be a new basic type for lists of HH formulas, built from the type signature $\mathtt{nil} : \mathtt{olist}$ and $(\mathtt{::}) : \mathtt{o} \to \mathtt{olist} \to \mathtt{olist}$. Then, list membership ($\mathtt{member}$) and list concatenation ($\mathtt{append}$) may be specified in $\mathcal{G}$ using the following definitional clauses:

$$\mathtt{member}\ X\ (X :: L) \triangleq \top. \qquad \mathtt{member}\ X\ (Y :: L) \triangleq \mathtt{member}\ X\ L.$$

$$\mathtt{append}\ \mathtt{nil}\ L\ L \triangleq \top. \qquad \mathtt{append}\ (X :: L_1)\ L_2\ (X :: L_3) \triangleq \mathtt{append}\ L_1\ L_2\ L_3.$$

As before, we use the convention of indicating universally closed variables with upper-case identifiers. Read positively, these clauses can be used in the standard backchaining style to derive atomic formulas: the goal must match with the head of a clause and the backchaining step reduces the goal to deriving the instances of corresponding body. Read negatively, they are used to do case analysis on an assumption: if an atomic formula holds, then it must be the case that it unifies with the head of some clause defining it and the body of the clause is derivable. It therefore suffices to show that the conclusion follows from each such possibility.

Fixed-point definitions can also be interpreted inductively or coinductively, leading to corresponding reasoning principles. We use an annotated style of reasoning to illustrate how induction works in $\mathcal{G}$. Its interpretation into the proof theory of $\mathcal{G}$ is described in [3]. When proving a formula $(\forall \bar{x}.\, D_1 \supset \cdots \supset A \supset \cdots \supset D_n \supset G)$ by induction on the atom A, the proof reduces to proving G with the inductive hypothesis $(\forall \bar{x}.\, D_1 \supset \cdots \supset A^* \supset \cdots \supset D_n \supset G)$ and assumptions $D_1, \ldots, A^@, \ldots, D_n$. Here, A^* and $A^@$ are simply annotated versions of A standing for *strictly smaller* and *equal sized* measures respectively. When $A^@$ is *unfolded* by using a definitional clause, the predicates in the body of the corresponding clause are given the * annotation. Thus, the inductive hypothesis can only be used on A^* that results from unfolding $A^@$ at least once.

The negative reading of fixed-point definitions in $\mathcal{G}$ requires some care. In the negative view, universally quantified variables are interpreted *extensionally*, *i.e.*, as standing for all their possible instances. To illustrate, if `nat` were defined by $\texttt{nat z} \triangleq \top$ and $\texttt{nat (s } X\texttt{)} \triangleq \texttt{nat } X$, then one can derive $\forall x.\, \texttt{nat } x \supset G$ by case-analysis of the assumption $\texttt{nat } x$, which amounts to proving $[\texttt{z}/x]G$ and $\forall y.\, \texttt{nat } y \supset [\texttt{s } y/x]G$. This extensional view of universal variables is not appropriate when reasoning about binding structures viewed as syntax, where the syntactic variables are not stand-ins for all terms but rather for *names*. To see this clearly, consider the formula $\forall w.\, (\lambda x.\, w) = (\lambda x.\, x) \supset \bot$. If equality of λ-terms were interpreted extensionally with $\forall$, we would be left with $\forall w.\, (\forall x.\, w = x) \supset \bot$ which is not provable. Yet, the λ-terms $(\lambda x.\, w)$ and $(\lambda x.\, x)$ denote the constant and identity functions, respectively, and are therefore intensionally different – neither is $\beta\eta$-convertible to the other.

To achieve this intensional view, we come to the final ingredient of $\mathcal{G}$: *generic reasoning*. Every type of $\mathcal{G}$ is endowed with an infinite number of *nominal constants*, and there is a quantifier $\nabla_\tau : (\tau \to \texttt{prop}) \to \texttt{prop}$ (for τ not containing `prop`) to abstract over such constants. The critical features of nominal constants are: (1) they are not $\beta\eta$-convertible, so $\nabla x.\, \nabla y.\, x = y \supset \bot$ is derivable; and (2) formulas on the left and right of a $\mathcal{G}$ sequent interact up to *equivariance*, which allows the nominal constants in one formula to be systematically permuted to match those of the other. This latter property allows the ∇ quantifier to permute with all other connectives, including disjunction. The rules for introducing ∇ quantified formulas both as assumption and conclusion are similar: a formula $\nabla x.A$ is reduced to $[c/x]A$ where c is a nominal constant that is not already present in A.

The general form of a $\mathcal{G}$ definitional clause therefore allows the ∇ quantifier in heads: $\forall \bar{x}.\, (\nabla \bar{z}.\, A) \triangleq B$. An instance of the clause must have $\bar{z}$ replaced by distinct nominal constants that are not already present in the formula. Since $\bar{x}$ is quantified outside of $\bar{z}$, their instantiations cannot contain $\bar{z}$, which is used to encode structural properties of terms in the definitions. For example, $(\nabla x.\, \texttt{name } x) \triangleq \top$ defines `name` such that $\texttt{name } X$ holds only if X is a nominal constant. Another example is $\forall T.\, (\nabla x.\, \texttt{fresh } x\ T) \triangleq \top$ which defines `fresh` such that $\texttt{fresh } X\ T$ holds only if X is a nominal constant that does not occur in T.

5.2 An Encoding of HH in $\mathcal{G}$

The HH proof system can be encoded as a fixed-point definition in $\mathcal{G}$. The logical constants, signatures, and terms are imported into $\mathcal{G}$ transparently from HH – this is possible since $\mathcal{G}$ and HH use the same simple type system. The sequents of HH are then encoded in $\mathcal{G}$ using the predicates $\texttt{seq} : \texttt{olist} \to \texttt{o} \to \texttt{prop}$ and $\texttt{bch} : \texttt{olist} \to \texttt{o} \to \texttt{o} \to \texttt{prop}$. The encodings and their abbreviated notations are as follows:

HH	$\mathcal{G}$	notation
$\Sigma; \Gamma \vdash F$	$\texttt{seq } \Gamma\ F$	$\{\Gamma \vdash F\}$
$\Sigma; \Gamma, [F] \vdash A$	$\texttt{bch } \Gamma\ F\ A$	$\{\Gamma, [F] \vdash A\}$

$$\texttt{seq}\ L\ (F \Rightarrow G) \triangleq \texttt{seq}\ (F :: L)\ G. \qquad \texttt{bch}\ L\ (G \Rightarrow F)\ A \triangleq \texttt{seq}\ L\ G \wedge \texttt{bch}\ L\ F\ A.$$
$$\texttt{seq}\ L\ (\Pi x{:}\tau.\ F\ x) \triangleq \nabla x{:}\tau.\ \texttt{seq}\ L\ (F\ x). \qquad \texttt{bch}\ L\ (\Pi x{:}\tau.\ F\ x)\ A \triangleq \exists t{:}\tau.\ \texttt{bch}\ L\ (F\ t)\ A.$$
$$\texttt{seq}\ L\ A \triangleq \texttt{atom}\ A \wedge \texttt{member}\ F\ L \wedge \texttt{bch}\ L\ F\ A. \qquad \texttt{bch}\ L\ A\ A \triangleq \top.$$

Figure 3 Encoding of HH rules as inductive definitions in $\mathcal{G}$.

The definitional clause for `seq` and `bch` are listed in Figure 3. Note that we use a list of formulas to represent the multiset context in HH. This is not a problem since we can prove that the ordering of clauses in the list in `seq` and `bch` formulas does not affect their derivability as theorems about `seq` and `bch` in $\mathcal{G}$.

The encoding of HH in $\mathcal{G}$ is adequate. The first two clauses defining `seq` exactly capture the reduce rule. Note that in the second clause we use ∇ to encode Π in HH, since Π introduces fresh and unanalyzable eigenvariables into the HH signature. The third clause encodes the focus rule, where `atom` is defined as

$$\texttt{atom}\ F \triangleq \big(\forall G.\ (F = \Pi x{:}\tau.\ G\ x) \supset \bot\big) \wedge \big(\forall G_1, G_2.\ (F = (G_1 \Rightarrow G_2)) \supset \bot\big).$$

The clauses defining `bch` exactly capture the backchain rule. Note that the two rules for introducing Π actually represents a collection of rules for each instance of τ. In other words, these rule are actually schematic rules. This is possible since the proof theory allows inductive definitions to have infinitely many clauses. We will often write elements in `olist` in reverse order separated by commas; the exact meaning of the comma depends on its context. For example, given $L_1, L_2 : \texttt{olist}$ and $A : \texttt{o}$, $\{L_1, L_2, A \vdash G\}$ stands for $\texttt{seq}\ (A :: L)\ G$ for some $L : \texttt{olist}$ such that $\texttt{append}\ L_1\ L_2\ L$ holds.

Theorems of HH specifications can be proved through this encoding in $\mathcal{G}$. As an example, consider proving the uniqueness of the encoding of typing in STLC shown in Section 3.2. The theorem can be stated as follows:

$$\forall L, M, T, T'.\ \{\Gamma, L \vdash \texttt{of}\ M\ T\} \supset \{\Gamma, L \vdash \texttt{of}\ M\ T'\} \supset T = T'.$$

where Γ represents the program clauses defining `of` in Section 3.2.[2]

6 Formalizing Independence

This section first describes the formalization of independence in terms of *strengthening* for HH sequents of a certain shape in $\mathcal{G}$. Then a general algorithm for automatically deriving such strengthening lemmas is presented. Lastly, an application of the formalization for pruning variable dependencies in $\mathcal{G}$ is described.

6.1 Independence as Strengthening Lemmas in $\mathcal{G}$

A strengthening lemma has the following form in $\mathcal{G}$: $\forall \bar{x}.\ \{\Gamma, F \vdash G\} \supset \{\Gamma \vdash G\}$. As we have discussed in Section 3, an HH derivation always starts with applying reduce to turn the right hand side to atomic form. Thus, it suffices to consider instances of strengthening lemmas where G is atomic. This lemma is usually proved by induction on the only assumption. The proof proceeds by inductively checking that a derivation of $\{\Gamma', F \vdash A\}$ cannot contain any application of focus rule on F. In Section 3 we have shown that an HH derivation starting

[2] This is not precisely correct, since the typing context L also needs to be characterized by some inductive property; a complete exposition on this encoding can be found in the Abella tutorial [1].

with an atomic goal can be seen as repeatedly applying bcred. At every point in such a proof, we check whether the atomic goal A matches the head of F. If such a match never occurs, we can drop the assumption F from every HH sequent.

We formalize this intuition to establish independence for STLC in terms of a kind of strengthening lemma in $\mathcal{G}$. Concretely, by the adequacy of the encoding of STLC in HH and the adequacy of the encoding of HH in $\mathcal{G}$, τ_2 is independent of τ_1 in Γ if the following strengthening lemma is provable in $\mathcal{G}$:

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], [\![\tau_1]\!]\ x \vdash [\![\tau_2]\!]\ (t\ x)\} \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash [\![\tau_2]\!]\ t'\}.$$

Note that the variables in the context Γ become nominal constants of appropriate types which are absorbed into formulas. The term $(t\ x)$ indicates the *possible* dependence of t on x. The conclusion $t = (\lambda y.\, t')$ asserts that t is $\beta\eta$-convertible to a term with a vacuous abstraction; this is indicated by the fact that t' is bound outside the scope of λ. To prove any instance of this lemma for particular types τ_1 and τ_2, we proceed by induction on the only assumption. The conclusion $\exists t'.\, t = (\lambda y.\, t')$ is immediately satisfied when the atomic goals in the derivation do not match $[\![\tau_1]\!]\ x$, since this is the only hypothesis where x occurs.

As an example, `ty` is independent of `tm` in $[\![\Gamma]\!]$ which contains the clauses in Figure 2 is formalized as the following lemma:

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t\ x)\} \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t'\}$$

By induction on the assumption and introducing the goals, we get an inductive hypothesis

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t\ x)\}^* \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t'\}$$

and a new hypothesis $\{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t\ x)\}^{@}$. Unfolding this hypothesis amounts to analyzing all possible ways to derive this using the definitional clauses of Figure 3. Since $\hat{\texttt{ty}}\ (t\ x)$ cannot match $\hat{\texttt{tm}}\ x$, the selected focus cannot be $\hat{\texttt{tm}}\ x$, so the only options are clauses selected from $[\![\Gamma]\!]$, of which only two clauses have heads compatible with $\hat{\texttt{ty}}\ (t\ x)$. In the first case, for the clause $\hat{\texttt{ty}}\ \texttt{b}$, we unite $t\ x$ with $\texttt{b}$, instantiating the eigenvariable t with $\lambda y.\, \texttt{b}$; this in turn gives us the witness for t' to finish the proof. In the other case, $t\ x$ is united with $\texttt{arr}\ t_1\ t_2$ (for fresh eigenvariables t_1 and t_2). Here, t_1 and t_2 are first *raised* over x to make the two terms have the same nominal constants, and bcred then reduces the hypothesis to the pair of hypotheses, $\{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t_1\ x)\}^*$ and $\{[\![\Gamma]\!], \hat{\texttt{tm}}\ x \vdash \hat{\texttt{ty}}\ (t_2\ x)\}^*$. The inductive hypothesis applies to both of these, so we conclude that $\{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t_1'\}^*$ and $\{[\![\Gamma]\!] \vdash \hat{\texttt{ty}}\ t_2'\}^*$ for suitable t_1' and t_2' that are independent of x. We can then finish the proof by forward-chaining on the definitional clauses for HH. Observe that this proof follows the informal proof described in Section 4.2

As another example, let's see why `tm` is not independent of `ty`, since abstractions contain their argument types. The relevant lemma would be:

$$\forall t.\, \nabla x.\, \{[\![\Gamma]\!], \hat{\texttt{ty}}\ x \vdash \hat{\texttt{tm}}\ (t\ x)\} \supset \exists t'.\, t = (\lambda y.\, t') \wedge \{[\![\Gamma]\!] \vdash \hat{\texttt{tm}}\ t'\}.$$

Here, a direct induction on the assumption would not work because we can now focus on the fourth clause of Figure 2 that would extend the context of the HH sequent with a fresh assumption of the form $\hat{\texttt{tm}}\ x'$. The inductive hypothesis is prevented from being used because the context has *grown*. This is a standard feature of HH derivations, and we therefore need to give an inductive characterization of the structure of the *dynamically growing* context. We use the technique of defining these dynamic extensions in terms of *context definitions*; for example, for the $\hat{\texttt{tm}}$ predicate, this definition has the following clauses:

$$\texttt{ctx}\ \texttt{nil} \triangleq \top; \qquad (\nabla x.\, \texttt{ctx}\ (\hat{\texttt{tm}}\ x :: L)) \triangleq \texttt{ctx}\ L.$$

Then we generalize the lemmas as follows:

$$\forall t, L. \nabla x. \texttt{ctx}\ L \supset \{[\![\Gamma]\!], L, \hat{\texttt{ty}}\ x \vdash \hat{\texttt{tm}}\ (t\ x)\} \supset \exists t'. t = (\lambda y. t') \wedge \{[\![\Gamma]\!], L \vdash \hat{\texttt{tm}}\ t'\}.$$

Now, by induction, when the fourth clause of Figure 2 is selected for focus, one of the hypotheses of the resulting bcred is $\{[\![\Gamma]\!], L, \hat{\texttt{ty}}\ x \vdash \hat{\texttt{ty}}\ (t_1\ x)\}$ (for a new eigenvariable t_1). It is entirely possible that $\hat{\texttt{ty}}\ (t_1\ x)$ is proved by selecting $\hat{\texttt{ty}}\ x$. Thus we cannot conclude the original assumption does not depend on $\hat{\texttt{ty}}\ x$ – so tm is not independent of ty.

6.2 Automatically Deriving Strengthening Lemmas

The above illustrations show that these strengthening lemmas have a predictable form and proof structure. We will now give an algorithm that extracts these proofs automatically. The key insight is that the strengthening lemmas can be provable because of a *failure* to match the heads of the encoded clauses against the right hand sides of the HH sequent. Therefore, we simply need to enumerate the possible forms of the right hand sides and generate a mutually inductive lemma to cover all possibilities. This intuition is not entirely trivial to implement, since the HH contexts can potentially grow on every bcred step, which must then be accounted for.

For $F = (\Pi\bar{x}. F_1 \Rightarrow \ldots \Rightarrow F_n \Rightarrow A)$, let $\mathcal{H}_p(F)$ stand for the head predicate in A and $\mathcal{L}_p(F)$ for $\{\mathcal{H}_p(F_i) \mid 1 \in i..n\}$. To prove $\forall\bar{x}. \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$, we proceed as follows:

1. For every predicate a in Γ, we compute the possible dynamic contexts that arise in proofs of atomic formulas of head a.
2. For every predicate a in Γ and A, we compute a collection of predicates $S(a)$ which contains the head predicates of atomic formulas that may occur as the goal in a derivation starting with an atomic goal formula A' for which $\mathcal{H}_p(A') = a$. That is, the provability of any goal A' of head a only depends on formulas whose heads are in $S(a)$.
3. If $\mathcal{H}_p(A) = a$ and $\mathcal{H}_p(F) \notin S(a)$, then for every predicate $a' \in S(a)$ and any atomic goal A' *s.t.* $\mathcal{H}_p(A') = a'$, $\forall\bar{x}. \{\Gamma, F \vdash A'\} \supset \{\Gamma \vdash A'\}$ is provable. The proof proceeds by a simultaneously induction on all these formulas.
4. Since $a \in S(a)$, the required lemma is just one of the cases of the simultaneous induction.

Before we elaborate on these steps in the following subsections, note that our algorithm is sound and terminating, but not complete. The existence of a decision procedure for strengthening lemmas is outside the scope of this paper.

6.2.1 Calculating the dynamic contexts

Let Γ be a context that contains only finite distinct clauses, A be an atomic goal and F be some program clause. We would like to prove the strengthening lemma $\forall\bar{x}. \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$ in $\mathcal{G}$. Let Δ be the set of predicates occurring in Γ. For every predicate $a \in \Delta$, let $C(a)$ denote a set of formulas that can possibly occur in the dynamic contexts of atomic formulas of head a. The only way formulas can be introduced into dynamic contexts is by applying bcred. Given a program clause $\Pi\bar{x}. G_1 \Rightarrow \cdots \Rightarrow G_n \Rightarrow A$, the dynamic context of $\mathcal{H}(G_i)$ is obtained by extending the context of A with program clauses in $\mathcal{L}(G_i)$ for $1 \leq i \leq n$. Algorithm 1 derives a set of constraints $\mathcal{C}$ on C based on this observation. It traverses all the program clauses and their sub-program clauses in formulas in Γ and collects a set $\mathcal{C}$ of constraints for dynamic contexts which must be satisfied by derivations starting with Γ as the context.

To compute a set of dynamic contexts that satisfies $\mathcal{C}$, we start with $C(a) = \emptyset$ for all $a \in \Delta$ and iteratively apply the constraint equations until the constraint is satisfied. It is

Algorithm 1 Collecting constraints on dynamic contexts

Let Γ' be a finite set equal to Γ and $\mathcal{C} \leftarrow \emptyset$
while $\Gamma' \neq \emptyset$ **do**
  pick some $D = (\Pi \bar{x}.\, G_1 \Rightarrow \cdots \Rightarrow G_n \Rightarrow A)$ from Γ'
  add equations $\{C(\mathcal{H}_p(G_i)) = C(\mathcal{H}_p(G_i)) \cup C(\mathcal{H}_p(A)) \cup \mathcal{L}(G_i) \mid i \in 1..n\}$ to $\mathcal{C}$
  remove D from Γ' and add clauses in $\bigcup_{i \in 1..n} \mathcal{L}(G_i)$ to Γ'
end while

Algorithm 2 Collecting constraints on the dependency relation

let Γ' be a finite set equal to Γ and $\mathcal{S} \leftarrow \emptyset$
for all $a \in \Delta$ **do**
  for all $D \in \Gamma' \cup C(a)$ where $D = (\Pi \bar{x}.\, G_1 \Rightarrow \ldots \Rightarrow G_n \Rightarrow A)$ and $\mathcal{H}_p(A) = a$ **do**
    add $(S(a) = S(a) \cup \bigcup_{i \in 1..n} S(\mathcal{H}_p(G_i)))$ to $\mathcal{S}$
  end for
end for

easy to see that this algorithm terminates: since the iterations never shrink $C(a)$ for $a \in \Delta$, the only way the algorithm goes on forever is to keep adding new clauses in every iteration. This is impossible since there are only finitely many distinct program clauses. In the end, we get a finite set of dynamic clauses $C(a)$ for every $a \in \Delta$ that satisfies constraints in $\mathcal{C}$. Suppose $C(a) = \{D_1, \ldots, D_n\}$ for $a \in \Delta$. We define the context relation $\mathtt{ctx}_a$ as follows:

$$\mathtt{ctx}_a\ \mathtt{nil} \triangleq \top; \qquad \mathtt{ctx}_a\ (D_1\ \mathtt{::}\ L) \triangleq \mathtt{ctx}_a\ L; \qquad \cdots \qquad \mathtt{ctx}_a\ (D_n\ \mathtt{::}\ L) \triangleq \mathtt{ctx}_a\ L.$$

6.2.2 Generating the dependency relation between predicates

By the bcred rule, given a program clause $\Pi \bar{x}.\, G_1 \Rightarrow \ldots \Rightarrow G_n \Rightarrow A$, the provability of A will depend on the provability of $\mathcal{H}(G_i)$ for $1 \in 1..n$. For any $a \in \Delta \cup \{\mathcal{H}_p(A)\}$, a can proved by backchaining on either some clause in Γ or in the dynamic contexts $C(a)$. Since both Γ and $C(a)$ are known and finite, we can derive a set $S(a)$ containing predicates that a depends on.

The steps for computing S are similar to that for C. First, we derive a set of constraints $\mathcal{S}$, by Algorithm 2. To generate the dependency relations, we start with $S(a) = \{a\}$ for all $a \in \Delta \cup \{\mathcal{H}_p(A)\}$ and iteratively apply the constraint equations until the constraints in $\mathcal{S}$ are satisfied. The algorithm terminates by an analysis similar to the previous one.

6.2.3 Constructing proofs for the strengthening lemmas

Now we are in a position to prove the strengthening theorem $\forall \bar{x}.\, \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$ in $\mathcal{G}$. Since the proof of A may depend on formulas with different heads, we generalize the strengthening lemma to take into account of related predicates.

▶ **Theorem 3.** *Let $S(\mathcal{H}_p(A)) = \{a_1, \ldots, a_n\}$, The generalized strengthening lemma is*

$$(\forall \Gamma', \bar{x}_1.\, \mathtt{ctx}_{a_1} \Gamma' \supset \{\Gamma, \Gamma', F \vdash a_1\ \bar{x}_1\} \supset \{\Gamma, \Gamma' \vdash a_1\ \bar{x}_1\}) \wedge \cdots \wedge$$
$$(\forall \Gamma', \bar{x}_n.\, \mathtt{ctx}_{a_n} \Gamma' \supset \{\Gamma, \Gamma', F \vdash a_n\ \bar{x}_n\} \supset \{\Gamma, \Gamma' \vdash a_n\ \bar{x}_n\})$$

If $\mathcal{H}_p(F) \notin S(\mathcal{H}_p(A))$, then this lemma has a proof in $\mathcal{G}$.

Proof. By simultaneous induction on $\{\Gamma, \Gamma', F \vdash a_i\ \bar{x}_i\}$ for $1 \leq i \leq n$. ◀

The original strengthening lemma is an immediate corollary of Theorem 3:

▶ **Corollary 4.** *If $\mathcal{H}_p(F) \notin S(\mathcal{H}_p(A))$, then $\forall \bar{x}.\, \{\Gamma, F \vdash A\} \supset \{\Gamma \vdash A\}$ has a proof in $\mathcal{G}$.* ◀

Since the proofs for Theorem 3 is constructive, we obtain a certifying algorithm to state and prove strengthening lemmas in $\mathcal{G}$ on demand.

6.3 Application

This section describes an application of the formalization of independence: to prune unnecessary dependence on variables in $\mathcal{G}$. When analyzing the terms containing nominal constants in $\mathcal{G}$, it is often necessary to introduce variable dependencies on such constants. As an example, the $\forall$ introduction rule creates a new eigenvariable that is raised over the nominal constants in the principal formula. However, if we can show that the type of the eigenvariable is independent from the types of the nominal constants, then we can suppress this dependency. In Abella, the theorem prover based on $\mathcal{G}$, an ad-hoc algorithm based on checking of subordination relations is used to prune such dependencies, but the exact logical basis of the algorithm has never been adequately formulated.

Now that we can derive independence lemmas, we can recast this pruning of nominal constants in terms of the closed-world reading of types. To give it a formal treatment in the logic, we reflect the closed-world reading of types into the proofs. This can be done by encoding the type theory of HH (which is STLC) into an HH specification as described in Section 4 and then to use the derived strengthening lemmas directly.

As an example, suppose we want to prove the following theorem

$$\forall X, T.\, \mathtt{name}\ X \supset \{\mathtt{of}\ X\ T\} \supset \cdots$$

where X has type $\mathtt{tm}$ and T has type $\mathtt{ty}$. By introducing the assumptions and case analysis on $\mathtt{name}\ X$, X will be unified with a nominal constant n and the dependence of T on n will be introduced, resulting in the hypothesis $\{\mathtt{of}\ n\ (T\ n)\}$. This dependence of T on n is vacuous as we have already seen – types cannot depend on terms in STLC. However, to formally establish the independence, we require T to be a well-formed type, so we need to change the theorem to

$$\forall X, T, L.\, \mathtt{ctx}\ L \supset \{\Gamma, L \vdash \hat{\mathtt{ty}}\ T\} \supset \mathtt{name}\ X \supset \{\mathtt{of}\ X\ T\} \supset \cdots$$

where Γ contains the program clauses described in Figure 2. For the purpose of demonstration, we assume the context definition $\mathtt{ctx}$ contains infinitely many nominal constants of type $\hat{\mathtt{tm}}$, which is defined as follows:

$$\mathtt{ctx}\ \mathtt{nil} \triangleq \top; \qquad (\nabla x.\, \mathtt{ctx}\ (\hat{\mathtt{tm}}\ x :: L)) \triangleq \mathtt{ctx}\ L.$$

We perform the same introduction and case analysis on $\mathtt{name}\ X$ and get hypotheses $\mathtt{ctx}\ (L\ n)$, $\{\Gamma, L\ n \vdash \hat{\mathtt{ty}}\ (T\ n)\}$ and $\{\mathtt{of}\ n\ (T\ n)\}$. By the definition of $\mathtt{ctx}$, when treated as a multiset, $L\ n$ must be equivalent to $(\hat{\mathtt{tm}}\ n :: L')$ where L' does not contain n. Thus $\{\Gamma, L', \hat{\mathtt{tm}}\ n \vdash \hat{\mathtt{ty}}\ (T\ n)\}$ holds. At this point, we can use the algorithm in Section 6.2 to derive and prove the following strengthening lemma

$$\forall T.\, \nabla x.\, \{\Gamma, L', \hat{\mathtt{tm}}\ x \vdash \hat{\mathtt{ty}}\ (T\ x)\} \supset \exists T'.\, T = (\lambda y.\, T') \wedge \{\Gamma, L' \vdash \hat{\mathtt{ty}}\ T'\}.$$

By applying it to $\{\Gamma, L', \hat{\mathtt{tm}}\ n \vdash \hat{\mathtt{ty}}\ (T\ n)\}$, we get $T = \lambda y.\, T'$ for some T' not containing y. Hence $\{\mathtt{of}\ n\ (T\ n)\}$ becomes $\{\mathtt{of}\ n\ T'\}$. Note that we choose a particular definition of $\mathtt{ctx}$ for demonstration; the exact context of well-formed terms can vary in practice, based on the signature. Nevertheless, pruning of nominal constants can always be expressed by proving and applying the strengthening lemmas derived from independence.

7 Related Work

The earliest formulation of dependency that we were able to find is by Miller [9, Definition 11.3], where it is defined in terms of derivability of a certain fragment of strict logic for use in a unification procedure. Although we do not have a proof of this, this notion appears to coincide with the negation of our notion of independence. It is unclear why this definition was never adopted in subsequent work on logical frameworks, but we can speculate that one reason is its inherent cost, since it requires proving an arbitrary theorem of relevant logic. Indeed, using independence in the core unification engine is probably inadvisable if the unification engine is to be trusted.

It is more popular to define subordination relations independently of proof-theory, as has been done for many variants of LF. In [15], Virga proposed a *dependence relation* between types and type families in LF to constraint higher-order rewritings to well-behaved expressions. Later, this relation was popularized as *subordination* and used in the type theory of canonical LF to show that canonical terms of one type τ is not affected by introduction of terms of another type that is not subordinate to τ [17, 7]. The subordination relations in these cases are defined to be strong enough so that the type theory only deals with well-formed instances.

For reasoning applications, in order to move from one context to another, it is often necessary to check if a term of type τ_1 can occur in the normal form of another type τ_2. Traditionally, the complement of subordination has been thought to be the right interpretation of independence. However, it is unclear how exactly it can be translated into evidence in the theory that supports the reasoning. Thus, *ad hoc* algorithms have been developed in systems like Twelf [11], Beluga [12] and Abella [18], which all lack formal definitions.

8 Conclusion and Future Work

We proposed a proof-theoretic characterization of independence in a two-level logic framework, and gave an example of such characterization by encoding the type theory of STLC in the logic HH and interpreting the independence relation as strengthening lemmas in a reasoning logic $\mathcal{G}$. We developed an algorithm to automatically establish the independence relation and strengthening lemmas and showed its application to pruning variable dependence in $\mathcal{G}$.

Interpreting independence as strengthening should be realizable in other logical frameworks that support inductive reasoning. We chose the two-level logic framework because it provides a first-class treatment of contexts, which makes proofs of strengthening lemmas easy. It would be worth investigating a similar formal development in logical frameworks such as Twelf and Beluga where contexts are either implicit or built into the type system.

The characterization of independence can be extended to more sophisticated type theories. Recently, it was shown that the encoding of LF in $\mathcal{G}$ [13] can be used transparently to perform inductive reasoning over LF specifications [14]. We plan to develop a characterization of independence of LF based on this approach, which will formalize the important concept of *world subsumption* for migrating LF meta-theorems between different LF context schema.

Finally, one benefit of a logical characterization that is almost too obvious to state is that it opens up independence to both external validation and user-guidance. In LF, where inhabitation is undecidable, the notion of independence proposed in this paper will generally not be automatically derivable. Presenting the user with unsolved independence obligations may be an interesting interaction mode worth investigating.

Acknowledgements. This work has been partially supported by the NSF Grant CCF-0917140 and the ERC grant ProofCert. The first author has been partially supported by the

Doctoral Dissertation Fellowship from the University of Minnesota, Twin Cities. Opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the University of Minnesota, or Inria.

References

1 David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

2 Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In *Conference on Automated Deduction*, volume 449 of *LNAI*, pages 221–235. Springer, 1990.

3 Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.

4 Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

5 Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.

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

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

8 Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

9 Dale Miller. Unification under a mixed prefix. *J. of Symbolic Comp.*, 14(4):321–358, 1992.

10 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

11 Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206. Springer, 1999.

12 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Fifth International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 15–21, 2010.

13 Zachary Snow, David Baelde, and Gopalan Nadathur. A meta-programming approach to realizing dependently typed logic programming. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 187–198, 2010.

14 Mary Southern and Kaustuv Chaudhuri. A two-level logic approach to reasoning about typed specification languages. In *34th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 557–569, December 2014.

15 Roberto Virga. *Higher-order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

16 Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. Reasoning about higher-order relational specifications. In *15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*, pages 157–168, September 2013.

17 Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, 2003. Revised, May 2003.

18 The Abella web-site. `http://abella-prover.org/`, 2015.