Reasoning About Higher-Order Relational Specifications

Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, Gopalan Nadathur

Reasoning About Higher-Order Relational Specifications

Yuting Wang
University of Minnesota, USA
yuting@cs.umn.edu

Kaustuv Chaudhuri
INRIA, France
http://chaudhuri.info

Andrew Gacek
Rockwell Collins, USA
andrew.gacek@gmail.com

Gopalan Nadathur
University of Minnesota, USA
gopalan@cs.umn.edu

February 11, 2013

Abstract

The logic of hereditary Harrop formulas (HH) has proven useful for specifying a wide range of formal systems. This logic includes a form of hypothetical judgment that leads to dynamically changing sets of assumptions and that is key to encoding side conditions and contexts that occur frequently in structural operational semantics (SOS) style presentations. Specifications are often useful in reasoning about the systems they describe. The Abella theorem prover supports such reasoning by explicitly embedding the specification logic within a rich reasoning logic; specifications are then reasoned about through this embedding. However, realizing an induction principle in the face of dynamically changing assumption sets is nontrivial and the original Abella system uses only a subset of the HH specification logic for this reason. We develop a method here for supporting inductive reasoning over all of HH. Our approach takes advantage of a focusing property of HH to isolate the use of an assumption and the ability to finitely characterize the structure of any such assumption in the reasoning logic. We demonstrate the effectiveness of these ideas via several specification and meta-theoretic reasoning examples that have been implemented in an extended version of Abella.

1 Introduction

Computational systems are often presented and reasoned about through descriptions in the structural operational semantics (SOS) style. We are concerned here with a formalization of this process. A first requirement towards this end is a logical framework that can be used to make precise the content of SOS style presentations. To be suited to this task, the framework must satisfy certain criteria: it should facilitate the description of relations in a rule-based fashion, it should support the treatment of syntactic objects that incorporate binding notions, and it should provide a means for formalizing side conditions and (changing) contexts that are often part of SOS style rules. Past work, such as that described in [13], has demonstrated that these requirements are adequately met by a logic programming language equipped with the ability to use (typed) λ-terms as data structures and the capability of treating universal and hypothetical judgments as goals. The logic of hereditary Harrop formulas (HH) that has come out of such work has in fact been used successfully in a variety of formalizations in areas such as programming languages, process calculi and proof systems.

Informal reasoning based on SOS style presentations often accords them an inductive, closed-world reading. This is in contrast to the usual interpretation of logic programming specifications: the meaning of a relation can always be extended by the addition of new assertions. In order to use such specifications effectively in formalized reasoning, it is necessary to somehow treat them instead as fixed point definitions. One way of doing this is by directly enriching the specification logic. Another approach, called the *two-level logic approach* [7, 10], is to embed the specification logic in a second *reasoning logic* in a way that results in the specifications being given the desired inductive interpretation. This is the approach we adopt here. The specific reasoning logic that we will use is called $\mathcal{G}$ [6]; this logic brings together a collection of ideas developed towards providing a foundation for this approach [9, 14, 20]. $\mathcal{G}$ provides a means for associating fixed-point definitions with atomic predicates. The two-level logic approach is realized in its

setting by encoding derivability in HH as an inductive predicate. The closed-world reading of specifications results then from the inductive treatment of the derivations that can be carried out from them.

There are several benefits to the two-level logic approach, chief among them being the ability to prove and use properties of the specification logic in reasoning. These benefits have been demonstrated through the implemented Abella system [4, 5]. There is, however, a difficulty in using the full expressive power of HH specifications in this form of reasoning. Typical arguments based on such specifications involve induction on derivations from them in the underlying logic. However, many specifications, especially ones concerning objects that manifest binding structure, make use of universal and hypothetical judgments, leading to derivations in the specification logic in which the assumption sets are changing. This complicates the form of inductive arguments since new ways of proving assertions become available as the derivation proceeds. The original Abella system restricts the permitted HH specifications to a form where only atomic assertions can be added to an existing program for this reason: this restriction allows the additional atomic assertions that can be proved to be characterized via a simple definition in the reasoning logic that can be used in the inductive proof.

This paper develops a general method for overcoming the above-mentioned difficulty, thereby facilitating reasoning over specifications that use the full power of HH. The method we present is based on three key observations. First, derivations in HH use assumptions in a focused fashion: to be useful in constructing a proof, an assumption must match an atomic judgment that is to be shown in a specific way and then leads immediately to new atomic judgments that must be proved. Thus, an inductive analysis of derivations can pick a generic formula from a set of assumptions and proceed based only on its structure, without having to consider an interleaved processing of varied assumptions. Second, all the assertions that are added arise from a fixed original specification. Thus, the structure of these assertions can be finitely characterized, leading thereby to a bounded case analysis in an argument that is inductive over HH derivations. Finally, definitions in the reasoning logic provide the power to describe the structure of the assumptions that can be dynamically added and to prove properties about these additions that are useful in the reasoning process.

The ideas that we have outlined above are conceptually simple, yet they lead to elegant and effective reasoning techniques related to specifications that include embedded implications, *i.e.*, to *higher-order* specifications. We demonstrate this fact in the rest of the paper by describing an extended Abella system that embodies them and exhibiting its power through a varied collection of reasoning examples. The next two sections develop the context for the work by describing the HH logic and its embedding within the reasoning logic in more detail; the latter description builds the focused treatment of assumptions into the structure previously existing in Abella. Section 4 then discusses the process of reasoning about HH specifications in this (extended) framework. In particular, it highlights the difficulty in inductive reasoning arising from dynamically changing assumption sets and exhibits our method for overcoming this difficulty. Section 5 illustrates our ideas through two further examples, one pertaining to subtyping in F_{sub} [1, 17] and another concerning the analysis of reduction in the λ-calculus. We conclude the paper in Section 6 with a discussion of related work.

2 The Specification Logic

The logic HH is a predicative fragment of Church's Simple Theory of Types [2]. The expressions of HH are simply typed λ-terms. The types are determined by the function type constructor, denoted by $\rightarrow$ and written as an infix, right associative operator, from a collection of primitive types. The primitive types are determined by the user relative to each specification but must include o, the type corresponding to propositions, and at least one other type. Terms are constructed as usual from an (initial) user-provided signature that identifies constants with specific types. We write $\Sigma \vdash t : \tau$ to denote that t is a well-formed term of type τ relative to the signature Σ. Well-formed terms of type o are called Σ-*formulas* or just *formulas* when the signature is implicit. Equality between terms is determined by the rules of α-, β-, and η-conversion.

Logic is introduced into this background through a set of constants that are given a special status via inference rules. The *logical* constants in the HH setting are the implication and conjunction symbols $\Rightarrow$ and $\&$ of type $\mathsf{o} \rightarrow \mathsf{o} \rightarrow \mathsf{o}$ and, for every type τ not containing o, the (generalized) universal quantifier Π_τ of type $(\tau \rightarrow \mathsf{o}) \rightarrow \mathsf{o}$. An atomic formula, denoted by A possibly with a subscript, is one that does not have a logical constant as its head symbol. We write $\Rightarrow$ and $\&$ in infix form, treating the former as right and the latter as left associative. We often omit the type subscript τ from Π_τ. We also use the abbreviations $\Pi x{:}\tau.\, F$ for $\Pi\,(\lambda x{:}\tau.\, F)$ and $\Pi x_1{:}\tau_1, \ldots, x_n{:}\tau_n.\, F$ for $\Pi x_1{:}\tau_1.\ \ldots \Pi x_n{:}\tau_n.F$. Finally, we write $\Pi x.\, F$ for $\Pi x{:}\tau.\, F$ when the type is irrelevant or can be inferred from context.

Asynchronous rules
$$\frac{\Sigma;\Gamma, F \vdash G}{\Sigma;\Gamma \vdash F \Rightarrow G}\Rightarrow_R \qquad \frac{\Sigma;\Gamma \vdash G_1 \quad \Sigma;\Gamma \vdash G_2}{\Sigma;\Gamma \vdash G_1 \,\&\, G_2}\,\&_R \qquad \frac{(c \notin \Sigma) \quad \Sigma, c{:}\tau;\Gamma \vdash (G\,c)}{\Sigma;\Gamma \vdash \Pi_\tau\, G}\,\Pi_R$$

Synchronous rules
$$\frac{\Sigma;\Gamma \vdash G \quad \Sigma;\Gamma,[F] \vdash A}{\Sigma;\Gamma,[G \Rightarrow F] \vdash A}\Rightarrow_L \qquad \frac{\Sigma;\Gamma,[F_i] \vdash A}{\Sigma;\Gamma,[F_1 \,\&\, F_2] \vdash A}\,\&_L \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma;\Gamma,[F\,t] \vdash A}{\Sigma;\Gamma,[\Pi_\tau\, F] \vdash A}\,\Pi_L$$

Structural rules
$$\frac{}{\Sigma;\Gamma,[A] \vdash A}\,\mathsf{init} \qquad \frac{(F \in \Gamma) \quad \Sigma;\Gamma,[F] \vdash A}{\Sigma;\Gamma \vdash A}\,\mathsf{decide}$$

Figure 1: Rules for HH. In $\&_L$, $i \in \{1, 2\}$.

The HH proof system is given in the form of a focused sequent calculus that can be seen as a suitable (logical) fragment of the system LJF from [8]. Sequents in this calculus are of two kinds. *Asynchronous sequents* are of the form $\Sigma;\Gamma \vdash G$ where Σ is a signature, Γ is a multiset of formulas that form the assumptions of the sequent and will be called the *program clauses*, and G is an arbitrary formula called the *goal* of the sequent. *Synchronous sequents* are of the form $\Sigma;\Gamma,[F] \vdash A$, where Σ and Γ are as before, A is an atomic formula, and F is an arbitrary formula called the *focus*.

Fig. 1 contains the inference rules of HH. Reading the rules as a computation of premise sequents from goal sequents, the *asynchronous rules* decompose the goal in an asynchronous sequent—using right-introduction rules—until it becomes atomic. Then, the decide rule is used to select a single program clause for the focus of a synchronous sequent. The *synchronous* rules, which are left-introduction rules, are then used to decompose the focus. Eventually, when the focus is an atomic formula (called the *head* of its parent program clause), it will be matched to the goal using the init rule, finishing the (branch of the) proof.

The HH language supports a higher-order approach to representing syntactic structure [12, 15] that is also known as λ-tree syntax [11]. Abstractions in the simply typed λ-terms of HH are used to encode binding operators present in formal objects. For example, consider representing (untyped) λ-terms in HH. We identify a type $\mathtt{tm}$ for such terms and add the term constructors as two signature constants $\mathtt{app} : \mathtt{tm} \to \mathtt{tm} \to \mathtt{tm}$ and $\mathtt{abs} : (\mathtt{tm} \to \mathtt{tm}) \to \mathtt{tm}$, We can then define an encoding $\ulcorner - \urcorner$ of λ-terms as terms of type $\mathtt{tm}$ in the natural way; for example, $\ulcorner \lambda x.\, \lambda y.\, x\, y \urcorner = \mathtt{abs}(\lambda x.\, \mathtt{abs}(\lambda y.\, \mathtt{app}\, x\, y))$. The binding character of abstractions in the (object-level) λ-terms is captured by abstractions in HH. The object-level notions of α-conversion and capture-avoiding substitution are realized directly in terms of the same operations in HH. The logical understanding of binding available through such a representation also simplifies the process of reasoning about specifications as we see later.

We are ultimately interested in formalizing object-level relations that are presented in a structural operational semantics (SOS) style. Each such relation is formalized as an atomic formula of HH, and its rules are specified as program clauses added to the initial goal sequent. For example, assume an (object-level) primitive type $\mathtt{b}$ and consider the typing relation over λ-terms that is described by the following rules:

$$\frac{}{\Phi, x{:}T \triangleright x : T} \qquad \frac{\Phi, x{:}S \triangleright M : T}{\Phi \triangleright \lambda x.\, M : S \to T} \qquad \frac{\Phi \triangleright M : S \to T \quad \Phi \triangleright N : S}{\Phi \triangleright M\, N : T}$$

The expression $\Phi \triangleright M : T$ is used here to denote the judgment that M has type T in a context Φ that assigns types to the free variables of M. The rule for typing abstractions has an implicit proviso that x does not already appear in the domain of Φ. To encode these rules in HH, we first represent the object types. Let $\mathtt{ty}$ be this (HH) type representing object types, and let the signature contain two constants $\mathtt{b} : \mathtt{ty}$ and $\mathtt{arr} : \mathtt{ty} \to \mathtt{ty} \to \mathtt{ty}$ to represent the basic types and function types respectively. Let $\ulcorner T \urcorner$ stand for the encoding of types as terms of type $\mathtt{ty}$. The typing relation $M : T$ is realized by the atomic predicate $\mathtt{of} : \mathtt{tm} \to \mathtt{ty} \to \mathtt{o}$ in the signature. The typing rules can be translated into the following program clauses for the $\mathtt{of}$ predicate.

$$\mathtt{of}\, M\, (\mathtt{arr}\, S\, T) \Rightarrow \mathtt{of}\, N\, S \Rightarrow \mathtt{of}\, (\mathtt{app}\, M\, N)\, T \qquad (R_1)$$

$$(\Pi x.\, \mathtt{of}\, x\, S \Rightarrow \mathtt{of}\, (M\, x)\, T) \Rightarrow \mathtt{of}\, (\mathtt{abs}\, M)\, (\mathtt{arr}\, S\, T) \qquad (R_2)$$

We use the convention here and elsewhere of indicating the outermost Π-prefix in clauses implicitly by using upper-case letters for the variables they bind. Let Σ be the signature described thus far and Γ be the above pair of clauses R_1, R_2. Then, the typing judgment $x_1{:}T_1, \ldots, x_n{:}T_n \triangleright M : T$ is realized by the HH sequent: $\Sigma;\Gamma, \mathtt{of}\, x_1\, \ulcorner T_1 \urcorner, \ldots, \mathtt{of}\, x_n\, \ulcorner T_n \urcorner \vdash \mathtt{of}\, \ulcorner M \urcorner\, \ulcorner T \urcorner$. For instance, the judgment $\cdot \triangleright \lambda x.\, \lambda y.\, (x\, y) : (\mathtt{b} \to \mathtt{b}) \to (\mathtt{b} \to \mathtt{b})$ is represented by the HH sequent

$$\Sigma;\Gamma \vdash \mathtt{of}\, (\mathtt{abs}\, (\lambda x.\, \mathtt{abs}\, (\lambda y.\, (\mathtt{app}\, x\, y))))\, (\mathtt{arr}\, (\mathtt{arr}\, \mathtt{b}\, \mathtt{b})\, (\mathtt{arr}\, \mathtt{b}\, \mathtt{b})).$$

The context is not explicit in the representation of typing judgments, *i.e.*, $\mathtt{of}$ is a relation between only a term and a type. Instead, the context is realized using an embedded hypothetical judgment in R_2, the

$$\frac{(B \approx B')}{\Xi; \Delta, B \vdash B'}\,\mathsf{id} \qquad \frac{\Xi; \Delta \vdash B \quad \Xi; \Delta, B' \vdash C \quad (B \approx B')}{\Xi; \Delta \vdash C}\,\mathsf{cut}$$

$$\frac{\Xi, \Sigma, C \vdash t : \tau \quad \Xi; \Delta, B\,t \vdash C}{\Xi; \Delta, \forall_\tau B \vdash C}\,\forall_L \qquad \frac{(h \notin \Xi) \quad (\bar{c} = \mathrm{supp}(B)) \quad \Xi, h; \Delta \vdash B\,(h\,\bar{c})}{\Xi; \Delta \vdash \forall_\tau B}\,\forall_R$$

$$\frac{(a \in C \setminus \mathrm{supp}(B)) \quad \Xi; \Delta, B\,a \vdash C}{\Xi; \Delta, \nabla_\tau B \vdash C}\,\nabla_L \qquad \frac{(a \in C \setminus \mathrm{supp}(B)) \quad \Xi; \Delta \vdash B\,a}{\Xi; \Delta \vdash \nabla_\tau B}\,\nabla_R$$

Figure 2: Selected rules of $\mathcal{G}$.

program clause corresponding to `abs`. In the course of proof-search, such assumptions are accumulated as new program clauses for `of` corresponding to the bound variables. As an example, the HH derivation for the sequent above leads to

$$\Sigma, z{:}\mathtt{tm}, w{:}\mathtt{tm}; \Gamma, \mathtt{of}\,z\,(\mathtt{arr}\,\mathtt{b}\,\mathtt{b}), \mathtt{of}\,w\,\mathtt{b} \vdash \mathtt{of}\,(\mathtt{app}\,z\,w)\,\mathtt{b}$$

This sequent encodes typing a term with no abstractions in a typing context containing typing assumptions for the free variables of the term.

3 The Two-Level Logic Approach to Reasoning

Presentations in the SOS style are usually given a *closed-world* reading, wherein relations are considered to be characterized fully by the given inference rules. Thus, the rules shown earlier for the typing judgment for λ-terms are used not only to relate well-typed terms to their types, but also to argue that a term such as $\lambda x.\,(x\,x)$ cannot be typed. The HH logic realizes only the positive part of such an interpretation. To provide a complete formalization of the intended meaning of SOS style rules, we use the logic $\mathcal{G}$ [6] that supports fixed-point definitions.

The basis for $\mathcal{G}$ is also an intuitionistic and predicative version of Church's Simple Theory of Types. Types are determined in $\mathcal{G}$ as in HH except that the type of formulas is `prop` rather than `o`. The logical constants of $\mathcal{G}$ consist initially of $\top$ and $\bot$ of type `prop`; $\wedge$, $\vee$ and $\supset$ of type $\mathtt{prop} \to \mathtt{prop} \to \mathtt{prop}$; for every type τ not containing `prop`, the quantifiers $\forall_\tau$ and $\exists_\tau$ of type $(\tau \to \mathtt{prop}) \to \mathtt{prop}$; and the equality symbol $=_\tau$ of type $\tau \to \tau \to \mathtt{prop}$. The proof system for $\mathcal{G}$ is presented as a sequent calculus with sequents of the form $\Xi; \Delta \vdash C$ where Δ is a set of formulas (*i.e.*, terms of type `prop`), C is a formula, and Ξ contains the free eigenvariables in Δ and C. The treatment of fixed-point definitions in $\mathcal{G}$ results in the eigenvariables being given an extensional interpretation; in other words, unfolding a definition on the left may instantiate some of the eigenvariables and introduce other eigenvariables. To provide the capability of reasoning about *open* λ-terms, which is necessary for many kinds of reasoning over λ-tree syntax, $\mathcal{G}$ also supports *generic* reasoning. Specifically, for every type τ not containing `prop`, $\mathcal{G}$ includes an infinite set of *nominal constants* of type τ, and a *generic quantifier* ∇_τ of type $(\tau \to \mathtt{prop}) \to \mathtt{prop}$ [14]. We use C to denote the collection of all nominal constants, and assume that it is disjoint from the eigenvariables contained in Ξ and the (logical and non-logical) constants contained in the signature, Σ. We write $\Xi, \Sigma, C \vdash t : \tau$ to mean that t is a well-formed term of type τ all of whose free variables, constants, and nominal constants are drawn from the respective sets to the left of $\vdash$. Like with HH, we often omit types and adopt the usual syntactic conventions for displaying the logical connectives.

Nominal constants are used to simplify generic judgments in the course of proof search. A correct formalization of this idea needs two provisos: that quantifier scopes be respected and that judgments that differ only in the names of nominal constants be identified. Figure 2 contains a few rules of $\mathcal{G}$ that show these conditions are realized.[1] The essential feature of nominal constants is *equivariance*: two terms B and B' are considered to be equal, written $B \approx B'$, if they are λ-convertible modulo a permutation of the nominal constants. We write $\mathrm{supp}(B)$—called the *support* of B—for the (finite) collection of nominal constants occurring in B. The rules for ∇ are the same on both sides of the sequent; in each case a nominal constant that doesn't already exist in the support of the principal formula is chosen to replace the ∇-quantified variable. In the $\forall_R$ rule of Fig. 2, the eigenvariable is *raised* over the support of the principal formula; this is needed to express permitted dependencies on these nominal constants in a situation where later substitutions for eigenvariables will not be allowed to contain them. Note, however, that nominal constants may be used in witnesses in the $\forall_L$ rule.

To accommodate fixed-point definitions, $\mathcal{G}$ is parameterized by sets of *definitional clauses*. Each such clause has the form $\forall \bar{x}.\,(\nabla \bar{z}.\,A) \triangleq B$ where A is an atomic formula (called the *head*) whose free variables are drawn from $\bar{x}$ and $\bar{z}$, and B is an arbitrary formula (called the *body*) whose free variables are also free

[1] The full system can be found in [6].

$$\begin{array}{rl@{\qquad}rl}
\mathtt{seq}\, L\, (G_1 \,\&\, G_2) & \triangleq \mathtt{seq}\, L\, G_1 \wedge \mathtt{seq}\, L\, G_2 & \mathtt{sync}\, L\, (F_1 \,\&\, F_2)\, A & \triangleq \mathtt{sync}\, L\, F_1\, A \vee \mathtt{sync}\, L\, F_2\, A \\
\mathtt{seq}\, L\, (F \Rightarrow G) & \triangleq \mathtt{seq}\, (F :: L)\, G & \mathtt{sync}\, L\, (G \Rightarrow F)\, A & \triangleq \mathtt{seq}\, L\, G \wedge \mathtt{sync}\, L\, F\, A \\
\mathtt{seq}\, L\, (\Pi_\tau\, G) & \triangleq \nabla x{:}\tau.\, \mathtt{seq}\, L\, (G\, x) & \mathtt{sync}\, L\, (\Pi_\tau\, F)\, A & \triangleq \exists t{:}\tau.\, \mathtt{sync}\, L\, (F\, t)\, A \\
\mathtt{seq}\, L\, A & \triangleq \mathtt{atomic}\, A \wedge \mathtt{member}\, F\, L & \mathtt{sync}\, L\, A\, A & \triangleq \top \\
& \quad \wedge \mathtt{sync}\, L\, F\, A & &
\end{array}$$

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

in $\nabla\bar{z}.\, A$. Each clause partially defines a relation named by the predicate in the head. In every definitional clause $\forall\bar{x}.\, (\nabla\bar{z}.\, A) \triangleq B$, we require that $\mathrm{supp}(\nabla\bar{z}.\, A) = \mathrm{supp}(B) = \emptyset$. Consistency of $\mathcal{G}$ also requires predicate occurrences in the body of a clause to also satisfy certain *stratification* conditions that we do not discuss explicitly here for lack of space; see [6].

$\mathcal{G}$ includes special rules for treating definitions. When an atom occurs on the right of a sequent, then any of the clauses with a matching head may be used to replace the atom by the corresponding body of the clause; in other words, clauses may be *backchained* on. Matching the head of a clause $\forall\bar{x}.\, (\nabla\bar{z}.\, A) \triangleq B$ against the atom A', we look for a collection of nominal constants $\bar{c}$ and witness terms $\bar{t}$ that do not contain any of the elements of $\bar{c}$ such that $[\bar{t}/\bar{x}, \bar{c}/\bar{z}]A \approx A'$. If these can be found, then A' is replaced on the right by $[\bar{t}/\bar{x}]B$. When an atom A occurs on the left in a sequent, for every clause and every way of *unifying* the head of the clause to that atom, a new premise is created with the corresponding body added to the context. This amounts to a *case analysis* over the clauses in a definition. Note that substitutions into the clause must respect the order of the $\forall$ and ∇ quantifiers at its head. Some of the eigenvariables may be instantiated in the premises thus created so the eigenvariable context should be modified to reflect the resulting changes. The final crucial component in $\mathcal{G}$ is the ability to mark certain predicates as being *inductive*, whereby the set of clauses for that predicate is interpreted as a least fixed point definition. For lack of space, we do not describe the induction mechanism of $\mathcal{G}$ here; see [6, 7] for the details.

The proof system HH can be represented as an inductive definition in $\mathcal{G}$. The resulting ability to reason inductively about derivations in HH then indirectly yields a similar reasoning ability for any SOS system that has been formalized in HH. The similarity in the terms and types of HH and $\mathcal{G}$ permits a shallow representation of the former in the latter: every HH signature Σ is imported unchanged into $\mathcal{G}$. We additionally use the $\mathcal{G}$ type `olist` to represent contexts as lists of HH formulas, with constructors `nil` : `olist` and (`::`) : `o` $\to$ `olist` $\to$ `olist` and a standard definition of `member` : `o` $\to$ `olist` $\to$ `prop`. The asynchronous HH sequent $\Gamma \vdash G$ is encoded as the defined atom $\mathtt{seq}\, L\, G$; likewise, the synchronous HH sequent $\Gamma, [F] \vdash A$ is encoded as the defined atom $\mathtt{sync}\, L\, F\, A$; in either case, L is a list representation of Γ. The encoding of HH rules as clauses for `seq` and `sync` can be found in Fig. 3.[2] We use the standard notational convention of omitting the $\forall$-prefix on clauses and using upper-case letters to indicate $\forall$-bound variables. Compare Fig. 3 to Fig. 1: each inference rule of HH becomes a single definitional clause in $\mathcal{G}$. We will use the evocative notation $\{L \vdash G\}$ and $\{L, [F] \vdash G\}$ to stand for $\mathtt{seq}\, L\, G$ and $\mathtt{sync}\, L\, F\, A$. We also use commas instead of `::` in this notation, *i.e.*, $\{L, F \vdash G\}$ stands for $\mathtt{seq}\, (F :: L)\, G$, *etc*. Meta-theorems of HH can be proven in $\mathcal{G}$ in terms of this representation.

Proposition 1. *The following $\mathcal{G}$ sequents, representing the properties of monotonicity, cut-admissibility, and instantiation for* HH*, are all derivable.*[3]

1. $\cdot; \cdot \vdash \forall L, L', G.\, (\forall F.\, \mathtt{member}\, F\, L \supset \mathtt{member}\, F\, L') \supset \{L \vdash G\} \supset \{L' \vdash G\}$.
2. $\cdot; \cdot \vdash \forall L, F, G.\, \{L \vdash F\} \supset \{L, F \vdash G\} \supset \{L \vdash G\}$.
3. $\cdot; \cdot \vdash \forall L, G.\, \nabla x.\, \{L\, x \vdash G\, x\} \supset \forall t.\, \{L\, t \vdash G\, t\}$. □

4 Reasoning About HH Specifications

Many interesting HH specifications have a higher-order nature. This complicates inductive reasoning because the context in an HH sequent can be extended dynamically when unfolding the definition of `seq`. In the framework described in the previous section, this problem can be dealt with by qualifying the theorem to be proved through a characterization of these dynamic extensions. We explain this method below through two examples. The first example highlights the mechanism of inductive reasoning in $\mathcal{G}$ over HH derivations, and the second demonstrates the technique for dealing with dynamic extensions of the context.

[2] The predicate `atomic` : `o` $\to$ `prop` used here holds only of atomic HH formulas.

[3] The full proofs of these theorems in Abella can be found in Appendix A.

Our first pedagogical example will be a simple property of determinacy of addition of natural numbers. Suppose we identify a type nat of natural numbers and add two constructors z : nat and s : nat → nat and a predicate add : nat → nat → nat → o to the signature. The add predicate defines addition relationally, containing the two program clauses add z N N (named Z below) and add K M N ⇒ add (s K) M (s N) (named S below). We would like to prove that these program clauses entail that add is deterministic in its third argument. In terms of the encoding in Sec. 3, this amounts to proving the following $\mathcal{G}$ formula:

$$\forall x, y, z, w.\, \{\vdash \mathtt{add}\, x\, y\, z\} \supset \{\vdash \mathtt{add}\, x\, y\, w\} \supset z = w.$$

The fixed set of *static* program clauses is always assumed to be implicitly present in every HH sequent depicted using the {} notation in the rest of the paper. Thus, we omit the clauses Z and S to the left of ⊢ in the theorem above.

We prove this theorem by induction on the first assumption, {⊢add x y z}. The only rule that applies to $Z, S \vdash$ add x y z in HH is decide with the choices for the foci being the two program clauses Z and S. Focusing on Z gives us:

$$\dfrac{\dfrac{\dfrac{(x = \mathtt{z}) \quad (y = n) \quad (z = n)}{Z, S, [\mathtt{add}\, \mathtt{z}\, n\, n] \vdash \mathtt{add}\, x\, y\, z}\,\mathsf{init}}{Z, S, [\Pi n.\, \mathtt{add}\, \mathtt{z}\, n\, n] \vdash \mathtt{add}\, x\, y\, z}\,\Pi_L}{Z, S \vdash \mathtt{add}\, x\, y\, z}\,\mathsf{decide}$$

Each rule above corresponds to unfolding one of the clauses in Fig. 3. The only way for the derivation to complete with this focus is if it finishes with init, which requires that x = z and $y = z = n$ for some n:nat. This means that the second assumption of the theorem is {⊢add z n w}; the only way to prove this is by a decide on the program clause Z, because the head of S does not match the goal. Hence, it must be that $w = n$, so $z = w$.

In the other case when the focus was on S, we have:

$$\dfrac{\dfrac{Z, S \vdash \mathtt{add}\, k\, m\, n \quad \dfrac{(x = \mathtt{s}\, k) \quad (y = m) \quad (z = \mathtt{s}\, n)}{Z, S, [\mathtt{add}\, (\mathtt{s}\, k)\, m\, (\mathtt{s}\, n)] \vdash \mathtt{add}\, x\, y\, z}\,\mathsf{init}}{Z, S, [\Pi k, m, n.\, \mathtt{add}\, k\, m\, n \Rightarrow \mathtt{add}\, (\mathtt{s}\, k)\, m\, (\mathtt{s}\, n)] \vdash \mathtt{add}\, x\, y\, z}\,\Pi_L, \Rightarrow_L}{Z, S \vdash \mathtt{add}\, x\, y\, z}\,\mathsf{decide}$$

Once again, since the fourth premise must finish in init, it must be the case that $x = \mathtt{s}\, k$, $y = m$, and $z = \mathtt{s}\, n$, for some k, m, and n. We now have an additional $\mathcal{G}$ assumption {⊢add k m n} that comes from the third premise of this derivation; moreover, it is a strict sub-derivation (because the associated measure is smaller) and hence may be used for the inductive hypothesis. By inversion on the second assumption of the theorem (focusing on S again), it must be the case that there is some n' such that $w = \mathtt{s}\, n'$ and {⊢add k m n'}. By applying the inductive hypothesis, it must be that $n = n'$. Hence, $z = \mathtt{s}\, n = \mathtt{s}\, n' = w$.

The general structure of such $\mathcal{G}$ theorems is to perform an induction on the structure of a given seq assumption, written using {}, followed by a case analysis of the ways in which it might have been derived in HH. In the simple addition example, the dynamic program context is never extended, and it suffices to consider applications of decide where the focus is one of the static program clauses. This guarantees a finite number of cases to consider whenever the definition of seq is unfolded. Obviously, case-analyses should remain finite even when reasoning about higher-order specifications. This can only be achieved if the dynamic program context can be finitely characterized and this characterization can be encoded in the reasoning framework. There is a general method for doing so for any given HH specification, which we illustrate with our next example.

This example concerns the translation of λ-terms to De Bruijn form. To represent terms in De Bruijn form we introduce a new type dtm, and three signature constants dvar : nat → dtm, dapp : dtm → dtm → dtm and dabs : dtm → dtm. The dvar constructor is used to represent variable occurrences using De Bruijn indexes, which are the natural numbers defined previously. We describe the correspondence between an λ-term in the notation where bound variables are named and the De Bruijn form through the atomic formula hodb m h d where m:tm, d:dtm and h:nat; here h represents the number of λ-abstractions in scope. The predicate hodb with this property is defined by the following program clauses.

$$\mathtt{hodb}\, M\, H\, D \Rightarrow \mathtt{hodb}\, N\, H\, E \Rightarrow \mathtt{hodb}\, (\mathtt{app}\, M\, N)\, H\, (\mathtt{dapp}\, D\, E). \tag{P}$$

$$(\Pi x.\, (\Pi i, k.\, \mathtt{add}\, H\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\, (\mathtt{dvar}\, k)) \Rightarrow \mathtt{hodb}\, (M\, x)\, (\mathtt{s}\, H)\, D) \Rightarrow \mathtt{hodb}\, (\mathtt{abs}\, M)\, H\, (\mathtt{dabs}\, D). \tag{B}$$

The relation defined above is deterministic in its first and third arguments, *i.e.*, it constitutes an isomorphism between the two representations of λ-terms. Suppose we want to prove in $\mathcal{G}$ that it is deterministic in its third argument, *i.e.*,

$$\forall m, h, d, e.\, \{\vdash \mathtt{hodb}\, m\, h\, d\} \supset \{\vdash \mathtt{hodb}\, m\, h\, e\} \supset d = e.$$

If we try to prove this theorem by induction on one of the assumptions, like we did for add before, then we will get stuck when focusing on the program clause B. This case adds a new dynamic program clause, but the theorem as stated (and hence the induction) can only support an empty set of dynamic clauses. The remedy is to generalize the theorem to account for the dynamic clauses that may arise in the derivation being inducted over. That is, we look in the HH program clauses to find all the dynamic clauses that may be added when searching for a HH proof of $P, B \vdash \mathtt{hodb}\, m\, h\, d$. Focusing on B for such a sequent looks as follows, where G is $\mathtt{hodb}\,(\mathtt{abs}\, m)\, h\,(\mathtt{dabs}\, d)$.

$$\dfrac{\dfrac{x{:}\mathtt{tm};\, P, B, \Pi i, k.\, \mathtt{add}\, h\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\,(\mathtt{dvar}\, k) \vdash \mathtt{hodb}\,(m\, x)\,(\mathtt{s}\, h)\, d}{P, B \vdash (\Pi x.\, (\Pi i, k.\, \mathtt{add}\, h\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\,(\mathtt{dvar}\, k)) \Rightarrow \mathtt{hodb}\,(m\, x)\,(\mathtt{s}\, h)\, d} \quad \overline{P, B, [G] \vdash G}}{\dfrac{P, B, [B] \vdash G}{P, B \vdash G}\ \mathsf{decide}}$$

The first premise has a new dynamic program clause. Repeating the analysis with proofs of this premise, we observe that applications of decide on this dynamic clause will not extend the dynamic context further. Thus, all extensions of the dynamic context come from foci on B, and each such extension is with a program clause of the form $\Pi i, k.\, \mathtt{add}\, h\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\,(\mathtt{dvar}\, k)$ for some term $h{:}\mathtt{nat}$ and for some signature extension $x{:}\mathtt{tm}$.

In $\mathcal{G}$, we can fully characterize such dynamic contexts in terms of an inductive definition $\mathtt{ctx} : \mathtt{olist} \to \mathtt{prop}$ with the following definitional clauses.

$$\mathtt{ctx}\, \mathtt{nil} \triangleq \top \qquad\qquad (\nabla x.\, \mathtt{ctx}\,((\Pi i, k.\, \mathtt{add}\, H\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\,(\mathtt{dvar}\, k)) :: L)) \triangleq \mathtt{ctx}\, L.$$

Note the occurrence of ∇x at the head of the second clause in the definition of ctx: it guarantees that x does not occur in H or L. Therefore, in any L for which $\mathtt{ctx}\, L$ holds, it must be the case that there is exactly one such dynamic clause for each such $x \in \mathrm{supp}(L)$. It is easy to establish this fact in terms of a pair of theorems in $\mathcal{G}$, both proven by induction on ctx.

$$\forall L, E.\, \mathtt{ctx}\, L \supset \mathtt{member}\, E\, L \supset \exists x, h.\, E = (\Pi i, k.\, \mathtt{add}\, h\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\,(\mathtt{dvar}\, k)) \wedge \mathtt{name}\, x. \qquad (\chi_1)$$

$$\forall L, x, h_1, h_2.\, \mathtt{ctx}\, L \supset \mathtt{member}\,(\Pi i, k.\, \mathtt{add}\, h_1\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\,(\mathtt{dvar}\, k))\, L \supset \mathtt{member}\,(\Pi i, k.\, \mathtt{add}\, h_2\, k\, i \Rightarrow \mathtt{hodb}\, x\, i\,(\mathtt{dvar}\, k))\, L \supset h_1 = h_2. \qquad (\chi_2)$$

In χ_1, the predicate $\mathtt{name} : \mathtt{tm} \to \mathtt{prop}$ is defined by $(\nabla x.\, \mathtt{name}\, x) \triangleq \top$. Thus $\mathtt{name}\, x$ is true only if x is a nominal constant.

We strengthen the determinacy theorem using ctx to the following.[4]

$$\forall L, m, h, d, e.\, \mathtt{ctx}\, L \supset \{L \vdash \mathtt{hodb}\, m\, h\, d\} \supset \{L \vdash \mathtt{hodb}\, m\, h\, e\} \supset d = e.$$

Now, when we induct on one of the HH derivations, we can apply the induction hypothesis assuming that we can establish ctx for the extended dynamic context we encounter when unfolding the definition of seq.

In analyzing HH derivations, we also have to consider the possibility of using the dynamically added clauses. It is possible to prove the HH goal $\mathtt{hodb}\, m\, h\, d$ that appears in the second assumption by focusing on an element of L using decide. In terms of the definition of seq (Fig. 3), this amounts to unfolding $\{L \vdash \mathtt{hodb}\, m\, h\, d\}$ to yield $\exists F.\, \mathtt{member}\ F\ L \wedge \{L, [F] \vdash \mathtt{hodb}\, m\, h\, d\}$. This does not mean that we have to generalize the theorem further to mention synchronous sequents. Rather, because $\mathtt{ctx}\, L$ and $\mathtt{member}\ F\ L$, we can use the finite characterization of L in lemmas χ_1 and χ_2 to reveal the structure of F. It will be a clause with an atomic head, which allows us to unfold the sync definition fully and produce a collection of asynchronous (seq) premises.

To summarize, to reason about a higher-order HH specification, we must first determine all the possible forms of the dynamic context extensions in a HH derivation of a sequent, then write a $\mathcal{G}$ definition of these dynamic contexts, and then strengthen the theorems to include this finite characterization of dynamic contexts as an additional assumption. Then, in the induction, we use this finite characterization of the dynamic clauses to make the case-analysis finite.

5 Some Complex Examples of Higher-Order Reasoning

We now present two further examples to demonstrate the generality of our approach to reasoning about higher-order HH specifications. Each example uses an inductive definition in $\mathcal{G}$ to characterize program clauses that are added dynamically. Structural properties of these clauses are then proved as lemmas and used in the proofs of the main theorems. We highlight only the key applications of higher-order reasoning in the examples.[5]

[4] The full development of this example in Abella can be found in Appendix B.

[5] The full developments appear in Appendices C and D.

5.1 Transitivity of Subtyping in System F_{sub}

In [17], a variant of the POPLMark challenge problem 1a of showing the transitivity of subtyping in system F_{sub} was presented using an elegant higher-order encoding of the subtyping rules in Twelf. This development was proposed as a *proof pearl* because not only was the encoding natural but also the use of higher-order clauses allowed a succinct treatment of substitution in terms of function application in the meta-language. Substantially the same development can be achieved with HH and $\mathcal{G}$, which we demonstrate here with an Abella development that can be directly compared to the Twelf development in [17].

Briefly, the encoding of system F_{sub} types uses a type `tp` and the following signature constants: `top` : `tp` for the topmost type, `arr` : `tp` → `tp` → `tp` to represent function types $S \to T$, and `all` : `tp` → (`tp` → `tp`) → `tp` to denote the bounded subtype-polymorphic type $\forall a{\leq}S.\, T$. The subtyping relation $\leq$ is represented by `sub` : `tp` → `tp` → `o` with the following program clauses.

$$\mathtt{sub}\ T\ \mathtt{top}. \tag{S_1}$$

$$\mathtt{sub}\ T_1\ S_1 \Rightarrow \mathtt{sub}\ S_2\ T_2 \Rightarrow \mathtt{sub}\ (\mathtt{arr}\ S_1\ S_2)\ (\mathtt{arr}\ T_1\ T_2). \tag{S_2}$$

$$\begin{aligned}\mathtt{sub}\ T_1\ S_1 \Rightarrow (\Pi a.\ \ & (\Pi u, v.\ \mathtt{sub}\ a\ u \Rightarrow \mathtt{sub}\ u\ v \Rightarrow \mathtt{sub}\ a\ v) \Rightarrow \\ & \mathtt{sub}\ a\ a \Rightarrow \mathtt{sub}\ a\ T_1 \Rightarrow \mathtt{sub}\ (S_2\ a)\ (T_2\ a)) \Rightarrow \\ & \mathtt{sub}\ (\mathtt{all}\ S_1\ S_2)\ (\mathtt{all}\ T_1\ T_2).\end{aligned} \tag{S_3}$$

The first two clauses S_1 and S_2 are straightforward. In the third clause S_3, the second assumption has embedded program clauses defining transitivity and reflexivity of sub-typing for the bound type a, and the fact that it is below T_1. Using these clauses, the statement of transitivity of subtyping in $\mathcal{G}$ is as follows:

$$\forall L, s, q, t.\, \mathtt{ctx}\ L \supset \{L \vdash \mathtt{sub}\ s\ q\} \supset \{L \vdash \mathtt{sub}\ q\ t\} \supset \{L \vdash \mathtt{sub}\ s\ t\}$$

Just as in Sec. 4, we finitely characterize the dynamic contexts L using a definition `ctx` : `olist` → `prop` in terms of the following definitional clauses.

$$\mathtt{ctx}\ \mathtt{nil} \triangleq \top. \qquad \Big(\nabla a.\, \mathtt{ctx}(\ \mathtt{sub}\ a\ T :: \mathtt{sub}\ a\ a :: (\Pi u, v.\ \mathtt{sub}\ a\ u \Rightarrow \mathtt{sub}\ u\ v \Rightarrow \mathtt{sub}\ a\ v) :: L)\Big) \triangleq \mathtt{ctx}\ L.$$

Note that the ∇ at the head in the second clause guarantees that a does not occur in T or L. This lets us derive that if `ctx` L and `member` E L hold, then E must be one of the forms added to the context in the second clause.

The proof of transitivity proceeds by induction on the structure of the `tp` q (which requires an ancillary definition that we have elided here to simplify the presentation); for each form of q, the argument proceeds by case-analysis on the encoded HH sequent $\{L \vdash \mathtt{sub}\ s\ q\}$. Most cases are straightforward and follow the structure of the example of Sec. 4. The only interesting case is when q has the form `all` $q_1\ q_2$, whence the case analysis on the assumption $\{L \vdash \mathtt{sub}\ (\mathtt{all}\ s_1\ s_2)\ (\mathtt{app}\ q_1\ q_2)\}$ produces a premise (corresponding to decide on the static program clause S_3)

$$\{L, (\Pi u, v.\ \mathtt{sub}\ a\ u \Rightarrow \mathtt{sub}\ u\ v \Rightarrow \mathtt{sub}\ a\ v), \mathtt{sub}\ a\ a, \mathtt{sub}\ a\ q_1 \vdash \mathtt{sub}\ (s_2\ a)\ (q_2\ a)\} \tag{$\star$}$$

where a is some nominal constant. However, in order to establish the conclusion $\{L \vdash \mathtt{sub}\ (\mathtt{all}\ q_1\ q_2)\ (\mathtt{all}\ t_1\ t_2)\}$ using the inductive hypothesis we need to show:

$$\{L, (\Pi u, v.\ \mathtt{sub}\ a\ u \Rightarrow \mathtt{sub}\ u\ v \Rightarrow \mathtt{sub}\ a\ v), \mathtt{sub}\ a\ a, \mathtt{sub}\ a\ t_1 \vdash \mathtt{sub}\ (s_2\ a)\ (q_2\ a)\} \tag{$\dagger$}$$

In usual formal proofs of transitivity of subtyping in system F_{sub}, at this point one needs to prove a narrowing lemma to relax the assumption $\mathtt{sub}\ a\ q_1$ to $\mathtt{sub}\ a\ t_1$. However, in our case we can deftly avoid this distraction by using the meta-theoretic properties we have proven about HH derivations in $\mathcal{G}$ (Thm. 1). In particular, we know that $\{(\Pi u, v.\ \mathtt{sub}\ a\ u \Rightarrow \mathtt{sub}\ u\ v \Rightarrow \mathtt{sub}\ a\ v), \mathtt{sub}\ t_1\ q_1, \mathtt{sub}\ a\ t_1 \vdash \mathtt{sub}\ a\ q_1\}$, so by cutting against ($\star$) we obtain

$$\{L, (\Pi u, v.\ \mathtt{sub}\ a\ u \Rightarrow \mathtt{sub}\ u\ v \Rightarrow \mathtt{sub}\ a\ v), \mathtt{sub}\ a\ a, \mathtt{sub}\ a\ t_1, \mathtt{sub}\ t_1\ q_1 \vdash \mathtt{sub}\ (s_2\ a)\ (q_2\ a)\} \tag{$\ddagger$}$$

and since we can independently establish $\{L \vdash \mathtt{sub}\ t_1\ q_1\}$ (by a different application of the inductive hypothesis), a second cut against ($\ddagger$) gets us to the desired form ($\dagger$). The rest of the proof, particularly reasoning about the dynamic clauses, follows the outline of the example of Sec. 4.

5.2 Preservation of λ-Paths by Beta-Reduction

A *path* through a λ-term is a way to reach any non-binding occurrence of a variable in the term [13, Chap. 4.2]. In HH, we can identify a type `p` with paths with the following constructors in the signature: `left`, `right` : `p` → `p` to descend to the function or the argument sub-trees in an application, and

$\texttt{bnd} : (\texttt{p} \to \texttt{p}) \to \texttt{p}$ to descend through a λ-abstraction. Crucially, bnd has the same binding structure as the λ-abstractions encountered along the path. The predicate $\texttt{path} : \texttt{tm} \to \texttt{path} \to \texttt{o}$ asserts that a given λ-term contains a given path; it is defined by the following three program clauses.

$$\texttt{path}\, M\, P \Rightarrow \texttt{path}\,(\texttt{app}\, M\, N)\,(\texttt{left}\, P). \qquad \texttt{path}\, N\, P \Rightarrow \texttt{path}\,(\texttt{app}\, M\, N)\,(\texttt{right}\, P).$$
$$(\Pi x, p.\, \texttt{path}\, x\, p \Rightarrow \texttt{path}\,(M\, x)\,(P\, p)) \Rightarrow \texttt{path}\,(\texttt{abs}\, M)\,(\texttt{bnd}\, P).$$

As these paths record the specific structure of a λ-term, β-reduction changes the paths in the term. On the other hand, a path through the result of reducing $\texttt{app}\,(\texttt{abs}\,(\lambda x.\, M x))\, N$ would be a path through $M\, x$ with the additional proviso that any path through N is also a path through the variable x.

Suppose we want to compute the paths in a term that results from reducing certain marked β-redexes. Formally, we can add a new constructor for marked redexes, $\texttt{beta} : (\texttt{tm} \to \texttt{tm}) \to \texttt{tm} \to \texttt{tm}$ with the understanding that $\texttt{beta}\, M\, N$ denotes the same λ-term as $\texttt{app}\,(\texttt{abs}\, M)\, N$, except that the redex is marked. We can then define a relation $\texttt{bred} : \texttt{tm} \to \texttt{tm} \to \texttt{o}$ that reduces all the marked β-redexes in a term, with the following program clauses:

$$\texttt{bred}\, M\, U \Rightarrow \texttt{bred}\, N\, V \Rightarrow \texttt{bred}\,(\texttt{app}\, M\, N)\,(\texttt{app}\, U\, V).$$
$$(\Pi x.\, \texttt{bred}\, x\, x \Rightarrow \texttt{bred}\,(M\, x)\,(U\, x)) \Rightarrow \texttt{bred}\,(\texttt{abs}\, M)\,(\texttt{abs}\, U).$$
$$(\Pi x.\, (\Pi u.\, \texttt{bred}\, N\, u \Rightarrow \texttt{bred}\, x\, u) \Rightarrow \texttt{bred}\,(M\, x)\, V) \Rightarrow \texttt{bred}\,(\texttt{beta}\, M\, N)\, V.$$

We also add a new program clause for paths through a marked redex.

$$(\Pi x.\, (\Pi q.\, \texttt{path}\, N\, q \Rightarrow \texttt{path}\, x\, q) \Rightarrow \texttt{path}\,(M\, x)\, P) \Rightarrow \texttt{path}\,(\texttt{beta}\, M\, N)\, P.$$

Note that this HH specification has two different higher-order formulations. Proofs of $\texttt{bred}\, M\, U$ will add dynamic clauses involving bred, while proofs of $\texttt{path}\, M\, P$ will add dynamic clauses involving path. We would like to prove that bred preserves path, so the statement of the theorem would have to account for proofs of both kinds, and hence for both kinds of dynamic clauses. In many systems such as Twelf or Beluga, such a theorem would be proved in a common context containing both kinds of dynamic clauses. In $\mathcal{G}$, however, we can keep the two kinds of dynamic contexts separate, but *relate* them through a definition. The following definition of $\texttt{ctx2} : \texttt{olist} \to \texttt{olist} \to \texttt{prop}$ achieves this.

$$\texttt{ctx2}\ \texttt{nil}\ \texttt{nil} \triangleq \top. \qquad (\nabla x, p.\, \texttt{ctx2}\,(\texttt{bred}\, x\, x :: K)\,(\texttt{path}\, x\, p :: L)) \triangleq \texttt{ctx2}\, K\, L.$$
$$(\nabla x.\, \texttt{ctx2}\,((\Pi u.\, \texttt{bred}\, N\, u \Rightarrow \texttt{bred}\, x\, u) :: K)\,((\Pi p.\, \texttt{path}\, N\, p \Rightarrow \texttt{path}\, x\, p) :: L)) \triangleq \texttt{ctx2}\, K\, L.$$

The ctx2 predicate not only defines the dynamic contexts, but says how any two such contexts are related. As before, the ∇-bound variables at the head guarantee that every variable has a unique dynamic clause in both contexts. The $\mathcal{G}$ formula stating that bred preserves path is then as follows.

$$\forall K, L, m, u, p.\, \texttt{ctx2}\, K\, L \supset \{K \vdash \texttt{bred}\, m\, u\} \supset \{L \vdash \texttt{path}\, m\, p\} \supset \{L \vdash \texttt{path}\, u\, p\}.$$

This theorem is proved by induction on $\{K \vdash \texttt{bred}\, m\, u\}$. The technique outlined in Sec.4 and 5.1 works for most cases involving backchaining on the static program clauses. To handle backchaining on dynamic clauses, we will sometimes need lemmas such as the following:

$$\forall K, L, n, p.\, \nabla a.\, \texttt{ctx2}\,(K\, a)\,(L\, a) \supset \texttt{member}\ (\Pi q.\, \texttt{path}\, n\, q \Rightarrow \texttt{path}\, a\, q)\ (L\, a) \supset$$
$$\{L\, a \vdash \texttt{path}\, a\, p\} \supset \{L\, a \vdash \texttt{path}\, n\, p\}$$

Such an inversion property holds because, if the dynamic clause $\Pi q.\, \texttt{path}\, n\, q \Rightarrow \texttt{path}\, a\, q$ occurs in $L\, a$, then it must be the sole clause in $L\, a$ mentioning a.

6 Conclusion

We have presented an extension to the two-level logic framework that allows for the full richness of HH to be used in formalizing SOS style descriptions and we have exposed a method for reasoning about higher-order specifications in this enriched framework. We have validated our design and methodology by implementing an extended Abella system and by using it to develop a number of non-trivial examples of reasoning over higher-order specifications.

There are three systems besides Abella that are, broadly speaking, based on a two-level or nested reasoning approach and support higher-order abstract syntax: Twelf [16], Beluga [18] and Hybrid [3]. Hybrid is limited to the second-order hereditary Harrop fragment for the specification level, which makes it largely similar to the earlier version of Abella described in [7]. Beluga uses the LF type system for its object level and a dependently typed functional programming language for its meta-level. Instead of proving properties of relational specifications, in Beluga one writes recursive functions to manipulate LF data. To this

end, Beluga supports sophisticated reasoning about *contextual terms*, including a case-coverage checker for pattern-matching over such terms. However, Beluga lacks a termination checker, so it cannot verify that a recursive function proves a corresponding theorem.

Of the three systems, only Twelf is designed to reason about higher-order relational specifications. It has been shown that specifications in the LF language of Twelf can be systematically and faithfully translated into HH [19]. The image of this encoding of an LF signature in HH uses higher-order features pervasively, and was an early inspiration for the present work of supporting reasoning over higher-order specifications in Abella. Twelf's meta-level is not a logic but a family of fully automated meta-theoretic tools that can check properties asserted about LF specifications. These tools include a means of checking that a given inductive type family defines a total relation, *i.e.*, that it proves a theorem by induction. The major examples in this paper have also been done in Twelf to serve as a comparison of the two systems.

To reason about higher-order specifications, Twelf uses user-provided context *schemas* built out of a simple regular language of context *blocks*. Schemas are similar to the dynamic context definitions from Sec. 4 and 5, but less expressive. The principal difference is that in a single run of the totality checker, the entire LF specification shares the same dynamic context; thus, in the λ-paths example of Sec. 5.2, the theorem is proved in a context that contains both `bred` and `path` assumptions. To obtain a level of modularity, Twelf uses a sophisticated system of *context subsumption*, wherein a proof in a smaller context schema can be imported into a larger schema. This is sometimes a benefit; in $\mathcal{G}$, we must prove any such subsumption lemmas manually. On the other hand, because Abella represents context definitions using logic, properties about context definitions can be proven using lemmas and used in a modular fashion, as we have done in the examples in Sec. 4 and 5. Twelf does not support any kind of reasoning *about* context schemas directly, which both limits the modularity and increases the verbosity of Twelf proofs. Finally, Twelf's meta-reasoning can only check Π^1_0 theorems, whereas $\mathcal{G}$ theorems are not limited to any fragment of intuitionistic logic. For instance, an interesting theorem about λ-paths that is possible to prove in Abella is that two λ-terms that have the same paths must β-reduce to a common term; the statement of this theorem uses a quantifier alternation that is unavailable in Twelf's meta-reasoning.[6]

Acknowledgements: We thank Dale Miller for useful discussions about all aspects of this work. This work has been partially supported by the NSF Grants OISE-1045885 (REUSSI-2) and CCF-0917140 and by the INRIA *équipe associée* RAPT. 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.

References

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer, 2005.

[2] A. Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.

[3] A. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *J. of Automated Reasoning*, 48:43–105, 2012.

[4] A. Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.

[5] A. Gacek et al. The Abella system and homepage. `http://abella-prover.org/`, 2013.

[6] A. Gacek, D. Miller, and G. Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

[7] A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.

[6] See the theorem `same_paths_joinable` in the Abella development in Appendix D.

[8] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[9] R. McDowell and D. Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[10] R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

[11] D. Miller. Abstract syntax for variable binders: An overview. In J. Lloyd and *et al.*, editors, *CL 2000: Computational Logic*, number 1861 in LNAI, pages 239–253. Springer, 2000.

[12] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In S. Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, Sept. 1987.

[13] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

[14] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, Oct. 2005.

[15] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[16] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.

[17] B. Pientka. Proof pearl: The power of higher-order encodings in the logical framework LF. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference (TPHOLs)*, volume 4732 of *LNCS*, pages 246–261. Springer, 2007.

[18] B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 15–21, 2010.

[19] Z. Snow, D. Baelde, and G. 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.

[20] A. Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, volume 173 of *ENTCS*, pages 3–18, 2006.

A Meta-Theorems of HH

The file below, hh_meta.thm, gives formal proofs of the meta-theoretic properties of seq and sync (Thm. 1).

```
% Reasoning on multisets

% The definition (remove G A D) asserts that G is D extended with A.
Define remove : olist -> o -> olist -> prop by
  remove (A :: G) A G
; remove (B :: G) A (B :: D) := remove G A D.

% If G is an extension of D, then all members of D are also members of G.
Theorem remove_incl :
  forall G D A B, remove G A D -> member B D -> member B G.
induction on 1. intros. case H1.
  search.
  case H2.
    search.
    apply IH to H3 H4. search.

% If G is D extended with A, then any member of G is either A or a
% member of D.
Theorem remove_charac :
  forall G A D B, remove G A D -> member B G -> A = B \/ member B D.
induction on 1. intros. case H1 (keep).
  case H2.
    search.
    search.
  case H2.
    search.
    apply IH to H3 H4. case H5.
      search.
      search.

% Types of formulas and terms
Kind fm, tm type.

% Terms are left abstract, while formulas have the following constructors
Type atm tm -> fm.
Type and fm -> fm -> fm.
Type top fm.
Type imp fm -> fm -> fm.
Type all (tm -> fm) -> fm.

% Contexts are lists of formulas, but instead of defining a type of
% formula lists, we just reuse the olist type and keep things of
% the form ($fm A) in them.  In a polymorphically typed extension
% of Abella, we can avoid this hack.
Type $fm fm -> o.

% We will need to induct on the structure of formulas, so we write
% an inductive definition of all formulas.
Define is_fm : fm -> prop by
  is_fm (atm A)
; is_fm (and A B) := is_fm A /\ is_fm B
; is_fm top
; is_fm (imp A B) := is_fm A /\ is_fm B
; is_fm (all A)   := forall x, is_fm (A x).

% The focused sequent calculus consists of two phases: seq and sync
%
% sync L F A   stands for    L ; [F] |- A     (F under focus on the left)
% seq L G      stands for    L |- G
Define
  seq   : olist       -> fm -> prop,
  sync  : olist -> fm -> tm -> prop
by
  seq L (atm A)    := exists F, member ($fm F) L /\ sync L F A

; seq L (and G1 G2)  := seq L G1 /\ seq L G2
; seq L top
; seq L (imp F G)  := seq ($fm F :: L) G
; seq L (all A)    := nabla x, seq L (A x)

; sync L (atm A) A
; sync L (and F1 F2) A := sync L F1 A \/ sync L F2 A
; sync L (imp G F) A := seq L G /\ sync L F A
; sync L (all F) A   := exists t, sync L (F t) A.

% Note: the third argument to sync always represents an atom.
% This is because the atomic formula P(t1, ..., tn) is represented as
% (atm (P t1 ... tn)), where P has type (tm -> ... -> tm -> tm).
```

```
Theorem $monotone :
    (forall L L' C, (forall E, member E L -> member E L') -> seq L C -> seq L' C)
/\  (forall L L' F A, (forall E, member E L -> member E L') -> sync L F A -> sync L' F A).
induction on 2 2. split.
  intros. case H2 (keep).
    apply IH1 to H1 H4. apply H1 to H3. search.
    apply IH to H1 H3. apply IH to H1 H4. search.
    search.
    assert forall E, member E ($fm F :: L) -> member E ($fm F :: L').
      intros. case H4. search. apply H1 to H5. search.
     apply IH to H4 H3. search.
    apply IH to H1 H3. search.
  intros. case H2 (keep).
    search.
    case H3.
      apply IH1 to H1 H4. search.
      apply IH1 to H1 H4. search.
    apply IH to H1 H3. apply IH1 to H1 H4. search.
    apply IH1 to H1 H3. search.

Split $monotone as monotone_seq, monotone_sync.

Theorem $weakening :
   (forall L L' B G, remove L' B L -> seq L G -> seq L' G)
/\ (forall L L' B F A, remove L' B L -> sync L F A -> sync L' F A).
split.
  intros.
   assert forall E, member E L -> member E L'. intros. backchain remove_incl.
   backchain monotone_seq.
  intros.
   assert forall E, member E L -> member E L'. intros. backchain remove_incl.
   backchain monotone_sync.

Split $weakening as weakening_seq, weakening_sync.

Theorem member_inst :
   forall F L, nabla (n:tm), member (F n) (L n) -> forall t, member (F t) (L t).
induction on 1. intros. case H1.
  search.
  apply IH to H2. apply H3 with t = t. search.

% The instantiation lemma
Theorem $inst :
   (forall L G, nabla (n:tm), seq (L n) (G n) -> forall t, seq (L t) (G t))
/\ (forall L F A, nabla (n:tm), sync (L n) (F n) (A n) -> forall t, sync (L t) (F t) (A t)).
induction on 1 1. split.
  intros. case H1.
    apply IH1 to H3. apply member_inst to H2.
     apply H4 with t = t. apply H5 with t = t. search.
    apply IH to H2. apply IH to H3.
     apply H4 with t = t. apply H5 with t = t. search.
    search.
    apply IH to H2. apply H3 with t = t. search.
    apply IH to H2. apply H3 with t = t. search.
  intros. case H1.
    search.
    case H2.
      apply IH1 to H3. apply H4 with t = t. search.
      apply IH1 to H3. apply H4 with t = t. search.
    apply IH to H2. apply IH1 to H3.
     apply H4 with t = t. apply H5 with t = t. search.
    apply IH1 to H2. apply H3 with t = t. search.

Split $inst as inst_seq, inst_sync.

% The main cut-admissibility theorem.
Theorem $cut :
   (forall L K F G, is_fm F ->
     seq K F -> remove L ($fm F) K -> seq L G  -> seq K G)
/\ (forall L K F F' A, is_fm F ->
     seq K F -> remove L ($fm F) K -> sync L F' A -> sync K F' A)
/\ (forall L F A, is_fm F ->
     seq L F ->                       sync L F A -> seq L (atm A)).
% We proceed by nested induction on:
%
%  - the structure of the cut-formula, then
%  - the structure of the container derivation (i.e., the derivation
%     that contains the cut-formula as a hypothesis).
induction on 1 1 1.
induction on 4 4 3. split.
  intros. case H4 (keep).
    apply remove_charac to H3 H5. case H7.
      case H1 (keep).
```

```
        case H6. search.
        case H6 (keep). case H10.
          apply IH4 to H1 H2 H3 H11.
           case H2. apply IH2 to H8 H13 H12. search.
          apply IH4 to H1 H2 H3 H11.
           case H2. apply IH2 to H9 H14 H12. search.
        case H6.
        case H6.
         apply IH3 to H1 H2 H3 H10.
         apply IH4 to H1 H2 H3 H11.
         case H2 (keep).
         apply IH to H8 H12 _ H14.
         apply IH2 to H9 H15 H13. search.
        case H6.
         apply IH4 to H1 H2 H3 H9.
         case H2. apply inst_seq to H11. apply H12 with t = t.
          apply H8 with x = t.
          apply IH2 to H14 H13 H10. search.
      apply IH4 to H1 H2 H3 H6. search.
    apply IH3 to H1 H2 H3 H5.
     apply IH3 to H1 H2 H3 H6. search.
    search.
    assert remove ($fm F1 :: L) ($fm F) ($fm F1 :: K).
     assert seq ($fm F1 :: K) F. backchain weakening_seq.
     apply IH3 to H1 H7 H6 H5. search.
    apply IH3 to H1 H2 H3 H5. search.
  intros. case H4 (keep).
    search.
    case H5.
      apply IH4 to H1 H2 H3 H6. search.
      apply IH4 to H1 H2 H3 H6. search.
    apply IH3 to H1 H2 H3 H5.
     apply IH4 to H1 H2 H3 H6. search.
    apply IH4 to H1 H2 H3 H5. search.
  intros. case H3 (keep).
    search.
    case H4.
      case H1 (keep). case H2 (keep). apply IH2 to H6 H8 H5. search.
      case H1 (keep). case H2 (keep). apply IH2 to H7 H9 H5. search.
    case H1 (keep). case H2 (keep).
     apply IH to H6 H4 _ H8.
     apply IH2 to H7 H9 H5. search.
    case H1 (keep). case H2 (keep).
     apply H5 with x = t.
     apply inst_seq to H6. apply H8 with t = t.
     apply IH2 to H7 H9 H4. search.

Split $cut as cut, cut_commutative, cut_principal.
```

B Translating Between Higher-Order and De Bruijn Representations

`debruijn.sig` — the HH signature

```
sig debruijn.

kind nat type.
type z nat.
type s nat -> nat.
type add nat -> nat -> nat -> o.

kind ty  type.
type b   ty.
type arr ty -> ty -> ty.

kind tm type.
type app tm -> tm -> tm.
type abs (tm -> tm) -> tm.

kind dtm type.
type dapp dtm -> dtm -> dtm.
type dabs dtm -> dtm.
type dvar nat -> dtm.

type dat   nat -> ty -> o.

type hodb tm -> nat -> dtm -> o.
```

`debruijn.mod` — the program clauses for the De Bruijn translation

```
module debruijn.

add z C C.
```

```
add (s A) B (s C) :- add A B C.

hodb (app M N) H (dapp DM DN) :- hodb M H DM, hodb N H DN.
hodb (abs R) H (dabs DR) :-
  pi x\ (pi H'\ pi DX\ add H DX H' => hodb x H' (dvar DX)) =>
    hodb (R x) (s H) DR.
```

debruijn.thm — determinacy proofs

```
Specification "debruijn".

%% General property of member

Theorem member_prune : forall E L, nabla (x:tm),
  member (E x) L -> exists F, E = y\F.
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

%% Properties of addition

Define nat : nat -> prop by
  nat z ;
  nat (s X) := nat X.

Define le : nat -> nat -> prop by
  le X X ;
  le X (s Y) := le X Y.

Theorem le_dec : forall X Y,
  le (s X) Y -> le X Y.
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem le_absurd : forall X,
  nat X -> le (s X) X -> false.
induction on 1. intros. case H1.
  case H2.
  case H2. apply le_dec to H4. apply IH to H3 H5.

Theorem add_le : forall A B C,
  {add A B C} -> le B C.
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem add_absurd : forall A C,
  nat C -> {add A (s C) C} -> false.
intros. apply add_le to H2. apply le_absurd to H1 H3.

Theorem add_zero : forall A C,
  nat C -> {add A C C} -> A = z.
intros. case H2.
  search.
  case H1. apply add_absurd to H4 H3.

% add is deterministic in its first argument
Theorem add_det1 : forall A1 A2 B C,
  nat C -> {add A1 B C} -> {add A2 B C} -> A1 = A2.
induction on 2. intros. case H2.
  apply add_zero to H1 H3. search.
  case H3.
    case H1. apply add_absurd to H5 H4.
    case H1. apply IH to H6 H4 H5. search.

% add is deterministic in its second argument
Theorem add_det2 : forall A B1 B2 C,
  {add A B1 C} -> {add A B2 C} -> B1 = B2.
induction on 1. intros. case H1.
  case H2. search.
  case H2. apply IH to H3 H4. search.

%% Theorems specific to our translation

Define ctx : olist -> nat -> prop by
  ctx nil z ;
  nabla x,
    ctx ((pi H'\ pi DX\ add H DX H' => hodb x H' (dvar DX)) :: L) (s H) :=
      ctx L H.

Define name : tm -> prop by
  nabla x, name x.
```

```
Theorem ctx_nat : forall L H,
  ctx L H -> nat H.
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem ctx_inv : forall E L H,
  ctx L H -> member E L ->
   exists X HX,
     E = pi H'\ pi DX\ add HX DX H' => hodb X H' (dvar DX) /\
     name X /\ le (s HX) H.
induction on 1. intros. case H1.
  case H2.
  case H2.
    search.
    apply member_prune to H4. apply IH to H3 H4. search.

Theorem ctx_unique1 : forall L H X H1 H2,
  ctx L H ->
    member (pi H'\ pi DX\ add H1 DX H' => hodb X H' (dvar DX)) L ->
    member (pi H'\ pi DX\ add H2 DX H' => hodb X H' (dvar DX)) L ->
      H1 = H2.
induction on 2. intros. case H2.
  case H3.
    search. case H1. apply member_prune to H4.
  case H3.
    case H1. apply member_prune to H4.
    case H1. apply IH to H6 H4 H5. search.

Theorem ctx_unique2 : forall L H X1 X2 HX,
  ctx L H ->
    member (pi H'\ pi DX\ add HX DX H' => hodb X1 H' (dvar DX)) L ->
    member (pi H'\ pi DX\ add HX DX H' => hodb X2 H' (dvar DX)) L ->
      X1 = X2.
induction on 2. intros. case H2.
  case H3.
    search.
    case H1. apply ctx_inv to H5 H4. apply ctx_nat to H5.
      apply le_absurd to H8 H7.
  case H3.
    case H1. apply ctx_inv to H5 H4. apply ctx_nat to H5.
      apply le_absurd to H8 H7.
    case H1. apply IH to H6 H4 H5. search.

Theorem add_ignores_ctx : forall L H A B C,
  ctx L H -> {L |- add A B C} -> {add A B C}.
induction on 2. intros. case H2.
  search.
  apply IH to H1 H3. search.
  apply ctx_inv to H1 H4. case H3.


%% hodb is deterministic in its third argument
%% ie, higher-order --> debruijn is unique
Theorem hodb_det3 : forall L M D1 D2 H,
  ctx L H -> {L |- hodb M H D1} -> {L |- hodb M H D2} -> D1 = D2.
induction on 2. intros. case H2.
  case H3. apply IH to H1 H4 H6. apply IH to H1 H5 H7. search.
    apply ctx_inv to H1 H7. case H6. case H8.
  case H3. apply IH to _ H4 H5. search.
    apply ctx_inv to H1 H6. case H5. case H7.
  apply ctx_inv to H1 H5. case H4.
    case H3. case H6. case H6.
      apply ctx_inv to H1 H10. case H9.
      apply ctx_unique1 to H1 H5 H10.
      apply add_ignores_ctx to H1 H8. apply add_ignores_ctx to H1 H13.
      apply add_det2 to H14 H15. search.

Theorem hodb_det3_simple : forall M D1 D2,
  {hodb M z D1} -> {hodb M z D2} -> D1 = D2.
intros. apply hodb_det3 to _ H1 H2. search.


%% hodb is deterministic in its first argument
%% ie, debruijn --> higher-order is unique
%% proof is mostly the same as hodb_det3 except with fewer cases
Theorem hodb_det1 : forall L M1 M2 D H,
  ctx L H -> {L |- hodb M1 H D} -> {L |- hodb M2 H D} -> M1 = M2.
induction on 2. intros. case H2.
  case H3. apply IH to H1 H4 H6. apply IH to H1 H5 H7. search.
    apply ctx_inv to H1 H7. case H6.
  case H3. apply IH to _ H4 H5. search.
    apply ctx_inv to H1 H6. case H5.
  apply ctx_inv to H1 H5. case H4.
```

```
    case H3. apply ctx_inv to H1 H10. case H9.
    apply add_ignores_ctx to H1 H8. apply add_ignores_ctx to H1 H13.
    apply ctx_nat to H1. apply add_det1 to H16 H14 H15.
    apply ctx_unique2 to H1 H5 H10. search.

Theorem hodb_det1_simple : forall M1 M2 D,
  {hodb M1 z D} -> {hodb M2 z D} -> M1 = M2.
intros. apply hodb_det1 to _ H1 H2. search.
```

It is instructive to compare this example to the Twelf development found in the Twelf wiki here:

http://twelf.org/wiki/Concrete_representation

In Abella we have no need for an ancillary induction measure. Moreover, the translation hodb is manifestly a bijection.

C Transitivity of Subtyping in System F_{sub}

`fsub.sig` — the HH signature

```
sig fsub.

kind tp         type.
type top        tp.
type arr        tp -> tp -> tp.
type all        tp -> (tp -> tp) -> tp.

type sub        tp -> tp -> o.
```

`fsub.mod` — the program clauses for subtyping

```
module fsub.

sub T top.
sub (arr S1 S2) (arr T1 T2) :- sub T1 S1, sub S2 T2.
sub (all S1 S2) (all T1 T2) :-
    sub T1 S1,
    pi a\
      (pi U\ pi V\ sub a U => sub U V => sub a V) =>
      sub a T1 =>
      sub a a =>
        sub (S2 a) (T2 a).
```

`fsub.thm` — the proof of transitivity

```
Specification "fsub".

Define name : tp -> prop by
  nabla n, name n.

Define ctx : olist -> prop by
  ctx nil;
  nabla a, ctx ((sub a a) :: (sub a T) ::
       (pi U\ pi V\ sub a U => sub U V => sub a V) :: L) := ctx L.

Define tp : tp -> prop by
  tp top ;
  nabla x, tp x ;
  tp (arr T1 T2) := tp T1 /\ tp T2 ;
  tp (all T1 T2) := tp T1 /\ nabla x, tp (T2 x).

Theorem ctx_mem : forall L F,
  ctx L -> member F L -> exists A, name A /\
        ((F = sub A A) \/
         (exists T, F = sub A T) \/
         (F = pi U\ pi V\ sub A U => sub U V => sub A V)).
induction on 1. intros. case H1.
  case H2.
  case H2. search. case H4. search. case H5. search.
  apply IH to H3 H6. search.

Theorem ctx_sync : forall A L T,
  ctx L -> member (sub A T) L ->
    member (pi U\ pi V\ sub A U => sub U V => sub A V) L.
induction on 1. intros. case H1. case H2.
  case H2. search. case H4. search. case H5.
  apply IH to H3 H6. search.

Theorem ctx_sub_name : forall L D G,
  ctx L -> member D L -> {L, [D] |- G} -> exists A T, G = sub A T /\ name A.
```

```
intros. apply ctx_mem to H1 H2. case H5.
  case H3. search.
  case H3. search.
  case H3. search.

Theorem transitivity :
  forall L S Q T,
    ctx L -> tp Q -> {L |- sub S Q} -> {L |- sub Q T} -> {L |- sub S T}.
induction on 2.
intros. case H3.

  % sub S top.
  case H4. search.
    apply ctx_sub_name to H1 H6 H5. case H7.

  % sub (arr S1 S2) (arr T1 T2)
  case H4. search.
    case H2. apply IH to H1 H9 H7 H5. apply IH to H1 H10 H6 H8. search.

    apply ctx_sub_name to H1 H8 H7. case H9.

  % sub (all S1 S2) (all T1 T2)
  case H4. search.
    case H2. apply IH to H1 H9 H7 H5.
    assert({pi U\pi V\sub n1 U => sub U V => sub n1 V, sub T4 T1, sub n1 T4 |- sub n1 T1}).
    cut H12 with H7. cut H6 with H13.
    apply IH to _ H10 H14 H8. search.

    apply ctx_sub_name to H1 H8 H7. case H9.

  % backchain on the context
  apply ctx_mem to H1 H6. case H8.

    % sub a a
    case H5. search.

    % sub a T1
    case H5. apply ctx_sync to H1 H6. search.

    % pi U\pi V\sub a U => sub U V => sub a V
    case H5. search.
```

To compare with the Twelf implementation, see the file `pearl.elf` from [17].

D Preservation of λ-Paths by Beta Reduction and Joinability of λ-Terms with the Same Paths

`bred.sig` — the HH signature

```
sig breduce.

kind tm   type.
type abs  (tm -> tm) -> tm.
type app  tm -> tm -> tm.
type beta (tm -> tm) -> tm -> tm.

kind p type.
type left,right p -> p.
type bnd  (p -> p) -> p.

type bred tm -> tm -> o.

type path tm -> p -> o.

type bfree tm -> o.
```

`bred.mod` — the program clauses for β-reduction and paths

```
module breduce.

bred (abs M) (abs U) :-
  pi x\ bred x x => bred (M x) (U x).
bred (app M N) (app U V) :-
  bred M U, bred N V.
bred (beta R N) V :-
  pi x\ (pi u\ bred N u => bred x u)
    => bred (R x) V.

path (abs M) (bnd P) :-
  pi x\ pi p\ path x p => path (M x) (P p).
path (app M N) (left P) :-
```

```
  path M P.
path (app M N) (right P) :-
  path N P.
path (beta R N) P :-
  pi x\
    (pi q\ path N q => path x q) =>
    path (R x) P.

bfree (abs M) :- pi x\ bfree x => bfree (M x).
bfree (app M N) :- bfree M, bfree N.
```

`bred.thm` — proofs of path preservation in both directions and joinability of λ-terms with the same paths

```
Specification "bred".

Close tm, p.

Define ctx2 : olist -> olist -> prop by
  ctx2 nil nil
; nabla x p,
    ctx2 (bred x x :: G) (path x p :: D) := ctx2 G D
; nabla x,
    ctx2 ((pi u\ bred N u => bred x u) :: G)
         ((pi q\ path N q => path x q) :: D) :=
      ctx2 G D.

Define name : tm -> prop by
  nabla n, name n.

Define fresh : tm -> tm -> prop by
  nabla n, fresh n X.

Define pname : p -> prop by
  nabla p, pname p.

Theorem ctx2_mem_G :
    forall G D F,
    ctx2 G D -> member F G ->
    (   (exists x, F = bred x x /\ name x)
     \/ (exists x N, F = (pi u\ bred N u => bred x u) /\ fresh x N)).
induction on 1. intros. case H1.
  case H2.
  case H2.
    search. apply IH to H3 H4. case H5. search. search.
  case H2.
    search. apply IH to H3 H4. case H5. search. search.

Theorem ctx2_mem_D :
  forall G D F,
  ctx2 G D -> member F D ->
  (   (exists x p, F = path x p /\ name x /\ pname p)
   \/ (exists x N, F = (pi q\ path N q => path x q) /\ fresh x N) ).
induction on 1. intros. case H1.
  case H2.
  case H2.
    search. apply IH to H3 H4. case H5. search. search.
  case H2.
    search. apply IH to H3 H4. case H5. search. search.

Theorem ctx2_uniform :
  forall G D X, nabla n,
  ctx2 (G n) (D n) ->
  member (pi u\ bred X u => bred n u) (G n) ->
  member (pi q\ path X q => path n q) (D n).
induction on 1. intros. case H1.
  case H2.
  case H2. apply IH to H3 H4. search.
  case H2. apply IH to H3 H4. search.
  case H2. apply IH to H3 H4. search.
  case H2. search. apply IH to H3 H4. search.

Theorem member_prune_D :
   forall G D E, nabla (n:tm),
   ctx2 G D ->
   member (E n) D -> exists F, E = x\ F.
induction on 1. intros. case H1.
  case H2.
  case H2. search. apply IH to H3 H4. apply IH to H3 H4. search.
  case H2. search. apply IH to H3 H4. apply IH to H3 H4. search.

Theorem member_D_determinate :
   forall G D X Y, nabla n,
   ctx2 (G n) (D n) ->
   member (pi q\ path X q => path n q) (D n) ->
```

```
   member (pi q\ path Y q => path n q) (D n) ->
   X = Y.
induction on 1. intros. case H1.
  case H2.
  case H2. case H3. apply IH to H4 H5 H6. search.
  case H2. case H3. apply IH to H4 H5 H6. search.
  case H2. case H3. apply IH to H4 H5 H6. search.
  case H2. case H3.
    search. apply member_prune_D to H4 H5. apply member_prune_D to H4 H5.

Theorem member_D_discrim :
   forall G D X P, nabla n,
   ctx2 (G n) (D n) ->
   member (pi q\ path X q => path n q) (D n) ->
   member (path n P) (D n) ->
   false.
induction on 1. intros. case H1.
  case H2.
  case H2. case H3. apply IH to H4 H5 H6.
  case H2. apply member_prune_D to H4 H5.
  case H2. case H3. apply IH to H4 H5 H6.
  case H3. apply member_prune_D to H4 H5.

Theorem jump_D_invert :
   forall G D X P, nabla n,
   ctx2 (G n) (D n) ->
   member (pi q\ path X q => path n q) (D n) ->
   { D n |- path n P } -> { D n |- path X P }.
intros. case H3.
apply ctx2_mem_D to H1 H5. case H6.
  case H4. apply member_D_discrim to H1 H2 H5.
  case H4. case H7. apply member_D_determinate to H1 H2 H5. search.

Theorem bred_ltr :
  forall G D M N P,
  ctx2 G D ->
  { G |- bred M N } ->
  { D |- path M P } -> { D |- path N P }.
induction on 2.
intros. case H2 (keep).
  case H3.
    apply IH to _ H4 H5. search.
    apply ctx2_mem_D to H1 H6. case H7.
      case H8. case H5. case H8. case H5.
  case H3.
    apply IH to _ H4 H6. search.
    apply IH to _ H5 H6. search.
    apply ctx2_mem_D to H1 H7. case H8.
      case H9. case H6. case H9. case H6.
  case H3.
    apply IH to _ H4 H5.
     inst H6 with n1 = N1.
     assert {D |- pi q\ path N1 q => path N1 q}.
     cut H7 with H8. search.
    apply ctx2_mem_D to H1 H6. case H7.
      case H8. case H5. case H8. case H5.
  apply ctx2_mem_G to H1 H5. case H6.
    case H7. case H4. search.
    case H7. case H4.
     assert {D n1 |- path X P}.
       apply ctx2_uniform to H1 H5.
       apply jump_D_invert to H1 H9 H3. search.
     apply IH to H1 H8 H9. search.

Theorem bred_rtl :
  forall G D M N P,
  ctx2 G D ->
  { G |- bred M N } ->
  { D |- path N P } -> { D |- path M P }.
induction on 2.
intros. case H2 (keep).
  case H3.
    apply IH to _ H4 H5. search.
    apply ctx2_mem_D to H1 H6. case H7.
      case H8. case H5. case H8. case H5.
  case H3.
    apply IH to _ H4 H6. search.
    apply IH to _ H5 H6. search.
    apply ctx2_mem_D to H1 H7. case H8.
      case H9. case H6. case H9. case H6.
  assert {D, (pi q\ path N1 q => path n1 q) |- path N P}.
   apply IH to _ H4 H5. search.
  apply ctx2_mem_G to H1 H5. case H6.
    case H7. case H4. search.
```

```
    case H7. case H4.
     apply IH to H1 H8 H3.
     apply ctx2_uniform to H1 H5. search.


%%%%%%%%%%%%%%%%%%%%%%%%
% Two lambda terms must beta-reduce to a common term
% if they have the same paths
%%%%%%%%%%%%%%%%%%%%%%%%

Define bfctx : olist -> olist -> prop by
  bfctx nil nil
; nabla n p, bfctx (bfree n :: L) (path n p :: K) := bfctx L K.


Theorem member_prune_path : forall E L, nabla (x:p),
  member (E x) L -> exists F, E = y\F.
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem bfctx_member1 : forall X L K,
  bfctx L K -> member X L ->
    exists E F, X = bfree E /\ name E /\ member (path E F) K /\ pname F.
induction on 1. intros. case H1.
  case H2.
  case H2.
    search.
    apply IH to H3 H4. search.

Theorem bfctx_member2 : forall X L K,
  bfctx L K -> member X K -> exists E F, X = path E F /\ name E /\ pname F.
induction on 1. intros. case H1.
  case H2.
  case H2.
    search.
    apply IH to H3 H4. search.

Theorem member_path_unique : forall L K X Y F,
  bfctx L K -> member (path X F) K -> member (path Y F) K -> X = Y.
induction on 2. intros. case H2.
  case H3.
    search.
    case H1. apply member_prune_path to H4.
  case H3.
    case H1. apply member_prune_path to H4.
    case H1. apply IH to H6 H4 H5. search.

Theorem path_exists : forall L K M,
  bfctx L K -> {L |- bfree M} -> exists P, {K |- path M P}.
induction on 2. intros. case H2.
  assert bfctx (bfree n1 :: L) (path n1 n2 :: K).
    apply IH to H4 H3. search.
  apply IH to H1 H3. search.
  apply bfctx_member1 to H1 H4. case H3. search.

Theorem bfree_beta_absurd : forall L K R N,
  bfctx L K -> {L |- bfree (beta R N)} -> false.
intros. case H2.
apply bfctx_member1 to H1 H4. case H5. case H3.

Theorem path_app : forall L K M N Y,
  bfctx L K -> {L |- bfree (app M N)} -> {L |- bfree Y} ->
    (forall P, {K |- path (app M N) P} -> {K |- path Y P}) ->
      exists YM YN, Y = app YM YN.
intros. case H2.
  apply path_exists to H1 H5.
    assert {K |- path (app M N) (left P)}.
    apply H4 to H8.
    case H9.
      search.
      apply bfree_beta_absurd to H1 H3.
      apply bfctx_member2 to H1 H11. case H13. case H10.
  apply bfctx_member1 to H1 H6. case H5. case H7.

Theorem path_abs : forall L K R Y,
  bfctx L K -> {L |- bfree (abs R)} -> {L |- bfree Y} ->
    (forall P, {K |- path (abs R) P} -> {K |- path Y P}) ->
      exists YR, Y = abs YR.
intros. case H2.
  assert bfctx (bfree n1 :: L) (path n1 n2 :: K).
    apply path_exists to H6 H5.
    assert {K |- path (abs R) (bnd P)}.
    apply H4 to H8.
```

```
    case H9.
      search.
      apply bfree_beta_absurd to H1 H3.
      apply bfctx_member2 to H1 H11. case H10. case H13.
  apply bfctx_member1 to H1 H6. case H5. case H7.


Theorem bfree_sames :
  forall L K M N,
  bfctx L K ->
  {L |- bfree M} -> {L |- bfree N} ->
  (forall p, {K |- path M p} -> {K |- path N p}) ->
  M = N.
induction on 2.
intros. case H2 (keep).

  % M = (abs M1)
  apply path_abs to H1 H2 H3 H4. case H3.

    assert forall p, {K, path n1 n2 |- path (M1 n1) p} ->
                     {K, path n1 n2 |- path (YR n1) p}.
      intros.
      assert {K |- path (abs M1) (bnd p)}.
      apply H4 to H8.
      case H9.
        search.
        apply bfctx_member2 to H1 H11. case H13. case H10.

      assert bfctx (bfree n1 :: L) (path n1 n2 :: K).
        apply IH to H8 H5 H6 H7. search.

    apply bfctx_member1 to H1 H7. case H8. case H6.

  % M = (app M1 N1)
  apply path_app to H1 H2 H3 H4. case H3.

    % Prove M1 = YM
    assert forall p, {K |- path M1 p} -> {K |- path YM p}.
      intros.
      assert {K |- path (app M1 N1) (left p)}.
      apply H4 to H10.
      case H11.
        search.
        apply bfctx_member2 to H1 H13. case H15. case H12.
    apply IH to H1 H5 H7 H9.

    % Prove N1 = YN
    assert forall p, {K |- path N1 p} -> {K |- path YN p}.
      intros.
      assert {K |- path (app M1 N1) (right p)}.
      apply H4 to H11.
      case H12.
        search.
        apply bfctx_member2 to H1 H14. case H16. case H13.
    apply IH to H1 H6 H8 H10.

    % Finish this case
    search.

    apply bfctx_member1 to H1 H8. case H9. case H7.

  % M is a variable
  apply bfctx_member1 to H1 H6. case H5.
  assert {K |- path M F1}.
  apply H4 to H10. case H9.
  case H11.
    apply bfree_beta_absurd to H1 H3.
    apply bfctx_member2 to H1 H13.
      case H12. apply member_path_unique to H1 H8 H13. search.

Define brctx : olist -> olist -> prop by
  brctx nil nil
; nabla x,
    brctx (bred x x :: L) (bfree x :: K) :=
      brctx L K
; nabla x,
    brctx ((pi u\ bred N u => bred x u) :: L) K :=
      brctx L K.

Theorem brctx_mem_1 :
   forall L K E, brctx L K -> member E L ->
   (   (exists x, E = bred x x /\ name x)
    \/ (exists x N, E = (pi u\ bred N u => bred x u) /\ fresh x N)).
induction on 1. intros. case H1 (keep).
```

```
  case H2.
  case H2. search.
    apply IH to H3 H4. case H5.
      search. search.
  case H2. search.
    apply IH to H3 H4. case H5.
      search. search.

Theorem brctx_sync :
   forall L K, nabla x, brctx (L x) (K x) ->
   member (bred x x) (L x) ->
   member (bfree x) (K x).
induction on 1. intros. case H1.
  case H2.
  case H2. apply IH to H3 H4. search.
  case H2. search. apply IH to H3 H4. search.
  case H2. apply IH to H3 H4. search.
  case H2. apply IH to H3 H4. search.

Theorem bred_makes_bfree :
   forall L K M U,
   brctx L K -> {L |- bred M U} -> {K |- bfree U}.
induction on 2. intros. case H2 (keep).
  apply IH to _ H3. search.
  apply IH to _ H3. apply IH to _ H4. search.
  apply IH to _ H3. search.
   apply brctx_mem_1 to H1 H4. case H5.
     case H3. case H6.
      apply brctx_sync to H1 H4. search.
     case H3. case H6. apply IH to H1 H7. search.

Theorem same_paths_joinable :
  forall M N U V,
  (forall P, {path M P} -> {path N P}) ->
  {bred M U} -> {bred N V} -> U = V.
intros.
apply bred_makes_bfree to _ H2.
apply bred_makes_bfree to _ H3.
backchain bfree_sames. intros.
  apply bred_rtl to _ H2 H6.
  apply H1 to H7.
  apply bred_ltr to _ H3 H8. search.
```

`path.elf` — the development of path preservation (*i.e.*, the equivalent of `bred_ltr` and `bred_rtl` above) carried out in Twelf. Note that Twelf's meta-logic cannot express the equivalent of the `same_paths_joinable` theorem above, because it is not a Π^1_0 statement.

```
% Desc  : A twelf proof for the presevation of paths under
%         marked beta reductions in the simply typed lambda calculus

% lambda terms
tm : type.
lam : (tm -> tm) -> tm.
app : tm -> tm -> tm.
beta : (tm -> tm) -> tm -> tm.

% paths
pth : type.
left : pth -> pth.
right: pth -> pth.
bnd  : (pth -> pth) -> pth.

% beta reduction
breduce : tm -> tm -> type.

br-lam : breduce (lam M) (lam U)
  <- {x:tm} breduce x x -> breduce (M x) (U x).
br-app : breduce (app M N) (app U V)
  <- breduce M U
  <- breduce N V.
br-beta: breduce (beta R N) V
  <- {x:tm}({u:tm} breduce N u -> breduce x u)
     -> breduce (R x) V.
%block br-lam-blk : block {x:tm}{_:breduce x x}.
%block br-beta-blk : some {N:tm} block
                       {x:tm}{_:{u:tm} breduce N u -> breduce x u}.
%worlds (br-lam-blk | br-beta-blk) (breduce _ _).

% paths of terms
path : tm -> pth -> type.

p-lam : path (lam M) (bnd P)
 <- {x:tm}{p:pth} path x p -> path (M x) (P p).
```

```
p-appl: path (app M N) (left P)
         <- path M P.
p-appr: path (app M N) (right P)
         <- path N P.
p-beta: path (beta R N) P
 <- {x:tm}({q:pth} path N q -> path x q) -> path (R x) P.
%block p-lam-blk : block {x:tm}{p:pth}{_:path x p}.
%block p-beta-blk : some {N:tm} block
                      {x:tm}{_:{q:pth} path N q -> path x q}.
%worlds (p-lam-blk | p-beta-blk) (path _ _).

% beta reduction preserves the path (one direction)
breduce-ltr : breduce M N -> path M P -> path N P -> type.
%mode breduce-ltr +D1 +D2 -D3.

% lambda abstraction
- : breduce-ltr
     (br-lam Db : breduce (lam M) (lam U))
     (p-lam  Dp : path (lam M) (bnd P))
     (p-lam  Dpu : path (lam U) (bnd P))
     <- {x:tm}{p:pth}
        {bx:breduce x x}{px:path x p}
        breduce-ltr bx px px ->
        breduce-ltr
          (Db x bx : breduce (M x) (U x))
          (Dp x p px : path (M x) (P p))
          (Dpu x p px : path (U x) (P p)).

% application
- : breduce-ltr
     (br-app Dbn Dbm : breduce (app M N) (app U V))
     (p-appl Dpm : path (app M N) (left P))
     (p-appl Dpu : path (app U V) (left P))
     <- breduce-ltr Dbm Dpm Dpu.

- : breduce-ltr
     (br-app Dbn Dbm : breduce (app M N) (app U V))
     (p-appr Dpn : path (app M N) (right P))
     (p-appr Dpv : path (app U V) (right P))
     <- breduce-ltr Dbn Dpn Dpv.

% marked beta reduction
- : breduce-ltr
     (br-beta Db : breduce (beta R N) V)
     (p-beta Dp : path (beta R N) P)
     % apply Dv to arguments to get rid of the dependency on the hypotheses
     (Dv N ([q:pth][p:path N q]p) : path V P)
     <- {x:tm}
        {bb:{u:tm} breduce N u -> breduce x u}
        {pb:{q:pth} path N q -> path x q}
        (% backchaining on the context
           {u:tm}{q:pth}
           {Dbn : breduce N u}
           {Dpn : path N q}
           {Dpq : path u q}
           breduce-ltr (bb u Dbn) (pb q Dpn) Dpq
             <- breduce-ltr Dbn Dpn Dpq) ->
        breduce-ltr (Db x bb) (Dp x pb) (Dv x pb).

%block bltr-lam-blk : block
                  {x:tm}{p:pth}
                  {bx:breduce x x}{px:path x p}
                  {_:breduce-ltr bx px px}.
%block bltr-beta-blk : some {N:tm} block
                   {x:tm}
                   {bb:{u:tm} breduce N u -> breduce x u}
                   {pb:{q:pth} path N q -> path x q}
                   {_ :
                         {u:tm}{q:pth}
                         {Dbn : breduce N u}
                         {Dpn : path N q}
                         {Dpq : path u q}
                         breduce-ltr (bb u Dbn) (pb q Dpn) Dpq
                           <- breduce-ltr Dbn Dpn Dpq}.

%worlds (bltr-lam-blk | bltr-beta-blk) (breduce-ltr _ _ _).
%terminates D (breduce-ltr D _ _).
%covers (breduce-ltr +D1 +D2 -D3).
%total D (breduce-ltr D _ _).

% beta reduction preserves the path (another direction)
breduce-rtl : breduce M N -> path N P -> path M P -> type.
%mode breduce-rtl +D1 +D2 -D3.
```

```
% lambda abstraction
- : breduce-rtl
      (br-lam Db : breduce (lam M) (lam U))
      (p-lam  Dp : path (lam U) (bnd P))
      (p-lam Dp' : path (lam M) (bnd P))
      <- {x:tm}{p:pth}
         {bx:breduce x x}{px:path x p}
         {_ : breduce-rtl bx px px}
         breduce-rtl (Db x bx) (Dp x p px) (Dp' x p px).

% application
- : breduce-rtl
      (br-app Dbn Dbm : breduce (app M N) (app U V))
      (p-appl Dpu : path (app U V) (left P))
      (p-appl Dpm : path (app M N) (left P))
      <- breduce-rtl Dbm Dpu Dpm.

- : breduce-rtl
      (br-app Dbn Dbm : breduce (app M N) (app U V))
      (p-appr Dpv : path (app U V) (right P))
      (p-appr Dpn : path (app M N) (right P))
      <- breduce-rtl Dbn Dpv Dpn.

% marked beta reduction
- : breduce-rtl
      (br-beta Db  : breduce (beta R N) V)
      (Dpv         : path V P)
      (p-beta Dpr : path (beta R N) P)
      <- {x:tm}
         {bb:{u:tm} breduce N u -> breduce x u}
         {pb:{q:pth} path N q -> path x q}
         (% backchaining on the context
            {u:tm}{p:pth}
            {Dpp: path u p}
            {Dbn : breduce N u}
            {Dpn : path N p}
            breduce-rtl (bb u Dbn) Dpp (pb p Dpn)
              <- breduce-rtl Dbn Dpp Dpn) ->
         breduce-rtl (Db x bb) Dpv (Dpr x pb).

%block brtl-lam-blk : block
      {x:tm}{p:pth}
      {bx:breduce x x}{px:path x p}
      {_ : breduce-rtl bx px px}.
%block brtl-beta-blk : some {N:tm} block
      {x:tm}
      {bb:{u:tm} breduce N u -> breduce x u}
      {pb:{q:pth} path N q -> path x q}
      {_ :
            {u:tm}{p:pth}
            {Dpp: path u p}
            {Dbn : breduce N u}
            {Dpn : path N p}
            breduce-rtl (bb u Dbn) Dpp (pb p Dpn)
              <- breduce-rtl Dbn Dpp Dpn}.
%worlds (brtl-lam-blk | brtl-beta-blk) (breduce-rtl _ _ _).
%terminates D (breduce-rtl D _ _).
%covers (breduce-rtl +D1 +D2 -D3).
%total D (breduce-rtl D _ _).
```